QuickCheck használata szoftvertereszteléshez

Hoch Csaba

Témavezetők:

Prof. John Derrick
Department of Computer Science
University of Sheffield

Dr. Kozsik Tamás
Programozási Nyelvek és Fordítóprogramok Tanszék
Eötvös Loránd Tudományegyetem

Eötvös Loránd Tudományegyetem
Informatikai Kar

2008.

Using QuickCheck to test software

Csaba Hoch

Supervisors:

Prof. John Derrick
Department of Computer Science
University of Sheffield

Dr. Tamás Kozsik
Programozási Nyelvek és Fordítóprogramok Tanszék
Eötvös Loránd University

Eötvös Loránd University
Faculty of Informatics

2008.

Contents

1 Introduction

The programming language Erlang was developed by Ericsson. It was designed to support the development of distributed, fault-tolerant applications. It was used to implement the Ericsson AXD 301 high capacity ATM switch, which is used to implement the backbone of the telephony network in the UK. The software of this switch consists of 1.7 million lines of Erlang code [1].

Ensuring that programs do what they are intended to do is a crucial subject in the software industry, since software errors can have serious consequences. Thus reliability is becoming a key objective in the industry. The most popular technique for eliminating software errors is software testing, which is an empirical method, where the system is run and its behaviour is observed in different circumstances (the input, the events, the environment can be different). In contrast, formal methods use mathematics for analysing the code of the software, without running it. Formal methods can prove that a piece of software satisfies certain properties or conforms to a certain specification. However, the use of formal methods is usually more difficult than the use of testing.

One of the testing tools for Erlang is QuickCheck. It has two different versions, an open source and a commercial one. The testing method provided by both versions of QuickCheck is property-based and random-based at the same time. Property-based testing means that properties can be written about the program and the testing tool can test the program against these properties. Random based testing means that the test inputs of the program are randomly generated. There are many testing tools, which aid the tester in writing test suits. The test suits consists of many test cases; writing and maintaining them is very time consuming. Using random-based testing, the tester does not have to write test cases, these are generated automatically. QuickCheck runs these automatically generated test cases and examines whether the software satisfies the specified properties.

This dissertation gives a description about both versions of QuickCheck. Both versions are analysed, their testing methods are explained, and they are compared with each other. Many examples are given, and a case study, which was examined in both versions of QuickCheck.

Overview of the dissertation In section 2 the background for the rest of the dissertation is given. It includes an introduction to the Erlang programming language, and to Linear Temporal Logic.

Section 3 describes how QuickCheck works, what its underlying logic is and how it is worth to use it. The basic usage of QuickCheck is explained via a few simple examples in section 3.1. Section 3.2 talks about the more advanced features of QuickCheck, such as collection of the distribution of the test data, data generators and shrinking. Section 3.3 describes the differences between testing pure functions and systems with impure functions.

Section 3.4 explains testing systems with QuickCheck1, which is based on trace analysis; section 3.5 discusses a few problems and solutions in connection with it. Section 3.6 examines the test method of QuickCheck2, and an alternative test engine implementation is presented. The philosophy of QuickCheck is outlined in section 3.7.

Section 4 contains a case study, which was used to show how a more complex application can be tested with QuickCheck1 and QuickCheck2.

Finally, section 5 summarizes the dissertation.

2 Background

In this section, the background of the dissertation will be explained. QuickCheck tests Erlang programs and is written in Erlang, thus the first section will be about the Erlang programming language. Then Linear Temporal Logic will be described, which can be used in QuickCheck to express properties. Finally, the basics of testing will be covered.

2.1 Erlang

Erlang is a functional programming language with strong support for concurrency. It was designed to support distributed, fault-tolerant, soft-real-time, non-stop applications [11]. It was developed by Ericsson, and was released as open source in 1998.

In this section, those parts of Erlang will be explained, which are used in the current dissertation.

Hello world First let's see a Hello World program in Erlang:

```
1    -module(hello).
2    -export([say_hello/0]).
3
4    say_hello() ->
5        io:format("Hello, World!~n",[]).
```

The file which contains the program should be saved as `hello.erl`, because the name of the module is `hello`, as specified in the first line. The `export` annotation specifies, which functions will be exported, i.e. which functions will be accessed by other modules: now the `say_hello/0` function will be. (`say_hello/0` refers to the `say_hello` function which takes zero argument.) The last two lines define the `say_hello` function, which invokes the `io:format` function, which has the name `format`, and is placed in module `io`. `io:format` has two arguments: a string, which can contain a format sequence similar to the `printf` function in C, and a list of expressions, which has a similar purpose as to the other arguments in `printf`. This function call prints the string `"Hello, World!"`, and starts a new line. The period (`.`) marks the end of the function.

The program can be run from the Erlang Shell. It can be started with the `erl` command from a Unix shell:

```
$ erl
Erlang (BEAM) emulator version 5.5.2 [source] [async-threads:0]
[kernel-poll:false]

Eshell V5.5.2  (abort with ^G)
1>
```

The last line is the prompt. The user can type `c(hello).`, in order to compile the program:

```
1> c(hello).
{ok,hello}
2>
```

The successful compilation is indicated by printing the `{ok,hello}` line, which is the return value of the compiling command. A file called `hello.beam` is created, which contains the compiled module. The `say_hello` function of the program can be invoked by typing `hello:say_hello.`

```
2> hello:say_hello().
Hello, World!
ok
3>
```

The program printed `Hello, World!`, then finished with return value `ok`. When the user types an expression into the shell and wants it to evaluate it, then a period (`.`) has to be written after it.

The user can quit the Erlang Shell using the `q()` expression:

```
3> q().
ok
$
```

Types Erlang is a dynamically typed language. It has a type system, but types of variables are not checked in compilation time. The types which will be used in this dissertation are explained here.

Numbers Erlang has integers and floating-point numbers. They are similar to numbers in other programming languages.

E.g. integers are `3` and `-2`, a floating-point number is `3.1416`. The standard arithmetic operators can be used (`+`, `-`, `*`, `/`, `div`, `rem`), and a few others.

Atoms Atoms are non-numeric constant values. They usually start with lowercase letter, which is followed by a sequence of alphanumeric characters, underscore and `@` sign. If an atom does not have this form, then it has to be written between single quotation marks.

E.g. atoms are `f`, `atom`, `'This is an atom'`. Atoms can be used for similar purposes as enumerated types in other languages, but they do not need to be defined beforehand. The days of the week can be represented by atoms: `sunday`, `monday`, `tuesday`, ... Atoms such as `ok` and `error` are often used as return values. There is no special boolean type, atoms `true` and `false` are used instead.

Tuples Tuples represent a number of elements, typically only a few one. They are implemented as arrays. All the elements can be accessed in constant time, and creating a new tuple is linear.

A tuple can be specified by writing the elements between curly braces and separated by commas. E.g. tuples are `{}`, `{1,2,3,1}`, `{true,1}`, and `{1,{1,2},{{a}}}`.

Lists Lists represent a number of elements, typically many, and the programmer typically do not know, how many exactly. The first element is called the head of the list; the list which contains all the other elements is called the tail of the list. If the list has one element, then the tail is an empty list. The empty list does not have a tail.

Lists are implemented as linked lists, and they therefore have very different attributes than tuples. Accessing the nth element is a linear operation, but an element can be inserted to the beginning of the list in constant time.

Lists can be specified using similar syntax as tuples use, with square brackets instead of curly braces (e.g. `[1,2,3]` is a list). A list can be also given by specifying its head and tail: a pipe character needs to be written between the head and tail, and square brackets around them: `[1|[2]]`, `[1|[2|[]]]` are the same as `[1,2]`.

Variables In Erlang, variables do not vary. They do not have to be declared before use. When they are first used, they are bound to a value, and this value will never change. Variables start with uppercase letter, which is followed by a sequence of alphanumeric characters, underscore and `@` sign.

The following examples will be typed into the Erlang Shell. In this example the user tries to bind `X` to `10`, which is correct, but after that he tries to bind `X` again, which will fail:

```
1> X = 10.
10
2> X = 11.

=ERROR REPORT==== 21-Jan-2008::16:54:26 ===
Error in process <0.77.0> with exit value:
{{badmatch,11},[{erl_eval,expr,3}]}

** exited: {{badmatch,11},[{erl_eval,expr,3}]} **

3>
```

After the first command, `X` is bound to `10`. The second command fails, because `X` is already bound to `10`, so it cannot be bound to `11`.

Pattern matching Erlang can also unpack list and tuples:

```
1> {A,B} = {sunday,monday}.
{sunday,monday}
2> A.
sunday
3> B.
monday
4> [Head|Tail] = [1,2,3,4].
[1,2,3,4]
5> Head.
1
6> Tail.
[2,3,4]
```

The list `[1,2,3,4]` is the same as `[1|[2,3,4]]`, so `Head` has to be `1` and `Tail` has to be `[2,3,4]` to make the two sides of the equation equal.

The `=` operator does pattern matching in fact, which means that it tries to match the left and right hand side, and it can bind the variables in the left hand side, if that is needed to match the two patterns.

There is a special pattern, the underscore (`_`), which can mean any value:

```
1> [First,Second|_] = [5,6,7,8,9].
[5,6,7,8,9]
2> First.
5
3> Second.
6
4> {_,_,_,Day} = {true,{1},42,monday}.
{true,{1},42,monday}
5> Day.
monday
```

In the first example, the `_` stood for the value `[7,8,9]`. As the second example shows, if there are several underscores, they are not related to each other, they may or may not stand for the same value.

Expressions

Comma Comma (`,`) can be used to evaluate two or more expressions after each other.

```
1> io:format("Hello ",[]), io:format("World~n",[]).
Hello World
ok
2> 1+1, 2+2, 3+3.
6
```

The value of the compound expression is the value of the last one.

Case The `case` structure can be used to construct conditional expressions. This structure uses pattern matching, similarly to the equal operator (`=`). The `case` expression has the following form:

```
case Expression of
    Pattern_1 [when Guard_expression_1] -> Expression_1;
    Pattern_2 [when Guard_expression_2] -> Expression_2;
    ...
    Pattern_n [when Guard_expression_n] -> Expression_n
end
```

When evaluating the `case` expression, first the `Expression` is evaluated, then if `Guard_expression_1` is true, then the interpreter tries to match it with `Pattern_1`, if `Guard_expression_2` is true, it tries with `Pattern_2`, etc. It stops after the first match, calculates the value of the expression belonging to it: this will be the value of the `case` expression. The guard expressions are optional, they will not be used in the examples.

A factorial function, using `case`:

```
factorial(N) ->
    case N of
        0 -> 1;
        _ -> factorial(N-1) * N
    end.
```

`N` is first matched to `0`, so `factorial(0)` is `1`. If it is not `0`, then it is matched to `_`, which always works, since `_` stands for 'anything'. So if `N` is not `0`, `factorial(N)` is `factorial(N-1)*N`. The value of the latter will be calculated and that will be the value of the former.

Another example is a program, which prints different strings, depending on whether the arguments are equal or not:

```
-module(compare).
-export([compare_print/2]).

compare_print(X,Y) ->
    case X==Y of
        true  -> io:format("They are equal.~n",[]);
        false -> io:format("They are not equal.~n",[])
    end.
```

The program uses that the return value of the equality operator (`==`) is an atom: either `true` or `false`.

Modules and functions Erlang modules consist of annotations and functions. Every Erlang module has to be a separate file, and the name of the file has to be `Module.erl`, where `Module` is the name of the module.

Annotations Annotations start with a hyphen (`-`).
The most important annotations are:

- `-module`: Specifies the name of the module.
- `-include`: Includes the given file similar as the `#include` C preprocessor directive.
- `-include_lib`: Similar to `-include`, but it should not point out an absolute file. Instead, the first path component is assumed to be the name of an application.
- `-export`: Takes a list of function specification as an argument. The given functions will be exported from this module, so it can be accessed from outside the module. A function specification is the name and the arity of the function, separated by a slash symbol.

An example, which uses all annotations mentioned:

```
-module(hello).
-include("quickcheck.hrl").
-include_lib("eqc/include/eqc.hrl").
-export([f/0,f/1,g/1]).

f() -> ok.          % exported
f(X) -> 2*X.        % exported
f(X,Y,Z) -> X*Y*Z. % not exported
g(X) -> 3*X.        % exported
h(X) -> 4*X.        % not exported
```

As the example shows, there can be more function with the same name, but with different arities. These functions have nothing to do with each other: e.g. `f/1` is exported, while `f/3` is not.

Functions In Erlang modules, functions can be defined. The general form of the function definitions is as follows:

```
Function_name(Arg_list_1) [when Guard_expression_1] ->
    Expression_1;
Function_name(Arg_list_2) [when Guard_expression_2] ->
    Expression_2;
...
Function_name(Arg_list_n) [when Guard_expression_n] ->
    Expression_n.
```

When the function is called, the actual argument list will be matched against the formal argument lists with the given guard sequences, similarly when evaluating a `case` expression. Then the selected expression will be evaluated, and that will be the return value of the function, in other words, the value of the function call.

A factorial function, without `case`:

```
factorial(0) -> 1;
factorial(N) -> factorial(N-1) * N.
```

Anonymous functions can be defined, as well. They can have guard expressions and more bodies, like the functions above, but now we will deal only with the simple anonymous functions, which have the following form:

```
fun(Arg_list) -> Body end
```

These are expressions, their values are function references, which point to the functions.

They can be used as the following example shows:

```
1> F = fun(X) -> 2*X end.
#Fun<erl_eval.6.56006484>
2> F(4).
8
```

Concurrent programming Erlang was developed so that it is easier to write concurrent programs in it than in other languages. The concurrent model of Erlang is the Actor model. There are processes, which are lightweight: it is cheap to create and delete them. They can communicate by asynchronous message passing. Every process has a unique process id (*pid* for short). Pid is a separate data type, it is printed as a composition of three integers enclosed in angle brackets (e.g. `<0.100.1>`). The `self` function can be used to get the pid of a process: it returns the pid of the process, in which it was invoked.

Processes can be start with the `spawn` function. The `spawn` function gets a function as an argument: it can be either a function specification (`fun Function_name/Arity`), or an anonymous function (`fun(Arg_list) -> Body end`). The return value of the `spawn` function is pid of the newly created process.

The following expression is a sequence. The first part will start a process, which waits for a second, then prints `hello`, and the second part will be evaluated as `ok`:

```
1> spawn(fun()-> timer:sleep(1000), io:format("hello~n",[]) end), ok.
ok
hello
```

The first line shows the value of the whole expression, which is the `ok` atom. The second line shows the `hello`, printed by the process.

Processes can communicate with each other only via messages. Each process has a mailbox, which stores its unread messages. A message can be sent to a process with the exclamation mark operator:

```
Process ! Message
```

`Process` is the pid of the process, to which `Message` is sent. The message can be any Erlang term.

Processes can register (with the `register` function), which means that they will have a unique atom, as a name. (E.g. `register(my_server,Process)`.) After registration they can be sent messages with their names:

```
my_server ! Message
```

A process can receive a message (i.e. read its mailbox) using the `receive` expression:

```
receive
    Pattern_1 -> Expression_1;
    Pattern_2 -> Expression_2;
    ...
    Pattern_n -> Expression_n
[after
    Timeout -> Expression_timeout]
end.
```

(The `receive` expression can have guards, too.) When evaluating this expression, the process will wait until there is a message in the mailbox which matches a pattern, and then the corresponding expression will be evaluated and that will be the value of the expression. If a timeout is given and there is no matching pattern in the mailbox in `Timeout` milliseconds, then the `Expression_timeout` will be evaluated, the result will be the value of the expression.

The example application is a doubler server, which will send back the double of every number, which it receives. First the `start` function of the module should be called, which starts and registers the server process with the name `double_server`. The server process will run the `loop` function, which waits for either a tuple or a `stop` atom.

The user could send the tuple directly to the server and receive the result, but it is more convenient if the module already contains a function which does this task. The `double` function is for this: it can be used to call the double server. When called, it sends a tuple containing the id of the process (that is the `self()` function) and the number which should be doubled. The server is waiting in the receive block. When it receives the message, it goes to the first branch, and sends the doubled number back to the process which invoked the `double` function. Then it calls the `loop` function again, so it will wait for the next number or a `stop` message. On the other side, the `double` function receives the message of the server and returns it to the caller.

The server can be stopped by calling the `stop` function, which sends a `stop` atom to the server. The server process gets it, evaluates the `ok` atom – and does nothing else, there are no more expressions to evaluate. When the server terminates, it automatically unregisters.

```
-module(double).
-export([start/0,stop/0,double/1]).

start() ->
    register(double_server,spawn(fun loop/0)).

loop() ->
    receive
        {Pid,N} ->
            Pid ! 2*N,
            loop();
        stop ->
            ok
    end.

stop() ->
    double_server ! stop.

double(N) ->
    double_server ! {self(),N},
    receive
        N2 -> N2
    end.
```

The program can be used from the Erlang Shell in the following way:

```
14> double:start().
true
15> double:double(3).
6
16> double:stop().
stop
```

Why is it good that variables cannot vary? This restriction in Erlang has several advantages when writing sequential programs, e.g. it is easier to prove theorems about the program. But it has other important advantages, when writing concurrent programs. We will talk about implementation of concurrency in general, so when we use the word *process*, it can also mean *thread*, in fact.

If several processes are in a system, then the part of the memory used by the system can be sorted into the following categories:

1. Own: this memory area is used (read and written) only by one process.
2. Shared, modifiable: this memory area is used by many processes, and at least one can write it.
3. Shared, read-only: this memory area is used by many processes, but none of them can write it.

The main problem is the second category. Processes working on the same, modifiable memory area, must use techniques, such as semafors and mutexes, in order to avoid undesired interference. These techniques are error-phrone, and it is a difficult task to use them correctly. Erlang does not have this kind of memory area.

Without this category, processes has to communicate via messages – which is a good thing, because it is easier to write correct programs in which the processes communicate via messages than those in which the processes communicate via modifying shared memory. Now we have to ensure, that efficient programs can be written.

The problem with the first category is exactly that, the efficiency. If a process sends a data structure to another, then the data has to be copied from the memory of the first process to the memory of the second process. (Note: there is a way in Erlang to use this kind of memory, it is called process dictionary. But it is seldom needed.)

The second category does not have this problem. When a process sends a data structure to the other, only its address has to be sent. Since neither process can modify the data, it is guaranteed to have the same effect as if the data was really sent – the only difference is that it consumes less memory and costs less time.

To sum up, we can choose among the following alternatives:

1. Using mutexes and semafors.
2. Being unefficient.
3. Not varying the variables.

The Erlang chose the third alternative, and with this choice it made it easy to write correct concurrent programs.

2.2 Linear Temporal Logic

Linear temporal logic (LTL) is a modal temporal logic with modalities referring to time, that allows one to write properties such as '*something will happen infinitely often*' and '*something has to happen before something else happens*'. Using LTL, properties of programs can be described, a key component in both verification and testing; e.g. the QuickCheck tool uses LTL.

Before introducing LTL, we briefly summarize propositional logic, because LTL is an extension of that. The propositional logic works over propositional variables, that are usually represented by letters, e.g. P, Q, R. If each propositional variable is either true or false, it is called an interpretation; more precisely, an interpretation is a function, which assigns true or

false to each propositional variable (e.g. $\{P \to true, Q \to false, R \to false\}$ is an interpretation). So formally, an interpretation is a function from the set of propositional variables to the set of logical values.

Using the syntax of propositional logic well-formed formulas (or simply formulas) can be written (e.g. $P \wedge Q \to R$ is a formula). Each formula can be evaluated over an interpretation, and the result will be either true or false (e.g. the previous formula will be true in the previous interpretation).

In contrast, LTL has a concept of time, and the same propositional variable can have different values in different moments. The time in LTL can be imagined as a finite or infinite sequence of states (or moments), where a state is very much like an interpretation in propositional logic, i.e., each propositional variable is either true or false in each state. The time has a beginning (this is the moment 'now'), but does not necessarily have an end. Thus the states can be indexed with natural numbers or with an interval of natural numbers.

Interpretation is called path in LTL. In contrast with the propositional logic, here an interpretation (or a path) assigns a value to each propositional variable in each state. (E.g. '$\{P \to true, Q \to false\}$ in the first state, $\{P \to false, Q \to false\}$ in the second state and $\{P \to true, Q \to true\}$ in all the other states' is a path.) Formally, a finite path is a function, $\sigma : [1..n] \times V \to \mathbb{L}$, for some $n \in \mathbb{N}$. An infinite path is a function, $\sigma : \mathbb{N} \times V \to \mathbb{L}$.

As in propositional logic, formulas can be written using the syntactic rules of LTL. If a formula is given, it can be evaluated over a path, and the result will be either true or false. (E.g. the LTL-formula $\Box P$ means that 'the P propositional variable is true in each state'. Evaluating it over the previously mentioned path, the result will be false, since P is false in the second state of the path.)

Syntax The syntax of LTL is similar to the syntax of propositional logic: there are variables, unary and binary operators. All the operators of propositional logic can be found in LTL, and there are four additional operators: $\circ$, read as 'atnext'; $\Box$, read as 'always' or 'globally'; $\Diamond$, read as 'eventually' or 'sometimes'; $\mathcal{U}$, read as 'until'.

Here the syntax of LTL is given (i.e. the rules, with which the LTL formulas can be constructed). The explanation of the meaning of different operators is contained by the 'Semantics' paragraph.

- If P is a propositional variable, then P is an LTL formula.
- If A is an LTL formula, then $\neg A$ is an LTL formula.
- If A and B are LTL formulas, then $A \wedge B$, $A \vee B$ and $A \to B$ are LTL formulas.
- If A is an LTL formula, then $\circ A$, $\Box A$ and $\Diamond A$ are LTL-formulas.
- If A and B are LTL formulas, then $A \,\mathcal{U}\, B$ is an LTL-formula.

The precedence order from the strongest to the weakest operator is: $\circ$, $\Box$, $\Diamond$, $\neg$, $\mathcal{U}$, $\wedge$, $\vee$, $\to$

An example Let's suppose that there is a program with a variable `x`, and the following property should be described:

If the variable x *is positive twice in a row, then at some future time it will become higher than 10 and remain so permanently.*

This property can be expressed in LTL with the following formula:

$$\Box((\mathtt{x} > 0) \wedge \circ(\mathtt{x} > 0) \to \Diamond\Box(\mathtt{x} > 10))$$

Informally, we can see that this expresses the correct condition as follows.

$\mathtt{x} > 0$ means that `x` is higher than 10 in the first state, while $\circ(\mathtt{x} > 0)$ means that `x` is higher than 10 in the second state. Thus the formula $(\mathtt{x} > 0) \wedge \circ(\mathtt{x} > 0)$ means that `x` is higher than 10 in the first two states.

$\Box(\mathtt{x} > 10)$ means that `x` is always higher than 10; in other words it is higher than 10 in the first state and in all the following states. $\Diamond A$ means that A happens sometimes, or in other

words, there is a state in the future, when A is true. $\Diamond\Box(\mathtt{x} > 10)$ means therefore that there is a state in the future, when $\mathtt{x}$ is higher than 10 in that state and in all the following states.

Thus the formula $F = (\mathtt{x} > 0) \wedge \circ(\mathtt{x} > 0) \rightarrow \Diamond\Box(\mathtt{x} > 10)$ means that if $\mathtt{x}$ is higher than 10 in the first two states, then at some future time it will become higher than 10 and remain so permanently.

We must express that the same will happen if $\mathtt{x}$ is higher than 10 in the second and third states, or in the third and fourth states, etc.

$\circ F$ means that F is true in the second state. With the previous definition of F it means that if $\mathtt{x}$ is higher than 10 in the second and third states, then at some future time it will become higher than 10 and remain so permanently. (Because the second state now will be the first state in the second state.) Similarly, $\circ\circ F$ means that the same happens if $\mathtt{x}$ is higher than 10 in the third and fourth states.

Informally, we can express the original property (which is our goal to express in LTL) using the $\circ$ operator: $F \wedge \circ F \wedge \circ\circ F \wedge \circ\circ\circ F \wedge \ldots$ This is not an LTL formula, because it is not a finite sequence of symbols. But $\Box F$ is an LTL formula, and it means that F is true is all states, which is equivalent with all F, $\circ F$, $\circ\circ F$, ... being true.

So $\Box F = \Box((\mathtt{x} > 0) \wedge \circ(\mathtt{x} > 0) \rightarrow \Diamond\Box(\mathtt{x} > 10))$ is a correct formula to express the property.

Semantics First the concepts like *path*, *state* and *evaluation* will be formalized. After that the semantics of the LTL-formulas will be given.

Some notation that will be used:

- $\mathbb{L} = \{true, false\}$ is the set of boolean values,
- $\mathbb{N}$ is the set of natural numbers (beginning with 1),
- $[a..b]$, where $a, b \in \mathbb{N}$, is an interval which contains the integer numbers that are greater or equal to a and less or equal to b.

Let V be the set of propositional variables. If σ is a $[1..n] \times V \rightarrow \mathbb{L}$ function, where $n \in \mathbb{N}$, then σ is a finite path. If σ is an $\mathbb{N} \times V \rightarrow \mathbb{L}$ function, then σ is an infinite path. In both cases the domain of σ can be written as $\mathcal{D}_\sigma$. The size of the path is equal to the size of the domain and can be denoted by vertical bars: $|\sigma| = |\mathcal{D}_\sigma|$

σ can be written with sequence syntax: $\sigma = \langle\sigma_1, \sigma_2, \sigma_3, \ldots\rangle$, where each σ_i is a $V \rightarrow \mathbb{L}$ function. Note: when $\langle\sigma_1, \sigma_2, \sigma_3, \ldots\rangle$ is written, it does not mean that σ has at least 3 states, it is just a notation. A σ_i is a state in the path (a moment in the time). σ_1 is the first state, which is called 'now'; σ_2 is the second state, etc. If P is a propositional variable, σ is a path and $i \in \mathcal{D}_\sigma$ is an index, then P has a value in the state σ_i, i.e. $\sigma_i(P) \in \mathbb{L}$.

If an LTL formula F and a path σ are given, then F can be evaluated over σ, and the result will be either true or false. With formal notation: $E(F, \sigma) \in \mathbb{L}$ (E is the evaluation function).

Often when talking about the formal expression $E(F, \langle\sigma_i, \sigma_{i+1}, \ldots\rangle)$, people say '$F$ is true in σ_i' instead of saying 'F is true on $\langle\sigma_i, \sigma_{i+1}, \ldots\rangle$'. The reason is that the former is easier to say and imagine than the latter.

The evaluation can be done with the following rules (A and B are LTL formulas). If an LTL formula is not true, then it is false.

- The LTL formula *true* is true iff the path is not empty.
  Formally: $E(true, \sigma) = (|\sigma| \geq 1)$
- The LTL formula $false$ is true iff the LTL formula *true* is false. (I.e. when the path is empty.)
  Formally: $E(false, \sigma) = \neg E(true, \sigma)$
- If P is a propositional variable, then the LTL formula P is true iff the path is not empty and the P propositional variable is true in the first state.
  Formally: If $P \in V$, then $E(P, \sigma) = (|\sigma| \geq 1 \wedge \sigma_1(P))$
- $\neg A$ is true iff A is false.
  Formally: $E(\neg A, \sigma) = \neg E(A, \sigma)$
- $A \wedge B$ is true iff A and B are true.
  Formally: $E(A \wedge B, \sigma) = E(A, \sigma) \wedge E(B, \sigma)$

- $A \vee B$ is true iff A or B is true.
  Formally: $E(A \vee B, \sigma) = E(A, \sigma) \vee E(B, \sigma)$

- $A \rightarrow B$ is true iff A implies B (i.e. A is false or B is true).
  Formally: $E(A \rightarrow B, \sigma) = E(A, \sigma) \rightarrow E(B, \sigma)$

- $\circ A$ is true iff A is true in the second state. More precisely: $\circ A$ is true iff the path is not empty and A is true over that suffix of the original path, which begins at the second state. Often the terminology of the previous sentence is used, but actually the second one is the more precise, as it can be seen from the formal definition.
  Formally:
  if $|\sigma| = 0$, then $E(\circ A, \sigma) = false$;
  if $|\sigma| \geq 1$, then $E(\circ A, \langle \sigma_1, \sigma_2, \sigma_3, \ldots \rangle) = E(A, \langle \sigma_2, \sigma_3, \ldots \rangle)$

- $\Box A$ is true iff A is true in all states. More precisely: $\Box A$ is true iff A is true on all the suffixes of the path.
  Formally: $E(\Box A, \sigma) = (\forall i \in \mathcal{D}_\sigma)(E(A, \langle \sigma_i, \sigma_{i+1}, \ldots \rangle))$

- $\Diamond A$ is true iff there is at least one state where A is true. More precisely: $\Diamond A$ is true iff there is at least one suffix of the path on which A is true.
  Formally: $E(\Diamond A, \sigma) = (\exists i \in \mathcal{D}_\sigma)(E(A, \langle \sigma_i, \sigma_{i+1}, \ldots \rangle))$

- $A \,\mathcal{U}\, B$ is true iff B holds until the first state when A is true. B does not have to be true in the first state where A is true. [1]
  Formally:
  $E(A \,\mathcal{U}\, B, \sigma) = (\forall i \in \mathcal{D}_\sigma)(first(A, i, \sigma) \rightarrow (\forall j \in [1..(i-1)])(E(B, \langle \sigma_j, \sigma_{j+1}, \ldots \rangle)))$, where
  $first(A, i, \langle \sigma_1, \sigma_2, \ldots \rangle) = E(A, \langle \sigma_i, \sigma_{i+1}, \ldots \rangle) \wedge (\forall j \in [1..(i-1)])(\neg E(A, \langle \sigma_j, \sigma_{j+1}, \ldots \rangle))$.
  $first(A, i, \sigma)$ means that 'i is the first index of σ, where A is true'.

Examples These examples may help understand LTL formulas.

When a path $\langle \sigma_1, \sigma_2, \ldots \rangle$ is given, the ith suffix of the path is the path $\langle \sigma_i, \sigma_{i+1}, \ldots \rangle$, i.e. that suffix of the path which has the original ith state as its first state. The first suffix of a path is the path itself.

Often when we have a path, we say A is true in the ith state of the path instead of saying that A is true on the ith suffix of the path (formally $E(A, \langle \sigma_i, \sigma_{i+1}, \ldots \rangle) = true$).

1. $\Box \neg P$: never P.
   $\Box \neg P$ means that $\neg P$ is always true, i.e. P is always false, i.e. P is never true.

2. $\Box(P \vee Q)$: P or Q is true in each state.

3. $\Diamond(\neg P \wedge \neg Q)$: there is at least one state, where both P and Q are false. This is the opposite of the previous example.

4. $\circ \Box P$: P is always true from the second state.
   $\circ A$ means that A is true in the second suffix of the path, which contains every state from the original path except for the first one. So $\circ \Box P$ means that P always has to hold except for the first state.

5. $\Box \Diamond P$: P is true infinite times in the path, if the path is infinite. (Note: If the path is finite, this formula means that P is true at the last state.)
   $\Box \Diamond P$ means that $\Diamond P$ is always true, i.e. $\Diamond P$ holds on each suffix of the path, i.e. each suffix of the path contains a state, where P is true. If the path is infinite, then it implies that P is true infinite times; otherwise (if P was true finite times) there would be a suffix of the path, in which there is not a state where P is true.

6. $\Diamond \Box P$: from one point in the path P will be always true.
   $\Box P$ means that P is true in each state; $\Diamond \Box P$ means that there is a suffix of the path when $\Box P$ is true.

[1] Conventionally $A \,\mathcal{U}\, B$ means that A holds until the first state when B is true. However, QuickCheck uses it in the other way, thus in this dissertation QuickCheck's notation will be used.

Also conventionally (but using QuickCheck's notation) $A \,\mathcal{U}\, B$ can be true only if A happens eventually. QuickCheck's semantics are used again in this dissertation.

7. *true*: The path is not empty.
   If the path is empty, then the formula *true* is false and the formula $false$ is true by definition.

8. $\circ true$: The path contains at least two states.
   The operator $\circ$ implies that the path is not empty, and $\circ true$ means that *true* is true on the second suffix of the original path. The latter means (see example 7) that the second suffix of the original path is not empty; so the original path has at least two states.

9. $true \wedge \Box \circ true$: The path is infinite.
   The LTL formula *true* means that the path is not empty, i.e. it has a first state (see example 7). Because of $\Box \circ true$, $\circ true$ is true on all the suffixes of the path. Its being true on the first suffix means that there is a second state, therefore a second suffix (see example 8). Its being true on the second suffix means that there is a third state, therefore a third suffix. We can continue it until any natural number, so the path is infinite.

10. $\Diamond(A \wedge \circ\Diamond A)$: A happens at least twice.
    $\circ\Diamond A$ means that A is true in a state in the second suffix, so it is true in a state, which is not the first one. $A \wedge \circ\Diamond A$ means that A is true in the first state and after that in another state. $\Diamond(A \wedge \circ\Diamond A)$ means therefore that there is a state in the path, when A is true and after that there is another one when A is true again.

11. $L \,\mathcal{U}\, \neg M$: M will not happen until L happens. Or in other words: M can happen only if L happened previously. This is basically the same LTL formula which is used in section 3.4.2 to describe a property of a program based on a specification.

The following equations are used in the QuickCheck's implementation to check whether a path (or a trace in the QuickCheck's terminology) satisfies a formula.

- $(|\sigma| \geq 1) \rightarrow (E(\Box A, \sigma) = E(A \wedge \circ\Box A, \sigma))$
  If a path is not empty, then 'always A' means that A is true in the first, second, third state, etc. In other words: A is true in the first state, and A is true in the second, third, fourth state, etc. The latter clause can be expressed with $\circ\Box A$.

- $\Diamond A = \neg\Box\neg A$.
  $\neg\Box\neg A$ means that A is not always false, which is the same as A is sometimes true.

- If σ is finite:
  $E(A \,\mathcal{U}\, B, \sigma) = (|\sigma| = 0) \vee E(A, \sigma) \vee (\neg E(B, \sigma) \wedge E(\circ(A \,\mathcal{U}\, B), \sigma))$
  Let's suppose that $A \,\mathcal{U}\, B$ has to be calculated for a given path σ. So the following should be decided: does A hold until B becomes true? If the path is empty ($|\sigma| = 0$) or A is true in the first state ($E(A, \sigma)$), then the answer is yes. Otherwise B has to be false ($\neg E(B, \sigma)$), because if it was true, and A was false, then $A \,\mathcal{U}\, B$ would be false. So now we can assume that the path is not empty and both A and B are false in the first state. In this case the $E(A \,\mathcal{U}\, B, \sigma)$ is the same as $E(\circ(A \,\mathcal{U}\, B), \sigma)$, because both formulas mean that either B is always false, or there is a state when A holds after the first state of σ and before the first state when B holds.

3 Testing with QuickCheck

QuickCheck is a testing tool originally developed for testing Haskell programs by Koen Claessen and John Hughes in 2000 [3]. It was created to automatize property-based testing. Basically it works in the following way: First the tester writes properties, which should be satisfied by the program or function under test (e.g. the output number is the greatest common divisor of the input numbers; the output list has the same elements as the input list, but it is sorted; deadlock never occurs; there is no starvation, etc.). After that QuickCheck generates random inputs, and the properties written about the program or function are checked with these inputs.

After the successful implementation of the original version, QuickCheck has been reimplemented in various languages: Erlang [4], Scheme [6], Common Lisp [7], Python [7], Ruby [8], Scala [9], and Standard ML [10].

Erlang/QuickCheck, the Erlang version of the tool was introduced by Thomas Arts and John Hughes in 2003 [4]. Since the introduction of the first version, which was open source, the tool has been reimplemented and made into a commercial product, which is called QuviQ QuickCheck.

This dissertation deals only with Erlang/QuickCheck, so the 'Erlang' prefix will be omitted and just 'QuickCheck' will be written. The first, open source version of Erlang/QuickCheck will be referred as 'QuickCheck1', and the second, commercial version as 'QuickCheck2'.

Both types of QuickCheck provide one type of testing method, which is property-based and random-based at the same time.

Property-based testing means that properties can be written about the program and the tool can test the program against these properties. QuickCheck provides library functions and macros which can be used to describe the properties.

Random based testing means that the test inputs of the program are randomly generated; so test cases do not have to be written by the tester. In QuickCheck, the distribution of the test inputs can be specified, and testers can define own test input generators, too.

To sum up, property-based testing specifies how the expected behaviour of the program is expressed, and random-based testing specifies the method how the expected behaviour is checked.

QuickCheck evaluates the property being tested many times with randomly generated input. Afterwards the result of the property is examined. If QuickCheck finds that the property holds in each case, we call the result of the testing *negative*, which means that for all the test inputs which the QuickCheck generated, the property was true. In other words, the property passed. Otherwise the result of the testing is called *positive*, i.e. there was a test input, with which the property was false. In other words, the property failed. In the latter case this test input is given to the tester as a counterexample, explaining why the property failed.

QuickCheck1 also allows us to test concurrent and distributed programs by means such as trace analysis. A trace is a sequence of events, which happened during running the program. Linear Temporal Logic can be used to describe properties of traces. Actually the trace generation is a special kind of data generation, while LTL expressions are a special kind of property.

In QuickCheck2, the tester can specify an abstract state machine, which models the system, and QuickCheck can test whether the system conforms to the specified state machine.

An important feature of QuickCheck is that QuickCheck itself is written in Erlang, and all the properties and test input generators which will be written by testers are also in Erlang. Thus no additional programming language has to be learnt by testers and no additional programs (compilers, libraries, etc.) have to be installed; the only programming language needed is Erlang and the only program needed is the Erlang Runtime Environment. Besides these only the QuickCheck library is needed to be able to use QuickCheck.

3.1 A few simple examples

First example First consider a simple example: Erlang's standard library contains a `lists:reverse` function. `lists:reverse(List)` returns a list with the elements of `List` in reverse order. E.g. `lists:reverse([1,2,3])` returns `[3,2,1]`.

The property which will be tested is the following:

If the `lists:reverse` *function is applied to a list and is applied again to the result, then the final result will be the original list.*

The mathematical representation of the property is the following:

$$\forall X \in \texttt{list(int())} : \texttt{lists:reverse(lists:reverse(}X\texttt{))} = X$$

The `list(int())` means 'lists that contain integers'.

When QuickCheck is used to test a property, the property has to be written as an Erlang function. Since all the Erlang functions have to be in a module, a module will be defined, and this module will contain the function, which describes the property.

If QuickCheck1 is used, the Erlang code of the module is the following:

```
-module(e1).
-include("quickcheck.hrl").
-export([prop_list_reverse/0]).

prop_list_reverse() ->
    ?FORALL(X, list(int()),
        lists:reverse(lists:reverse(X)) == X).
```

The tester will not run this module directly, but will run QuickCheck and give it this function as an argument (see later), and QuickCheck will invoke this property function as many times as it is needed.

Let's walk through the code.

First the name of the module is given (this is the `e1` module):

```
-module(e1).
```

Then QuickCheck's header is included:

```
-include("quickcheck.hrl").
```

The function which describes the property is exported, because otherwise the QuickCheck module would not be able to use it (`prop_list_reverse/0` means that this function has zero arguments):

```
-export([prop_list_reverse/0]).
```

The property itself is written as a function:

```
prop_list_reverse() ->
    ?FORALL(X, list(int()),
        lists:reverse(lists:reverse(X)) == X).
```

This function is called *property function.* It looks very similar to the mathematical representation of the property.

But actually the `FORALL` is an Erlang macro and the `list(int())` is a test input generator function, both provided by QuickCheck. The `FORALL` macro takes three arguments: The first argument is the name of the variable (now `X`) into which the generated test cases will be put. The second argument (now `list(int())`) is called *data generator* or *test input generator.* It specifies a function, which returns random test cases for the variable in the first argument. The last argument is called *data property.* It specifies the expression to be tested whose return value will be either false or true depending on whether the test fails or passes successfully. It is called data property, because it is the property of the data. The property against which the system is tested ($\forall X \in$ `list(int())` : ...) is not the property of any data.

The module will be compiled in the usual way from the Erlang's shell:

```
> c(e1).
{ok,e1}
```

After compilation the tester will invoke QuickCheck. The property function will be the argument of the tester function:

```
> qc:quickcheck(e1:prop_list_reverse()).
..........................................................
..................................
OK, passed 100 tests
```

If the number of the tests is not specified, QuickCheck will do 100 tests by default. In each test QuickCheck will use the test input generator (`list(int())` function) to generate test cases randomly, and it will check if the result has the property described (the property described is `lists:reverse(lists:reverse(X)) == X`, where the generated test input will be actual value of `X`).

The output shows that all the 100 tests were passed, i.e. all the generated test inputs satisfied the property. (Each dot is a passed test.)

QuickCheck generates a tester function from the code above, and this tester function will be invoked 100 times. The data flow diagram of the tester function is shown in figure 1. First the test data is generated by the `list(int())` generator function. Then this data (which is called `X`) is given to the `list:reverse()` function, and the output of that is given to the `list:reverse()` function again. The output of that and the original `X` is given to the `==` function, the output of which is a boolean, which determines whether the property holds for the concrete data `X`. The box around the `list:reverse` and `==` function shows that from the QuickCheck's point of view, that box is a black box, i.e. QuickCheck does not know what is inside. The generator function is not in the box, because QuickCheck knows that `list(int())` is generating the data `X` which will be given to the black box, and it will print the current value of `X` to the screen if the property does not hold for it.

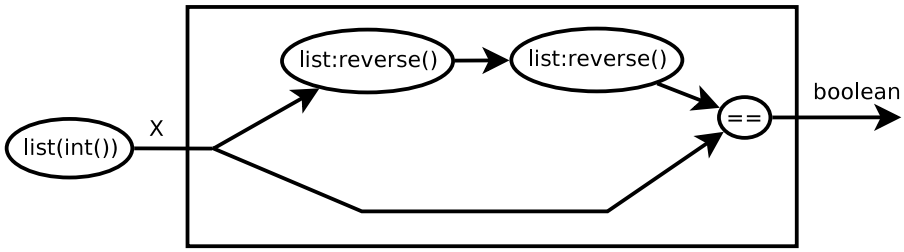

Figure 1: The data flow diagram of the property of first example.

Now let's examine, what happens if the test fails. If the `prop_list_reverse` function contained `lists:reverse(X) == X` instead of `lists:reverse(lists:reverse(X)) == X`, then the test output would be something like this:

```
> qc:quickcheck(e1:prop_list_reverse()).
...
Falsifiable, after 3 successful tests:
[-3,-2]
```

This output means that the first 3 generated test inputs passed the test (they were lists consisting of zero or one element). The third generated test input was the list above (`[-3,-2]`), and the property was false for that list, so the test failed and the QuickCheck stopped the test.

When a test fails, it is because one of the generated test inputs did not satisfy the property. Thus this test input will be a counterexample for the property, so a counterexample will always be given, when a test fails.

If QuickCheck2 is used, the only difference in the tester module is that the line which contains the `-include` directive is different. This is the equivalent code for QuickCheck2:

```
-module(e1).
-include_lib("eqc/include/eqc.hrl").
-export([prop_list_reverse/0]).

prop_list_reverse() ->
    ?FORALL(X, list(int()),
        lists:reverse(lists:reverse(X)) == X).
```

If the property holds, then the output of the testing is similar to the output of the testing with QuickCheck1:

```
> eqc:quickcheck(e1:prop_list_reverse()).
.....................................................
.....................................
OK, passed 100 tests
```

If the property does not hold, QuickCheck2 will also try to shrink the counterexample. The property above holds, but if it is modified not to, QuickCheck2 will try shrinking, as follows:

```
> eqc:quickcheck(e1:prop_list_reverse()).
..........Failed! After 11 tests.
[-1,2]
Shrinking..(2 times)
[0,1]
false
```

This output says that after 10 successful tests QuickCheck found a list for which the property did not hold. This is the list `[-1,2]`; this is the counterexample to the property. Then the QuickCheck tried to find a smaller and simpler counterexample; this process is called shrinking. Finally, after two attempts, the result of the shrinking is the list `[0,1]`, which is not smaller, but can be considered as simpler than the original one. Shrinking will be explained more detailed in section 3.2.3.

Second example: `IMPLIES` Now let's try to test the `lists:nth` function, which is also part of Erlang's standard library. `lists:nth(N,List)` returns the `N`th element of `List`. E.g. `lists:nth(2,[a,b,c,d])` returns `b`.

The test property is the following:

> *If a list is not empty, then its first element and its head element are equal.*

With mathematical notation:

$$\forall L \in \texttt{list(int())} : \texttt{length}(L) \neq 0 \rightarrow (\texttt{lists:nth}(1, L) = \texttt{hd}(L))$$

QuickCheck has an `IMPLIES` macro, which can be used to express this property:

```
prop() ->
    ?FORALL(L, list(int()),
        ?IMPLIES(
            not (length(L) == 0),
            (lists:nth(1,L) == hd(L))
        )
    ).
```

The data flow diagram of this property is in figure 2. The upper box represents the first argument of the `IMPLIES` expression, and the lower box represents the second one.

The output of QuickCheck1 is similar to the output showed before, but the output of QuickCheck2 is a little different: it prints `x` characters instead of dot characters when a test was successful *because* the first argument of the `IMPLIES` was false and thus the second argument was not examined.

A possible output:

```
> eqc:quickcheck(example2:prop()).
x.x..xx.....xxxx........xx..x...x.xx...x.........xx....x.xx
...................x........x.......x...................
.....
OK, passed 100 tests
```

It is a useful feature, because when there are too many `x` characters, the tester knows that something is wrong.

Unfortunately there are no `AND` and `OR` macros in either QuickCheck.

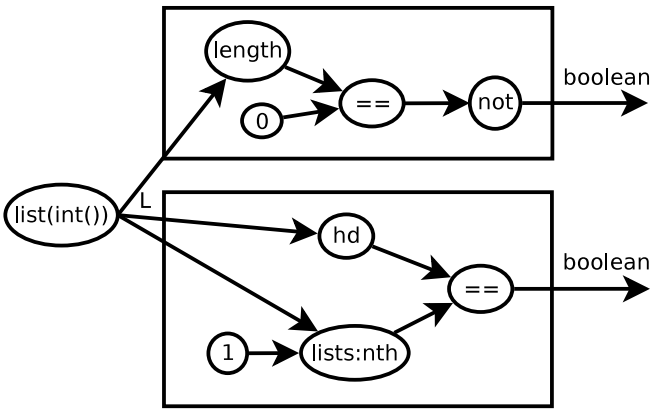

Figure 2: The data flow diagram of the property of second example.

Third example: nested `FORALL` macros This example tests the `lists:append` function against associativity. `lists:append(List1,List2)` returns a new list which is made from the elements of `List1` followed by the elements of `List2`. E.g. `lists:append([1,2,3],[4,5,6])` returns `[1,2,3,4,5,6]`.

The test property is the following:

The `lists:append` *function is associative, i.e.* `lists:append(L1,lists:append(L2,L3))` *is equal to* `lists:append(lists:append(L1,L2),L3)` *for all* `L1`*,* `L2`*,* `L3` *lists.*

With mathematical notation:

$$\forall L_1 \in \texttt{list(int())} : \forall L_2 \in \texttt{list(int())} : \forall L_3 \in \texttt{list(int())} :$$
$$\texttt{lists:append}(L_1, \texttt{lists:append}(L_2, L_3)) = \texttt{lists:append}(\texttt{lists:append}(L_1, L_2), L_3)$$

This property can be expressed in QuickCheck using 3 nested `FORALL` macros:

```
prop() ->
    ?FORALL(L1, list(int()),
        ?FORALL(L2, list(int()),
            ?FORALL(L3, list(int()),
                lists:append(L1,lists:append(L2,L3)) ==
                lists:append(lists:append(L1,L2),L3)
            )
        )
    ).
```

The data flow diagram of this property is in figure 3.

What does QuickCheck provide? The testing method of QuickCheck implies that it provides the followings.

If the test property holds, i.e. the data property is true for each input, the result of the testing will be negative. In other words, if the result of the testing is positive, the program or the property is definitely not correct.

QuickCheck does not provide that it is true in the other way as well. If a test property does not hold, i.e. there are inputs for which the data property is false, it is not guaranteed that QuickCheck will find any of these inputs, so the result of the testing may be negative. This is called false negative.

3.2 Features of QuickCheck

Here we explain a few features of QuickCheck that are useful in defining tests.

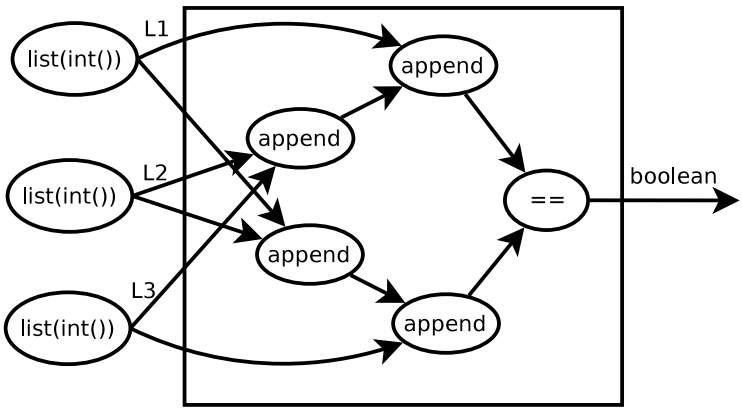

Figure 3: The data flow diagram of the property of third example.

3.2.1 `collect`

With the `collect` function the test cases can be collected into categories as defined by the tester. QuickCheck will print which categories occurred how many times. This way the tester will get the statistical distribution of the test cases, which will help him to identify what types of test cases were considered when testing the property. The tester can specify which aspect should be used to create the categories.

The following code collects the test cases by the length of the input lists:

```
prop_list_reverse() ->
    ?FORALL(X, list(int()),
        collect(length(X),
            lists:reverse(lists:reverse(X)) == X)).
```

Below the usual output of QuickCheck, the frequency of the partitions can be seen:

```
12> qc:quickcheck(e2:prop_list_reverse()).
..................................................
........................................
OK, passed 100 tests
14% 2
11% 0
11% 1
7% 7
7% 6
7% 3
6% 5
5% 12
5% 9
4% 15
4% 13
4% 10
4% 4
3% 14
3% 8
2% 18
1% 21
1% 16
1% 11
```

Every row is a test category. The first column shows in percentage that how often the input belonged to this category. The second column shows the aspect with which the categories were created, which is now the size of the generated list. E.g. the first row means that ‘in 16% of the test cases was the length of the input list equal to 2’.

The categories are exhaustive and exclusive, i.e. each test belongs exactly to one category. The first argument of the `collect` function is used to determine the category of the test. The test (a data property) is the second argument. A category is equivalent to an Erlang term. The return value of the data property is used to determine the Erlang term, and by this, the category of the test. In the example above, all the Erlang terms of the categories are numbers.

3.2.2 Test data generators

In the previous examples QuickCheck functions were used to generate data (the `int/0` and `list/1` functions). Some of the functions of QuickCheck can be manipulated by the tester to specify the distribution of the data, and the tester can define new test data generators as well. The data generators are similar in the two versions of the QuickCheck tool, but there are some differences.

Defining new data generator in QuickCheck1 Data generator functions are not generating the data themselves, but they return a function that generates data. The returned functions are called tester functions. They have one argument: the size of the test case. The tester will not use this argument now. The size argument is described and discussed in section 3.5.3.

In this paragraph the `lists:reverse` function will be tested again, but against another property than previously. The current test property is:

For all I indices of all L lists, the element at index I in the reversed list is the same as the element at index $length(L) - I + 1$ in the original list.

With mathematical notation:

$$\forall L \in \texttt{list(int())} : \forall I \in \texttt{index}(L) :$$
$$\texttt{lists:nth}(I,\texttt{lists:reverse}(L)) == \texttt{lists:nth(length}(L)\texttt{-}I\texttt{+1},L)$$

The expression `index(`L`)` represents the index set of the L list; but there is no such function as `index` in QuickCheck, so the tester has to write it. The function will be called `gen_index` instead of `index`, because then the code will be more readable. It returns the tester function, which returns a random index of the list:

```
gen_index(List) ->
    fun(_Size) ->
        random:uniform(length(List))
    end.
```

The property is similar to its mathematical form. The main difference is that since the `gen_index` function does not work with empty lists, these cases should be handled separately. The `IMPLIES` macro is used to solve this problem.

```
prop() ->
    ?FORALL(L,list(int()),
        ?IMPLIES(
            length(L)>0,
            ?FORALL(I,gen_index(L),
                lists:nth(I,lists:reverse(L)) ==
                lists:nth(length(L)-I+1,L)
            )
        )
    ).
```

This example also shows that a generator can use the data generated by the previous generators.

There is a performance issue, when more `FORALL` macros are used in one property. QuickCheck will generate only one index for one list. Suppose that the cost of creating the two data inputs are very different. They are different now, but the difference can be larger, e.g. when the first data is a dictionary or an event trace (see section 3.4), while the second one is an index. In this situation the reasonable testing method is that after generating the first data, many tests should be performed, i.e. many different second data should be generated to the same first data. QuickCheck does not have support for that, the tester has to implement this on his own.

Defining new data generator in QuickCheck2 QuickCheck2 works in a similar way, but not exactly like QuickCheck1 when defining data generators. The test property can be exactly the same, but in QuickCheck2 the data generator function *is* the tester function, instead of returning it. (It does not mean that QuickCheck2 does not have the 'size' feature, but the `Size` parameter is not argument of the tester function there.)

So a correct data generator in QuickCheck2:

```
gen_index(List) ->
    random:uniform(length(List)).
```

`frequency` **function** This function, which is provided by both QuickChecks, gets a list of `{Weight,Output}` tuples; so the list has the form of `[{Weight_1,Output_1},{Weight_2, Output_2},...]`. The return value of the function is nondeterministically one of the `Output` values. The distribution is determined by the `Weight` values: the probability that `Output_i` will be returned is:

$$\frac{\texttt{Weight\_i}}{\texttt{Weight\_1} + \ldots + \texttt{Weight\_n}}$$

An example:

```
gen_bool() ->
    frequency([
            {3,true},
            {1,false}
        ]).
```

The probability of generating true is $\frac{3}{4}$, the probability of false is $\frac{1}{4}$. The function is used in the chat server example, on page 44.

3.2.3 Shrinking

If QuickCheck1 finds a counterexample, it stops and prints it. In contrast, QuickCheck2 tries to shrink the counterexample. An example of shrinking can be seen on page 17.

```
> eqc:quickcheck(e1:prop_list_reverse()).
..........Failed! After 11 tests.
[-1,2]
Shrinking..(2 times)
[0,1]
false
```

A way of shrinking data can be specified by the tester.

3.3 Testing impure functions with QuickCheck

The functions which were tested in the previous section were pure functions. Testing impure functions is more difficult than testing pure ones.

A function is pure if the result of the function depends only on its arguments, and it is side-effect free, so its only output is its return value.

In contrast, an impure function can use and modify the state of the system: it can print messages to the user, read and write files, communicate with other processes, read and set global variables (process dictionary and registry in Erlang), etc. If impure functions need to be tested, the test property has to consider all these activities beside the arguments and the return values. The question: how will the property know what the functions did, what the original state of the system was and what side-effects were caused.

When talking about testing functions, it does not mean that we are testing one function at a time. A single property can involve more than one function. E.g. if `S` is a stack and `E` is an element, and the property `top(push(S,E))==E` is tested, then it is not testing of one function – it is testing the properties of the data type. Similarly, a property can express a property of a module, a process, a program, etc. When testing impure functions, we usually do not test the functions individually, but write properties about the behaviour of the whole system (e.g. a

chat server or a dining philosophers simulation). Such a system may consist of many functions, and may contain records, or in other programming languages classes, objects, templates, etc.

When testing impure functions, both QuickChecks have the approach that we are testing systems, not individual functions, and not types or records or modules. A system is started, the environment communicates with it, and the aim of the tester is to examine whether the system behaves correctly. This system will be called SUT (system under test).

Even though the two QuickChecks use the same method for testing pure functions, they choose very different approaches for testing impure functions. The method of QuickCheck1 is to observe the events happening during the running of the SUT, and afterwards the property (created by the tester) will examine this event trace. Using QuickCheck2, the tester creates an abstract state machine, which models the states of the SUT and the properties of the states. In this case QuickCheck is not a passive observer, but it controls the SUT: it decides which state transition should happen, and after each transition it examines the test property.

3.4 Testing impure functions with QuickCheck1

Let's suppose the following program needs to be tested:

```
-module(nat_gen).
-export([start/0]).

start() ->
    Nat_receiver = spawn_link(fun nat_receiver/0),
    nat_sender(Nat_receiver,0).

nat_receiver() ->
    receive
        Nat ->
            io:format("Received this number: ~w~n",[Nat]),
            nat_receiver()
    end.

nat_sender(Nat_receiver,N) ->
    Nat_receiver ! N,
    nat_sender(Nat_receiver,N+1).
```

This program is a natural number generator, in which there are two processes, and one process sends natural numbers to the other one in increasing order, starting from 0.

The program can be started with the `start` function. It starts two processes: one process runs the `nat_receiver` function, the other one runs the `nat_sender` process. The `nat_sender` sends natural numbers to the `nat_receiver`, which will print them.

The property which needs to be tested is that the first printed natural number is 0 and all the other ones are the successors of the previous one.

The tools of QuickCheck1 for testing this program will be introduced in the next paragraph. After that a possible solution will be given.

3.4.1 Traces and events

The main idea of testing systems with impure functions in QuickCheck1 is the idea of examining traces. A trace is a sequence of events, which happened during the execution of the SUT. Before testing, the tester not only has to write the test property, but he has to place events in the code. QuickCheck will execute the SUT many times, just as in the case of testing pure functions. But now during the execution of the SUT QuickCheck will record the trace (the list of events), and the test property will examine this trace instead of the return value.

The events which will appear in the traces should be placed in the original code. The events will be represented by `event` function calls. These function calls will be called event expressions. They can be placed anywhere, where Erlang expressions can be placed (because an event expression is an Erlang expression). The `event` function has one argument. This argument can be any Erlang term; it will be put into the trace. E.g. the following function contains two events:

```
computation(X) ->
    event(computation_started),
    % ... doing computation
    event(computation_finished).
```

When this function starts to run, the `computation_started` event will be put into the trace; similarly, when it finishes, the `computation_finished` event will be added.

The generated trace is a list which consists of event-tuples. There are three kinds of event-tuple, which have different meanings:

- {`event,Pid,Event`}: the event `Event` occurred in process `Pid`, i.e. an `event(Event)` expression was evaluated.
- {`exit,Pid,Reason`}: the process `Pid` exited.

- {wait,N}: N milliseconds passed with no events recorded.

Traces can be generated by the TRACE macro call, which is provided by QuickCheck. It takes two arguments: The first argument is a Timeout; it will be discussed in section 3.5.3. The second argument is the function to be executed.

E.g. when testing the following property, QuickCheck generates traces from the execution of the start function (with a timeout of 3 milliseconds), and then examines the trace with the p function:

```
prop() ->
    ?FORALL(T,?TRACE(3,start()),p(T)).
```

nat_gen example Let's test the nat_gen program, which was introduced above. First the tester has to include QuickCheck's header and put event expressions into the code:

```
-module(nat_gen).
-export([start/0,prop/0]).
-include("quickcheck.hrl").

start() ->
    Nat_receiver = spawn_link(fun nat_receiver/0),
    event(start),
    nat_sender(Nat_receiver,0).

nat_receiver() ->
    event({receiver,waiting}),
    receive
        Nat ->
            event({receiver,received,Nat}),
            io:format("Received this number: ~w~n",[Nat]),
            nat_receiver()
    end.

nat_sender(Nat_receiver,N) ->
    Nat_receiver ! N,
    event({sender,sent,N}),
    nat_sender(Nat_receiver,N+1).
```

There is no general rule about how to put event expressions. For our purpose only the event({receiver,received,Nat}) call is important. (It represents that the receiver process received the Nat message.)

Then the tester has to write the property. The property is that if the start function is executed, the event trace should be correct:

```
prop() ->
    ?FORALL(T,?TRACE(3,start()),trace_correct(T)).
```

Now only one task remained, to write the trace_correct function, which will determine whether a trace is correct. The trace is correct if all the received messages are natural numbers, the first one is 0 and all the other ones are the successors of the previous one. One way to write the trace_correct function is the following:

```
trace_correct(T) ->
    trace_correct(0,T).

trace_correct(_,[]) ->
    true;
trace_correct(N,[{event,_Pid,{receiver,received,N}}|T]) ->
    trace_correct(N+1,T);
trace_correct(_N,[{event,_Pid,{receiver,received,_M}}|_T]) ->
    false;
```

```
trace_correct(N,[_H|T]) ->
    trace_correct(N,T).
```

`trace_correct(N,T)` means that 'all the received messages are natural numbers, the first one is `N` and all the other ones are the successors of the previous one'. `trace_correct(T)` is equivalent to `trace_correct(0,T)`.

The code of `trace_correct/2` means the following: If the trace contains no element, then it is correct. If the next number which should be received is `N` and the first event is `{receiver,received,N}`, then the remainder of the trace should be checked as well. If the first event is `{receiver,received,N}`, but `N` is not the next number, then the trace is not correct. If the first element of the list is something else, then the other elements of the list should be checked.

The program can be tested in the usual way:

```
> qc:quickcheck(nat_gen:prop()).
```

Designing and implementing a testing strategy is non-trivial, and usually there are two potential pitfalls. If we find an 'error', it could be that the SUT is erroneous, or it could be that the property was incorrectly formulated. Using the testing method which was described above, there is an additional pitfall: the event calls can be incorrect, as well. Event calls can be incorrect in two ways: they can be incorrect themselves, or they can be in wrong places in the SUT. An example for the former error is if the `event({receiver,received,Nat})` line was mistyped and the code contained `event({receiver,receiver,Nat})`, instead of the original one. An example for the latter error is if the original line would be inserted two lines below (below line `nat_receiver()`), instead of its current place. In both cases, the property would fail. However, unfortunate errors may happen, where both the SUT and the events are incorrect, but they pass the correct properties.

3.4.2 Writing properties in LTL

In addition, Linear Temporal Logic (see section 2.2) can be used to analyse the traces. LTL expressions can be built using the functions and macros provided by QuickCheck, and QuickCheck also can evaluate the LTL expressions, which can be used in test properties.

The `MATCHES` macro of QuickCheck can be used to express that the first element of the trace matches a pattern. This can be expressed as `?MATCHES(Pattern)`; this is the simplest LTL expression. A `MATCHES` macro call in the QuickCheck is similar to a propositional variable in LTL. E.g. `?MATCHES({event,_Pid,start})` means that the first event happened matches the pattern `{event,_Pid,start}`, so the first event was raised by an `event(start)` function call.

New LTL expressions can be built from existing ones with the following functions:

- `tnot(L)`: the LTL expression `L` is false.
- `tand(A,B)`: the LTL expressions `A` and `B` are both true.
- `tor(A,B)`: `A` or `B` is true.
- `timplies(A,B)`: `A` implies `B`.
- `next(L)`: `L` is true in the next state.
- `always(L)`: `L` is always true.
- `eventually(L)`: `L` is eventually true.
- `until(A,B)`: until `A` becomes true, `B` has to hold.
- `empty()`: the trace is empty.

For the exact semantics of these operators and for examples see section 2.2.

Examples for LTL expressions in QuickCheck:

- `?MATCHES({event,_,a})`: the first event is event `a`.
- `tnot(?MATCHES({event,_,a}))`: the first event is not event `a`.

- `tor(?MATCHES({event,_,a}),?MATCHES({event,_,b}))`: the first event is event `a` or event `b`.
- `next(?MATCHES({event,_,a}))`: the second event is event `a`.
- `next(tor(?MATCHES({event,_,a}),?MATCHES({event,_,b})))`: the second event is event `a` or event `b`.
- `always(tnot(?MATCHES({event,_,a})))`: all the event does not match `{event,_,a}`, i.e. there is no event `a` in the trace.

The `satisfies` function can be used to integrate the LTL expressions into test properties. The function takes two arguments: the trace and the LTL expression. The result is true if the trace satisfies the LTL expression, false otherwise. An example of a test property which uses LTL:

```
prop() ->
     ?FORALL(T,
         ?TRACE(3,start()),
         satisfies(T,
             always(tnot(?MATCHES({event,_,a})))
         )
     ).
```

It says that the traces should not contain any event `a`.

LTL expressions are on another level than property functions. Property functions can be built using `FORALL` and `IMPLIES` macros, data generators (such as the `list(int())` function and the `TRACE` macro), and data properties (Erlang expressions with a boolean return value). QuickCheck LTL expressions represent mathematical LTL expressions, which can be built using the `MATCHES` macro and the LTL builder functions described above (`tnot`, `tand`, etc.). These two levels can be connected, but cannot be mixed: e.g. a `FORALL` macro cannot be used inside a `tand` function, and the `tand` function cannot be used directly inside a `FORALL` macro. The `IMPLIES` macro and the `timplies` function are very different, only their names are similar.

These two levels can be connected using the `satisfies` function. This function converts a variable which represents a trace, and a QuickCheck LTL expression into a data property. The data property will be the property of the trace (which is just like any other data input in the philosophy of QuickCheck), and it says that 'the LTL expression is true over the trace'. In the example above, the data property consists of the following three lines:

```
satisfies(T,
    always(tnot(?MATCHES({event,_,a})))
)
```

which represents that 'there is no event `a` in trace `T`'.

Example: `missile_simulation` This is an example of using QuickCheck for testing a program. The property tested will be expressed in LTL.

The specification The task is to write a simulation of a missile controller system. In the simulation there is a missile, a missile controller and a missile handler.

The missile controller has 3 buttons. The buttons are: `launch_button`, `turn_off_button`, and `turn_on_button`. The `launch_button` launches the missile if the missile controller is turned on, and the controller can be turned off and on with the other two buttons. It is turned on when the simulation starts.

The missile handler is a person who is sitting in front of the missile controller and pushes the buttons. We do not know his logic, so in our simulation he will push the buttons randomly.

The following property should be tested with QuickCheck:

The missile can be launched only if the `launch_button` *was pressed previously.*

Expressing the property in QuickCheck The property in other words:

The missile cannot be launched until the `launch_button` *was not pressed.*

It can be expressed in LTL:

$$\texttt{launch\_button\_pressed}\ \mathcal{U}\ \neg\texttt{missile\_launched}$$

With QuickCheck notation:

```
until(
    ?MATCHES({event,_,launch_button_pressed}),
    tnot(?MATCHES({event,_,missile_launched}))
)
```

The program The simulation program with the QuickCheck events is a little complicated, because the main process will wait for the other ones to finish. (Otherwise QuickCheck will not work as it is expected; see section 3.5.5 for the explanation.)

First the module name, including the QuickCheck's header and exporting the functions:

```
-module(missile_simulation).
-include("quickcheck.hrl").
-export([prop_simulation/0,simulation/0]).
```

The missile can get two kinds of messages. If it gets `launch`, then it will be launched. It can get a `{Pid,end_of_simulation}` message either before or after the `launch` message. When the missile got this message, it will send back its own process id (`self()`) to the process `Pid`.

```
missile() ->
    receive
        launch ->
            event(missile_launched),
            io:format("BANG!",[]),
            receive
                {Pid,end_of_simulation} ->
                    Pid ! self()
            end;
        {Pid,end_of_simulation} ->
            Pid ! self()
    end.
```

The `missile_controller` can receive three kinds of messages. If it receives `launch_button`, then it will `launch` the missile (e.g. it will send a `launch` message to the `Missile` process). If it receives `turn_off_button`, then it will wait for a `turn_on_button` message. In any situation it can get the `{Pid,end_of_simulation}` message, and then its behaviour is similar to the behaviour of the `missile` function.

```
missile_controller(Missile) ->
    receive
        launch_button ->
            event(launch_button_pressed),
            Missile ! launch,
            missile_controller(Missile);
        turn_off_button ->
            %%%
            event(turned_off),
            receive
                turn_on_button ->
                    event(turned_on),
                    missile_controller(Missile);
                {Pid,end_of_simulation} ->
```

```
                    Pid ! self()
            end;
        {Pid,end_of_simulation} ->
            Pid ! self()
    end.
```

The first two lines of the `missile_handler/1` is the initialisation of the Erlang's random number generator; then it starts a `missile_handler` with argument 10, which means that after 10 commands it should stop. The `missile_handler/2` (when the number of commands is not zero) generates a random number, and based on this, it sends a (random) message to the `Missile_controller`.

```
missile_handler(Missile_controller) ->
    {A1,A2,A3} = now(),
    random:seed(A1,A2,A3),
    missile_handler(Missile_controller,10).

missile_handler(_,0) ->
    end_of_simulation;
missile_handler(Missile_controller,I) ->
    case random:uniform(3) of
        1 -> Missile_controller ! launch_button;
        2 -> Missile_controller ! turn_off_button;
        3 -> Missile_controller ! turn_on_button
    end,
    missile_handler(Missile_controller,I-1).
```

The `simulation` starts the other two processes and then the `missile_handler` function. When finished, it waits for the others to finish and then terminates.

```
simulation() ->
    event(simulation_start),
    Missile =
        spawn_link(fun missile/0),
    Missile_controller =
        spawn_link(fun() -> missile_controller(Missile) end),
    missile_handler(Missile_controller),
    Missile_controller ! {self(),end_of_simulation},
    receive Missile_controller -> ok end,
    Missile ! {self(),end_of_simulation},
    receive Missile -> ok end.
```

The function which checks the property is made from the LTL expression above:

```
prop_simulation() ->
    ?FORALL(T,
        ?TRACE(3,simulation()),
        satisfies(T,
            until(
                ?MATCHES({event,_,launch_button_pressed}),
                tnot(?MATCHES({event,_,missile_launched}))
            )
        )
    ).
```

If e.g. a `Missile ! launch` expression is written into the line marked with `%%%`, the test will fail (because there will be some traces, where the missile will be launched before the `launch_button` was pressed).

User defined LTL expressions New functions can be defined to build LTL expressions. They can be based on the ones of QuickCheck or they can manipulate directly with the trace.

A few examples (these are all based on QuickCheck's operators):

- All A-s have to happen before any B happens.

```
order(A,B) -> until(tnot(eventually(A)),tnot(B)).
```

- A has to happen a minimum of N times.

```
tmin(0,_A) -> fun(_) -> true end;
tmin(1,A) -> eventually(A);
tmin(N,A) -> eventually(tand(A,next(tmin(N-1,A)))).
```

- A has to happen a maximum of N times.

```
tmax(0,A) -> tnot(eventually(A));
tmax(N,A) -> always(timplies(A,next(tmax(N-1,A)))).
never(A)  -> tmax(0,A).
```

- A has to happen exactly N times.

```
tnumber(N,A) ->
    tand(tmin(N,A),tmax(N,A)).
```

- Given a list of properties, all of these properties have to hold.

```
tand([A]) ->
    A;
tand([A,B]) ->
    tand(A,B);              % QuickCheck's tand
tand([A,B|Others]) ->
    A_and_B = tand(A,B),    % QuickCheck's tand
    tand([A_and_B|Others]).
```

Summary To summarize, what QuickCheck1 provides: it records the events which happened during running the SUT and aids in the analysis of the collection of events.

The tester has to:

- place the events in the code,
- write the code which analyses the event trace, or
- write LTL expressions about the event trace.

QuickCheck will:

- record the event trace,
- examine whether the LTL expressions hold for the event trace.

3.5 Testing impure functions with QuickCheck1: problems and tricks

In this section various problems will be examined. Some of the problems can be solved by a trick, and some of them cannot be. This section tries to give a better understanding of how QuickCheck works and a few pieces of advice about how to write programs which can be easily tested with QuickCheck.

3.5.1 Events everywhere

To test a program with QuickCheck using trace generation, a lot of `event` function calls have to be placed into the code (`qc:event` means that the `event` function in the `qc` module, which is the module of QuickCheck):

```
% *.erl
...
f() ->
    ...
    qc:event(a),  % event 'a' happened
    ...
...
```

After testing the program, it should be able to run and be used without QuickCheck, since that is a testing tool and not a runtime environment. If the program is not changed, then when an `event` function is called, it will be an `error`, because the `event` function cannot find the QuickCheck running.

An own `event` function One way to solve this problem is to remove all the `event` function calls from the code, but that's not a very good solution, because later another test may be necessary, and, in this case, all the `event` calls should be put back.

A better way is to use an own `event` function, which will invoke the QuickCheck's `event` function. Here the own `event` function is called `ev`, and this is inserted into the program being tested:

```
% *.erl

ev(Event) ->
    qc:event(Event).  % QuickCheck's event function

...
f() ->
    ...
    ev(a),  % event 'a' happened
    ...
...
```

Instead of writing `qc:event(a)` into the code being tested, `ev(a)` will be written. In other words this function is a wrapper for the QuickCheck's `event` function.

When the test is finished, the `ev` function should be modified, because then nothing should happen when the `ev` function is called:

```
% *.erl

ev(Event) ->
    ok.     % do nothing

...
f() ->
    ...
    ev(a),  % event 'a' happened
    ...
...
```

After that one modification the program will be able to run without the QuickCheck module, since it will not invoke any function from that.

Using this solution the program has an overhead of calling the `ev` function, instead of calling QuickCheck's `event` function directly (when the program is tested), or instead of calling nothing (when the program is used). It is not a problem, when the program is tested, because

the overhead of QuickCheck itself is much higher; but it may be a problem, when the program is used.

The `ev` function can be used to show additional information about testing. E.g. if the tester defines an event called `end_of_test`, and he wishes to observe its occurrence, he can use the following `ev` function, which will print a comma every time the `end_of_test` event happened:

```
ev(Event) ->
    case Event of
        end_of_test -> io:format(",",[]);
        _ -> ok
    end,
    event(Event).
```

This trick is useful, because many times QuickCheck kills the program under test before it could finish, and this function allows the tester to see when the program managed to finish (see section 3.5.4).

Similarly, it can be used to print every event:

```
ev(Event) ->
    io:format("Event happened: Pid: ~w, event: ~w.~n",[self(),Event]),
    event(Event).
```

It prints the process id of the process which raised the event (`self()`), and the event itself.

Putting the own `event` function into a separate module All these different behaviours can be put into one function, which can be parameterized by what to do, so it does not have to be modified, when a different behaviour is expected.

If there is more than one module in the program, and the tester wants to have only one `ev` function, then it should be placed into a separate module and should be exported (it means that other modules can use it).

```
% my_qc.erl

-module(my_qc).

ev(Event,Test) ->
    case Test of
        on_show_no_event ->
            qc:event(Event);
        on_show_only_end_of_test ->
            case Event of
                end_of_test -> io:format(",",[]);
                _ -> ok
            end,
            qc:event(Event);
        on_show_all_events ->
            io:format("Pid: ~w, event: ~w.~n",[self(),Event]),
            qc:event(Event);
        off ->
            ok
    end.

% *.erl

...
-define(TEST,on_show_no_event).

...
f() ->
    ...
```

```
    my_qc:ev(a,?TEST),  % event 'a' happened
    ...
    my_qc:ev(b,?TEST),  % event 'b' happened
    ...
...
```

Using this solution, the program under test will call this function (`my_qc:ev`), instead of calling the QuickCheck's `event` function.

If `Test=off`, it means that the program is not under a test, so nothing should happen when the `ev` function is called. In the other cases `Test` begins with `on`, which means that a test is running, so the QuickCheck's `event` should be called. If `Test` is `on_show_only_end_of_test`, then the `end_of_test` event will be shown by a comma; if it is `on_show_all_events`, then all information about all the events will be printed onto the screen; if it is `on_show_no_event`, then nothing will happen except for calling the QuickCheck's `event` function.

If the tester in the example above wants to print the events, he just have to modify the definition of `TEST`:

```
-define(TEST,on_show_all_events).
```

When the test is finished and the program is used, this will be the definition of the `TEST` macro:

```
-define(TEST,off).
```

A second own `event` function In this paragraph a pattern will be introduced to help the developers to use QuickCheck when there are more Erlang projects under development and testing in the system, and every project may contain several modules.

The problem with the previous solution is that if several modules are in a project, and the testing mode should be changed (e.g. the tester wants to parameterize the `my_qc:ev` function with `off` instead of `on_show_no_event`), then the `TEST` macros should be redefined in all modules. A good solution to this problem is to place this `TEST` macro in a header file, which will be included into every `erl` file in the project.

But there is an alternative way, too, which will be discussed now. In contrast with the solution with header file, it is a bit more elegant, because nothing has to be included, and the tester will not write `ev(a,?TEST)`, but will write `ev(a)` instead.

The main idea is to use two kinds of modules:

1. Using the `my_qc` module as a library module. It means that there will be one `my_qc` module in the whole system, which may have a lot of features, depending on the tester's needs.

2. Using a different `my_ev` module for each project. The other modules of the projects will invoke the `my_qc:ev` function via this module.

Sample Erlang code of the pattern:

```
% my_qc.erl
%% the same as in the previous paragraph

% my_ev.erl
ev(E) ->
    my_qc:ev(E,on_show_only_end_of_test).

% *.erl
...
f () ->
    ...
    my_ev:ev(a),  % event 'a' happened
    ...
...
```

At first sight this pattern given here seems to be complicated, but the advantage of it is that the code which has the real functionality and the code which decides how to behave in this moment are in different modules. The first one is in module `my_qc`, while the second one is in module `my_ev` .

Thus the same `my_qc` file can be used by different projects (so e.g. it can be used as a library function), but for one project, to change the behaviour of all the `event` calls in all modules, only one line of the project has to be changed (the second line of the `my_ev` module of the project).

If the decision about how the `event` calls should behave was made in the `my_qc` module, then every project would have to use a different `my_qc`. (Because maybe one project is at the beginning of the testing phase and the developers want to see every event on the screen, while in another project the developers want to see only the `end_of_test` events, and the third project is used, so nothing has to happen if an `event` is called.) If the decision was made in the modules, then each module would have to be changed when the behaviour of the `event` calls should be changed.

The summary of the pattern: We have a system where more projects are developed and each project has more modules. The modules in our system are the following (as in the sample Erlang code above):

- Module `my_qc`: contains an `ev/2` function (`ev` function with 2 arguments), which can call the QuickCheck's `event` function in various ways. This module can be a library function, so there is only one `my_qc` module in the whole system.
- Module `my_qc`: contains an `ev/1` function, which calls the `my_qc:ev` function, and nothing else. The way to call it is hard coded in this function. There is one `my_qc` module in each project.
- All the other modules: they invoke the `my_qc:ev` functions, if an event happens.

3.5.2 Using Erlang variables in LTL formulas

The problem If a program uses the `MATCHES` macro of QuickCheck, Erlang variables inside the macro arguments cannot be used. E.g. consider this program:

```
1    -module(e1).
2    -include("quickcheck.hrl").
3    -export([main/0,prop_1/0]).
4
5    main() ->
6        event(b).
7
8    prop_1() ->
9        A = a,
10       ?FORALL(T,
11           ?TRACE(3,main()),
12           satisfies(T,
13               ?MATCHES({_,_,A})
14           )
15       ).
```

Here the property described in the `prop_1` function is that for all traces (which were generated by running the `main` function), the tuple `{_,_,A}` matches the first event. The question is that when the tuple `{_,_,A}` matches an event.

The expected behaviour would be that `A` is bound by `a`, and `?MATCHES({_,_,A})` would mean `?MATCHES({_,_,a})`. Therefore the test should fail, because the property says that the first event should be `a`. However, this cannot happen in the above program, since the only event is event `b` (in line 6).

But instead warnings are printed when compiling the program:

```
./e1.erl:9: Warning: variable 'A' is unused
./e1.erl:13: Warning: variable 'A' is unused
./e1.erl:13: Warning: variable 'A' shadowed in 'fun'
```

And the output of QuickCheck will be:

```
10> qc:quickcheck(e1:prop_1()).
.....................................................
.......................................
OK, passed 100 tests
```

The cause of the problem The cause of the problem is the definition of the `MATCHES` macro. The `?MATCHES({_,_,A})` will become this:

```
1    fun ([]) ->
2            false;
3        ([Event|_]) ->
4            case catch (fun({_,_,A}) -> true end)(Event) of
5                true -> true;
6                _ -> false
7            end
8    end.
```

This function gets a trace and returns true if the pattern matches the trace and false otherwise. The problem is that if the event is e.g. `{_,_,b}`, the function will return true, despite that we would expect it to return false. It is because the `A` in the 4th line has nothing to do with the `A` variable outside (line 9 in the original program), which was bound by `a`. This `A` is a formal parameter, and thus it will match `b`.

A solution The problem can be solved by rewriting the `MATCHES` macro. This is the new definition of the macro:

```
-define(MY_MATCHES(X),
    fun (T) ->
        case T of
            [X|_] -> true;
            _ -> false
        end
    end
).
```

When this definition of the macro is used (i.e. `?MY_MATCHES({_,_,A})` is written instead of `?MATCHES({_,_,A})`), the result will be this code:

```
fun(T) ->
    case T of
        [{_,_,A}|_] -> true;
        _ -> false
    end
end
```

This will work fine, `A` will be considered to be bound by `a`, so if the trace begins with `{_,_,b}`, the function will return false.

Note: even though this macro will always work, there is still a little problem: if the variable `T` is used in the block which contains the macro call, a warning will be printed that the `T` in that block is shadowed. But it is only a warning, the program will run correctly.

3.5.3 Test size

QuickCheck's generator functions have a `Size` argument. The entities generated by the generators will not have a greater size than this `Size`. If the generated object is a list, it will be the maximum length of the list. If a natural number is generated, it will be between 0 and `Size`. If a trace is generated, it will be the maximum number of events.

When QuickCheck makes tests, `Size` starts at 2 and will be increased by one after every fifth test. So the sizes of the tests are: 2,2,2,2,2,3,3,3,3,3,4,4,4,4,4, ...

`Size` problem The `Size`-feature can give rise to problems. Consider an example, when processes are tested with random inputs. The property will be something like this:

```
?FORALL(Input,gen_input(),
    ?FORALL(T,?TRACES(3,my_program(Input)),
        ...
    )
)
```

This property means that for all `Input` which was generated by the `gen_input` function, and for all `T` trace which was generated by the `?TRACE` trace generator (with `my_program(Input)` as an argument), the expression standing instead of the '`...`' has to be true (the expression can contain the variables `Input` and `T`).

When QuickCheck does a test, first it will call the `gen_input` function; the return value will be placed in the variable `Input`. Then the `?TRACE` macro will call the `my_program` function with `Input` as an argument. After that QuickCheck will evaluate the expression.

The problem is that both `gen_input` and `?TRACE` will use the `Size` variable, which will always be the same for both. But e.g. if `Input` is a list, and the processing of each element of `Input` raises a few events, then the trace will be always a few times bigger than the size of the list, and it will be a rare case when the program will be able to finish processing the whole list. E.g. if processing of each list element raises n events and there are no other events, then the rate of finished test cases will be around $\frac{1}{n}$.

Two possible (but not very nice) solutions are to use hard coded sizes or change the meaning of the `Size` argument (e.g. a function uses `Size/2` instead of `Size`).

Timeout The trace generator uses another parameter beside `Size`. When generating a trace, a `Timeout` argument has to be specified. If `Timeout` milliseconds pass with no event recorded, QuickCheck will stop the current test (i.e. it will kill all the processes) and start the next one.

The testers should be aware of this feature, too, because if there is a long computation or a timer in the program, then it will always be killed at that point by QuickCheck.

The advantage of `Size` and `Timeout` This may be the reason why the developers of QuickCheck decided to introduce these features.

The `Size` and `Timeout` together can guarantee, that every test will be finished in a finite time, because:

1. the maximum number of events is `Size` and
2. the maximum number of milliseconds between two events is `Timeout`,

thus the test will be certainly finished in `Size` $\times$ `Timeout` milliseconds.

3.5.4 Properties and end of test

When QuickCheck generates traces, processes will be killed after `Size` events. (See section 3.5.3.) This implies that any property, which begins with the sometimes operator ($\Diamond A$) may fail during testing, because the system may be killed at any time by QuickCheck (so A may not happen).

Testing programs which terminate: `end_of_test` event If the program terminates, then this problem can be solved by using an `end_of_test` event. (This is not a QuickCheck feature, `end_of_test` is just an ordinary atom, it could have any other name.) When all the processes terminated, an `end_of_test` event can be raised, and in the property the LTL formula $\Diamond$`end_of_test` $\rightarrow B$ can be written instead of the LTL formula B itself. (So $\Diamond A$ becomes $\Diamond$`end_of_test` $\rightarrow \Diamond A$.) This new LTL formula will always be true if the event `end_of_test` did not happened; if it happened, then it will have the same value as B. So the test cases which failed using the original property (B) because the processes were killed before the test ended, will pass the test now.

Using this trick the testers should be aware that if a program never finishes then all the tests will be passed. The method described in section 3.5.1 can be used to observe whether `end_of_test` happened.

Testing programs which do not terminate When testing a system which will never finish, something else should be used instead of `end_of_test`. In the worst case a clock can be used, as in the next example.

An example: `dining_philosophers` A dining philosophers program will be tested against deadlock.

The `clock` process first raises an event, just to be a QuickCheck-registered process; then it tells the main process about it. (All the other processes will do the same.) Afterwards it raises a `tick` event in every 100 milliseconds until it is killed.

```
-module(e1).
-export([main/0,prop/0]).
-include("quickcheck.hrl").

clock_start(Main) ->
    event({clock,started}),
    Main ! registered,
    clock_loop().

clock_loop() ->
    event(tick),
    timer:sleep(100),
    clock_loop().
```

A philosopher process becomes hungry, then tries to grab the left and the right fork. Then it eats, releases the forks, thinks and starts it again.

```
philosopher_start(Main,Index,Left_fork,Right_fork) ->
    event({philosopher_started,Index}),
    Main ! registered,
    philosopher_loop(Index,Left_fork,Right_fork).

try_to_grab_fork(Fork,Index) ->
    Fork ! {self(),try_to_grab},
    receive
        {Fork,engaged} ->
            event({try_again,Index}),
            try_to_grab_fork(Fork,Index);
        {Fork,ok} ->
            ok
    end.

release_fork(Fork) ->
    Fork ! {self(),release}.

philosopher_loop(Index,Left_fork,Right_fork) ->
    event({hungry,Index}),
    event({try_to_grab_left_fork,Index}),
    try_to_grab_fork(Left_fork,Index),
    event({left_fork_grabbed,Index}),
    event({try_to_grab_right_fork,Index}),
    try_to_grab_fork(Right_fork,Index),
    event({right_fork_grabbed,Index}),
    event({started_eating,Index}),
    release_fork(Left_fork),
    release_fork(Right_fork),
    event({thinking_started,Index}),
    philosopher_loop(Index,Left_fork,Right_fork).
```

A fork acts like a server. It has two states: `free` and `engaged`. First it is `free`.

If a philosopher tries to grab it, it can act based on its state: If its `free`, it will become `engaged` and will answer `ok` to the philosopher. It its `engaged`, it will answer `engaged`.

If a philosopher releases it, it will become `free`. (In a real program an assertion would be useful here that `State` is `engaged`; just to make sure, that the program works correctly.)

```
fork_start(Main,Index) ->
    event({fork_started,Index}),
    Main ! registered,
    fork_loop(free,Index).

fork_loop(State,Index) ->
    receive
        {Philosopher, try_to_grab} ->
            case State of
                free ->
                    Philosopher ! {self(),ok},
                    fork_loop(engaged,Index);
                engaged ->
                    Philosopher ! {self(),engaged},
                    fork_loop(engaged,Index)
            end;
        {_Philosopher,release} ->
            fork_loop(free,Index)
    end.
```

The `main` process spawns all fork and philosopher processes and gives them the correct indices and other arguments. After that it waits for a `registered` message from each child processes. (See advice 2 in section 3.5.5.)

```
main() ->
    Main = self(),
    spawn_link(fun() -> clock_start(Main) end),
    Forks = [
        spawn_link(fun() -> fork_start(Main,1) end),
        spawn_link(fun() -> fork_start(Main,2) end),
        spawn_link(fun() -> fork_start(Main,3) end),
        spawn_link(fun() -> fork_start(Main,4) end),
        spawn_link(fun() -> fork_start(Main,5) end)
    ],
    spawn_link(fun() -> philosopher_start
                (Main,1,lists:nth(5,Forks),lists:nth(1,Forks)) end),
    spawn_link(fun() -> philosopher_start
                (Main,2,lists:nth(1,Forks),lists:nth(2,Forks)) end),
    spawn_link(fun() -> philosopher_start
                (Main,3,lists:nth(2,Forks),lists:nth(3,Forks)) end),
    spawn_link(fun() -> philosopher_start
                (Main,4,lists:nth(3,Forks),lists:nth(4,Forks)) end),
    spawn_link(fun() -> philosopher_start
                (Main,5,lists:nth(4,Forks),lists:nth(5,Forks)) end),

    receive_message_from_clients(10,registered),
    event({main,finished}).

% Receives the given message from the given count of
% clients.
receive_message_from_clients(0,_Message) ->
    ok;
receive_message_from_clients(Client_no,Message) ->
    receive
        {_Client,Message} -> ok
```

```
    end,
    receive_message_from_clients(Client_no-1,Message).
```

The property should describe that deadlock does not occur. Instead of trying to describe the deadlock in a very sophisticated way, the following sentence will be used:

Somebody is hungry now, nobody will eat in the future, and the clock will tick twice in the future.

If this sentence is true, it means practically that deadlock occurred. The '*clock will tick twice*' is needed because if it was not, then the sentence could be easily true without a deadlock. Consider the case when a philosopher becomes hungry and in the next moment the system is killed by QuickCheck.

The sentence translated into LTL:

$$\texttt{hungry} \wedge (\Box\neg\texttt{started\_eating}) \wedge \Diamond(\texttt{tick} \wedge \circ\Diamond\texttt{tick})$$

The explanation:

- `hungry` means that '*somebody is hungry now*',
- ($\Box\neg$`started_eating`) means that '*nobody will eat in the future*' (`started_eating` is always not true, so it is never true),
- $\Diamond$(`tick` $\wedge \circ\Diamond$`tick`) means that *'the clock will tick twice in the future'*.

The property function of the LTL formula uses the `never/1` and `tand/1` functions, which were introduced on page 29. The property function is as follows:

```
prop() ->
    ?FORALL(T,?TRACE(1000,main()),
        satisfies(T,
            never(tand([
                ?MATCHES({event,_,{hungry,_}}),
                never(?MATCHES({event,_,{started_eating,_}})),
                eventually(
                    tand(?MATCHES({event,_,tick}),
                         next(eventually(?MATCHES({event,_,tick})))))])))).
```

The code of the `never/1` and `tand/1` functions:

```
tmax(0,A) -> tnot(eventually(A));
tmax(N,A) -> always(timplies(A,next(tmax(N-1,A)))).
never(A)  -> tmax(0,A).
tand([A]) -> A;
tand([A,B]) -> tand(A,B);
tand([A,B|Others]) ->
    A_and_B = tand(A,B),
    tand([A_and_B|Others]).
```

Deadlock can occur in the program, if all philosophers take the left fork first and then try to take the second.

QuickCheck will probably find this error after a few hundred tests. The tester has to specify that instead of 100 test cases, 1000 test cases should be examined, otherwise the property will pass the tests:

```
> qc:quickcheck(1000,1000,e3:prop()).
```

The test was run 20 times, and the number of successful test cases before the counterexample was between 329 and 448. One test (the few hundred test cases) runs for one minute in average.

3.5.5 Safe shutdown of the system

When QuickCheck runs the system under test, it kills it after `Size` events happened or if there were not any events since `Timeout` time (see section 3.5.3). The developers of the program being tested have to make sure that when it happens, the system will terminate properly. Otherwise useless processes can keep running; e.g. servers waiting for a message which will never, in fact, arrive (because the other parts of the system, which could send a message, were killed).

Registering the processes When QuickCheck is generating traces for a system (i.e. it is running the system), it does not know about every process. It knows only about the generator process (which was started by QuickCheck itself) and about processes which raised an event at least once. These processes will be called *QuickCheck-registered processes* in this dissertation.

When QuickCheck decides to kill the system, it will kill the QuickCheck-registered processes, but it will not kill the others, since it does not know them.

If every process is a QuickCheck-registered process, then the system will be killed properly. However, it is hard to guarantee this, because it may happen that the system is killed just after a process was started and the process did not have time to register. Fortunately, Erlang has a link mechanism, which can help solve this problem.

Linking processes Two existing processes can be linked with the `link` function. If a process spawns another process with the `spawn_link` function, then they will be linked immediately. `link` and `spawn_link` are built-in functions in Erlang.

If two processes are linked and one of them terminates because of an error, then the other will terminate, too (unless it is a supervisor process, but now let's suppose it is not). This continues recursively: if other processes were linked to the second one, those will terminate too.

The live processes of the system under test can be imagined as an undirected graph, where:

- the nodes are processes;
- there is an edge between two processes if they are linked;
- there are special nodes, the QuickCheck-registered processes.

If the following statement is true in every moment of the system's life, then each process of the system will be killed properly when the QuickCheck decides to do that:

Every node of this graph is reachable from at least one QuickCheck-registered node.

Two pieces of advice will be written, which can help write more stable systems (in the point of view of testing with QuickCheck).

Advice 1: use `spawn_link` instead of `spawn` If a process was started with `spawn`, and it is not QuickCheck-registered, it will not be killed when the others will be, and it will not become QuickCheck-registered until the first event raised by this process. So if the system is killed after spawning the process but before its first event, then it will not receive the kill-message.

It can happen with any process, and sometimes the developer does not want a process to be registered at all. The following code is an example for the latter case:

```
-module(e1).
-export([main/0,prop/0]).
-include("quickcheck.hrl").

clock() ->
    receive
        stop -> ok
    after
        0 ->
            io:format("~w ~w~n",[time(),self()]),
            timer:sleep(1000),
            clock()
    end.

main() ->
```

```
    Clock = spawn(fun clock/0),
    event(a),
    event(b),
    Clock ! stop.

prop() ->
    ?FORALL(_T,?TRACE(3,main()),
        true
    ).
```

The `clock` process prints the time in every second. The main process spawns the clock, then two events happen, then the clock is stopped. It seems to be good. But if the `prop` property is tested with QuickCheck, the result will be that sometimes QuickCheck kills the system after spawning the clock but before stopping it, and the clock will run after the whole test was finished.

If `spawn_link` is used instead of `spawn`, this behaviour can be avoided, because when the main process is killed, the clock process will get a message that it should terminate, too.

(Note: of course the property will always hold.)

Advice 2: if the child can survive the parent, register the child If a process spawns a new one, it sometimes waits for the other to be finished. But sometimes it does not: in these cases, the new process should be registered, otherwise if the first process terminates normally, and after that the system is killed, the second process will not be notified about the shutdown of the system.

In the following example this is exactly what happens:

```
-module(e2).
-export([main/0,prop/0]).
-include("quickcheck.hrl").

clock() ->
    io:format("~w ~w~n",[time(),self()]),
    timer:sleep(1000),
    clock().

main() ->
    event(main_started),
    spawn_link(fun clock/0).

prop() ->
    ?FORALL(_T,?TRACE(10,main()),
        true
    ).
```

The `main` function spawns a `clock` process, and terminates. Then QuickCheck will think that this is the end of the test, because every registered process terminated; but the `clock` process will not be killed.

So it is not enough to use `spawn_link` instead of `spawn`, in this case this advice is worth to take, too.

3.6 Testing impure functions with QuickCheck2

Let's suppose the following program needs to be tested:

```
-module(nat_gen).
-export([start/0,stop/0,next/0]).

start() ->
    register(nat_gen_server,spawn_link(fun() -> loop(0) end)).

loop(N) ->
    receive
        {Pid,next} ->
            Pid ! {self(),N},
            loop(N+1);
        stop ->
            ok
    end.

stop() ->
    nat_gen_server ! stop.

next() ->
    nat_gen_server ! {self(),next},
    receive
        {_,N} -> N
    end.
```

This program is a natural number generator server. It has to be started by calling the `start` function, which starts the `loop` function and registers the server process with the name `nat_gen_server`. The `loop` function is the actual server, which remembers the following natural number (`N`). The server can be stopped by calling the `stop` function of the modul, which will send the `stop` message to the `nat_gen_server` process. The next number can be obtained by calling the `next` function, which will send a message to the server and wait for the answer, which will be returned to the caller.

The states and possible transitions (function calls) of the server are shown in figure 4.

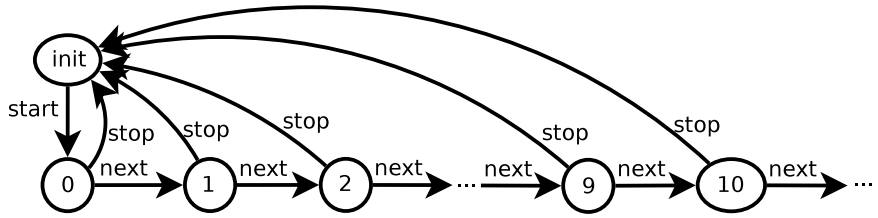

Figure 4: The states and possible transitions of the natural number generator server.

3.6.1 Abstract state machines

In QuickCheck2, abstract state machines (ASM) can be used to test impure functions. As explained previously, QuickCheck2 does not concentrate on the actual functions, but on the SUT (system under test), as a whole.

The basic idea is that the tester specifies the expected behaviour of the SUT via an ASM.

The states of this machine represent the states of the SUT, and the transitions of the machines represent the operations on the SUT: these operations are called *commands*.

(The word *command* will always be used in this sense.) These commands have preconditions and postconditions. QuickCheck will use this ASM to generate command sequences. Then the command sequences will be executed, and after each command the postcondition of the command will be examined, and if it is false, the test will be finished and reported as failed.

The ASM is specified via callback functions, which needs to be implemented by the tester. From this 5 functions, 4 are invoked during generation time (when the command sequence is generated), and 4 during runtime (when the commands are actually executed). The generation time starts with invoking the `initial_state` function, which returns the initial state of the system. The system state can be any Erlang term. Then the `command`, the `precondition` and the `next_state` functions are used to generate the command sequence. The `command` function returns the next command to be put into the command sequence, the `precondition` determines whether this command should be put into the sequence (this will be explained later), and `next_state` specifies the state of the system after executing the command (which was generated by `command`). All these functions can use the current state of the machine. During runtime the commands in the command sequence are executed after each other, one by one. The return value of the executed command is examined before executing the next one. The functions `initial_state`, `precondition` and `next_state` are used here, as well. But this time `postcondition` is used too, to examine after the execution of each command, whether the state, the command and the result of the command are correct or not.

The signature of the callback functions:

- `initial_state()`: returns the state in which each test case starts. This function always has to return the same value.
- `precondition(S,C)`: returns true if the call `C` can be performed in state `S`. This function should be deterministic as well.
- `command(S)`: returns the next command, which should be performed in state `S`, or returns the `stop` atom, if the command sequence should end. This function is probably nondeterministic, because probably there are states in the system where more transition can occur. E.g. when a chat server is being tested and two clients are connected, then maybe one of the clients will disconnect or maybe one of them will send a message; so this function should generate nondeterministically either a 'disconnect' or a 'send message' command.
- `next_state(S,R,C)`: returns the next state of the system, where `S` was the previous state, `C` the performed command and `R` the return value of the command. This function should be deterministic.
- `postcondition(S,C,R)`: returns true if the SUT seems to be correct and the `R` return value is the correct return value of the command `C`, when the system was in state `S` before performing `C`. This function should be deterministic as well.

The property function will use the `commands` and `run_commands` functions of QuickCheck. The `commands` function generates the command sequence. The running of this function is called generation time. The `run_commands` function gets the command sequence and executes it. The running of this function is called runtime.

The basic form of the property function is as shown:

```
prop() ->
    ?FORALL(Commands,commands(Module),
        begin
            {_H,_S,Result} = run_commands(Module,Commands),
            Result == ok
        end
    ).
```

The `Module` is the module which implements the callback functions of the ASM. The `FORALL` macro works like described before: In each test it runs the `commands` function, the result of which is the command sequence. The command sequence is put into the `Commands` variable. Then the `begin-end` block is called. This block runs the `run_commands` function, which executes the commands sequence contained by the `Commands` variable. The result of the `run_commands` function call is a tuple, which contains the history, the final dynamic state and the result. The latter is the atom `ok` if the test passed, and if this is the case, the value of the `begin-end` block is true; otherwise, it is false.

The testing can be run in the usual way, and QuickCheck does the test described above 100 times, as usual:

```
> eqc:quickcheck(e1:prop()).
.....................................................
.....................................
OK, passed 100 tests
```

Symbolic states As mentioned above, the command generator generated command sequences. Each command has the following form:

```
{set, {var,N}, {call, Module, Function, Args}}
```

When QuickCheck runs this command, the function `Module:Function` will be executed with the argument list `Args`. `N` is a natural number, which is individual for the command. The `{var,N}` term represents the return value of this function. It is called a symbolic variable. It is symbolic, because it does not have a value here, but during runtime (when the command sequence will be run), it will be replaced by the real value: the return value of the `Module:Function` function.

E.g. if the first command in the sequence is

```
{set, {var,1}, {call,chat_test,start_server,[]}}
```

and the second command is

```
{set, {var,2}, {call,chat_test,connect,[{var,1}]}}
```

then it means that when these two commands are actually run after each other, the return value of the first one, which is supposed to be the process id of the started server, will be given to the second one as an argument, instead of the symbolic variable `{var,1}`.

Example: `nat_gen` Let's see the solution of the example `nat_gen`!

The ASM, which will be implemented is shown in figure 5.

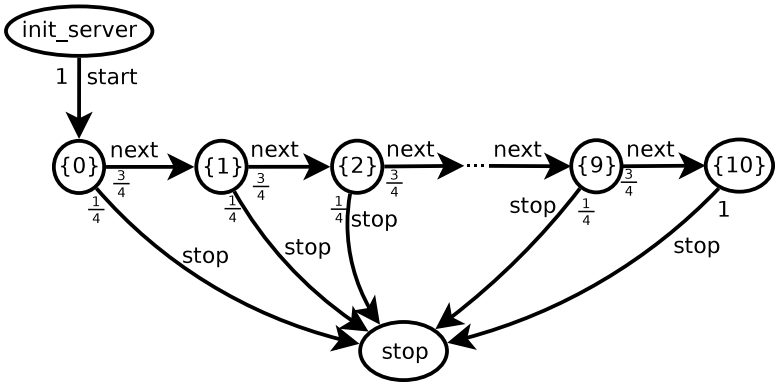

Figure 5: The ASM, which will be implemented by the tester and used by QuickCheck to test the `nat_gen` module.

The basic idea of the implementation of the ASM is that first the system goes to state `init_server`, where the number generator server is not started yet. Then the `nat_gen:spawn()` command is called, which starts the server, and the system goes to state `{0}`, which means that 0 is the following number which should be generated. Then with $\frac{1}{4}$ probability the system stops: it makes a `nat_gen:stop()` call and finishes the test, or with $\frac{3}{4}$ probability it makes a `nat_gen:next()` call. In the latter case the result of the function call is examined, and it should be the same as the number in the state, which is 0. The new state is `{1}`. A command is generated again, the system may stop or may not, the result is examined. This may continue until the state is `{10}`, when the system has to stop. (The tester assumes that if the system works until 10, then it works over 10, as well.)

The concrete implementation in QuickCheck2 follows. First the module name, the list of exported functions and used libraries are specified:

```
-module(nat_gen_tester).
-export([initial_state/0,precondition/2,command/1,
         postcondition/3,next_state/3,prop/0]).
-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").
```

The first defined callback function is the `initial_state`. It returns the `init_server` atom. The tester uses this atom to describe the first state, when the number generator server has to be initialized.

```
initial_state() ->
    init_server.
```

Now the precondition is always true, every command is accepted:

```
precondition(_S,_C) ->
    true.
```

The `command(S)` function specifies that if the system is in state `S`, which commands can run.

The following implementation of the function says, that if the current state is `init_server`, then generate a `{call,nat_gen,start,[]}` symbolic function call (which will do a `nat_gen:start()` real function call, when the command sequence is run). If the state is `{N}`, that means that the server already runs and the next number which it should send is `N`. If `N` is lower than 10, then the `frequency` function of QuickCheck (which is explained on page 21) is used to generate a nondeterministic function call. In this case `frequency` gets a list of `{Weight,Command}` tuples. It chooses between the given commands considering the given weights. E.g. now the `{call,nat_gen,next,[]}}` symbolic function call is generated with probability $\frac{3}{4}$, while the `{call,nat_gen,stop,[]}}` function call with probability $\frac{1}{4}$. If `N` is at least 10, then definitely the `{call,nat_gen,stop,[]}` symbolic function call is generated. If the state is `stop`, then the generated command is `stop`. The `stop` atom as a state is not special to QuickCheck, only the tester decided to use that to indicate that the test should stop. The `stop` as a command, however, is a special QuickCheck command, which indicates to QuickCheck that generating the command sequence is finished. (The `stop` atom will not be put into the command sequence.)

```
command(init_server) ->
    {call,nat_gen,start,[]};
command({N}) when N<10 ->
    frequency([
            {3,{call,nat_gen,next,[]}},
            {1,{call,nat_gen,stop,[]}}
        ]);
command({_}) ->
    {call,nat_gen,stop,[]};
command(stop) ->
    stop.
```

The `postcondition` function has to return a boolean: true if the state, the executed command and the return value of the latter are correct, and false otherwise.

If the state is `{N}`, the executed command is `nat_gen:next()`, and the return value of it is `R`, then the postcondition is that `N` (the number which the return value should be) and `R` (the actual return value) are equal. If the state or the call has an other form, then the postcondition is true, so these cases are not examined.

```
postcondition({N},{call,nat_gen,next,_},R) ->
    R == N;
postcondition(_,_,_) ->
    true.
```

The function `next_state` should return the new state of the system.

The current implementation says that if the state is the `init_server` atom, then the new state is `{0}`. If the state is `{N}` and the executed command was a `nat_gen:next()`, then the next state is `{N+1}`. Otherwise the command has to be a `nat_gen:stop()` command, and in this case the new state is `stop`.

```
next_state(S,_R,C) ->
    case {S,C} of
        {init_server,_} -> {0};
        {{N},{call,nat_gen,next,[]}} -> {N+1};
        {_,{call,nat_gen,stop,[]}} -> stop
    end.
```

The property function is almost the same as the template given above. The only difference is that here the `Module` is `?MODULE`, which is an Erlang macro, which will be replaced by the name of the current module by the compiler.

```
prop() ->
    ?FORALL(Commands,commands(?MODULE),
        begin
            {_H,_S,Result} = run_commands(?MODULE,Commands),
            Result == ok
        end
    ).
```

The testing can be done in the usual way:

```
> eqc:quickcheck(nat_gen_tester:prop()).
.....................................................
.....................................
OK, passed 100 tests
```

An important feature of this testing method is, in contrast with the testing method of QuickCheck1, that the module under test does not have to be modified at all; QuickCheck2 does not even need the source code of that.

3.6.2 The description of the testing method

The detailed description of the tester mechanism It is difficult to understand how this testing mechanism works without reading the exact description. That's why the flow charts of these functions are given here beside the textual description of the testing system.

A note about what the flow charts mean. These are specifications of what the `commands` and `run_commands` function do. The functions may not follow these exact steps in fact, but the result of the functions is supposed to be the same as the result of executing the algorithm specified by the flow charts.

Two things has to be examined: how the `commands` function and how the `run_commands` function work.

The `commands` function The `commands` function is used to generate the command sequence. It uses the following 4 callback functions of the ASM: `initial_state`, `command`, `precondition`, and `next_state`.

The algorithm of this function is shown in figure 6.

The `commands` function first evaluates the `initial_state` callback function, which returns the initial state. The command counter (`I`) is set to 1, and the command sequence (`Commands`) is set to an empty list.

Then a loop is started, which has the condition inside. First the `command` callback function is called with the current state. The return value should be either a symbolic command (with the form of `{call,Module,Function,Args}`), or the `stop` atom. In the latter case the test generation is finished. In the former case the `precondition` callback function is called. If it is false, it means that the generated command is not good for some reason, so the algorithm goes

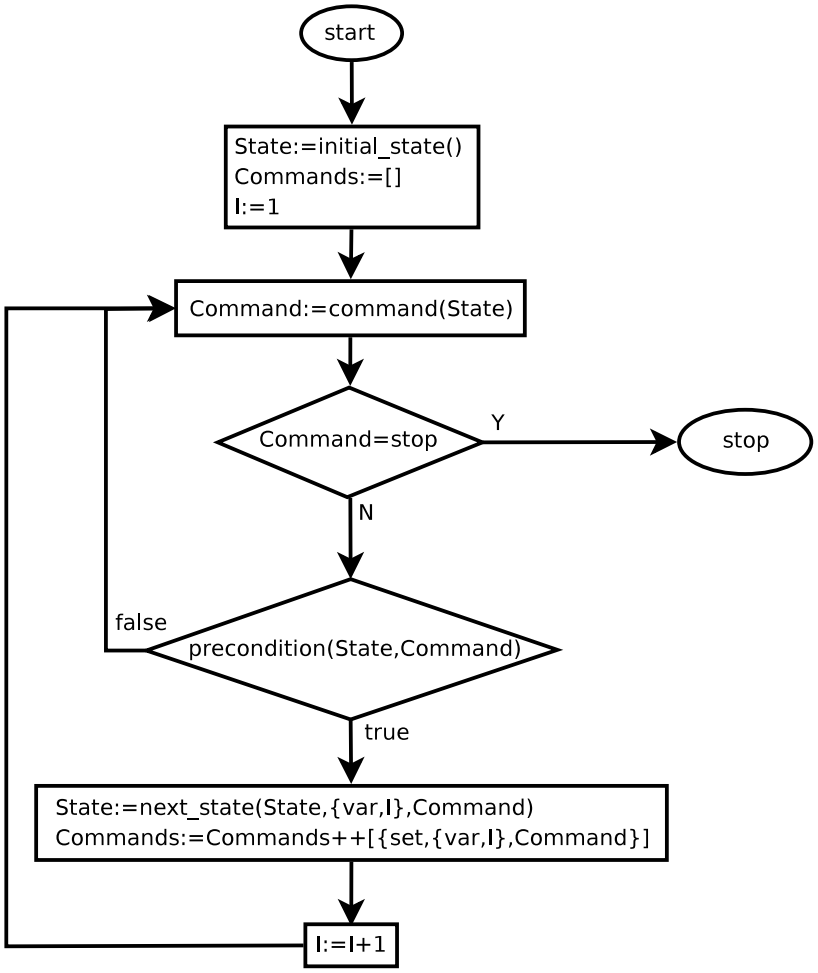

Figure 6: The flow chart of the `commands` function of QuickCheck.

back to the command generation. (If the `command` is unable to produce a command which is accepted by th precondition, then the `commands` function will do an infinite loop, and the tester has to stop it.) When the `Command` is accepted by the `precondition`, it is appended to the command sequence (`Commands`). (In fact, not the command is appended, but it is wrapped into a symbolic command: `{set,{var,I},Command}` is the new command, where `I` is the command counter.) Then the `next_state` is called to compute the next state of the system. Finally the command counter (`I`) is increased, and the loop starts again.

The `precondition` callback function can be useful when the tester does not want to correct the program immediately when he found a bug, or when the program is not finished. In these cases the `precondition` function can prevent generating command sequences about which the tester knows that they are buggy or unfinished.

The `run_commands` function The `run_commands` function is used to run the previous generated command sequence. It uses the following 4 callback functions of the ASM: `initial_state`, `precondition`, `postcondition`, and `next_state`.

The algorithm of this function is shown in figure 7.

The `run_commands` function first evaluates the `initial_state` callback function, which returns the initial state.

Then follows a loop, which runs for each command in the command sequence. The current command is `Commands[I]`. If there is a problem, then the loop will be broken; if there is no problem, then after the execution of all commands, the test will be passed. The loop first examines whether the precondition of the current command is true; if it is not, then the test fails. (This should not happen, because in this case the precondition should have failed in generation time, and a new command should have been generated instead of the actual one.)

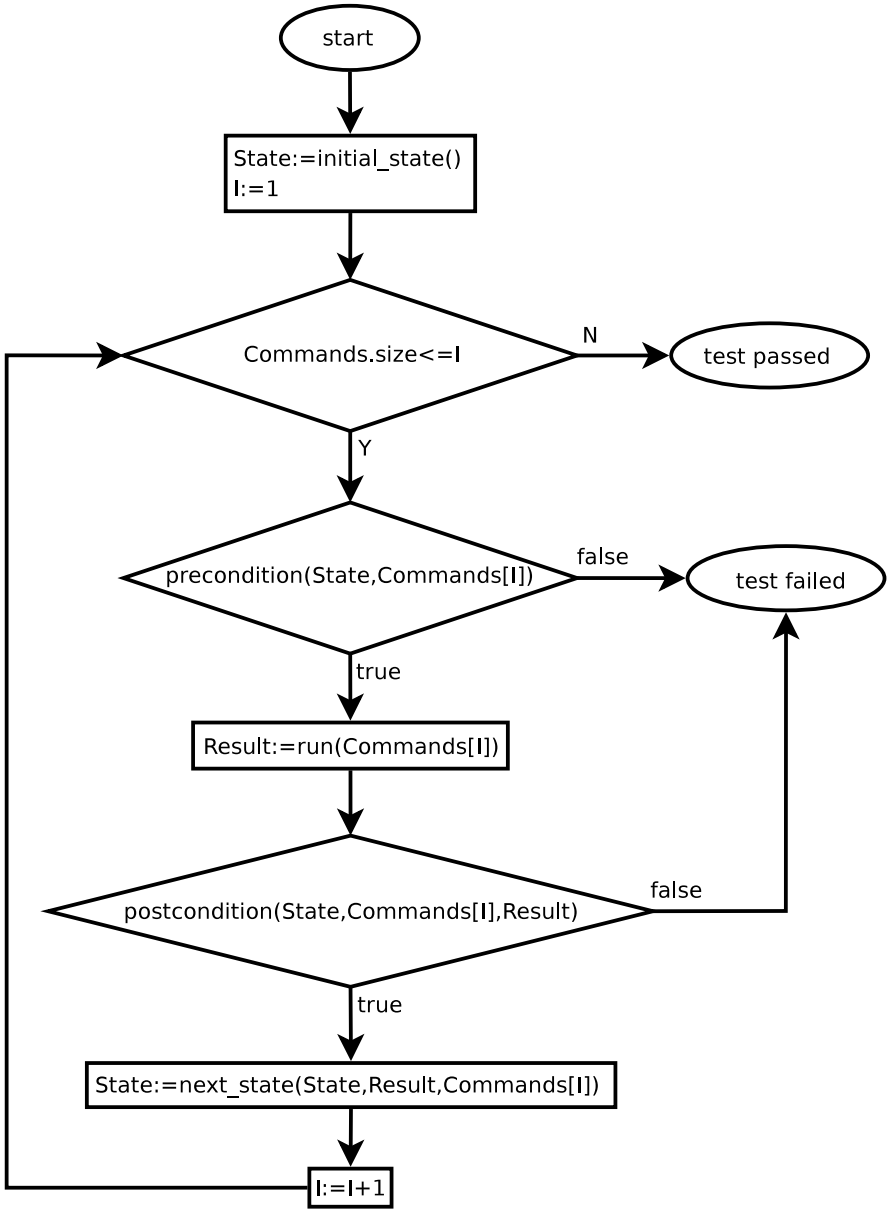

Figure 7: The flow chart of the `run_commands` function of QuickCheck.

After examining the precondition, the command is executed. Then the `postcondition` callback function is invoked to check the postcondition. If it is false, the test fails. Otherwise the next state is calculated with `next_state` and the loop starts again.

The states during the running of this function are not the same as the states during the running of the `commands` function. There are symbolic states with symbolic values (`{var,I}`), and here are dynamic states with real values.

The tester process of QuickCheck2 is less robust than the tester process of QuickCheck1. QuickCheck1 kills the SUT (see 3.5.3) after a certain time, while QuickCheck2 does not. If the ASM or an operation has a function which does not terminate, then the tester process will not terminate either. The disadvantage of QuickCheck2's solution is that it is more difficult to find out what happened in the mentioned cases, since the tester will not get any information from QuickCheck (it just runs and never stops). The advantage is that there is no need to specify timeouts, and there is no need to make the SUT always ready to terminate. That is a difficult task, and probably the code of SUT has to be modified to achieve this.

What does QuickCheck2 test? QuickCheck2 tests whether the SUT (the implementation) *conforms* to the ASM (the specification). However, this is not a complete answer, unless we specify what *conform* means.

The SUT is considered to be a black box by QuickCheck, with an interface consisting of operations. These operations are Erlang functions. QuickCheck2's commands are function calls that call these operations. The ASM specifies which command sequences can be run, and which condition has to be true after running each command. QuickCheck tests the following:

For all route of the ASM that starts in the initial state and ends in a final state, the command sequence of the route has to be able to be performed on the SUT, and after every performed command, the postcondition has to hold.

A state is final state, if it can be the last element of a command sequence; technically it means that the `command` callback function may generate `stop` atom in that state.

The principle behind the sentence above is the following: if the SUT is used correctly, then it works correctly. If it is used incorrectly, anything can happen. There is a way, however, to test what happens, when the SUT is used incorrectly, see later.

We may be interested in examining the testing method, if we do not consider the SUT to be a black box, as QuickCheck does, but to be a state machine. This state machine is similar to the ASM, but it does not have probabilities on the transitions. (The state machine of the implementation of the `nat_gen` example is in figure 4 on page 41.)

We cannot give a better definition here for *conform* then we did previously, but examples can be shown for what is tested by QuickCheck and what is not.

QuickCheck tests whether traces of the ASM are traces of the implementation. Therefore the implementation shown in figure 8 does not conform to the ASM, because the trace ⟨`b`⟩ is contained by the ASM, but not by the implementation.

Figure 8: The implementation does not conform to the ASM, because it does not contain every trace of it.

QuickCheck tests these traces – it runs them and checks the postcondition. But it does not check those traces, which are not contained by the ASM. Therefore the implementations shown in figure 9 both conforms to the ASM. So the traces of the ASM has to be traces of the implementation, but it is not true in the other way around.

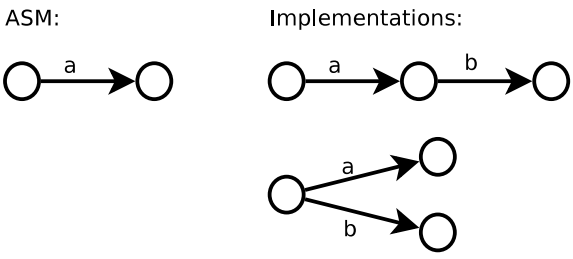

Figure 9: Both implementations conform to the ASM, if the postconditions do not say otherwise.

In other words, it tests, what happens, when the SUT is used correctly, but it does not test, what happens, when it is used incorrectly. Incorrect usage means that the command sequence is not a correct command sequence according to the ASM, i.e. the ASM does not contain it. Even though QuickCheck does not support testing incorrect command sequences, testing these can be simulated using the `command`, `next_state` and `postcondition` functions, as shown in figure 10. In this solution, all the incorrect commands can be generated, not only the correct ones, and they lead to state `error`, where the postcondition is false.

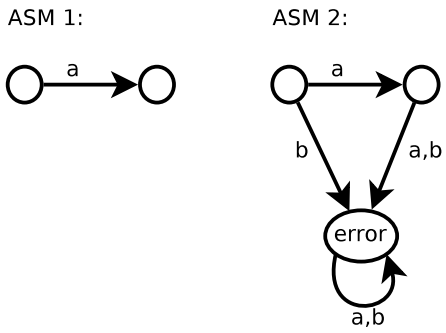

Figure 10: In the case of ASM 1 QuickCheck tests only the command sequence $\langle \mathtt{a} \rangle$, while in the case of ASM 2, it tests every possible command sequence.

QuickCheck checks the states only via the `postcondition` function; it does not examines whether two states are the same or not. Depending on `postcondition`, the implementation shown in figure 11 can conform to the ASM-s. So, again, checking whether two states are equivalent or not, can be done using the `postcondition` function.

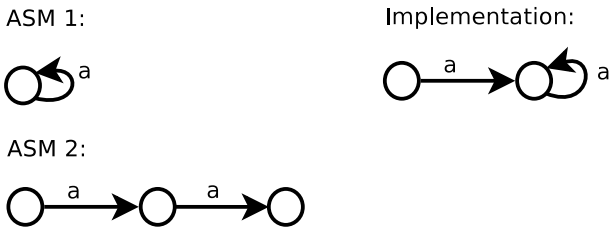

Figure 11: The implementation conforms to both ASM-s (if the postconditions do not say otherwise), because all the traces contained by ASM 1 are contained by the implementation, and the same is true for ASM 2. QuickCheck does not examine the relation of states.

We can have a question: could not QuickCheck test the SUT against the properties above? Could not it test it against using it in an incorrect way, or test whether the different states in the ASM are really different states in the SUT, too? It could, but there are two reasons why not to test these. The first reason is a technological reason. Even though we analyse the connection between the ASM and the state machine of the implementation, QuickCheck does not know about the latter. For QuickCheck, the implementation is only a black box. If the tester would need to do the tests above, a different testing environment should be implemented. The other reason is that it is not obvious, what should the test environment do, when testing the SUT against incorrect traces. Should it expect that the SUT will terminate, or that it will do nothing? And what should it do, when two states are the same in the ASM, but not in the SUT? It does not mean automatically that there is an error in the SUT. If a test environment wants to have these features, the answers to these questions must be able to be specified.

Symbolic and dynamic states In QuickCheck2, generating and running the commands are two separate activities. The former contains only symbolic states, while the latter deals with both.

State sequences When testing, first the whole command sequence is generated – during this time, which is called *generation time*, only the ASM is used, the SUT cannot affect the command sequence in any way. Then the command sequence is run – during this time, which is called *runtime*, the SUT has to perform the fix command sequence. The SUT is expected to have the same states as the ASM had during generation time; with the exception that the SUT has dynamic values, where the ASM had symbolic variables.

Let's explain the last sentence with an example. First consider an example of a `nat_gen` command sequence. If the generated command sequence was:

```
{call,nat_gen,start,[]},
{call,nat_gen,next,[]},
{call,nat_gen,next,[]},
{call,nat_gen,stop,[]}
```

and the states were the following:

```
init_server,
    {0},
    {1},
    {2},
    stop
```

then the states of the SUT should be the same sequence. QuickCheck will not check this, but it is considered as a wrong solution, if an ASM does not follow this rule.

Symbolic and dynamic states Comparing the state sequences is more complicated, when there are differences between the symbolic and dynamic command sequences.

First let's modify the `nat_gen` example! The only difference will be, that the `nat_gen_server` process was registered there, while it will not be registered here. The pid of the server process will be stored in the state of the ASM/SUT.

The code of `nat_gen_2` is as follows:

```
-module(nat_gen_2).
-export([start/0,stop/1,next/1]).

start() ->
    spawn_link(fun() -> loop(0) end).

loop(N) ->
    receive
        {Pid,next} ->
            Pid ! {self(),N},
            loop(N+1);
        stop ->
            ok
    end.

stop(Nat_gen_server) ->
    Nat_gen_server ! stop.

next(Nat_gen_server) ->
    Nat_gen_server ! {self(),next},
    receive
        {_,N} -> N
    end.
```

The code of `nat_gen_2_tester` is as follows:

```
-module(nat_gen_2_tester).
-export([initial_state/0,precondition/2,command/1,
         postcondition/3,next_state/3,prop/0]).
-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").

initial_state() ->
    init_server.

precondition(_S,_C) ->
    true.
```

```
command(init_server) ->
    {call,nat_gen_2,start,[]};
command({Server,N}) when N<10 ->
    frequency([
            {3,{call,nat_gen_2,next,[Server]}},
            {1,{call,nat_gen_2,stop,[Server]}}
        ]);
command({Server,_}) ->
    {call,nat_gen_2,stop,[Server]};
command(stop) ->
    stop.

postcondition({_Server,N},{call,nat_gen_2,next,_},R) ->
    R == N;
postcondition(_,_,_) ->
    true.

next_state(S,R,C) ->
    case {S,C} of
        {init_server,_} -> {R,0};
        {{Server,N},{call,nat_gen_2,next,_}} -> {Server,N+1};
        {_,{call,nat_gen_2,stop,_}} -> stop
    end.

prop() ->
    ?FORALL(Cmds,commands(?MODULE),
        begin
                {_H,_S,Result} = run_commands(?MODULE,Cmds),
                Result == ok
        end
    ).
```

Here a possible command sequence is:

```
   {call,nat_gen_2,start,[]},
{call,nat_gen_2,next,[{var,1}]},
{call,nat_gen_2,next,[{var,1}]},
{call,nat_gen_2,stop,[{var,1}]}
```

The corresponding symbolic state sequence:

```
init_server,
{{var,1},0},
{{var,1},1},
{{var,1},2},
    stop
```

The term `{var,N}` represents the return value of the `N`th command. Here the return value of the first command (`{var,1}`) gets into the state.

When the SUT is tested, dynamic states will be used instead of the symbolic ones. The command sequence will contain symbolic states, but these will be transformed into dynamic ones. The dynamic state of the system is `init_server` in the beginning. The `{call,nat_gen_2, start,[]}` command, i.e. the `nat_gen_2:start()` function call will be performed. The pid of the started server will be its return value; e.g. it can be `<0.100.0>`. QuickCheck will remember that the value of the first variable (`{var,1}`) is `<0.100.0>`. When a command has `{var,1}` inside, it will be replaced by `<0.100.0>`. After the command the new state of the system is `{<0.100.0>,0}`, which is calculated by the `next_state` function. (The `next_state` function should not have a different return value as it had when it was called with a symbolic state. Technically, it can happen, but as said before, it is considered to be a wrong technique.) The

second command is `{call,nat_gen_2,next,[{var,1}]}`, where `{var,1}` will be replaced by `<0.100.0>`, so the `nat_gen_2:next(<0.100.0>)` function call will be performed.

After the execution of all commands, the dynamic state sequence will be the following:

```
 init_server,
{<0,100,0>,0},
{<0,100,0>,1},
{<0,100,0>,2},
     stop
```

3.6.3 An alternative implementation of the ASM testing engine

Testing with ASM in QuickCheck2 has some problems. The root of these problems is the difference between generation time and runtime.

The most important problems:

- It is difficult to understand and use the concepts of symbolic and dynamic states.
- The `next_state` function has to be deterministic. Some problems would be easier to solve if it was not a restriction.
- The testing environment is not able to adapt to the decisions of the SUT. The SUT cannot have any impact to the command sequence. To be more concrete: the `next_state` function cannot use the return values of the operations to determine the next state. It can copy them, but it cannot read them. A lot of problems would be easier to solve if the next state could consider the decisions made by the SUT. See the example later.
- `{var,N}` tuples cannot be used in the program, because QuickCheck tries to replace them by return values of commands. (See appendix B for an example.)

An alternative ASM test engine will be described here. The specification of it is very similar to the specification of the ASM test engine of QuickCheck2, but without symbolic states and without the restrictions above. In this solution there is one algorithm instead of two, shown in figure 12.

Here after one command is generated, it is executed. The next state and next command are generated only after the execution is finished. Otherwise the logic of the solution is similar to the logic of QuickCheck's solution.

The features of this solution:

- If a correct SUT conforms to a correct ASM in QuickCheck2 using the standard property function (which was used in the examples), it will conform here as well. Correct means that it does not use the implementation details of QuickCheck2, only the behaviour documented in this dissertation.
- The `next_state` function can be nondeterministic.
- The `next_state` function can use the dynamic state of the system.
- There are no symbolic states, so `{var,N}` is not a special tuple, it can be used anywhere.

Example: `nat_gen_3` The `nat_gen_3` is another natural number generator. It is similar to `nat_gen_2`, but here the next number which is generated does not have to be the successor of the previous one – it just has to be higher. In the example implementation, it can be higher maximum by 10, but that is irrelevant to the ASM which will test it.

The code of the module is as follows:

```
-module(nat_gen_3).
-export([start/0,stop/1,next/1]).

start() ->
    spawn_link(fun init/0).

init() ->
```

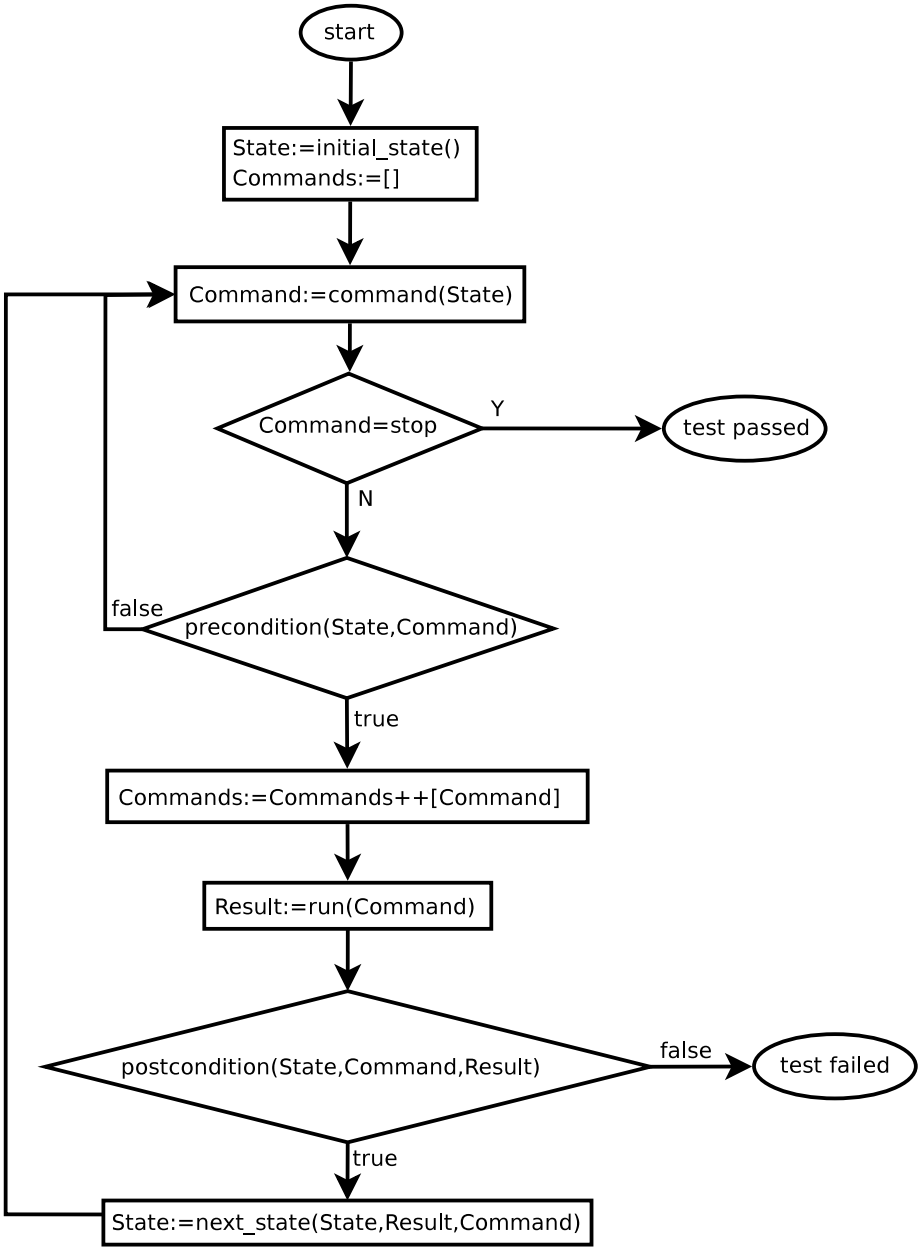

Figure 12: The flow chart of the `asm_tester`, an alternative ASM testing engine.

```
    {A,B,C} = now(),
    random:seed(A,B,C),
    loop(random:uniform(10)-1).

loop(N) ->
    receive
        {Pid,next} ->
            Pid ! {self(),N},
            loop(N+random:uniform(10));
        stop ->
            ok
    end.

stop(Nat_gen_server) ->
    Nat_gen_server ! stop.

next(Nat_gen_server) ->
    Nat_gen_server ! {self(),next},
    receive
        {_,N} -> N
    end.
```

The code of the tester module is:

```
1   -module(nat_gen_3_tester).
2   -export([initial_state/0,precondition/2,command/1,
3            postcondition/3,next_state/3]).
4   -import(asm_tester,[frequency/1]).
5
6   initial_state() ->
7       init_server.
8
9   precondition(_S,_C) ->
10      true.
11
12  command(init_server) ->
13      {call,nat_gen_3,start,[]};
14  command({Server,N}) when N<10 ->
15      frequency([
16              {3,{call,nat_gen_3,next,[Server]}},
17              {1,{call,nat_gen_3,stop,[Server]}}
18          ]);
19  command({Server,_}) ->
20      {call,nat_gen_3,stop,[Server]};
21  command(stop) ->
22      stop.
23
24  postcondition({_Server,N},{call,nat_gen_3,next,_},R) ->
25      R >= N;
26  postcondition(_,_,_) ->
27      true.
28
29  next_state(S,R,C) ->
30      case {S,C} of
31          {init_server,_} -> {R,0};
32          {{Server,_},{call,nat_gen_3,next,_}} -> {Server,R+1};
33          {_,{call,nat_gen_3,stop,_}} -> stop
34      end.
```

The differences between this and `nat_gen_2_tester`:

- Line 4: using the `frequency` function of the `asm_tester` module and not using the QuickCheck2 libraries.
- Line 25: `R>=N` instead of `R==N`. It is because now the specification is that the next number has to be higher then the previous one (which was `N-1`).
- Line 32: `{Server,_}` instead of `{Server,N}`: `N` is not used now. (The compiler would give a warning that it is not used if the previous version remained.)
- Line 32: `{Server,R+1}` instead of `{Server,N+1}`: the next number has to be at least `R+1` and not at least `N+1`.
- We do not have a property function. (The property is contained implicitly by the `asm_tester` module.)

`asm_tester` can test the implementation module `nat_gen_3` against the ASM module `nat_gen_3_tester`:

```
> asm_tester:test(nat_gen_3_tester).
.....................................................
.....................................
All tests passed.
```

QuickCheck2 cannot test this program against this ASM, because the `next_state` uses the dynamic state of the system, which is not allowed in QuickCheck2.

Implementation of `asm_tester` The implementation of `asm_tester`, which performs similar task as the ASM test engine of QuickCheck2, is not complicated. It is basically the implementation of the flow chart. The source code can be found in appendix A. It is not robust, it was made only to show that this algorithm can be coded easily.

Evaluating `asm_tester` and comparing to QuickCheck's ASM tester engine The advantages of `asm_tester` over QuickCheck was discussed in the previous paragraph; this paragraph talks about its problems.

An obvious problem of the current implementation of `asm_tester` is that it is not integrated into QuickCheck, so it cannot be used with all the infrastructure that are given by QuickCheck: data generators, shrinking, `FORALL`, `IMPLIES`, etc. It would be possible to make the QuickCheck's ASM testing engine similar to this, or integrate this into QuickCheck2 as alternative ASM testing engine. In these cases this problem would not exist any more.

A bigger problem is that QuickCheck's philosophy is to make a distinction between the data generator and the data property, and it has a very good reason to do that. This distinction makes possible e.g. shrinking, as it is explained in section 3.7 on page 57. That philosophy also remains when writing the property function for a module which will be tested with ASM:

```
prop() ->
    ?FORALL(Commands,commands(Module),
        begin
            {_,_,Result} = run_commands(Module,Commands),
            Result == ok
        end
    ).
```

The `asm_tester` does not have this distinction, so the tester cannot manipulate the command sequence after it is ready. (He can manipulate it gradually, after the generation of each command and after the execution of each command.) And, which is worse, QuickCheck cannot manipulate it, so e.g. shrinking cannot be done.

If a data generator (such as `commands`) is implemented, it has to return the generated data. If a simple data property (such as the `begin-end` block above, using the functionality of the `run_commands` function) is implemented, it has to return a boolean. But if the ASM tester was to be integrated into QuickCheck2, a construction similar to the `FORALL` macro should be implemented, in the sense that it should return the information whether the property passed

and if not, then also the command sequence (the data) should be returned. It is not difficult to implement this, it just means that the data property of this solution would be in a different category as the other, simpler data properties.

So the biggest problem is that this solution cannot give symbolic counterexamples, which may be used to reproduce the error. It would be worth to think about whether the advantages of the two solutions could be unified, i.e. could not we make a testing engine, where the SUT can affect the test cases, and still, the test cases would be symbolic.

Summary To summarize, what QuickCheck2 provides: the tester can specify the behaviour of the SUT, and QuickCheck generates, runs and evaluates the test cases.

The tester has to:

- specify the abstract state machine.

QuickCheck will:

- generate test cases,
- run and evaluate them.

3.7 Philosophy of QuickCheck

The word *property* can be used to describe related terms, and now a distinction will be made between them.

A general form for QuickCheck *property functions* is:

```
prop() ->
    ?FORALL(X,gen(),p(X)).
```

The data flow diagram of this property function can be seen in figure 13.

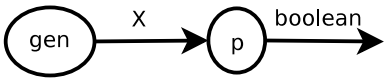

Figure 13: The data flow diagram of a general form of the QuickCheck property.

The property function is an Erlang function. (In the example, it is the `prop()` function.) This will be invoked once by QuickCheck in the beginning of the testing and the return value of the function is a tester function or a boolean. (It is mostly a tester function, so this case will be considered in the followings.) The tester function will be invoked by QuickCheck many times, and it also has a `Size` argument, as described in section 3.5.3.

Generally the tester does not need to think about the tester function. He writes the *complete property expression* (or *complete property*), using the macros and functions of QuickCheck, and its value is the tester function; this value will be calculated by Erlang. (In the example the `?FORALL(X,gen(),p(X))` expression is the complete property).

The complete property is an Erlang expression, and it is complete because it represents a closed, first-order logic formula, so its truth value is either true or false. (In the example, the logic formula which is represented by the complete property is the following: $\forall$`X` $\in$ `gen()` : `p(X)`.) QuickCheck's aim is to compute the truth value of the complete property, because if it is false, then the test should fail, otherwise it should pass. However, QuickCheck will not necessarily completely determine the truth value of the complete property. If QuickCheck's result is that the test passed, which suggests that the truth value is true, the truth value may be false. On the other hand, if the result is that the test failed, then the property is definitely false.

Complete properties can contain *data generators* and *data properties*. (Our example contains only one data generator and one data property, but other complete properties can contain more. In section 3.1 the second example is about a property with two data properties (`not(length(L)==0)` and `lists:nth(1,L)==hd(L)`), and the third example is about a property with three data generators (all the three data generators are `list(int())` functions).)

The data generator is an Erlang expression, which generates the test data. (In the example, the `gen()` expression is the data generator.) The data property is a boolean expression inside the complete property. (In the example, `p(X)` is the data property.) It is called data property, because it is the property of the data: its truth value depends on the data, and it will be calculated completely, unlike the truth value of the complete property.

The data generator may or may not have an input, but it always has an output. The data property almost always has an input, and the output is a boolean value. The output of the data generator is usually the input of the data property, as it can be seen in figure 13.

The boundary between the data generator and data property is very important, since the data which crosses this boundary is what the user gets if the complete property turns out to be false. Actually, this data is the counterexample. QuickCheck manipulates this data, too: when it tries to shrink the counterexample, this is the data that QuickCheck tries to simplify. Usually this counterexample can be used to reproduce the error. This is not true, however, when the SUT is not deterministic or in the case of QuickCheck1's trace generation.

This theory about writing data generators, data properties and connecting them with `FORALL` and `IMPLIES` macros works well with pure functions. A function is pure if the result of the function depends only on its arguments, and it is side-effect free, so its only output is its return value.

If a function is not pure, it is called impure. The behaviour of an impure function may depend on the state of the system and it may modify this state: it may print messages to the user, read and write files, communicate with other processes, read and set global variables, etc.

There are two problems arising when testing impure functions. The first is a technical one: it is easy to manipulate the input of a pure function and access the output of it, because they are only the arguments and the return value. But when one wants to manipulate the input of an impure function (files, communication, global variables), and access its output (again: files, communication, global variables), it is a more difficult task to do. The other problem is that the correct behaviour of an impure function depends on the state of the system, so this state needs to be accessed and considered by the tester function.

Neither of the QuickChecks can solve these problems completely, but they give tools to the tester so he can solve them more easily. The tool of QuickCheck1 is the trace generator, which helps solve the half of the first problem and the second problem. The tool of QuickCheck2 is the abstract state machine, which helps solve the second problem.

QuickCheck1 In the previous paragraph it was said that QuickCheck1 helps solve the half of the first problem: this was that manipulating the input of and accessing the output of impure functions is difficult.

QuickCheck1's trace generator was made to make it easier to access the output of impure functions. It helps some to access the system state as well, since the system state can be (partly or totally) calculated from the event trace; this was the second problem.

QuickCheck1's solution for accessing the output of an impure function is to allow to look inside the function. The tester writes event function calls into the code of the SUT, and these function calls will be collected together into a list (called event trace), during the execution of the SUT. The tester can put any value into the trace, so in this way the tester property can see what was happening in the SUT and it can decide whether it behaved correctly.

QuickCheck integrated the trace generation into the testing system: the trace generator is a data generator. It was mentioned before that there is an important boundary between the data generators and the data properties. The trace generator's being a data generator means that the event trace is among the data which will be provided to the tester if the property fails.

There is an additional benefit of using QuickCheck1's trace generator, which helps when testing concurrent programs. The Erlang scheduler is too predictable, and the trace generator has its own scheduler, so it is more random that which process runs next.

QuickCheck1 does not give any tool which could help solve the other part of the first problem: manipulating the input of the program being tested. The trace generator is a passive observer, which collects data, and then gives it to the property to examine it. It is good for programs which does not need input, e.g. a simulation. But if a service, e.g. a server needs to be tested, then the tester has to write complete test clients. In other words, just a complete system can be tested (like a simulation or a server and a client together), a component (like a server alone) cannot be. This is the field where QuickCheck2 is better then QuickCheck1, because it was made to test a component (but unfortunately it has problems with complete systems).

The problem and the solution can be demonstrated with the data flow diagram, as well.

A possible data flow diagram of a QuickCheck property is in figure 14.

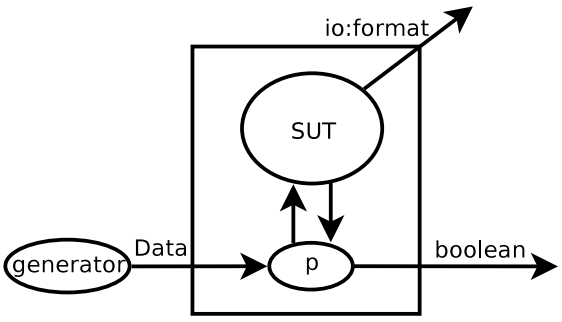

Figure 14: A possible the data flow diagram of a QuickCheck property.

The `generator` generates `Data`, and `p` is the data property expression. `p` should start, supervise, and stop the SUT. `p` should give all the inputs to it, get all the outputs and analyse them considering the current state of the SUT. Finally it should return a boolean value.

The figure shows a problem, that e.g. a kind of output of the SUT which is difficult to get is that was written to the standard output.

But aside from this, `p` has to implement a lot of things: it may have to create new processes, manipulate files, etc.

QuickCheck1 suggests another solution: the use of the trace generator. A possible form of using the trace generator of QuickCheck1:

```
prop() ->
    ?FORALL(T,?TRACE(3,start()),p(T)).
```

The data flow diagram of this property function can be seen in figure 15. It shows that the SUT is put into the data generator part instead of the data property part. The trace generator starts and stops the SUT. In this solution the SUT has to raise events: all the important events and variables has to be sent to the trace generator. The expression `p` do not have to manage the execution of the SUT.

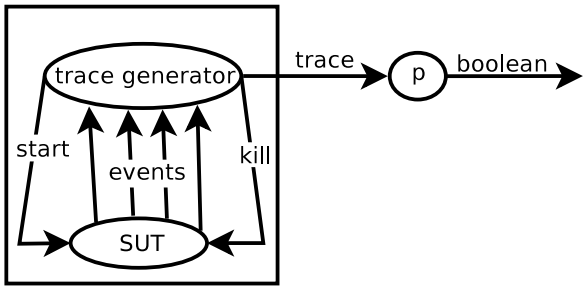

Figure 15: The data flow diagram of a possible form of using the trace generator. The components inside the box are the data generator.

QuickCheck2 The two problems of testing impure functions were that it is difficult to manipulate the input and access the output of impure functions, and that the correct behaviour of the SUT depends on its state, so this state needs be accessed and considered by the tester function.

While QuickCheck1 concentrates on the first question (accessing the output of the SUT), QuickCheck2 concentrates on the second one. In QuickCheck2 the tester can specify an abstract state machine, which describes the states and state transitions of the SUT. QuickCheck2 uses this state machine to determine what input to generate to the SUT, and what is the correct output. Technically, these inputs are function calls and the outputs are return values. The tester can write the functions which are called by QuickCheck2, so these functions serve as a bridge between QuickCheck2 and the SUT. In QuickCheck1, there is no such bridge needed: and it costs much, because there, the original code has to be modified (with event function calls). Here the original code does not have to be touched; instead, another layer needs to be written: the bridge, which was mentioned before. With this architecture, the input of the SUT can be manipulated, the output can be accessed and can be taken into account when changing the state; and in the other way around, the manipulation of the input can take into account the current state of the system. So this architecture is the answer of QuickCheck2 to both questions.

In QuickCheck1, the decisions about what will happen are made in the SUT. The system can be either deterministic or nondeterministic – however, the latter is more probable in the case of testing a system (and not just one function). In contrast, in QuickCheck2, the SUT has to be deterministic. The abstract state machine can be nondeterministic, but it models the nondeterminism of the environment and not the nondeterminism of the SUT. When the SUT is in a state (in runtime), it cannot determine the next state, because it already has been determined by the abstract state machine (in generation time). [2] In other words, all these decisions (what is the next state) are made in the abstract state machine, before even starting running the SUT. Thus the SUT has to conform to the nondeterminism of the environment and

[2] Technically, in can be done, but the code of such an abstract state machine is considered to be bad code.

be itself deterministic. This philosophy suits for testing e.g. a server, where the decisions are made by clients, and the server just has to do its duty; but it may not fit e.g. for a simulation.

The general form of the property function in QuickCheck2 (when using an abstract state machine):

```
prop() ->
    ?FORALL(Commands,commands(Module),
        begin
            {_,_,Result} = run_commands(Module,Commands),
            Result == ok
        end
    ).
```

The data flow diagram of this property function is shown in figure 16. The `commands` function (provided by QuickCheck) uses the other 4 functions (written by the tester) to generate a command sequence. This is the data generator part: the command sequence is the data. The center of the data property is the `run_commands` function (also provided by QuickCheck), which uses the generated command sequence to run the commands.

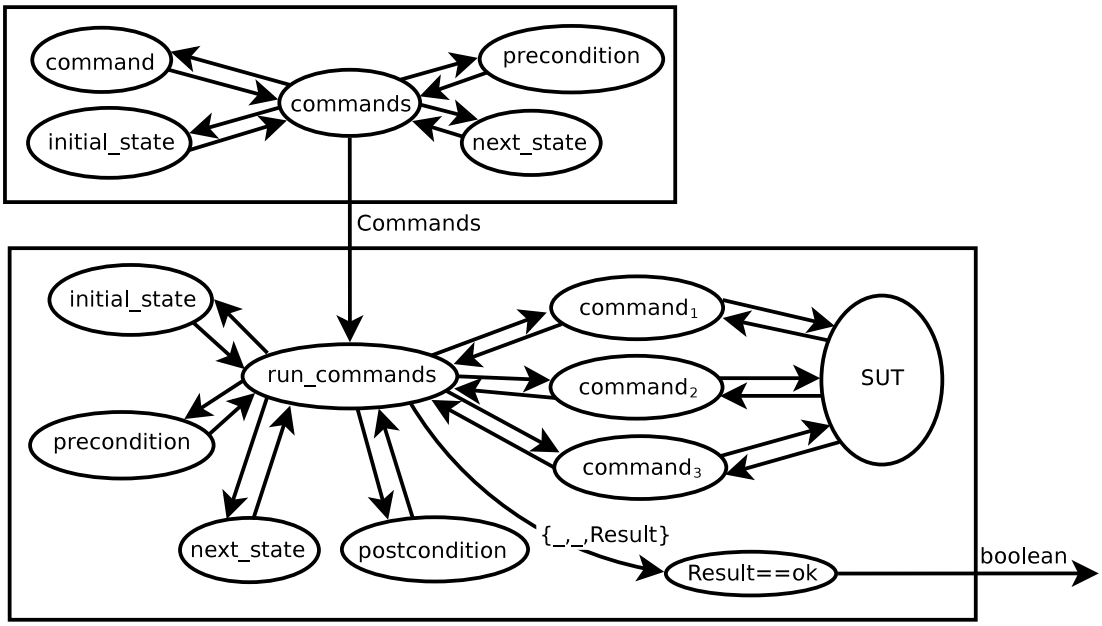

Figure 16: The data flow diagram of a QuickCheck2 property function.

Comparison Finally, we compare the two QuickChecks and their solutions to the problems, which were as follows:

1. It is difficult for the testing system to manipulate the input of the SUT.
2. It is difficult for the testing system to access the output of the SUT.
3. The correct behaviour of a SUT depends on the state of the system, so this state needs be accessed and considered by the tester function.

QuickCheck1:

1. It does not do anything about manipulating the input of the SUT.
2. It solves accessing the output, but this solution has a high price: the original code has to be modified very heavily.

3. It gives the trace to the data property written by the tester, so it is the tester's task to calculate the state of the SUT and decide whether it behaved correctly considering that state; so QuickCheck1 helps some in solving this problem.

QuickCheck2:

1. The tester has to write a bridge between the abstract state machine and the system, so QuickCheck2 does not help how to manipulate the system technically; but it recommends an architecture, and will generate the input to be given to the system.
2. The answer to accessing the output is the same as the answer to the previous questions.
3. The tester has to write the callback functions of the abstract state machine; these functions will consider the states and determine which transition can occur, what should happen at which transitions, and when is a state correct.

So if the input has to be manipulated and a component is tested, then QuickCheck2 is the better choice, but if the system needs to be observed and the decisions has to be made in the system being tested, then QuickCheck1 is better.

4 Case study: testing a chat server

The source code of these modules can be found in appendix C.

4.1 Module chat_server

The chat_server is a very simple program, with which two processes can communicate with each other.

4.2 Testing with QuickCheck1

4.2.1 Modules

Module my_ev This module is the my_ev module of the pattern introduced in section 3.5.1, on page 32.

Module my_qc This module is similar to a library, it contains functions and macros which are useful for every program working with QuickCheck1.

This module contains:

- A part of the pattern introduced in section 3.5.1, on page 32.
- Definition of the MY_MATCHES macro, described in section 3.5.2 on page 34.
- User defined LTL functions, as described in section 3.4.2 on page 29.

Module chat_test This module contains two properties, and the generator functions and the clients used by them.

The properties The first property says that 'every process should be finished, which was started'.

The second property says the followings:

- If two clients send messages to a third one, then it should get them exactly once.
- If two clients send messages to a third one, and one was posted before the other, then the former should be received first.
- A client should not get its own message.

Calling the generate_nonempty_message_table function In the testing module, the generate_message_table() function makes a generate_nonempty_message_table(5) function call instead of generate_nonempty_message_table(Size). It is because the effect that was described in section 3.5.3 on page 35.

We compared how the testing runs in the two cases. The full stops indicates that a test was run, and the commas that a test succeeded to finish. One thousand tests were run in both cases.

When generate_nonempty_message_table(5) was called:

```
> qc:quickcheck(1000, 1000, chat_test:prop_1()).
............................................................
.............,..................,...............,........,.,
.....,....,..,...,.,..,....,.,.,..,..,.,..,..,.....,.,...,..
,.,..,...,.,.,.,.,.,.,.,.,...,.,.,.,.,.,.,...,.,.,.,.,.,.,.,
.,.,.,.,.,.,.,.,.,.,.,.,.,..,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
```

```
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.,.
OK, passed 1000 tests
ok
```

When `generate_nonempty_message_table(Size)` was called:

```
> qc:quickcheck(1000, 1000, chat_test:prop_1()).
............................................................
............................................................
....................,.....................,.................
...,................................................,.......
..,..............,.,........................,...............
......,.............,...,........,........,...,.............
.......................,........................,...........
.....,.......................,..............................
.............,.............................,....,...........
..........,.......,..................,.........,..........,.
........................................................,.,.
..............................,........................,...,
................................,.....,.....................
.........,.............................,...........,........
.........,.......,........,...............,.................
..........................,.................................
......,..,.................,......,..............,.....,....
.,..........,................
OK, passed 1000 tests
```

In the first case, almost all the test cases was finished after the 200th. In contrast, in the second case, there is no such tendency, here only 49 test cases were finished out of the 1000.

4.2.2 Running the tests

Now the tester wants to test two property functions, so the usual `qc:quickchech` function has to be called twice:

```
> qc:quickcheck(1000, 1000, chat_test:prop_1()).
[...]
> qc:quickcheck(1000, 1000, chat_test:prop_2()).
[...]
```

The number of test cases were given: this is the first argument, so QuickCheck will generate and examine 1000 test cases. The second argument of the `qc:quickcheck` function is a limit, which is 1000 by default.

Errors in the system In this section examples are shown about what happens when the test properties do not hold.

When the first property fails In this example, the testing system will be modified (and not the SUT).

If the 4th line of the `client_disconnect` function is modified by writing `finishedx` instead of `finished`:

```
client_disconnect(_Client_id,Server,Main) ->
    Server ! {self(),disconnect},
    Main ! {self(),finished},
    ev({client,finishedx}).
```

then a possible testing output is the following:

```
> qc:quickcheck(1000, 1000, chat_test:prop_1()).
.....................................................
................................,
Falsifiable, after 92 successful tests:
[[],[]]
[{event,<0.15159.0>,{tester,started}},
 {wait,2},
 {event,<0.15161.0>,{client,started}},
 {event,<0.15162.0>,{client,started}},
 {event,<0.15161.0>,{client,connected}},
 {wait,1},
 {event,<0.15162.0>,{client,connected}},
 {wait,1},
 {wait,1},
 {event,<0.15159.0>,{tester,every_client_connected}},
 {event,<0.15162.0>,{2,all_messages_sent}},
 {event,<0.15161.0>,{1,all_messages_sent}},
 {wait,1},
 {event,<0.15159.0>,{tester,every_client_sent_all_messages}},
 {wait,2},
 {event,<0.15162.0>,{2,received_end_of_test}},
 {event,<0.15161.0>,{1,received_end_of_test}},
 {event,<0.15161.0>,{client,finishedx}},
 {wait,1},
 {event,<0.15162.0>,{client,finishedx}},
 {event,<0.15159.0>,{tester,finished}},
 {event,<0.15159.0>,end_of_test},
 {wait,1},
 {exit,<0.15162.0>,normal},
 {exit,<0.15161.0>,normal},
 {exit,<0.15159.0>,normal},
 {wait,3},
 timeout]
```

It shows that the property failed. The output contains an incorrect trace.

When the second property fails In this example, the SUT will be modified. The current implementation of the `{Client,Msg}` branch of the server's loop is:

```
{Client,Msg} ->
    case Client of
        Client1 ->
            case Client2 of
                null -> ok;
                _ -> Client2 ! {self(),msg,Msg}
```

```
            end,
            loop(Mem);
        Client2 ->
            case Client1 of
                null -> ok;
                _ -> Client1 ! {self(),msg,Msg}
            end,
            loop(Mem);
        Other ->
            io:format("chat_server: unexpected arg: ~w~n",[Other]),
            loop(Mem)
    end;
```

It can be modified to the following, incorrect implementation:

```
{_Client,Msg} ->
    case Client2 of
        null -> ok;
        _ -> Client2 ! {self(),msg,Msg}
    end,
    case Client1 of
        null -> ok;
        _ -> Client1 ! {self(),msg,Msg}
    end,
    loop(Mem);
```

It is incorrect, because when a client sends a message, the server will deliver it not only to the other client, but to the sender, as well.

If this `chat_server` is tested, a possible result is the following:

```
4> qc:quickcheck(1000, 1000, chat_test:prop_2()).
.....................................................
.....................................,
Falsifiable, after 97 successful tests:
[[],["n"]]
{{2,"n"},{2,"n"},2}
[{event,<0.597.0>,{tester,started}},
 {event,<0.599.0>,{client,started}},
 {event,<0.600.0>,{client,started}},
 {event,<0.599.0>,{client,connected}},
 {event,<0.600.0>,{client,connected}},
 {event,<0.597.0>,{tester,every_client_connected}},
 {event,<0.600.0>,{2,started,"n"}},
 {event,<0.599.0>,{1,all_messages_sent}},
 {wait,1},
 {event,<0.600.0>,{2,finished,"n"}},
 {event,<0.599.0>,{1,received,"n"}},
 {wait,6},
 {event,<0.600.0>,{2,received,"n"}},
 {wait,1},
 {event,<0.600.0>,{2,all_messages_sent}},
 {wait,2},
 {event,<0.597.0>,{tester,every_client_sent_all_messages}},
 {event,<0.600.0>,{2,received_end_of_test}},
 {event,<0.599.0>,{1,received_end_of_test}},
 {event,<0.599.0>,{client,finished}},
 {wait,2},
 {event,<0.600.0>,{client,finished}},
 {event,<0.597.0>,{tester,finished}},
 {event,<0.597.0>,end_of_test},
```

```
{exit,<0.599.0>,normal},
{exit,<0.600.0>,normal},
more]
```

It shows that the property failed. The output contains an incorrect trace.

4.3 Testing with QuickCheck2

4.3.1 Module `chat_test`

The tester program is only one module here (the `chat_test` module). The module contains the abstract state machine, one property, and the operations, which implements the connection between the testing engine and the SUT.

4.3.2 Running the tests

The usual `eqc:quickcheck` function can be used to test the property. Now the `eqc:numtests` function will be used, as well, with which the tester can set the number of test cases. Now 1000 test cases will be generated and examined.

The output of the testing is:

```
> eqc:quickcheck(eqc:numtests(1000,chat_test:prop())).
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................................
.....................................
OK, passed 1000 tests
true
```

Error in the system Actually, QuickCheck2 managed to find a real bug in the software.

This was the original implementation of the `{Client,Msg}` branch of the server's `receive`:

```
{Client,Msg} ->
    case Client of
        Client1 ->
            Client2 ! {self(),msg,Msg}
            loop(Mem);
        Client2 ->
            Client1 ! {self(),msg,Msg}
            loop(Mem);
        Other ->
            io:format("chat_server: unexpected arg: ~w~n",[Other]),
            loop(Mem)
    end;
```

The problem is that when one of the clients is `null` (or both), the server still tries to send them the message.

When QuickCheck2 is run with that version, the following is a possible output:

```
> eqc:quickcheck(eqc:numtests(1000,chat_test:prop())).

=ERROR REPORT==== 31-Jan-2008::12:41:36 ===
Error in process <0.17760.5> with exit value:
{{case_clause,<0.17771.5>},[{chat_server,loop,1}]}

timeout at client_loop/1
Failed! After 1 tests.
[{set,{var,1},{call,chat_test,start_server,[]}},
 {set,{var,2},{call,chat_test,connect,[{var,1}]}},
 {set,{var,3},{call,chat_test,send,[{var,2},message]}},
 {set,{var,4},{call,chat_test,send,[{var,2},message]}},
 {set,{var,5},{call,chat_test,disconnect,[{var,2}]}},
 {set,{var,6},{call,chat_test,connect,[{var,1}]}},
 {set,{var,7},{call,chat_test,send,[{var,6},message]}},
 {set,{var,8},{call,chat_test,connect,[{var,1}]}},
 {set,{var,9},{call,chat_test,send,[{var,6},message]}},
 {set,{var,10},{call,chat_test,client_receive,[{var,8},message]}},
 {set,{var,11},{call,chat_test,disconnect,[{var,8}]}},
 {set,{var,12},{call,chat_test,nop,[]}},
 {set,{var,13},{call,chat_test,disconnect,[{var,6}]}},
 {set,{var,14},{call,chat_test,nop,[]}},
 {set,{var,15},{call,chat_test,stop_server,[{var,1}]}}]

=ERROR REPORT==== 31-Jan-2008::12:41:38 ===
Error in process <0.17809.5> with exit value:
{{case_clause,<0.17820.5>},[{chat_server,loop,1}]}

timeout at client_loop/1
Shrinking.(1 times)
[{set,{var,1},{call,chat_test,start_server,[]}},
 {set,{var,2},{call,chat_test,connect,[{var,1}]}},
 {set,{var,3},{call,chat_test,send,[{var,2},message]}}]
false
```

It shows that QuickCheck2 found an error, and it gives two command sequences, with which the error occurred. The first command sequence is a long one, it was the first one with which the SUT behaved erroneously. The second sequence is a shorter one, this is the result of the shrinking mechanism of QuickCheck.

5 Conclusion

In this dissertation we show how the two QuickChecks work and how they can be used. They test pure functions similarly: the tester writes properties about the functions, and QuickCheck generates random data to test whether these properties are actually true. There are differences here between the two QuickChecks, but not fundamental ones; the most notable is that QuickCheck2 (the commercial version of QuickCheck) has a 'shrinking' feature, which means that it can simplify the counterexamples.

Testing systems with impure functions is very different in the two tools. QuickCheck1 (the open source version of QuickCheck) does trace generation and trace analysis. The tester has to modify the source code of the SUT: he has to place `event` function calls in it. Then, QuickCheck1 runs the SUT and observes the events that happened (this is *trace generation*). The trace (the list of events) is analysed by the property, which was written by the tester (this is *trace analysis*).

There are problems both in trace generation and trace analysis. The main problems with the trace generator of QuickCheck1 are that the source code of the SUT has to be modified, and various tricks has to be used to ensure that the testing will work properly. The system has to be implemented with QuickCheck1 in the mind. An additional problem is that there is no support in QuickCheck1 for controlling the input of the SUT, so there is no support for generating appropriate test cases. (The tester has to write the test case generator himself, if he does not want to test any random input; and he has to solve how to give this input to the system.) These problems could be solved by a much more sophisticated implementation of the test generator, which can get the events from the SUT without making the tester modify it.

The problem of trace analysis is that there is not an appropriate way to describe properties of traces. There are basically two ways: the tester can write either Erlang functions or LTL expressions, which analyse the traces. The former is not a high level description, the tester probably has to use, e.g., recursion. The problem of the latter is that the version of LTL implemented is not powerful enough for describing properties, since it does not have quantifiers and non-boolean variables. This problem could be solved by making it possible to reason about the trace using another tool, which is high level and powerful at the same time.

QuickCheck2 does not do trace generation and trace analysis to test systems with impure functions. Instead, the tester specifies an ASM (abstract state machine), and QuickCheck2 tests whether the implementation conforms to the ASM. The word *conform* means that if the system is used correctly, it will work correctly. It does not test that the SUT cannot be used differently than specified in the ASM. QuickCheck2 knows how the SUT should work (using the ASM), so it can generate appropriate test cases. (QuickCheck1 has no idea how it works – it just starts it and records the events.)

In my experience, QuickCheck2's testing method is more appropriate for testing typical Erlang programs (servers, protocol implementations) than QuickCheck1. This is because in QuickCheck1, if a server is wanted to be tested, the tester has to write a client simulation which can do everything that a client can do, and write a property, which analyses a complete trace. On the other hand, in QuickCheck2, the tester only has to write simple clients and specify an ASM, which models the SUT – probably the abstraction level of the ASM is much closer to the specification level of the SUT than a client simulation and a trace analysis written in Erlang. Then, QuickCheck2 generates test cases from the ASM, which suit to the expected behaviour of the SUT.

Another huge advantage of QuickCheck2 over QuickCheck1 is that the source code of the SUT does not have to be modified - actually, the source code is not needed at all, if the ASM can be constructed from the specification of the SUT.

The weakness of the method of QuickCheck2 is that the SUT cannot be nondeterministic, since the testing engine first examines the SUT and generates the whole command sequence, which should be executed in the SUT; just after that it is really executed, so there is no feedback from the SUT to the test cases. An alternative ASM testing engine is presented in this paper, which does not have this problem, but unfortunately it has other problems. But despite those, it may be built into QuickCheck2.

As a final conclusion, even though there are possible improvements on QuickCheck2, it seems to be a good choice for testing software systems.

A Implementation of asm_tester

```
-module(asm_tester).
-export([test/1,test/2,frequency/1]).

test(ASM_module) ->
    test(ASM_module,100).

test(ASM_module,I) ->
    {A,B,C} = now(),
    random:seed(A,B,C),
    test_(ASM_module,I).

test_(_ASM_module,0) ->
    io:format("~nAll tests passed.~n",[]);
test_(ASM_module,I) ->
    case test_once(ASM_module) of
        passed ->
            io:format(".",[]),
            test_(ASM_module,I-1);
        {failed,postcondition,Commands} ->
            io:format("~nTest failed: postcondition was false"
                      ".~nCommands:~n~w~n",[Commands])
    end.

test_once(ASM_module) ->
    test_once(apply(ASM_module,initial_state,[]),[],ASM_module).

test_once(State,Commands,ASM_module) ->
    case Command = apply(ASM_module,command,[State]) of
        stop ->
            passed;
        {call,Module,Function,Args} ->
            case apply(ASM_module,precondition,[State,Command]) of
                false ->
                    test_once(State,Commands,ASM_module);
                true ->
                    Commands_2 = [Command|Commands],
                    Result = apply(Module,Function,Args),
                    case apply(ASM_module,postcondition,[State,Command,Result])
                        of
                        false ->
                            {failed,postcondition,lists:reverse(Commands_2)};
                        true ->
                            State_2 = apply(ASM_module,
                                next_state,[State,Result,Command]),
                            test_once(State_2,Commands_2,ASM_module)
                    end
            end
    end.

frequency(List) ->
    Weight_sum = lists:foldl(
        fun({Weight,_},Weight_sum) ->
                Weight_sum + Weight
        end,
        0,
        List
    ),
```

```
    N = random:uniform(Weight_sum),
    frequency_(List,N).

frequency_([{Weight,Element}|Tail],N) ->
    case (N =< Weight) of
        true ->
            Element;
        false ->
            frequency_(Tail,N-Weight)
    end.
```

B A program that shows how QuickCheck works with {var,N} tuples

The output of this program reveals that when the tester tests random tuples, he has to be aware, because QuickCheck2 replaces the {var,N} tuples by dynamic return values of the function calls.

The program is as follows:

```
-module(var_n).
-compile(export_all).
-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").

init() ->
    ok.

f(Tuple) ->
    io:format("Tuple=~w~n",[Tuple]).

initial_state() ->
    init.

precondition(_S,_C) ->
    true.

command(init) ->
    {call,?MODULE,init,[]};
command(f) ->
    {call,?MODULE,f,[{var,1}]};
command(_S) ->
    stop.

postcondition(_S,_C,_R) ->
    true.

next_state(init,_,_) ->
    f;
next_state(f,_,_) ->
    stop.

prop() ->
    ?FORALL(Cmds,commands(?MODULE),
        begin
            {_,_,Result} = run_commands(?MODULE,Cmds),
            Result == ok
        end
    ).
```

The output of the testing is as follows:

```
> eqc:quickcheck(var_n:prop()).
Tuple=ok
.Tuple=ok
.Tuple=ok
.Tuple=ok
.Tuple=ok
.Tuple=ok
[...]
.Tuple=ok
.Tuple=ok
.Tuple=ok
.Tuple=ok
.Tuple=ok
.
OK, passed 100 tests
```

C chat_server example

C.1 Module chat_server

```
-module(chat_server).
-export([start_link/0]).

start_link()->
    spawn_link(fun init/0).

init() ->
    loop({null,null}).

loop(Mem={Client1,Client2}) ->
    receive
        {Pid,stop} ->
            case Client1 of
                null -> ok;
                _ -> Client1 ! {self(),server_stopped}
            end,
            case Client2 of
                null -> ok;
                _ -> Client2 ! {self(),server_stopped}
            end,
            Pid ! {self(),ok},
            ok;
        {Client,connect} ->
            case {Client1,Client2} of
                {null,_} ->
                    Client ! {self(),ok},
                    loop({Client,Client2});
                {_,null} ->
                    Client ! {self(),ok},
                    loop({Client1,Client});
                _ ->
                    Client ! {self(),server_is_full},
                    loop(Mem)
            end;
        {Client,disconnect} ->
            case Client of
                Client1 ->
                    loop({null,Client2});
```

```
                Client2 ->
                    loop({Client1,null});
                Other ->
                    io:format("chat_server: unexpected arg:  ~w~n",[Other]),
                    loop(Mem)
            end;
        {Client,number_of_clients} ->
            Client ! {
                self(),
                case {Client1,Client2} of
                    {null,null} -> 0;
                    {null,_} -> 1;
                    {_,null} -> 1;
                    _ -> 2
                end
            },
            loop(Mem);
        {Client,Msg} ->
            case Client of
                Client1 ->
                    case Client2 of
                        null -> ok;
                        _ -> Client2 ! {self(),msg,Msg}
                    end,
                    loop(Mem);
                Client2 ->
                    case Client1 of
                        null -> ok;
                        _ -> Client1 ! {self(),msg,Msg}
                    end,
                    loop(Mem);
                Other ->
                    io:format("chat_server: unexpected arg: ~w~n",[Other]),
                    loop(Mem)
            end;
        Other ->
            io:format("chat_server: unexpected arg:  ~w~n",[Other]),
            loop(Mem)
    end;
loop(Other) ->
    io:format("chat_server:loop: unexpected arg:  ~w~n",[Other]).
```

C.2 Testing with QuickCheck1

C.2.1 Module `my_ev`

```
-module(my_ev).
-export([ev/1]).

ev(E) ->
    my_qc:ev(E,on_show_only_end_of_test).
```

C.2.2 Header `my_qc.hrl`

```
-define(MY_MATCHES(X),
    fun(T3) ->
        case T3 of
            [X|_] -> true;
            _ -> false
        end
```

```
    end
).
-define(M(Event),?MY_MATCHES({event,_,Event})).
```

C.2.3 Module my_qc

```
-module(my_ev).
-export([ev/1]).

ev(E) ->
    my_qc:ev(E,on_show_only_end_of_test).
```

C.2.4 Module chat_test

```
-module(chat_test).
-include("quickcheck.hrl").
-include("my_qc.hrl").
-import(my_ev,[ev/1]).
-import(my_qc,[order/2,tmin/2,tmax/2,tnumber/2,tand/1,
               finished/0,iffinished/1]).
-export([prop_1/0,prop_2/0]).

%%%%% properties %%%%%

% Property 1:
% "Every process should be finished, which was started."
prop_1() ->
    ?FORALL(Message_table,generate_message_table(),
        ?FORALL(T, ?TRACE(3,generate_trace(Message_table)),
            ?FORALL(Pid, gen_random_pid(T),
                satisfies(T,
                    data_prop_1(Pid)
                )
            ))).

% Generates a random pid from the given trace.
gen_random_pid(T) ->
    fun(_Size) ->
            L = lists:filter(
                fun (X) ->
                        case X of
                            {_,_,{_,started}} -> true;
                            {_,_,{_,finished}} -> true;
                            _ -> false
                        end
                end,
                T
            ),
            case length(L) of
                LL when LL > 0 ->
                    R = random:uniform(LL),
                    {_,Pid,{_,_}} = lists:nth(R,L),
                    Pid;
                _ ->
                    0
            end
    end.

% The LTL formula of property 1 for a given pid.
```

```
data_prop_1(Pid) ->
    my_qc:iffinished(
        timplies(
            eventually(?MY_MATCHES({event,Pid,{_,started}})),
            eventually(?MY_MATCHES({event,Pid,{_,finished}}))
        )
    ).

% Property 2:
%
% Let Sender1,Msg1,Sender2,Msg2 and Client be such that
% Sender1, Sender2 and Client are clients,
% Sender1 sent Msg1, and Sender2 sent message Msg2.
%
% data_prop_2(Sender1,Msg1,Sender2,Msg2,Client) has to be
% true in this case.

prop_2() ->
    ?FORALL(Message_table,generate_message_table(),
        case length(lists:flatten(Message_table)) > 0 of
            true ->
                ?FORALL(
                    {{Sender1,Msg1},{Sender2,Msg2},Client},
                    {generate_message(Message_table),
                     generate_message(Message_table),
                     generate_client(Message_table)},
                    ?FORALL(T,
                        ?TRACE(3,generate_trace(Message_table)),
                        satisfies(T,
                            data_prop_2(Sender1,Msg1,Sender2,Msg2,Client)
                        )
                    ));
            _ ->
                true
        end
    ).

% This LTL formula states the following about the
% Sender1,Msg1,Sender2,Msg2,Client variables:
%
% If Client is different from both Senders, it should
% receive both Msgs once.
% If Client is equal to one of the Senders, but
% different from the other, it should receive only the
% Msg which was sent by the different one.
% If Client if equal to both senders, it should not receive
% either Msg.
% AND:
% If the first message was posted earlier than the second
% one, i.e. Sender1 finished sending its message before
% Sender2 started it, and the Client got both messages, then
% it got the first message before the second one.
%
% Note: all msgs have to be different, otherwise the
% property does not hold.
% The message function generator of this module works in
% such a way, that it generates only such messages tables,
% in which all the values are unique.
data_prop_2(Sender1,Msg1,Sender2,Msg2,Client) ->
```

```
    my_qc:iffinished(
        tand([
            % if Client is different from both Sender, it
            % gets both messages
            timplies(
                fun (_) -> ((Client /= Sender1) and (Client /= Sender2)) end,
                tand(
                    my_qc:once(?M({Client,received,Msg1})),
                    my_qc:once(?M({Client,received,Msg2}))
            ),
            % if Client is the first sender, it gets only
            % the second message
            timplies(
                fun (_) -> ((Client == Sender1) and (Client /= Sender2)) end,
                tand(
                    my_qc:never(?M({Client,received,Msg1})),
                    my_qc:once(?M({Client,received,Msg2}))
            ),
            % if Client is the second sender, it gets only
            % the first message
            timplies(
                fun (_) -> ((Client /= Sender1) and (Client == Sender2)) end,
                tand(
                    my_qc:once(?M({Client,received,Msg1})),
                    my_qc:never(?M({Client,received,Msg2}))
            ),
            % if Client is both sender, it does not get
            % either of the messages
            timplies(
                fun (_) -> ((Client == Sender1) and (Client == Sender2)) end,
                tand(
                    my_qc:never(?M({Client,received,Msg1})),
                    my_qc:never(?M({Client,received,Msg2}))
            ),
            timplies(
                % {Sender1,Msg1} was sent before {Sender2,Msg2}
                my_qc:order(
                    ?M({Sender1,finished,Msg1}),
                    ?M({Sender2,started,Msg2})
                ),
                % Client got the messages in the correct order
                timplies(
                    fun (_) ->
                        ((Client /= Sender1) and (Client /= Sender2))
                    end,
                    my_qc:order(
                        ?M({Client,received,Msg1}),
                        ?M({Client,received,Msg2})
                    )
                )
            )
            ])
    ).

%%%%% generators %%%%%

% Generates a random character.
char() ->
    fun(_Size) ->
```

```
        random:uniform(26)+96
    end.

% Returns true iff every element of the list is unique.
all_unique(List) ->
    all_unique(List,sets:new()).

% Returns true iff
% every element of the list is unique and is not in the set;
% so it is not in the list elsewhere nor in the set.
all_unique([],_Set) ->
    true;
all_unique([Head|Tail],Set) ->
    case sets:is_element(Head,Set) of
        false ->
            all_unique(Tail,sets:add_element(Head,Set));
        true ->
            false
    end.

% Generates a message table, where all the messages are
% unique.
generate_nonempty_message_table(Size) ->
    L1 = (list(list(char())))(Size),
    L2 = (list(list(char())))(Size),
    case all_unique(L1++L2) of
        true ->
            [L1,L2];
        false ->
            generate_nonempty_message_table(Size)
    end.

% Generates a message table.
generate_message_table() ->
    fun(Size) ->
        case Size of
            0 -> [];
            _ -> generate_nonempty_message_table(5)
        end
    end.

% Random element of a list.
random_element(L) ->
    lists:nth(random:uniform(length(L)),L).

% Generate a random message from the message table.
generate_message(Message_table) ->
    fun(_) ->
        case length(Message_table) > 0 of
            true ->
                Sender = random:uniform(length(Message_table)),
                Message_list = lists:nth(Sender,Message_table),
                case length(Message_list) > 0 of
                    true ->
                        {Sender,random_element(Message_list)};
                    false ->
                        (generate_message(Message_table))(0)
                end;
            _ ->
```

```
                null
        end
    end.

generate_client(Message_table) ->
    fun(_) ->
        random:uniform(length(Message_table))
    end.

%%%%% trace generator %%%%%

generate_trace(Message_table) ->
    ev({tester,started}),
    Client_no = length(Message_table),
    Main = self(),
    Server = chat_server:start_link(),

    % spawn the clients
    Clients = spawn_clients(Client_no,Message_table,Main,Server),

    % receive clients' 'connected'
    receive_message_from_clients(Client_no,connected),
    ev({tester,every_client_connected}),

    % send 'start_talking' to the clients
    lists:foreach(
        fun(Client) ->
                Client ! {self(),start_talking}
        end,
        Clients
    ),

    % receive clients' 'all_messages_sent'
    receive_message_from_clients(Client_no,all_messages_sent),
    ev({tester,every_client_sent_all_messages}),

    % quit the server and receive its finished
    Server ! {self(),stop},
    receive {Server,ok} -> ok end,

    % send 'end_of_test' to the clients
    lists:foreach(
        fun(Client) ->
                Client ! {self(),end_of_test}
        end,
        Clients
    ),

    % receive clients' 'finished'
    receive_message_from_clients(Client_no,finished),

    ev({tester,finished}),
    ev(end_of_test).

% Spawn all the clients.
spawn_clients(Client_no,Message_table,Main,Server) ->
    spawn_clients(0,Message_table,[],Client_no,Main,Server).

spawn_clients(Client_no,_,Clients,Client_no,_,_) ->
```

```
    Clients;
spawn_clients(I,[Message_list|Message_table_tail],Clients,Client_no,Main,
    Server) ->
    Client = spawn_link(
        fun() -> client(I+1,Server,Main,Message_list) end
    ),
    spawn_clients(I+1,Message_table_tail,[Client|Clients],Client_no,Main,
        Server).

% Receives the given message from the given count of
% clients.
receive_message_from_clients(0,_Message) ->
    ok;
receive_message_from_clients(Client_no,Message) ->
    receive
        {_Client,Message} -> ok
    after
        5000 -> timeout()
    end,
    receive_message_from_clients(Client_no-1,Message).

%%%%% client simulation %%%%%

% Starts a client.
client(Client_id,Server,Main,Message_list) ->
    ev({client,started}),
    Server ! {self(),connect},
    receive {Server,ok} -> ok after 1000 -> timeout() end,
    ev({client,connected}),
    Main ! {self(),connected},
    receive {Main,start_talking} -> ok after 1000 -> timeout() end,
    client_loop(Client_id,Server,Main,Message_list).

% The loop of the client which sends the messages.
client_loop(Client_id,Server,Main,Message_list) ->
    client_read_loop(Client_id,Server,Main),
    case Message_list of
        [Message|Message_list_tail] ->
            ev({Client_id,started,Message}),
            Server ! {self(),Message},
            ev({Client_id,finished,Message}),
            client_loop(Client_id,Server,Main,Message_list_tail);
        _ ->
            ev({Client_id,all_messages_sent}),
            Main ! {self(),all_messages_sent},
            client_read_all_loop(Client_id,Server,Main)
    end.

% The loop of the client which receives the messages.
client_read_loop(Client_id,Server,Main) ->
    case random:uniform(2) of
        1 ->
            receive
                {Server,msg,Message} ->
                    ev({Client_id,received,Message}),
                    client_read_loop(Client_id,Server,Main)
            after
                0 -> ok
            end;
```

```
        2 -> ok
    end.

% The client reads all the messages.
client_read_all_loop(Client_id,Server,Main) ->
    receive
        {Server,msg,Message} ->
            ev({Client_id,received,Message}),
            client_read_all_loop(Client_id,Server,Main);
        {Main,end_of_test} ->
            ev({Client_id,received_end_of_test}),
            client_disconnect(Client_id,Server,Main)
    after
        5000 -> timeout()
    end.

% The client disconnects.
client_disconnect(_Client_id,Server,Main) ->
    Server ! {self(),disconnect},
    Main ! {self(),finished},
    ev({client,finished}).

%%%%% misc %%%%%

timeout() ->
    ev(timeout),
    erlang:error(timeout).
```

C.3 Testing with QuickCheck2

C.3.1 Module chat_test

```
-module(chat_test).
-compile(export_all).
-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").
-record(state,{
        main_state,
        server,
        number_of_clients = 0,
        clients = [],
        max_number_of_clients = 2,
        messages_queue = queue:new(),
        commands = 0,
        max_commands = 10
    }).

% c(chat_test).
% eqc:quickcheck(eqc:numtests(1000,chat_test:prop())).

%%%%% operations %%%%%

client(Spawner,Server) ->
    Server ! {self(),connect},
    receive
        {Server,ok} ->
            Spawner ! {self(),ok},
            client_loop(Server)
    after
```

```
        2000 -> io:format("timeout at client/2~n",[])
    end.

client_loop(Server) ->
    receive
        {Pid,disconnect} ->
            Server ! {self(),disconnect},
            Pid ! {self(),ok};
        {Pid,send,Message} ->
            Server ! {self(),Message},
            Pid ! {self(),ok},
            client_loop(Server);
        {Pid,recv,Timeout} ->
            receive
                {Server,msg,Message} ->
                    Pid ! {self(),ok,Message}
            after
                Timeout ->
                    Pid ! {self(),timeout}
            end,
            client_loop(Server)
    after
        2000 ->  io:format("timeout at client_loop/1~n",[])
    end.

connect(Server) ->
    Spawner = self(),
    Client = spawn(fun() -> client(Spawner,Server) end),
    receive
        {Client,ok} -> Client
    after
        2000 -> {error,timeout}
    end.

disconnect(Client) ->
    Client ! {self(),disconnect},
    receive
        {Client,ok} -> ok
    after
        2000 -> {error,{timeout,disconnect}}
    end.

send(Client,Message) ->
    Client ! {self(),send,Message},
    receive
        {Client,ok} ->
            ok
    after
        2000 -> {error,{timeout,send}}
    end.

client_receive(Client,Message) ->
    Client ! {self(),recv,1000},
    receive
        {Client,ok,Message_from_client} ->
            case Message == Message_from_client of
                true ->
                    ok;
                false ->
```

```
                    {error,message_not_ok}
            end;
        {Client,timeout} ->
            {error,{message_not_ok,timeout}}
    after
        2000 -> {error,{timeout,client_receive}}
    end.


start_server() ->
    %io:format("start_server~n",[]),
    chat_server:start_link().

stop_server(Server) ->
    Server ! {self(),stop},
    receive
        {Server,ok} -> ok
    after
        2000 -> {error,timeout}
    end.

number_of_clients(Server) ->
    Server ! {self(),number_of_clients},
    receive
        {Server,Number_of_clients} -> Number_of_clients
    after
        2000 -> {error,timeout}
    end.

nop() ->
    ok.

%%%%% ASM callback functions %%%%%

initial_state() ->
    #state{main_state=init}.

precondition(_S,_C) ->
    true.

command(State = #state{
        main_state = Main_state,
        server = Server,
        number_of_clients = Number_of_clients,
        clients = Clients,
        max_number_of_clients = Max_number_of_clients,
        messages_queue = Message_queue,
        commands = _Commands,
        max_commands = Max_commands
    }) ->

    case State of

        #state{main_state = init} ->
            {call,?MODULE,start_server,[]};

        #state{main_state = stopping, number_of_clients = 0} ->
            {call,?MODULE,nop,[]};
```

```
        #state{main_state = normal, commands = Max_commands,
            max_commands = Max_commands} ->
            {call,?MODULE,nop,[]};

        #state{main_state = stop_server, server = Server} ->
            {call,?MODULE,stop_server,[Server]};

        #state{main_state = stop} ->
            stop;

        _Any when ((Main_state == normal) or (Main_state == stopping)) ->

            Commands_to_do =
            % connect
            case (Number_of_clients < Max_number_of_clients) and
                (Main_state == normal) of
                true ->
                    [{1,{call,?MODULE,connect,[Server]}}];
                false ->
                    []
            end
            ++
            % disconnect
            case Number_of_clients > 0 of
                true ->
                    Client = lists:nth(
                        random:uniform(length(Clients)),Clients),
                    [{1,{call,?MODULE,disconnect,[Client]}}];
                false ->
                    []
            end
            ++
            % send
            case (Number_of_clients > 0) and (Main_state == normal) of
                true ->
                    Client = lists:nth(
                        random:uniform(length(Clients)),Clients),
                    Message = message,
                    [{2,{call,?MODULE,send,[Client,Message]}}];
                false ->
                    []
            end
            ++
            case queue:is_empty(Message_queue) of
                false ->
                    {Message, Clients_to_send} = queue:head(Message_queue),

                    Client = lists:nth(
                        random:uniform(length(Clients_to_send)),
                        Clients_to_send),
                    [{4,{call,?MODULE,client_receive,[Client,Message]}}];
                true ->
                    []
            end,
            frequency(Commands_to_do)
    end.

postcondition(State,Command,Result) ->
    case {State,Command,Result} of
```

```
            {_,_,{error,_Error}} ->
                false; % postcondition failed: "error"
            _ ->
                case next_state(State,Result,Command) of
                    #state{main_state=stop} ->
                        true;
                    #state{server=Server,number_of_clients=Number_of_clients} ->
                        case number_of_clients(Server) == Number_of_clients of
                            false ->
                                false; % postcondition failed: the
                                       % number_of_clients at the server does
                                       % not match with the number_of_clients
                                       % at the ASM
                            true ->
                                true
                        end;
                    _ ->
                        true
                end
        end.

    % Deletes a client from a queue.
    delete_client(Client,Q) ->
        delete_client(Client,queue:new(),Q).

    % Deletes a client from queue Q and appends it to queue Acc.
    delete_client(Client,Acc,Q) ->
        case queue:is_empty(Q) of
            true ->
                Acc;
            false ->
                {{value,{Message,Clients_to_send_1}},Q_tail} = queue:out(Q),
                case lists:delete(Client,Clients_to_send_1) of
                    [] ->
                        delete_client(Client,Acc,Q_tail);
                    Clients_to_send_2 ->
                        Acc_2 = queue:in({Message,Clients_to_send_2},Acc),
                        delete_client(Client,Acc_2,Q_tail)
                end
        end.

    next_state(
        State = #state{
            main_state = _Main_state,
            server = _Server,
            number_of_clients = Number_of_clients,
            clients = Clients,
            max_number_of_clients = _Max_number_of_clients,
            messages_queue = Message_queue,
            commands = Commands,
            max_commands = Max_commands
        },
        Result,
        Call) ->

        case {State,Result,Call} of

            {#state{main_state = init},New_server,_} ->
                State#state{
```

```
            main_state = normal,
            server = New_server
        };

    {#state{main_state = stopping, number_of_clients = 0},_,_} ->
        State#state{
            main_state = stop_server
        };

    {#state{main_state = normal, commands = Max_commands,
        max_commands = Max_commands},_,_} ->
        State#state{
            main_state = stopping
        };

    {_,Client,{call,_,connect,_}} ->
        State#state{
            number_of_clients = Number_of_clients + 1,
            clients = [Client|Clients],
            commands = Commands + 1
        };

    {_,_,{call,_,disconnect,[Client]}} ->
        State#state{
            number_of_clients = Number_of_clients - 1,
            clients = lists:delete(Client,Clients),
            messages_queue = delete_client(Client,Message_queue),
            commands = Commands + 1
        };

    {_,_,{call,_,send,[Client,Message]}} ->
        case Clients of
            [Client] ->
                State#state{
                    commands = Commands + 1
                };
            _ ->
                State#state{
                    messages_queue = queue:in({Message,
                            lists:delete(Client,Clients)},Message_queue),
                    commands = Commands + 1
                }
        end;

    {_,_,{call,_,client_receive,[Client,_]}} ->

        {{value,{Message,Clients_to_send}},Other_messages_to_send} =
            queue:out(Message_queue),

        Message_queue_2 =
        case lists:delete(Client,Clients_to_send) of
            [] ->
                Other_messages_to_send;
            Clients_to_send_2  ->
                queue:cons({Message,Clients_to_send_2},
                    Other_messages_to_send)
        end,

        State#state{
```

```
                messages_queue = Message_queue_2,
                commands = Commands + 1
            };

        {_,_,{call,_,stop_server,_}} ->
            State#state{
                main_state = stop
            }
    end.

%%%%% the property %%%%%

prop() ->
    ?FORALL(Cmds,commands(?MODULE),
        begin
            {_H,_S,Result} = run_commands(?MODULE,Cmds),
            Result == ok
        end
    ).
```

References

[1] Joe Armstrong. Concurrency Oriented programming in Erlang. *Lightweight Languages Workshop 2002 (LL2)*

[2] Joe Armstrong. Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf, 2007. ISBN 193435600X

[3] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*, pages 268-279, 2000.

[4] Zubair Sheikh. Transfer Report Towards Testing for Erlang Code. 2007.

[5] Thomas Arts, John Hughes, Joakim Johansson and Ulf Wiger. Testing Telecoms Software with Quviq QuickCheck. Phil Trinder (editor), *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, 2006.

[6] Scheme-Check: Randomized Unit Testing for PLT Scheme. `http://web.archive.org/web/20050212183945sh_re_/www.inf.ufrgs.br/~carlossch/scheme-check/`

[7] Clickcheck and Peckcheck. `http://www.accesscom.com/~darius/software/clickcheck.html`

[8] RushCheck. `http://rushcheck.rubyforge.org/`

[9] ScalaCheck. `http://code.google.com/p/scalacheck/`

[10] QCheck/SML. `http://contrapunctus.net/league/haques/qcheck/`

[11] Open-source Erlang - White Paper. `http://erlang.org/white_paper.html`