Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Αυτόματος Τυχαίος Έλεγχος Συστημάτων με Εσωτερική Κατάσταση βάσει Μοντέλου

Διπλωματική Εργασία

της

Ειρήνης Αρβανίτη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Ιούλιος 2011

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Αυτόματος Τυχαίος Έλεγχος Συστημάτων με Εσωτερική Κατάσταση βάσει Μοντέλου

Διπλωματική Εργασία

της

Ειρήνης Αρβανίτη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18$^{\eta}$ Ιουλίου, 2011.

....	....	....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Κώστας Κοντογιάννης
Αν. Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2011

....
Ειρήνη Αρβανίτη
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Το PropEr είναι ένα εργαλείο ανοιχτού λογισμικού για τον αυτόματο τυχαίο έλεγχο ιδιοτήτων συναρτήσεων, γραμμένων στη γλώσσα Erlang, από τις προδιαγραφές τους. Αρχικά ήταν επικεντρωμένο στον έλεγχο αγνών συναρτήσεων. Ωστόσο, οι εφαρμογές που υλοποιούνται σε Erlang συνήθως αποτελούνται από κώδικα με εσωτερική κατάσταση. Στην παρούσα διπλωματική εργασία, έχουμε επεκτείνει το PropEr με δύο βιβλιοθήκες που υποστηρίζουν τον τυχαίο ελέγχο συστημάτων με εσωτερική κατάσταση βάσει μοντέλου. Ο χρήστης καλείται να προσδιορίσει ένα μοντέλο της συμπεριφοράς του συστήματος υπό έλεγχο. Δεδομένου αυτού του μοντέλου, ελέγχουμε ένα σύστημα παράγοντας και εκτελώντας ακολουθίες κλήσεων προς αυτό, ενώ καταγράφουμε τις αποκρίσεις του ώστε να επιβεβαιώσουμε ότι το σύστημα παρουσιάζει την αναμενόμενη συμπεριφορά.

Η πρώτη βιβλιοθήκη, που ονομάζεται proper_statem, έχει σχεδιαστεί για τον έλεγχο γενικευμένων εξυπηρετητών και άλλων συστημάτων των οποίων η διεπαφή παρουσιάζει εσωτερική κατάσταση. Οι παρενέργειες των συστημάτων προσδιορίζονται μέσω μίας αφηρημένης μηχανής κατάστασης. Η ίδια μηχανή κατάστασης μπορεί επίσης να χρησιμοποιηθεί για την παραγωγή ακολουθιών κλήσεων, οι οποίες θα εκτελεστούν παράλληλα για τον εντοπισμό συνθηκών ανταγωνισμού. Η δεύτερη βιβλιοθήκη, που ονομάζεται proper_fsm, προσφέρεται για τον έλεγχο συστημάτων που παρουσιάζουν συμπεριφορά μηχανής πεπερασμένης κατάστασης, αφού είναι σχεδιασμένη ώστε να φέρνει την περιγραφή του μοντέλου του συστήματος πολύ κοντά σε ένα διάγραμμα κατάστασης. Ο προσδιορισμός ενός μοντέλου για το σύστημα υπό έλεγχο δεν είναι σε καμία περίπτωση τετριμμένη διαδικασία. Γι'αυτό το λόγο, παρουσιάζουμε λεπτομερή επεξηγηματικά παραδείγματα σχετικά με την αποτελεσματική χρήση των νέων βιβλιοθηκών για τον έλεγχο συστημάτων με εσωτερική κατάσταση.

Λέξεις Κλειδιά

έλεγχος λογισμικού βάσει μοντέλου, έλεγχος λογισμικού βάσει ιδιοτήτων, αυτόματος έλεγχος, τυχαίος έλεγχος, παράλληλος έλεγχος, προδιαγραφές μηχανής κατάστασης

Abstract

PropEr is an open-source tool for automated random testing of Erlang function properties from specifications. Its original focus was on testing pure functions. Nevertheless, projects implemented in Erlang typically consist of stateful code. In this thesis, we have extended PropEr with two library modules that support random model-based testing of stateful reactive systems. The model behaviour of the system under test should be defined in a callback module. Given this model, we test a stateful system by generating and performing sequences of calls to that system, while monitoring its actual responses to ensure the system behaves as expected.

The first module, proper_statem, is designed to test generic servers and other systems with stateful interfaces, whose side-effects are specified via an abstract state machine. The same state machine specification can also be used for generating sequences of calls that will be executed in parallel to test for race conditions. The second module, proper_fsm, offers a convenient way to test systems exhibiting a finite state machine behaviour, as it is designed to bring the callback module specification very close to a state diagram. Defining a model of the system under test is by no means a trivial task. To compensate for that, we present detailed tutorials on effectively using the new library modules to test stateful systems.

Keywords

model-based testing, property-based testing, automated testing, random testing, parallel testing, state machine specifications

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Κωστή Σαγώνα, για την έμπνευση του θέματος της διπλωματικής, τη διαρκή καθοδήγηση και τη συνεργασία μας κατά τη διάρκεια του τελευταίου έτους. Επιπλέον, για την εμπιστοσύνη που μου έδειξε από την πρώτη στιγμή ανάθεσης της διπλωματικής.

Θα ήθελα επίσης να ευχαριστήσω το Μανώλη Παπαδάκη, για τις πολύτιμες συμβουλές του, το χρόνο που διέθεσε και την προθυμία του να απαντήσει σε κάθε είδους απορία σχετικά με το PropEr.

Ευχαριστώ τους καθηγητές και τους φοιτητές του Εργαστηρίου Λογισμικού για τις εποικοδομητικές συζητήσεις και τη φιλική ατμόσφαιρα στο εργαστήριο. Ιδιαιτέρως τον κ. Νίκο Παπασπύρου, ο οποίος συνέβαλε στην απόφασή μου να ασχοληθώ με την Πληροφορική από το πρώτο έτος των σπουδών μου.

Τέλος, ευχαριστώ την οικογένειά μου και όσους ήταν τα τελευταία χρόνια δίπλα μου στηρίζοντας κάθε μου επιλογή.

Ειρήνη Αρβανίτη

Contents

List of Figures

List of Listings

Chapter 1

Motivation

Test-Driven Development (TDD), a practice suggesting that tests should be the driving force in software creation [4], is very popular within the Erlang community. Testing is used in all phases of software development to reveal software discrepancies, or increase confidence about their absence. TDD calls for automating the test process as much as possible, in an attempt to produce concise and reusable test suites that can accomplish a thorough testing of the system under test. Unit testing frameworks (e.g. EUnit[1] for Erlang) are commonly used towards this direction, since they can automatically execute large test suites and capture related output. Nevertheless, it is the user's manual task to provide valid test input and decide on the correctness of the output.

In the recent years, there has been a growing interest in Property-Based Testing (PBT). This is a novel approach to software testing, where the test process is driven by properties, which depict the system's intended behaviour, and generators, which produce directed random test input data. The tester only needs to specify the appropriate generators, along with a number of properties which are expected to hold for every valid input produced by the generators. A property-based testing tool, when supplied with this information, will automatically produce progressively more complex random valid inputs, then apply those inputs to the program while monitoring its execution, to ensure that it behaves according to its specification.

The first implementation of a property-based testing tool was Quickcheck [7], written for Haskell in 2000. Since then, the idea of property-based testing has spread into many programming languages.[2] The first PBT tool for Erlang is Quviq Quickcheck [3], a proprietary closed-source tool, marketed as a commercial product. More recent PBT tools include Triq [16] and PropEr [13, 14], which are released as open-source. PropEr's exclusive feature, compared to existing similar tools for Erlang, is that it offers a tight integration of the language of types and function specifications of Erlang[3] with properties.

Erlang is a programming language mainly used for developing telecom software, web services or database management systems. Such applications typically consist of reactive systems, i.e. systems that continuously interact with their environment through events. This ongoing interaction with the environment forms the *internal state* of the system. The internal state is based on previous operations and affects subsequent ones, therefore

[1]`http://erlang.org/doc/apps/eunit/users_guide.html`
[2]`http://en.wikipedia.org/wiki/QuickCheck`
[3]`http://www.erlang.org/doc/reference_manual/typespec.html`

introducing side-effects in the system's behaviour. Since PropEr's initial focus was on testing pure functions, it could not be easily used for testing code with side-effects. In order to test a stateful system, users would have to provide generators that manipulate the system's internal state. This is a non-trivial task that requires considerable effort. For this reason, we decided to equip PropEr with additional library modules so as to facilitate testing of stateful systems.

The ideas that we have implemented come from the area of Model-Based Testing (MBT), where a system is tested against an abstract model of itself. These ideas are already implemented in Quviq Quickcheck [11]. Having no access to the commercial software or its code, we have implemented from scratch a 'stateful testing' subsystem for PropEr, which can serve as a basis for further research into the areas of property-based and model-based testing. Last but not least, we have integrated PropEr's exclusive features into the new 'stateful testing' subsystem.

The rest of the thesis is organised as follows. Chapter 2 is a short introduction to the Erlang programming language and OTP behaviours. In Chapter 3 we give an overview of property-based testing with PropEr. Chapter 4 is the main chapter of the thesis, where we present our implementation of a 'stateful testing' subsystem for PropEr. In Chapter 5 there are examples on effectively using PropEr to test impure code, in the form of tutorials. Finally, in Chapter 6 we present our concluding remarks and future work.

Chapter 2

A Short Introduction to Erlang and OTP

2.1 The Erlang Programming Language

Erlang is a high-level, general-purpose programming language and runtime system with built-in support for concurrency, distribution and fault-tolerance [2, 6]. Its sequential subset is a strict, dynamically-typed functional programming language. Originally developed in the mid-1980s in Ericsson's research laboratories, Erlang was designed with a clear goal in mind: to support and facilitate the development of scalable, robust, massively concurrent, soft real-time, non-stop applications, such as telecommunication systems. As it happens that web services, commercial banking, messaging systems, and database management systems, among others, share the same requirements as telecom software, Erlang is becoming increasingly popular in these sectors as well. Since its open-source release in December 1998, Erlang has been used in several proprietary and open-source projects, including Apache CouchDB[1], Amazon SimpleDB[2], the distributed database Riak[3], the XMPP instant messaging server Ejabberd[4], the AMQP messaging protocol implementation RabbitMQ[5], and Yaws web server.[6]

Erlang's main strength lies in its lightweight, no memory-sharing concurrency model. Erlang processes are lightweight in that the Erlang VM does not create a new OS thread for every newly-spawned process. Processes are created, scheduled and handled in the VM, independently of the underlying operating system. Erlang implements the Actor Concurrency Model, where processes do not share memory, instead each process executes in its own memory space and owns its own heap and stack. Processes communicate with each other via asynchronous message passing instead of shared variables, which removes the need for locks. Incoming messages are retrieved from the process mailbox selectively via pattern matching.

[1]`http://couchdb.apache.org/`
[2]`http://aws.amazon.com/simpledb/`
[3]`http://www.basho.com/products_riak_overview.php`
[4]`http://www.ejabberd.im/`
[5]`http://www.rabbitmq.com/`
[6]`http://yaws.hyber.org/`

The Open Telecom Platform (OTP)[7] provides a framework to structure Erlang systems offering robustness and fault-tolerance together with a set of tools and libraries. In the next sections, we present in more detail the design patterns that can be followed when building software with Erlang/OTP.

2.2 OTP Design Principles

The OTP Design Principles is a set of principles for how to structure Erlang code in terms of processes, modules and directories. In OTP systems are built in the following hierarchical manner [1]:

Releases
Releases are at the top of the hierarchy. A release contains all the information necessary to build and run a system. Internally a release consists of zero or more applications.

Applications
Applications are simpler than releases, they contain all the code and operating procedures necessary to implement some specific functionality. The simplest kind of application does not have any processes, but consists of a collection of functional modules. Such an application is called a library application. An application with processes is usually implemented as a supervision tree, that is a process structuring model based on the idea of workers and supervisors.

Supervisors
OTP applications are commonly built from a number of instances of supervisors. Supervisors are processes which monitor the behaviour of workers. A supervisor can restart a worker if something goes wrong.

Workers
OTP supervisors supervise worker nodes. Workers are processes which perform computations, that is, they do the actual work.

2.3 Behaviours

In a supervision tree, many of the processes have similar structures, they follow similar programming patterns. Behaviours are formalizations of these common patterns. The idea is to divide the code for a process in a generic part (a behaviour module) and a specific part (a callback module). The behaviour module is part of Erlang/OTP. To implement a process such as a supervisor, the user only has to implement the callback module which should export a pre-defined set of functions (the callback functions) for the supervisor behaviour.

Many worker processes are servers in a server-client relation, finite state machines, or event handlers such as error loggers. These are implemented as instances of the *gen_server*, *gen_fsm* and *gen_event* behaviours respectively. In the next sections, we will present in more detail the *gen_server* and *gen_fsm* behaviours.

[7] `http://www.erlang.org/doc/`

2.3.1 A generic server behaviour

The client-server model is characterized by a central server and an arbitrary number of clients and is generally used for resource management operations, where several different clients want to share a common resource. The server is responsible for managing this resource. gen_server is a behaviour module for implementing the server of a client-server relation.

We will describe the gen_server's and the callback module's interfaces through an example: a system consisting of one master and multiple slave processes. The master process is implemented using the gen_server behaviour. The main concept is that the master exchanges ping and pong messages with all slave processes, which do not interact with each other. We can think of the slave processes as ping-pong players interacting only with their trainer, i.e. the gen_server process. External clients should be able to attach and detach player processes to and from the server. Additionaly, the server has to handle ping messages and to keep track of the scores (i.e. the number of ping-pong message exchanges) of the attached player processes.

We can start the server by calling `gen_server:start_link({local, Name}, Mod, InitArgs, Opts)`. This function starts and links to a server localy registered as `Name`. An option for globally registering the server is also available. `Mod` is the name of the callback module. `InitArgs` is a term which is passed as is to the callback function `Mod:init/1` and `Opts` is a list of additional options. Then, the first routine to be called in the callback module is `Mod:init(InitArgs)`, which must return `{ok, State}`. `State` is the internal state of the gen_server process. In our case, it is a dictionary containing the scores of attached player processes. This is specified in Listing 2.1.

```
start_link() ->
    gen_server:start_link({local, ?MASTER}, ?MASTER, [], []).

init([]) ->
    {ok, dict:new()}.
```

Listing 2.1: Starting the ping-pong master

Listing 2.2 describes how to make synchronous requests to the server and also how such requests are handled in the callback module. Synchronous requests are implemented using `gen_server:call(Name, Request)`. The request is made into a message and sent to the gen_server registered as `Name`. When the request is received, the gen_server calls `handle_call(Request, From, State)`, where `From` is the PID of the requesting client process and `State` is the current state of the gen_server. This is expected to return a tuple `{reply, Reply, NewState}`. `Reply` is the reply which should be sent back to the client and `NewState` is a new value for the state of the gen_server.

Asynchronous communication with the server can be achieved via `gen_server:cast(Name, Request)`, which implements a cast (i.e. an asynchronous request). The request is made into a message and sent to the gen_server registered as `Name`. When the request is received, the gen_server calls `Mod:handle_cast(Request, State)`, which is expected to return a tuple `{noreply, NewState}`.

We can use `gen_server:cast/2` to stop the ping-pong server. To this end, we have to

```
add_player(Name) ->
    gen_server:call(?MASTER, {add_player, Name}).

remove_player(Name) ->
    gen_server:call(?MASTER, {remove_player, Name}).

ping(FromName) ->
    gen_server:call(?MASTER, {ping, FromName}).

get_score(Name) ->
    gen_server:call(?MASTER, {get_score, Name}).

% ---------------------------------------------------------------------

handle_call({add_player, Name}, _From, Dict) ->
    case whereis(Name) of
        undefined ->
            Pid = spawn(fun () -> ping_pong_player(Name) end),
            true = register(Name, Pid),
            {reply, ok, dict:store(Name, 0, Dict)};
        Pid when is_pid(Pid) ->
            {reply, ok, Dict}
    end;
handle_call({remove_player, Name}, _From, Dict) ->
    Pid = whereis(Name),
    exit(Pid, kill),
    {reply, {removed, Name}, dict:erase(Name, Dict)};
handle_call({ping, FromName}, _From, Dict) ->
    {reply, pong, dict:update_counter(FromName, 1, Dict)};
handle_call({get_score, Name}, _From, Dict) ->
    Score = dict:fetch(Name, Dict),
    {reply, Score, Dict}.
```

Listing 2.2: Calls to the ping-pong master

define a callback clause `Mod:handle_cast(stop, State)` that returns `{stop, normal, NewState}`. The second element (the atom `normal`) is used as the first argument to the callback `Mod:terminate(Reason, NewState)`, which is called upon termination in order to give the server a chance to perform any final operations that it wishes to perform before exiting. When `Mod:terminate/2` returns, the generic server will be stopped and all name registrations removed. Stopping the server is depicted in Listing 2.3.

```
stop() ->
    gen_server:cast(?MASTER, stop).

handle_cast(stop, Dict) ->
    {stop, normal, Dict}.

terminate(_Reason, Dict) ->
    Players = dict:fetch_keys(Dict),
    lists:foreach(fun (Name) -> exit(whereis(Name), kill) end, Players).
```

Listing 2.3: Stopping the ping-pong master

For the gen_server to be able to receive other messages than pre-defined requests, the callback function `Mod:handle_info(Info, State)` must be implemented to handle them. Examples of other types of messages are exit messages and trapped exit signals. Last but not least, the server can dynamically update its code while running via the callback function `Mod:code_change(OldVsn, State, NewVsn)`.

2.3.2 A generic finite state machine behaviour

A Finite State Machine (FSM) can be described as a set of relations of the form:

$$S \times E \xrightarrow{A} S'$$

Meaning that if we are in state S and the event E occurs, we should perform the actions A and make a transition to the state S'.

For an FSM implemented using the gen_fsm behaviour, the state transition rules are written as a number of Erlang functions which conform to the following convention.

```
StateName(Event, StateData) ->
.. code for actions here ...
{next_state, NextStateName, NewStateData}
```

Concrete examples are given later in Listings 2.6–2.8.

The state is split into `StateName` and `StateData`. The `StateName` denotes a named state of the finite state machine, whereas the `StateData` denotes the internal state of the process implementing the gen_fsm.

Consider the state diagram in Figure 2.1. It describes the life of a strange creature that feeds on cheese, grapes and lettuce but never eats the same kind of food on two consecutive days.

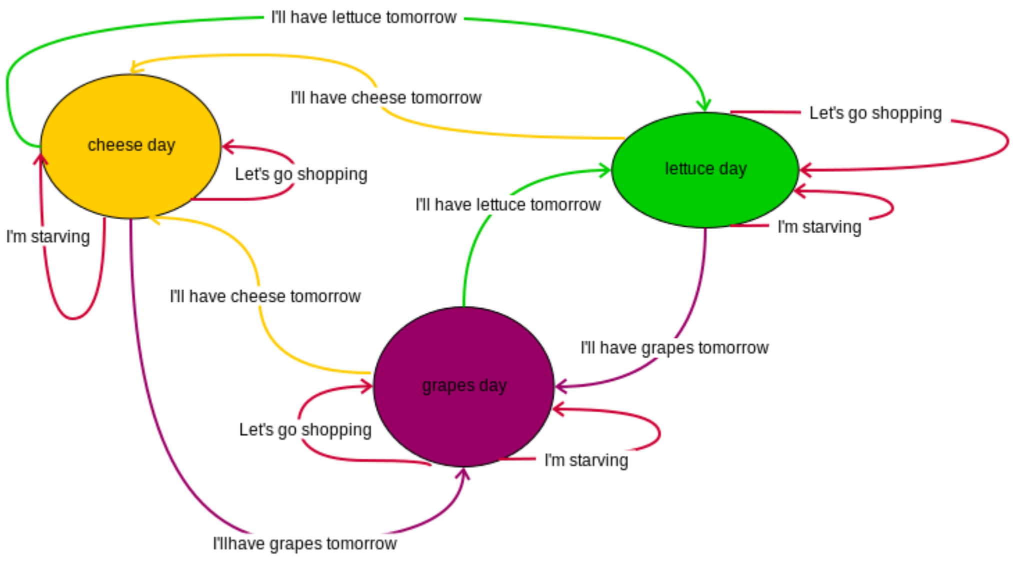

Figure 2.1: State Diagram of creature's behaviour

As we can see from the state diagram, days are categorized into *cheese_ days*, *lettuce_ days* and *grapes_ days*. On a *cheese_ day* the creature eats only cheese, on a *lettuce_ day* only lettuce and so on. When it gets hungry, it consumes a fixed portion of the daily food, which is kept in the food storage. To make life less boring and, more importantly, to bring food to the storage, the creature goes shopping from time to time. Finally, every night it decides what to eat the next day. There is only one rule regarding this decision: never eat the same food for two days in a row.

Let us assume that this creature has implemented a finite state machine describing its daily routine as a gen_fsm callback module. The internal state of the finite state machine is a record that represents the food storage, as we can see in Listing 2.4. The record fields contain the quantity of food that is currently stored.

```
-type quantity() :: non_neg_integer().

-record(storage, {cheese  = 5 :: quantity(),
                  lettuce = 5 :: quantity(),
                  grapes  = 5 :: quantity()}).
```

Listing 2.4: State data representation

We can start and link to the finite state machine in a way similar to starting and linking to the gen_server, using `gen_fsm:start_link({local, Name}, Mod, InitArgs, Opts)`. Then, the callback function `Mod:init(InitArgs)` will be called, which should now return `{ok, StateName, StateData}`. `StateName` is the name of the initial state of the finite state machine and `StateData` is the initial internal state of the gen_fsm. Starting the gen_fsm is depicted in Listing 2.5. The input argument `Day` to `start_link/1` specifies the kind of day on which the finite state machine will be started. It can be a `cheese_day`, `lettuce_day` or `grapes_day`.

```
start_link(Day) ->
    gen_fsm:start_link({local, creature}, ?MODULE, [Day], []).

init([Day]) ->
    {ok, Day, #storage{}}.
```

Listing 2.5: Starting the fsm

The functions `Mod:StateName(Event, StateData)` and `Mod:StateName(Event, Caller, StateData)` correspond to the states of the finite state machine. These states specify a context in which a given event is handled. Let us suppose that the initial call is `start_link(cheese_day)`. Then, the `Mod:init/1` callback will return the tuple `{ok, cheese_day, #storage{}}`. This means that the finite state machine will be initially in a *cheese_ day* state. Whenever the gen_fsm process receives an event, either the function `Mod:cheese_day/2` or `Mod:cheese_day/3` will be called. `Mod:cheese_day/2` is called for asynchronous events, whereas `Mod:cheese_day/3` for synchronous ones. The arguments for `Mod:StateName/2` are `Event` (i.e. the actual message sent as an event) and the `state data`. `Mod:StateName/3` takes the PID of the caller process as an extra argument. Both callbacks usually return a tuple of the form `{next_state, NextStateName, NewStateData}`.

We can simulate the condition when the creature is hungry by sending a synchronous 'eat' event to the gen_fsm. Synchronous events to be handled by `Mod:StateName/3` are sent to a gen_fsm, registered as `Name`, with `gen_fsm:sync_send_event(Name, Event)`. After the event has been handled, a reply is sent back to the client using `gen_fsm:reply(Caller, Reply)`. In our example, the reply is the quantity of available food at the moment when the creature is ready to start eating. This is described in Listing 2.6.

```
hungry() ->
    gen_fsm:sync_send_event(creature, eat).

cheese_day(eat, Caller, #storage{cheese = Cheese} = S) ->
    gen_fsm:reply(Caller, {cheese_left, Cheese}),
    {next_state, cheese_day, S#storage{cheese = Cheese - 1}}.

lettuce_day(eat, Caller, #storage{lettuce = Lettuce} = S) ->
    gen_fsm:reply(Caller, {lettuce_left, Lettuce}),
    {next_state, lettuce_day, S#storage{lettuce = Lettuce - 1}}.

grapes_day(eat, Caller, #storage{grapes = Grapes} = S) ->
    gen_fsm:reply(Caller, {grapes_left, Grapes}),
    {next_state, grapes_day, S#storage{grapes = Grapes - 1}}.
```

Listing 2.6: Example of synchronous event handling

We can, also, simulate the situation when the creature goes shopping to buy some new food. This is handled via an asynchronous event, as described in Listing 2.7. Asynchronous events to be handled by `Mod:StateName/2` are sent with `gen_fsm:send_event(Name, Event)`. Last but not least, we can simulate the beginning of a new day by asynchronously specifying the creature's food for that day. This is described in Listing 2.8.

```erlang
buy(Food, Quantity) ->
    gen_fsm:send_event(creature, {store, Food, Quantity}).

cheese_day({store, Food, Quantity}, S) ->
    case Food of
        cheese ->
            {next_state, cheese_day,
            S#storage{cheese = S#storage.cheese + Quantity}};
        lettuce ->
            {next_state, cheese_day,
            S#storage{lettuce = S#storage.lettuce + Quantity}};
        grapes ->
            {next_state, cheese_day,
            S#storage{grapes = S#storage.grapes + Quantity}}
    end;
lettuce_day({store, Food, Quantity}, S) ->
    ...
grapes_day({store, Food, Quantity}, S) ->
    ...
```

Listing 2.7: Example of asynchronous event handling

```erlang
new_day(Food) ->
    gen_fsm:send_event(creature, {new_day, Food}).

cheese_day({new_day, lettuce}, S) ->
    {next_state, lettuce_day, S};
cheese_day({new_day, grapes}, S) ->
    {next_state, grapes_day, S}.

lettuce_day({new_day, cheese}, S) ->
    {next_state, cheese_day, S};
lettuce_day({new_day, grapes}, S) ->
    {next_state, grapes_day, S}.

grapes_day({new_day, cheese}, S) ->
    {next_state, cheese_day, S};
grapes_day({new_day, lettuce}, S) ->
    {next_state, lettuce_day, S}.
```

Listing 2.8: Another example of asynchronous event handling

Sometimes an event can arrive at any state of the gen_fsm. Instead of sending the message with `gen_fsm:send_event/2` and writing one clause handling the event for each state function, the message can be sent with `gen_fsm:send_all_state_event(Name, Event)` and handled with `Mod:handle_event(Event, StateName, StateData)`. This technique can be used to stop a gen_fsm that is not part of a supervision tree. The callback function handling the stop event should return a tuple `{stop, normal, NewStateData}`, where `normal` specifies that it is a normal termination and `NewStateData` is a new value for the state data of the gen_fsm. This will cause the gen_fsm to call `Mod:terminate(normal, StateName, NewStateData)` and then terminate gracefully. Stopping the gen_fsm is depicted in Listing 2.9.

```
stop() ->
    gen_fsm:send_all_state_event(creature, stop).

handle_event(stop, _StateName, _StateData) ->
    {stop, normal, []}.

terminate(_Reason, _StateName, _StateData) ->
    ok.
```

Listing 2.9: Stopping the finite state machine

Finally, gen_fsm can handle spontaneous messages such as exit signals, or update its code dynamically while running in a way similar to gen_server. The callback functions handling these cases are `Mod:handle_info(Info, StateName, StateData)` and `Mod:code_change(OldVsn, StateName, StateData, NewVsn)` respectively.

Chapter 3

Property-Based Testing, Model-Based Testing and PropEr

3.1 An overview of property-based testing with PropEr

PropEr is a property-based testing tool for programs written in the Erlang programming language. The core PropEr modules focus on testing the behaviour of pure functions. The input domain of functions is specified through the use of a type language, modeled closely after the type language of Erlang itself. Properties are written using Erlang expressions, with the help of a few predefined macros. Should a property fail to pass a test, the failing test case is automatically simplified to a minimal test case, whose every part is essential in reproducing the failure. This process is called shrinking. Failing test cases can be saved and re-applied to the same property, allowing users to easily check if they have successfully fixed the problem.

PropEr's salient feature is that it offers a tight integration of the language of types and specs of Erlang with properties [15]. Apart from types specified in the PropEr's type-system notation, native Erlang types can also be used as generators. In addition, any function specification can be directly used as simple property of a function. Last but not least, PropEr offers support that significantly simplifies the task of writing generators for recursive and opaque data types, i.e. data types whose internal representation is not supposed to be visible outside of their defining module.

PropEr's capabilities are thoroughly described in other publications [13, 14]. In this section, we will present the characteristics directly related to our work.

3.1.1 Properties

Properties are written using Erlang expressions, with the help of a few predefined macros. A simple property can be specified by wrapping a boolean expression with a `?FORALL` wrapper, which has the following syntax:

`?FORALL(Xs, Xs_type, Prop)`
: The `Xs` field can contain either a single variable, a tuple of variables or a list of variables. The `Xs_type` field must then contain a singlePropEr type (for more on

the PropEr type system, see the next section), a tuple of types of the same size as the tuple of variables or a list of types of the same length as the list of variables, respectively. Tuples and lists can be arbitrarily nested, as long as `Xs` and `Xs_type` are compatible. All the variables inside `Xs` can (and should) be present as free variables inside the wrapped property `Prop`. When a `?FORALL` wrapper is encountered, a random instance of `Xs_type` is produced and each variable in `Xs` is replaced inside `Prop` by its corresponding instance.

A simple property is specified in Listing 3.1. In this example, we wish to test our implementation of a function that inserts an integer into its right place in a sorted list of integers. The property states that, after inserting an integer into a sorted integer list, the list will remain sorted. Sorted lists of integers are produced as instances of the generator `orderedlist(integer())`, provided by PropEr.

```
add_sorted(X, List) ->
    add_sorted(X, List, []).

add_sorted(X, [], Acc) ->
    lists:reverse(Acc) ++ [X];
add_sorted(X, [H|Tail] = List, Acc) ->
    case X < H of
        true ->
            lists:reverse(Acc) ++ [X] ++ List;
        false ->
            add_sorted(X, Tail, [H|Acc])
    end.

is_sorted(List) ->
     List =:= lists:sort(List).

%-----------------------------------------------------------------------------

prop_is_sorted() ->
    ?FORALL({I, List},
            {integer(), orderedlist(integer())},
            is_sorted(add_sorted(I, List))).
```

Listing 3.1: Example of a property

Additionally, PropEr comes with macros and functions for analyzing test case distribution, trapping exit signals and providing debugging information on failing test cases. Some examples are the following:

`aggregate(Categories, Prop)`
The `Categories` field can be an expression or statement block that evaluates to a list of categories - the test case produced will be categorized under one of these categories. In case no test fails, all produced categories are printed at the end of testing along with the percentage of test cases belonging to each category.

`?TRAPEXIT(Prop)`
If the code inside `Prop` spawns and links to a process that dies abnormally, PropEr will catch the exit signal and treat it as a test failure, instead of crashing. `?TRAPEXIT` cannot contain any more wrappers.

`?WHENFAIL(Action, Prop)`
: The `Action` field should contain an expression or statement block that produces some side-effect (e.g. prints something to the screen). In case the `Prop` fails, `Action` will be executed.

3.1.2 Types and generators

The input domain of functions is specified through the use of a custom type language, modeled closely after the type language of Erlang itself. Given a type specification, PropEr can produce a valid instance of that type. What is more, PropEr knows what shrinking strategy to apply so as to simplify failing instances of each type. The shrinking behaviour can also be fine-tuned through user-defined shrinking strategies. Apart from types specified in the PropEr's type-system notation, native erlang types (both module-local and remote) can also be used as generators; PropEr knows how to create a shrinker for them.

Examples of supported types are integers, floats, atoms, bitstrings, lists, tuples. These are denoted as `integer()`, `float()`, `atom()`, `bitstring()`, `list()`, `loose_tuple()`, respectively. PropEr also supports various more specialized types, such as integers within a given range, denoted as `range(Low, High)`, lists containing elements of a specific type `ElemType`, denoted as `list(ElemType)` and many more.

Additionally, PropEr offers functions and macros that can be applied to types in order to produce new ones. Some of the most commonly used are:

`union(ListOfTypes)`
: The union of all types in `ListOfTypes`, which cannot be empty. The random instance generator is equally likely to choose any one of the types in `ListOfTypes`. It can also be written as **`oneof(ListOfTypes)`** and **`elements(ListOfTypes)`**.

`weighted_union(ListOfTypes)`
: A specialization of `union(ListOfTypes)`, where each type in `ListOfTypes` is assigned a frequency, i.e. a positive integer. Types with larger frequencies are more likely to be chosen by the random instance generator. It can also be written as **`wunion(ListOfTypes)`** and **`frequency(ListOfTypes)`**.

`?LET(Xs, Xs_type, In)`
: To produce an instance of this type, all appearances of the variables in `Xs` inside `In` are replaced by their corresponding values in a randomly generated instance of `Xs_type`. It's OK for the `In` part to evaluate to a type - in that case, an instance of the inner type is generated recursively.

`?SUCHTHAT(X, Type, Condition)`
: This produces a specialization of `Type`, which only includes those members of `Type` that satisfy the constraint `Condition`, i.e. those terms for which the function `fun(X)` → `Condition end` returns 'true'. Testing will be stopped in case a constraint tries limit is reached.

`noshrink(Type)`
: Creates a new type which is equivalent to `Type`, but whose instances are never shrunk by the shrinking subsystem.

PropEr also supports user-defined recursive data types. To guide the generation process of a recursive data type, the user must handle the 'size' parameter (i.e. the parameter that controls the maximum size of produced instances) manually: she has to write a recursive generator function that accepts a `Size` parameter, which should be distributed among all recursive calls of each recursion path. The following macros and functions are mainly useful for writing recursive type declarations:

`?SIZED(Size, Generator)`
: Constructs a new type from a sized generator. To produce instances of this type, PropEr will apply the current value of 'size' to the function `fun(Size)` → `Generator end`.

`resize(NewSize, Type)`
: This declaration instructs PropEr to use `NewSize` instead of the value of the 'size' parameter to produce instances of `Type`. One use of this function is to modify types to grow faster or slower, like so: `?SIZED(Size, resize(Size * 2, list(integer()))`. The above specifies a list type that grows twice as fast as normal lists.

`?LAZY(Type)`
: This construct delays the generation of `Type`. Users should wrap each recursive choice in the non-zero-size clause of a recursive generator with a `?LAZY` macro, so as to achieve linear generation time.

3.1.3 Symbolic representation

When writing properties that involve abstract data types, such as dictionaries or sets, it is usually best to avoid dealing with the ADTs' internal representations directly. Working, instead, with a symbolic representation of an ADT's construction process (series of API calls) has several benefits, as it makes failing test cases easier to read and understand and, more importantly, easier to shrink.

In order to allow for *blackbox testing* of abstract data types, PropEr supports the symbolic representation of function calls, using the following syntax: `{call, Module, Function, Arguments}`. A term like that represents a call to the API function `Module:Function` with arguments `Arguments`. Each of the arguments may be a symbolic call itself or contain other symbolic calls in lists or tuples of arbitrary depth. Additionally, PropEr supports the symbolic representation of variables, using the syntax: `{var, VarId}`. This construct serves as a placeholder for values that are not known at type construction time. It will be replaced by the actual value of the variable during evaluation. The `eval/1` function can be used to evaluate a symbolic instance, i.e. calculate the concrete term it represents.

3.2 Using PropEr to test stateful systems: the idea

Typical Erlang applications consist of telecom software, database management systems, web servers and other kinds of reactive systems, i.e. systems that maintain an ongoing interaction with their environment through events. The behaviour of a reactive system is highly dependent on the internal state of that system, as well as on the state of its environment. The term *internal state* refers to the implicit state of a system, that is continuously changing as a result of the system's interaction with its environment. Usually, the state is not memory-less but depends on previous events. Given the internal state of a system, one should be able to fully determine the possible side-effects. The main problem associated with testing stateful systems is that their internal state is not observable, as it is not directly accessible to the user from the API of the system under test.

In order to be able to test stateful reactive systems in an automated way, a formal specification of their internal state and, therefore, of the possible side-effects is necessary. Such systems are typically represented as abstract models. State transition systems (FSMs, Extended FSMs (EFSMs), state charts, flow charts, Markov chains etc.) representing the possible configurations of the system under test are most often used as models. Executable paths through such a state-based specification can then be translated into test cases. This technique is generally described as Model-Based Testing [9, 17].

3.2.1 Model-Based Testing

Model-Based Testing (MBT) is a test method in which test case generation and evaluation are based on an abstract model of the system under test (SUT) and/or its environment. The model is a depiction of the system's intended behaviour.

Model-based testing typically involves the following activities [17, 9, 5, 8]:

1. Building an abstract model of the system under test, based on functional requirements or existing specification documents.
2. Defining the test selection criteria. These criteria will be used to choose a subset of behaviours of the model that are most likely to detect severe failures.
3. Transforming test selection criteria into concrete test case specifications. Given a model and a test case specification, some automatic test case generator must be able to derive a test suite.
4. Validating the model against system requirements so as to detect possible deficiencies.
5. Generating test cases deriving information from the model and the test case specifications. These test cases are expressed in terms of the abstractions used in the model.
6. Translating abstract test cases into an executable form.
7. Executing test cases and assigning them a pass/fail verdict.
8. Analyzing the execution results.

A great advantage of model-based testing is that it facilitates automatic generation of large and compact test suites, exploiting the rich theoretical background that is available for state-based models. Moreover, the model serves as a checkable, shareable and reusable partial specification of the system under test. It provides information on tests that have been run, but also gives insight into what tests have not been run yet.

On the other hand, this technique requires sizeable initial effort. Selecting the type of model, partitioning system functionality into multiple parts of a model and finally building the model can be labour-intensive tasks when it comes to complicated systems. The model should be simpler and easier to check than the SUT, but, at the same time, it should be comprehensive and precise so as to serve as basis for meaningful test case generation [9].

When implementing a model-based testing tool, one has to determine the relevant timing and possible interaction between test case generation and execution. There are two options to consider:

1. On-line testing suggests that test case generation is mixed with execution, so that the generation algorithm can react to the actual outputs of the SUT. This technique is usually applied to handle the non-determinism inherent in reactive or concurrent systems behaviour.

2. Off-line testing suggests that test cases are generated strictly before they are run. The main advantage of this approach is related to regression testing, since test cases can be generated once and then executed multiple times on the system under test. Last but not least, it allows for analyzing and shrinking failing test cases.

3.2.2 Our approach to testing stateful systems

Our approach to testing stateful systems follows the model-based testing paradigm. PropEr is used in all different phases of the test process that we described in the previous section. A behavioral model of the system under test is specified in a callback module, using Erlang and PropEr expressions. Testing is driven by PropEr-supported types and generators, which are included in the callback functions to produce directed random test data.

The symbolic representation supported for abstract data types can be readily used for symbolically representing test cases for stateful systems. Thus, an off-line test method is adopted: test cases are symbolically generated strictly before they are run. The reasons for this choice are explained in the next chapter. As a next step, symbolic test cases are evaluated using the `eval/1` function and available information from the model. The pass/fail verdict can be assigned by formulating and testing a property stating that the system under test behaves as specified in the model. Last but not least, we can use PropEr's statistics-collecting and debugging macros and functions to analyze the results of execution.

In the next chapter, we describe in detail how PropEr can be used for random model-based testing of stateful reactive systems.

Chapter 4

Testing Stateful Systems

We have implemented two library modules, namely proper_statem and proper_fsm, for model-based testing of stateful code. The model behaviour of the system under test (SUT) is defined in a callback module. Given this model, we test a stateful system by generating and performing sequences of calls to that system, while monitoring its actual responses to ensure the system behaves as expected.

4.1 PropEr Statem

Reactive systems are typically specified using abstract models. Most often, these models are state transition systems representing the possible configurations of the system under test. Adopting this idea, we implemented proper_statem to test reactive stateful systems whose internal state and side-effects are specified via an abstract state machine. When referring to *model state*, we mean the state of the abstract state machine.

4.1.1 General concepts

Test case representation

Test cases generated for testing a stateful system are lists of *symbolic* API calls to that system. Symbolic representation has several benefits, listed here in increasing order of importance:

- Generated test cases are easier to read and understand.
- Failing test cases are easier to shrink.
- The generation phase is side-effect free and this results in repeatable test cases, which is essential for correct shrinking of failing testcases and for re-checking a counterexample against the property that it falsified.

Symbolic calls are not executed during test case generation. Therefore, the actual result of a call is not known at generation time and cannot be used in subsequent operations. To remedy this situation, symbolic variables are used. A *command* is a symbolic term, used to bind a symbolic variable to the result of a symbolic call.

Listing 4.1 contains a command sequence that could be used to test the process dictionary, a process-local destructively-updated store owned by every Erlang process. In this example, the first call stores the pair `{a,42}` in the process dictionary, while the second one deletes it. Finally, a new pair `{b, {var,2}}` is stored. `{var,2}` is a symbolic variable bound to the result of `erlang:erase/1`. This result is not known at generation time since none of these operations is actually performed at that time. After evaluating the command sequence at runtime, the process dictionnary will eventually contain the pair `{b,42}`.

```
[{set, {var,1}, {call,erlang,put,[a,42]}},
 {set, {var,2}, {call,erlang,erase,[a]}},
 {set, {var,3}, {call,erlang,put,[b,{var,2}]}}]
```

Listing 4.1: Example of a command sequence

Generation time vs runtime

The method used for testing stateful systems is a combination of symbolic with actual execution of test cases. Each test consists of two phases:

- As a first step, PropEr generates random symbolic command sequences deriving information from the callback module implementing the abstract state machine. These command sequences model the operations in the system under test (SUT). During this phase, PropEr might interact with the SUT only to initialize the model state.
- As a second step, symbolic command sequences are evaluated in order to check that the system behaves as expected. Upon failure, the shrinking mechanism attempts to find a minimal command sequence provoking the same error. During this phase, PropEr interacts with the SUT by performing API calls to it, while monitoring its responses.

The *model state* is used to model the internal state of the system under test, which is not accessible to the user from the system's API. During each test, the model state can be either symbolic or dynamic:

- During command generation time, we use symbolic variables to bind the results of symbolic calls. Therefore, the model state might contain symbolic variables and/or symbolic calls, which are necessary to operate on symbolic variables. Thus, we refer to it as *symbolic state*. For example, assuming that the internal state of the process dictionary is modeled as a property list (i.e. an association list between atom-based keys and values), the model state after generating the command sequence in Listing 4.1 will be [`{b, {var,2}}`].
- During command execution time, symbolic calls are evaluated and symbolic variables are replaced by their corresponding real values. Now we refer to the state as *dynamic state*. After running the command sequence in Listing 4.1, the model state will be [`{b,42}`].

4.1.2 Defining a callback module

The following functions must be exported from the callback module implementing the abstract state machine:

`initial_state()`
: Specifies the symbolic initial state of the state machine. This state will be evaluated at command execution time to produce the actual initial state. The function is not only called at generation time, but also in order to initialize the state every time the command sequence is run (i.e. during normal execution, while shrinking and when re-checking a counterexample). For this reason, it should be deterministic and self-contained.

`command(SymbState)`
: Generates a symbolic call, given the current state `SymbState` of the abstract state machine. This function will be repeatedly called to produce the next symbolic call to be included in the test case. PropEr automatically binds the result of generated symbolic calls to symbolic variables. In other words, symbolic calls are automatically translated to commands. However, before the call is actually included, a precondition is checked.

`precondition(SymbState, Call)`
: Specifies the precondition that should hold so that `Call` can be included in the command sequence, given the current `SymbState` of the abstract state machine. In case a precondition does not hold, a new call is chosen using the `command/1` generator. If preconditions are very strict, it will take a lot of tries for PropEr to randomly choose a valid command. Since preconditions are imposed via a `?SUCHTHAT` macro (described in Chapter 3), testing will be stopped in case a constraint tries limit is reached. If this is the case, the tester should modify the `command/1` generator so that it produces valid commands more frequently.

`postcondition(DynState, Call, Result)`
: Specifies the postcondition that should hold about the `Result` of evaluating the symbolic `Call`, given the dynamic state `DynState` of the abstract state machine prior to performing the call. This function is called during command execution time, this is why the state is dynamic.

`next_state(State, Result, Call)`
: Specifies the next state of the abstract state machine, given the current `State`, the symbolic `Call` chosen and its `Result`. This function is called both at command generation and command execution time in order to determine the next state, therefore `State` and `Result` can be either symbolic (i.e. containing symbolic variables and/or symbolic calls) or dynamic (i.e. containing real values).

Defining preconditions to exclude invalid operations is an example of *positive testing*, i.e. testing aimed at showing that software works when provided with valid input. In some cases, we may wish to perform *negative testing*, i.e. testing aimed at showing that exceptions are raised in case of invalid input. This can be achieved by moving constraints from preconditions to postconditions. When removing a constraint from a precondition, we need to catch the exceptions raised from invalid operations. Then, when adding the constraint to a postcondition, we should check that when executing an invalid operation, the expected kind of exception is indeed raised.

4.1.3 Implementation details

In this section we describe how we implemented the symbolic command generators, using only PropEr's API, and how symbolic command sequences are executed in order to assign a pass/fail verdict to each test case.

Command generation phase

The following functions can be used to generate test cases for stateful systems:

`commands(Module)`
: Generates random symbolic command sequences, deriving information from the abstract state machine specified in `Module`. The initial state is computed by `Module:initial_state/0`.

`commands(Module, InitialState)`
: Similar to `commands/1`, but the generated command sequences always start at `InitialState`. The first command is `{init, InitialState}` and is used to initialize the state every time the command sequence is run. In this case, `Module:initial_state/0` is never called.

The callback functions used during command generation are `initial_state/0`, `command/1`, `precondition/2` and `next_state/3`. The next symbolic call to be included in the test case is determined by the `command/1` callback, which takes as argument the model state. On the other hand, each symbolic call chosen may update the model state via the `next_state/3` callback. Due to this mutual dependency between the model state and the symbolic calls, commands have to be generated recursively. Also, each command included in the test case must satisfy a precondition. This constraint is expressed via a `?SUCHTHAT` macro. Apart from the name of the callback module (`Module`) and the model state (`State`), the recursive generator takes two extra arguments: an integer `N` to appear in the symbolic variable `{var, N}` that will be used to bind the result of the symbolic call and an integer `Size` parameter that controls the probability of adding one more command to the sequence. The higher the value of `Size`, the higher the probability of generating one more command. Each time a command is added to the sequence, the `Size` parameter is decreased by one. Therefore, we can be sure that, at some point, it will take a zero value and command generation will halt. The recursive generator is depicted in Listing 4.2.

Given the recursive command generator, it is straightforward to implement the API function `proper_statem:commands/1`, as described in Listing 4.3. The remote call to `Module:initial_state/0` is performed to initialize the model state. The `?SIZED` macro is used to introduce the `Size` parameter to the recursive generator `commands/4`. A point that requires special attention is the effective shrinking of command sequences. The recursive generator `commands/4` is enclosed in a `noshrink/1` wrapper to prevent PropEr from applying the predefined shrinking strategies. If we had omitted this wrapper, PropEr

```
-spec commands(size(), mod_name(), symbolic_state(), pos_integer()) ->
         proper_types:type().
commands(Size, Module, State, N) ->
    ?LAZY(
       proper_types:frequency(
         [{1, []},
          {Size, ?LET(Call,
                      ?SUCHTHAT(X, Module:command(State),
                                Module:precondition(State, X)),
                      begin
                          Var = {var,N},
                          NextState = Module:next_state(State, Var, Call),
                          ?LET(Cmds,
                               commands(Size-1, Module, NextState, N+1),
                               [{set,Var,Call}|Cmds])
                      end)}])).
```

Listing 4.2: Recursive command generator

would try new commands in place of the existing ones in an attempt to produce a simpler test case. But our goal is to eliminate commands that do not contribute to failure. This goal could have been achieved via a `?LETSHRINK` macro, which is normally used for shrinking recursive generators. However, since test cases are *lists* of commands, it seemed more intuitive and effective to use the shrinking strategies that PropEr applies to simple lists. To this end, we introduced a special generator, **`shrink_list(InputList)`**. At generation time this generator has no effect, since it will generate exactly its input argument. However, it can be useful during shrinking, as it inherits the shrinking strategies that PropEr applies to simple lists. That is, shrinking a list by discarding elements that do not contribute to failure. In our case, the elements of the list are commands. Additionally, we require that shrunk instances should be *valid* command sequences. A command sequence is considered *valid* when all commands satisfy preconditions and use only symbolic variables bound to the results of preceding calls in the same sequence.

```
-spec commands(mod_name()) -> proper_types:type().
commands(Module) ->
    ?LET(InitialState, Module:initial_state(),
         ?SUCHTHAT(
            Cmds,
            ?LET(List,
                 ?SIZED(Size,
                        proper_types:noshrink(
                          commands(Size, Module, InitialState, 1))),
                 proper_types:shrink_list(List)),
            is_valid(Module, InitialState, Cmds, []))).
```

Listing 4.3: Commands/1 implementation

The generator `proper_statem:commands/2` is implemented in a similar way. The only difference lies in the initialization of the model state, which is achieved by adding an **`{init, InitialState}`** command as head of the sequence.

Command execution phase

Symbolic command sequences can be evaluated using the following functions:

`run_commands(Module, Cmds)`
: Evaluates a given symbolic command sequence `Cmds` according to the abstract state machine specified in `Module`. First, the model state is initialized via the `Module:initial_state/0` callback. Then, for each symbolic call in the test case, its arguments are evaluated using the `proper_symb:eval/1` function and its precondition is re-checked. Normally, there should not be an error at this point, except for when re-checking a counterexample. Then, the symbolic call is performed and, unless an exception is raised, the postcondition is checked. The test passes if all postconditions are true and no (unhandled) exception is raised. The result is a triple of the form `{History, DynamicState, Result}`, where:

  - `History` contains the execution history of all commands that were run without raising an exception. It contains tuples of the form `{DynState, CmdResult}`, specifying the dynamic state prior to command execution and the actual result of each command.
  - `DynamicState` contains the dynamic state of the abstract state machine at the moment when execution stopped.
  - `Result` specifies the overall outcome of command execution. A test passes if `Result` is the atom `ok`.

`run_commands(Module, Cmds, Environment)`
: Similar to `run_commands/2`, but also accepts an `Environment`, used for symbolic variable evaluation during command execution. The `Environment` consists of `{Key, Value}` pairs. Keys may be used in symbolic variables (i.e. `{var, Key}`) whithin the command sequence `Cmds`. These symbolic variables will be replaced by their corresponding `Value` during command execution.

In order to collect useful information about the commands that were executed, we have implemented the following additional functions:

`command_names(Cmds)`
: Extracts the names of the calls from a given command sequence, in the form of MFAs (i.e. in the form `{Module, Function, Arity}`).

`zip(ListA, ListB)`
: Behaves like `lists:zip/2`, but the input lists do no not necessarily have equal length. Zipping stops when the shortest list stops. This is useful for zipping a command sequence with its (failing) execution history.

Workflow

As mentioned before, each test consists of two phases. First, PropEr generates random symbolic command sequences, deriving information from the callback module implementing the abstract state machine. This is the role of `commands/[1,2]` generators. As a second step, command sequences are executed so as to check that the system behaves as expected. This is the role of the functions `run_commands/[2,3]`. These two phases are encapsulated in the property specified in Listing 4.4, which can be used for testing stateful systems with PropEr. When testing code with side-effects, it is very important to keep each test self-contained. For this reason, almost every property for testing stateful systems contains some set-up and/or clean-up code. Such code is necessary to put the system in a known state, so that the next test can be executed independently from previous ones.

```
prop_stateful() ->
    ?FORALL(Cmds, commands(?MODULE),
            begin
                {_History, _State, Result} = run_commands(?MODULE, Cmds),
                cleanup(),
                Result =:= ok
            end).
```

Listing 4.4: Property to test stateful systems

The workflow of proper_statem, when testing a property similar to that specified in Listing 4.4, is summarized in Figure 4.1.

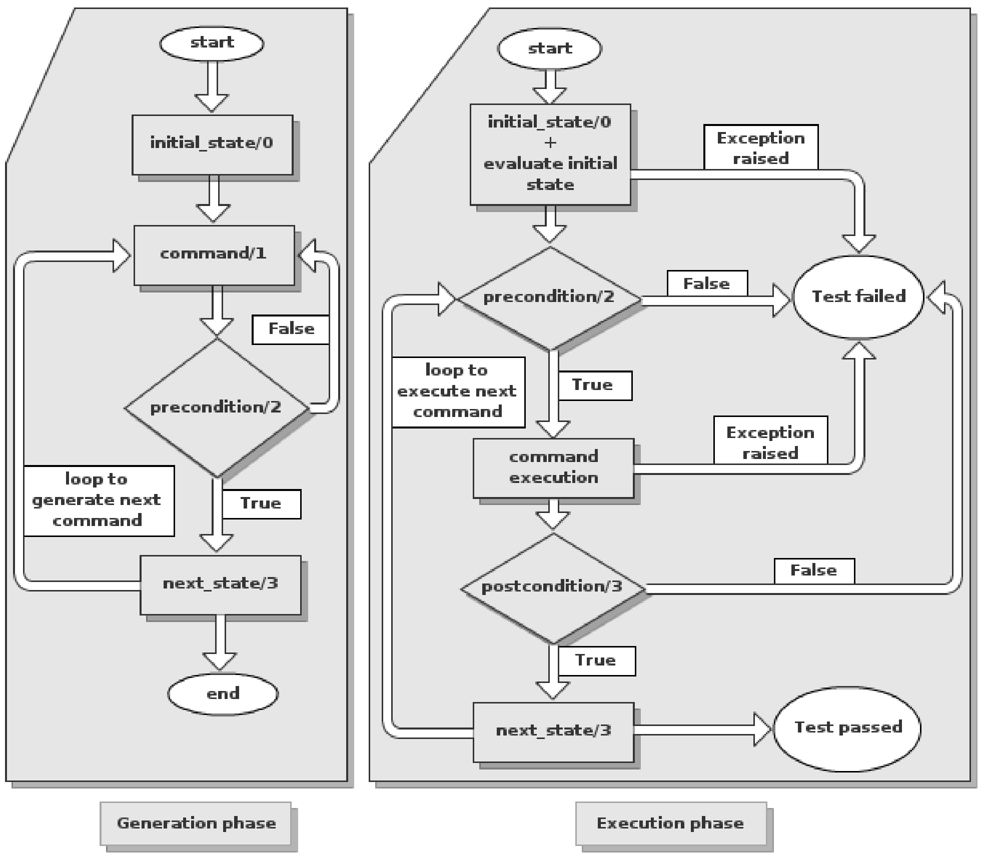

Figure 4.1: PropEr statem workflow

4.1.4 Parallel statem testing

Property-based testing can also be used to debug concurrent applications. The Quickcheck tool from Quviq has been efficiently used, in combination with a custom randomizing scheduler, to detect race conditions [12]. In that case study, the authors formulate a property for testing parallel code and describe the outline of a method for generating parallel test cases. The specification used to determine correct parallel behaviour is that the API calls to the system under test should behave atomically. A test passes iff the results observed could have been produced by a possible sequential execution of the operations in the parallel test case.

These ideas have been implemented in the proper_statem library module. The abstract state machine specification that is used for sequential testing can also be used for generating command sequences that will be executed in parallel to test for race conditions. A parallel test case consists of a sequential and a parallel component. The sequential component is a command sequence that is run first to put the system in a random state. The parallel component is a list of command sequences that are executed in parallel, each of them in a separate newly-spawned process. After running a parallel test case, PropEr uses the state machine specification to check if the results observed could have been produced by a possible sequentialization of the parallel component. If no such sequentialization is possible, then an atomicity violation has been detected. In this case, the shrinking mechanism attempts to produce a counterexample that is minimal in terms of concurrent operations.

Parallel test case generation

A parallel test case consists of a sequential prefix component and a list of concurrent tasks. Generating parallel test cases involves the following actions. Initially, we generate a command sequence deriving information from an abstract state machine specification, as in the case of sequential statem testing. Then, we parallelize a random suffix of the command sequence by splitting it into a pre-defined number of concurrent tasks. The initial relevant ordering of commands must be preserved within each task. We will have to explore all possible interleavings of the concurrent tasks in order to generate an appropriate parallel test case. Therefore, it is important to keep the number of such interleavings feasible to enumerate. The `?LIMIT` macro is used to specify the maximum length of a suffix to be parallelized and the `?WORKERS` macro is used to specify the number of concurrent tasks. These two parameters can be modified for experimentation without the need to make further changes to the code. However, care has to be taken since the number of possible command interleavings is exponential in both the number of parallel `?WORKERS` and the `?LIMIT` length of each command sequence. More precisely, given n parallel processes that each executes m commands, the number of possible interleavings is $\frac{(n*m)!}{m!^n}$.

In our current implementation, a suffix no longer than 12 commands is split in 2 lists of approximately the same length, if possible. Limitations arise from the fact that each list should be a *valid* command sequence (i.e. all commands should satisfy preconditions and use only symbolic variables bound to the results of preceding calls in the same sequence). Furthermore, we apply an additional check: we have to ensure that preconditions are satisfied in all possible interleavings of the concurrent tasks. Otherwise, an exception might be raised during parallel execution and lead to unexpected (and unwanted) test failure. In case this constraint cannot be satisfied for a specific test case, the test case will be executed sequentially. Then an ‘f’ is printed on screen to inform the user. This usually means that preconditions need to become less strict for parallel testing to work. Moving constraints from preconditions to postconditions is a simple and efficient way to handle such cases.

Parallel test cases can be generated using `parallel_commands(Module)`, where `Module` is the name of the callback module that contains the abstract state machine specification. Our implementation of `parallel_commands/1` is described in Listing 4.5. The innermost `?LET` is used to generate the parallel test case and provide the main shrinking strategy, while the outermost `?LET` provides an additional shrinker for parallel test cases. These steps will be explained in further detail in the rest of this section.

```erlang
-define(LIMIT, 12).
-define(WORKERS, 2).

-spec parallel_commands(mod_name()) -> proper_types:type().
parallel_commands(Module) ->
    ?LET({ShrunkSequential, ShrunkParallel},
         ?LET({Sequential, Parallel},
              noshrink(parallel_gen(Module)),
              parallel_shrinker(Module, Sequential, Parallel)),
         move_shrinker(ShrunkSequential, ShrunkParallel, ?WORKERS)).
```

Listing 4.5: Parallel test case generator

In order to generate a parallel test case, we have to split a selected suffix of the initial command sequence into `?WORKERS` subsequences that satisfy the aforementioned constraints. To this end, we lazily produce possible ways of splitting the original command sequence, until we reach one that satisfies the constraints. As it is more convenient to lazily produce possible combinations when working with integers instead of commands, we create a lookup dictionary that associates each command with an integer denoting the position of the command in the original list. The initial combination for splitting 10 commands into 3 sublists would look like so: $[\{1,[7,8,9,10]\},\{2,[4,5,6]\},\{3,[1,2,3]\}]$. This means that commands associated with the indexes 7, 8, 9, 10 are assigned to the first concurrent task, commands associated with indexes 4, 5, 6 to the second task and so on. For a combination to be valid, the indices in each sublist must be in increasing order and no duplicates are permitted.

After producing a valid combination, we check whether it satisfies specific constraints (this point will be elaborated in the next paragraph). In case the constraints are satisfied, a parallel test case has been generated. If this is not the case, we need to produce the next combination. To achieve this, we initially try to re-order the indices in the first two sublists. In case of `?WORKERS` $= 2$, this is all that can be done. In the general case of `?WORKERS` ≥ 2, after trying all possible valid re-orderings of the indices in the first two sublists, we make a change to the third sublist and then try again the new possible re-orderings of the first two sublists. This technique is generalised to handle an arbitrary number of parallel `?WORKERS`. In case the constraints are not satisfied for any produced test case, we change the expected length of the sublists, assigning more commands to the first concurrent task. Thus, in the worst case, the whole test case is assigned to only one task and is executed sequentially.

Parallel test cases must satisfy specific constraints. Initially, we have to ensure that each sublist is a *valid* command sequence. If this is the case, we can move on to check the second constraint: that preconditions are satisfied in all possible interleavings of the concurrent tasks. Sharing Donald Knuth's opinion on the perils of 'premature optimization', we have kept our approach as simple as possible for the time being; we produce all possible interleavings and then check if preconditions are satisfied. The method is computationally feasible as long as we carefully choose the values of the ?WORKERS and ?LIMIT macros.

Interleavings of two command sequences are produced using the function **insert_all(L1, L2)**, as specified in Listing 4.6. This function returns all possible insertions of the elements of the first list, preserving their order, into the second list. The implementation of **insert_all/2** depends on **all_insertions/3** and **index/2**. The function **all_insertions(Element, Limit, List)** returns all possible insertions of **Element** into **List** with the constraint that **Element** should be inserted in a position with index less than **Limit**. Indexing is zero-based. The function **index(X, List)** returns the position of the first occurence of **X** in **List**. The head of the list is at position 1. The technique is easily generalised to produce all possible interleavings between an arbitrary number of command sequences. This is also described in Listing 4.6, where the function **possible_interleavings/2** is specified.

```
-spec insert_all([term()], [term()]) -> [[term()]].
insert_all([], List) ->
    [List];
insert_all([X], List) ->
    all_insertions(X, length(List) + 1, List);

insert_all([X|[Y|_] = Tail], List) ->
    [L2 || L1 <- insert_all(Tail, List),
           L2 <- all_insertions(X, index(Y, L1), L1)].

-spec possible_interleavings([command_list()]) -> [command_list()].
possible_interleavings([P1,P2]) ->
    insert_all(P1, P2);
possible_interleavings([P1|Rest]) ->
    [I || L <- possible_interleavings(Rest),
          I <- insert_all(P1, L)].
```

Listing 4.6: Producing command interleavings

Shrinking of parallel test cases does not depend on the generation process. For this reason, as we can notice in Listing 4.5, the generator **parallel_gen/1** is enclosed in a **noshrink/1** wrapper that prevents PropEr from applying predefined shrinking strategies to parallel test cases. Since the sequential prefix and the concurrent tasks are lists of commands, they should inherit the methods that PropEr uses to shrink command lists. This is the role of **parallel_shrinker/3**, which is applied via the innermost ?LET in Listing 4.5. Furthermore, the commands of the parallel component may (and usually do) depend on the sequential prefix. Since we are interested in counterexamples that are minimal in terms of concurrent operations, we attempt to shrink the parallel component first, then the sequential. The main shrinking strategy applied to parallel test cases is depicted in Listing 4.7.

In Listing 4.8 we describe an additional shrinker that is applied on top of the main shrinkers

```
-spec parallel_shrinker(mod_name(), command_list(), [command_list()]) ->
          proper_types:type().
parallel_shrinker(Mod, Sequential, Parallel) ->
    I= Mod:initial_state(),
    ?SUCHTHAT({Seq1, Parallel1},
              ?LET(ShrunkParallel,
                   [proper_types:shrink_list(P) || P <- Parallel],
                   ?LET(ShrunkSequential,
                        proper_types:shrink_list(Sequential),
                        {ShrunkSequential, ShrunkParallel})),
              lists:all(
                  fun(P) -> is_valid(Mod, I, Seq1 ++ P, []) end,
                  Parallel1)).
```

Listing 4.7: Main shrinker for parallel test cases

and attempts to move as many commands as possible from the parallel to the sequential component. This is achieved via a `?SHRINK(Type, AltGens)` macro that allows the tester to provide a list of additional generators, `AltGens`, that serve as primary shrinking targets for the specific `Type`. Upon failure, these generators are first run to produce hopefully simpler instances of `Type`. In our case, the alternative generators `AltGens` remove command slices of different lengths from the parallel component and add them to the sequential prefix.

```
-spec move_shrinker(command_list(), [command_list()], pos_integer()) ->
          proper_types:type().
move_shrinker(Sequential, Parallel, 1) ->
    ?SHRINK({Sequential, Parallel},
            [{Sequential ++ Slice, remove_slice(1, Slice, Parallel)}
             || Slice <- get_slices(lists:nth(1, Parallel))]);
move_shrinker(Sequential, Parallel, I) ->
    ?LET({ShrunkSequential, ShrunkParallel},
         ?SHRINK({Sequential, Parallel},
                 [{Sequential ++ Slice, remove_slice(I, Slice, Parallel)}
                  || Slice <- get_slices(lists:nth(I, Parallel))]),
         move_shrinker(ShrunkSequential, ShrunkParallel, I-1)).
```

Listing 4.8: Additional shrinker for parallel test cases

Testing for race conditions

Parallel test cases are run using `run_parallel_commands(Module, TestCase)`, where `Module` is the name of the callback module and `TestCase` is the paralllel test case to be executed. This function evaluates the sequential prefix using `proper_statem:run_commands/2` and then executes the concurrent tasks in separate newly-spawned processes, while collecting their results. The function used to execute a command sequence has to be mapped to each of the concurrent tasks. To this end, we have implemented an SMP alternative of `lists:map/2`, called `pmap/2`, inspired from a talk from Ulf Wiger [18]. Using `pmap/2`, we spawn and link to `?WORKERS` new processes and instruct them to execute the concurrent tasks. Upon completion of its task, each worker process reports back the command execution history and terminates gracefully. Preconditions are satisfied in all possible interleavings of the parallel processes, since this was a prerequisite for generated parallel test cases. Thus, no unhandled exceptions are normally raised during parallel execution.

As a next step, we attempt to construct a sequentialization of the parallel component which explains the results observed. In other words, we look for an interleaving of the parallel processes in which the postconditions specified in the abstract state machine are satisfied. This is achieved by examining the available commands in a depth-first order. As soon as the result of a command does not satisfy a postcondition, then this interleaving and all subsequent ones can be pruned. However, this does not suggest that we can discard the specific command, since it may become valid after one or more commands from the other concurrent tasks have been added to the sequentialization. Our goal is to consume all commands in an order that explains the results observed.

The search algorithm has been implemented in a generic way to account for an arbitrary number of concurrent tasks. It is depicted in Listing 4.9. The lists `Tried` and `ToTry` contain the tasks that have already been (unsuccessfully) tried and the ones that remain to be tried, respectively. After consuming one or more commands from a task, it is necessary to explore all other non-empty tasks once again. This is why they are stored in the `Tried` list, to be retrieved later. We also keep a flag `Changed` to record whether some command has been consumed while examining the `ToTry` list of tasks. In case all tasks have been moved to the `Tried` list and the flag is still 'false' (i.e. no command has been consumed), then there is no possible sequentialization that explains the results observed.

```
-spec check(mod_name(), dynamic_state(), proper_symb:var_values(),
            boolean(), [parallel_history()], [parallel_history()]) -> boolean().
check(_Mod, _State, _Env, _Changed, [], [])  -> true;
check(_Mod, _State, _Env, false, _Tried, []) -> false;
check(Mod, State, Env, true, Tried, []) ->
    check(Mod, State, Env, false, [], Tried);
check(Mod, State, Env, Changed, Tried, [P|ToTry]) ->
    case P of
        [] ->
            check(Mod, State, Env, Changed, Tried, ToTry);
        [H|Tail] ->
            {{set, {var,N}, {call,M,F,A}}, Res} = H,
            M_ = proper_symb:eval(Env, M),
            F_ = proper_symb:eval(Env, F),
            A_ = proper_symb:eval(Env, A),
            Call = {call,M_,F_,A_},
            case Mod:postcondition(State, Call, Res) of
                true ->
                    Env2 = [{N, Res}|Env],
                    NextState = proper_symb:eval(
                                    Env2, Mod:next_state(State, Res, Call)),
                    check(Mod, NextState, Env2, true, Tried, [Tail|ToTry])
                        orelse check(Mod, State, Env, Changed,
                                     [P|Tried], ToTry);
                false ->
                    check(Mod, State, Env, Changed, [P|Tried], ToTry)
            end
    end.
```

Listing 4.9: Attempt to construct a sequentialization of the results observed

The result of `run_parallel_commands/2` is a triple of the form `{Sequential_history, Parallel_history, Result}`, where:

- `Sequential_history` contains the execution history of the sequential prefix.
- `Parallel_history` contains the execution history of each of the concurrent tasks.
- `Result` specifies the outcome of the attemp to serialize command execution, based on the results observed. It can be one of the atoms:
  - `ok`
  - `no_possible_interleaving`

The procedure that we described for parallel testing for race conditions can be encapsulated in the property described in Listing 4.10.

```
prop_parallel_testing() ->
   ?FORALL(Testcase, parallel_commands(?MODULE),
           begin
               {_Sequential, _Parallel, Result} =
                   run_parallel_commands(?MODULE, Testcase),
               cleanup(),
               Result =:= ok
           end).
```

Listing 4.10: Property for parallel testing

As a final remark, we should note that the actual interleaving of commands of the parallel component depends on the Erlang scheduler, which is too deterministic. For PropEr to be able to detect race conditions using parallel statem testing, the code of the system under test should be instrumented with `erlang:yield/0` calls to the scheduler.

4.2 PropEr FSM

We have developed another library module, proper_fsm, which offers a convenient way to test systems exhibiting a finite state machine behaviour. That is, systems that can be easily modeled as a finite collection of named states and transitions between them. Proper_fsm is closely related to proper_statem and is, in fact, implemented in terms of that. Test cases generated using proper_fsm will be on precisely the same form as test cases generated using proper_statem. The difference lies in the way the callback modules are specified. A typical finite state machine representation of a system is the state diagram. Proper_fsm is designed to bring the callback module specification very close to a state diagram.

The relation between proper_statem and proper_fsm is similar to that between gen_server and gen_fsm in OTP libraries. The abstract state machine modeled by a proper_statem callback module can be considered to be a server and all events may arrive at any time. The finite state machine modeled by a proper_fsm callback module limits the possibility to events that can only happen in a certain state.

4.2.1 Changes in the callback module

State representation

Following the convention used in gen_fsm behaviour, the state is separated into `StateName` and `StateData`. `StateName` is used to denote a named state of the FSM and `StateData` is any relevant information that has to be stored in the model state. States are fully represented as tuples `{StateName, StateData}`.

`StateName` is usually an atom (i.e. the name of the state), but can also be a tuple. In the latter case, the first element of the tuple must be an atom specifying the name of the state, whereas the rest of the elements can be arbitrary terms specifying state attributes. For example, when implementing a finite state machine of an elevator which can reach `N` different floors, the `StateName` for each floor could be `{floor, i}, 1` $\leq$ `i` $\leq$ `N`. `StateData` can be an arbitrary term, but is usually a record.

Splitting the model state into `StateName` and `StateData` makes the states of the finite state machine more explicit and allows the specification to be closer to a state diagram of the system under test. The `proper_statem:command/1` generator is replaced by a collection of callback functions, one for each reachable state of the FSM. Each callback function is named after the state it represents and takes the `StateData` as argument. If the `StateName` is a tuple, the callback is named after its first element. In this case, the state attributes are passed as arguments, along with the `StateData`.

The state is initialized via the `initial_state/0` and `initial_state_data/0` callbacks. The former specifies the initial state of the finite state machine, while the latter specifies what the state data should initially contain. As with `proper_statem:initial_state/0`, these functions are called both at generation time and at runtime to correctly initialize the model state.

Specifying transitions

As mentioned above, there is a separate callback function for each state of the finite state machine. This function specifies a list of possible transitions from that state. A transition is represented as a tuple `{TargetState, SymbCall}`. This means that performing the specified symbolic call, while being at the current state of the finite state machine, will lead to `TargetState`. The atom `history` can be used as `TargetState` to denote that a transition does not change the current state of the FSM.

Let us assume that we want to specify the finite state machine of a television. Listing 4.11 shows how to model the possible transitions from the two states of the FSM. In this simple example, the `StateData` represents the current tv channel and is not actually used in the transition specifications. The `StateData` is updated via a separate callback function that will be described later.

At command generation time, the callback function corresponding to the current state of the finite state machine will be called to return the list of possible transitions from that state. Then, PropEr will randomly choose a transition and, according to that, generate the next symbolic call to be included in the command sequence.

By default transitions are chosen with equal probability. In order to fine-tune the frequency with which each transition is chosen, we can define the optional callback

```
tv_on(_Channel) ->
    [{history, {call,tv,turn_on,[]}},
     {history, {call,tv,switch_channel,[channel()]}},
     {tv_off, {call,tv,turn_off,[]}}].

tv_off(_Channel) ->
    [{history, {call,tv,turn_off,[]}},
     {tv_on, {call,tv,turn_on,[]}}].
```

Listing 4.11: Specifying transitions for the FSM of a tv

`weight(FromState, TargetState, SymbCall)`. This callback assigns an integer weight to transitions from `FromState` to `TargetState` triggered by symbolic call `SymbCall`. Each transition is chosen with probability proportional to the weight assigned.

Additionally, PropEr detects transitions that would raise an exception of class <*error*> at generation time (not earlier) and does not choose them. This feature was introduced for compatibility with Quviq Quickcheck and can be used to include conditional transitions that depend on the `StateData`.

Other callback functions

Preconditions and postconditions are also imposed on the finite state machine specification. Only now these callbacks take a slightly different form. Instead of the `State` argument that was provided in proper_statem callbacks, the callbacks in proper_fsm take into account the origin of a transition (`FromState`), the target of a transition (`TargetState`) and the `StateData`. The set of callback functions corresponding to the states of the FSM update part of the model state, since they specify the target state of a transition. In order to update the `StateData`, we have to define a separate callback function.

`precondition(FromState, TargetState, StateData, SymbCall)`
: Specifies the precondition that should hold so that `SymbCall` can be included in the command sequence. In case the precondition doesn't hold, a new transition is chosen using the callback function corresponding to the current state of the FSM. It is possible for more than one transitions to be triggered by the same symbolic call and lead to different target states. In this case, the precondition callback must return true for at most one of the target states. Otherwise, PropEr will not be able to detect which transition was chosen and an exception will be raised.

`postcondition(FromState, TargetState, StateData, SymbCall, Result)`
: Specifies the postcondition that should hold about the `Result` of the evaluation of `SymbCall`. This function is only called at runtime, therefore the `StateData` contains real values.

`next_state_data(FromState, TargetState, StateData, Result, SymbCall)`
: Specifies how the transition from `FromState` to `TargetState` triggered by `SymbCall` affects the `StateData`. `Result` refers to the result of `SymbCall` and can be either symbolic or dynamic.

4.2.2 Implementation details

The proper_fsm library is implemented as a proper_statem callback module. The model state is a record containing the name of the proper_fsm callback module, as well as the `StateName` and `StateData` of the FSM. The model state is initialized as described in Listing 4.12. The initial `StateName` and `StateData` are easily computed from the corresponding proper_fsm callbacks, but the name of the callback module has to be passed as input argument to the function that will initialize the model state, otherwise it cannot be retrieved. Therefore, the initilization callback will be `initial_state/1` instead of `initial_state/0`. This difference does not cause an important issue, as we will see in the next section where we describe our implementation of the proper_fsm API functions. In the rest of this section, we will describe how the other proper_statem callbacks are defined, so as to provide a proper_fsm interface.

```
-record(state, {name :: state_name(),  %% fsm state name
                data :: state_data(),  %% fsm state data
                mod  :: mod_name()}).  %% fsm callback module
-type state() :: #state{}.

-spec initial_state(mod_name()) -> state().
initial_state(Module) ->
    StateName = Module:initial_state(),
    StateData = Module:initial_state_data(),
    #state{name = StateName, data = StateData, mod = Module}.
```

Listing 4.12: Initializing the model state

The `command/1` generator should return the next symbolic call to be included in the command sequence, deriving this information from the set of callbacks corresponding to the FSM states. Our implementation is depicted in Listing 4.13. The function `get_transitions(Module, FromState, StateData)` returns all possible transitions from `FromState` by applying the callback function associated with this state. Then `choose_transition/3` makes a random choice, taking into account the weight assignment, if such an assignment exists. In fact, `choose_transition/3` discards the target state specified in the transition pattern and only returns the symbolic call chosen (i.e. the trigger of the transition), since the `command/1` callback is expected to return a symbolic call generator, not a transition pattern. When the optional callback `weight/3` is not defined (or not exported), a symbolic call is chosen by applying a `union/1` generator to the list of possible symbolic calls. On the other hand, when this callback is defined and exported, we apply a `weighted_union/1` generator to the list of possible symbolic calls zipped with their corresponding weights.

A subtle point is that we would like to detect transitions that raise an exception of class <*error*> at generation time and not choose them. To achieve that, we have defined internal exception-catching versions of the `union/1` and `weighted_union/1` generators and use these instead. When a specific transition raises an exception of class <*error*> at generation time, the internal generators simply ignore that transition and choose another one instead.

The statem `precondition/2` callback should just propagate the result of `precondition/4`, as specified in the proper_fsm callback module. However, `precondition/4` takes as in-

```
-spec command(state()) -> proper_types:type().
command(#state{name = FromState, data = StateData, mod = Module}) ->
    choose_transition(Module, FromState,
                      get_transitions(Module, FromState, StateData)).
```

Listing 4.13: Choosing the next symbolic call

put argument the target of the transition, which is not directly available. The function `target_states(Module, FromState, StateData, Call)` is used to retrieve all possible target states. A state `TargetState` is a possible target when there is at least one transition from `FromState` leading to `TargetState`, triggered by a symbolic call generator with the same MFA representation as the given symbolic `Call`.

As mentioned before, there might be more than one transitions from a given state, trigerred by the same function call but leading to different targets (for different arguments of the function call). At most one of these target states may have a true precondition. Therefore, we filter the list of possible targets and return true in case there is exactly one target state with a true precondition. If no such target state exists, we return false. In case of multiple targets, an exception is raised. This is described in Listing 4.14.

```
-spec precondition(state(), symb_call()) -> boolean().
precondition(#state{name = FromState, data = StateData, mod = Module}, Call) ->
    Targets = target_states(Module, FromState, StateData, Call),
    case [To || To <- Targets,
                Module:precondition(FromState, cook_history(FromState, To),
                                    StateData, Call)] of
        []   ->
            false;
        [_T] ->
            true;
        _ ->
            erlang:error(too_many_targets)
    end.

-spec cook_history(state_name(), state_name()) -> state_name().
cook_history(From, history) -> From;
cook_history(_, To)         -> To.
```

Listing 4.14: precondition/2 callback

The statem `postcondition/3` callback should propagate the result of `postcondition/5`, specified in the proper_fsm callback module. At this point, we can be sure that there is a unique valid target for the chosen transition, since the `precondition/2` callback has preceded. Therefore, we retrieve the target state of the transition with `transition_target/4` and apply the `postcondition/5` callback. This is depicted in Listing 4.15.

Finally, the statem `next_state/3` callback is used to update the model state. After the update, the `state_name` field of the state record should contain the target of the transition just performed and the `state_data` field should contain the result of the proper_fsm `next_state_data/5` callback. This is depicted in Listing 4.16.

```
-spec postcondition(state(), symb_call(), result()) -> boolean().
postcondition(#state{name = From, data = StateData, mod = Module}, Call, Res) ->
    To = cook_history(From, transition_target(Module, From, StateData, Call)),
    Module:postcondition(From, To, StateData, Call, Res).
```

Listing 4.15: postcondition/3 callback

```
-spec next_state(state(), symb_var() | result(), symb_call()) -> state().
next_state(S = #state{name = From, data = StateData, mod = Module}, Var, Call) ->
    To = cook_history(From, transition_target(Module, From, StateData, Call)),
    S#state{name = To,
            data = Module:next_state_data(From, To, StateData, Var, Call)}.
```

Listing 4.16: next_state/3 callback

4.2.3 API description and implementation

The following functions can be used to test finite state machine specifications:

`commands(Module)`
: A special PropEr generator which generates random command sequences, according to a finite state machine specification. The function takes as input the name of a callback module, which contains the FSM specification. The initial state is computed by `{Module:initial_state/0, Module:initial_state_data/0}`.

`commands(Module, InitialState)`
: Similar to `commands/1`, but the generated command sequences always start at `InitialState`. In this case, the first command is always `{init, InitialState = {StateName, StateData}}` and is used to correctly initialize the model state every time the command sequence is run.

`run_commands(Module, Cmds)`
: Evaluates a given symbolic command sequence `Cmds` according to the finite state machine specified in `Module`. The result is a triple of the form `{History, DynamicState, Result}`, similar to `proper_statem:run_commands/2`.

`run_commands(Module, Cmds, Environment)`
: Similar to `run_commands/2`, but also accepts an `Environment` used for symbolic variable evaluation, exactly as described for `proper_statem:run_commands/3`.

`state_names(History)`
: Extracts the names of the states from a given command execution `History`. It is useful to collect statistics about state transitions during command execution.

Given the proper_statem callback module described in the previous section, it is straightforward to define the proper_fsm API functions in terms of the corresponding proper_statem ones. Our implementation of the proper_fsm command generators is specified in Listing 4.17. As we have noted, the proper_statem callback module does not define an `initial_state/0` function. Therefore, the model state has to be explicitly initialized via `proper_statem:commands/2`. In the case of `proper_fsm:commands/1`, the `?LET` macro simply discards the `{init, InitialState}` command that had been placed as head of the sequence by `proper_statem:commands/2`. In the case of `proper_fsm:commands/2`, the `?LET` macro fixes the form of the `InitialState` that is used in the initilization command. This is necessary because the model state of the proper_statem callback is a `#state{}` record, whereas the model state of the proper_fsm callback should have the form `{StateName, StateData}`.

```
-spec commands(mod_name()) -> proper_types:type().
commands(Module) ->
    ?LET([_|Cmds],
         proper_statem:commands(?MODULE, initial_state(Module)),
         Cmds).

-spec commands(mod_name(), fsm_state()) -> proper_types:type().
commands(Mod, {StateName,StateData} = InitialState) ->
    State = #state{name = StateName, data = StateData, mod = Module},
    ?LET([_|Cmds],
         proper_statem:commands(?MODULE, State),
         [{init,InitialState}|Cmds]).
```

Listing 4.17: Implementation of proper_fsm command generators

The functions `proper_fsm:run_commands/[2,3]` are implemented in terms of `proper_statem:run_commands/[2,3]`, as specified in Listing 4.18. The helper function `tmp_commands(Module, Cmds)` adds an extra initialization command to the sequence `Cmds`, to ensure that the model state is correctly initialized before the command sequence is run.

```
-spec run_commands(mod_name(), command_list()) ->
         {history(),fsm_state(),fsm_result()}.
run_commands(Module, Cmds) ->
    run_commands(Module, Cmds, []).

-spec run_commands(mod_name(), command_list(), proper_symb:var_values()) ->
         {history(),fsm_state(),fsm_result()}.
run_commands(Module, Cmds, Env) ->
    Cmds1 = tmp_commands(Module, Cmds),
    {H,S,Result} = proper_statem:run_commands(?MODULE, Cmds1, Env),
    History = [{{StateName,StateData}, R}
               || {#state{name = StateName, data = StateData}, R} <- H],
    State = {S#state.name, S#state.data},
    {History, State, Result}.
```

Listing 4.18: Implementation of proper_fsm:run_commands/2,3

A property that can be used to test finite state machine specifications is specified in Listing 4.19. Due to name conflicts with functions automatically imported from proper_statem, a fully qualified call is needed in order to use the API functions of proper_fsm.

```
prop_fsm() ->
    ?FORALL(Cmds, proper_fsm:commands(?MODULE),
            begin
                {_History, _State, Result} =
                    proper_fsm:run_commands(?MODULE, Cmds),
                cleanup(),
                Result =:= ok
            end).
```

Listing 4.19: Property for testing finite state machines

Chapter 5

Some PropEr Tutorials

In this chapter we will present two tutorials on the correct use of PropEr for testing stateful reactive systems. The first example is about testing generic servers (gen_server behaviour) and process interaction, whereas the second one covers finite state machine testing (gen_fsm behaviour).

5.1 PropEr statem tutorial

In this tutorial, we will use PropEr to test the interaction of the ping-pong server, described in Chapter 2, with the player processes. We can assume that we have already successfully tested the stand-alone behaviour of the server. There are still interesting bugs to discover when the server starts interacting with the player processes.

5.1.1 The ping-pong players

A ping-pong player is a process spawned and registered as `Name` that executes the loop described in Listing 5.1.

```
ping_pong_player(Name) ->
    receive
        ping_pong ->
            ping(Name);
        {tennis, From} ->
            From ! maybe_later;
        {football, From} ->
            From ! no_way
    end,
    ping_pong_player(Name).
```

Listing 5.1: Ping-pong player's loop

When a player is asked by an external client to play ping-pong, she will send a ping message to the ping-pong master. On the other hand, if asked to play tennis or football, the player replies with a message expressing her dislike for any sport other than ping-pong. The API for interacting with a ping-pong player is described in Listing 5.2.

```
play_ping_pong(Player) ->
    Player ! ping_pong,
    ok.

play_tennis(Player) ->
    Player ! {tennis, self()},
    receive
        Reply -> Reply
    end.

play_football(Player) ->
    Player ! {football, self()},
    receive
        Reply -> Reply
    end.
```

Listing 5.2: Ping-pong player's API

5.1.2 It's ping-pong time!

It's now time to test that the system behaves as expected when the ping-pong players interact with the master. To this end, we will specify an abstract state machine modeling the master's internal state, just as we would do to test the stand-alone behaviour of the master. We choose to base our state machine specification on the master process because this is the main component of the system under test. But now, instead of making `ping/1` calls directly to the master, we will instruct the ping-pong players to do so by performing the asynchronous `play_ping_pong/1` call. Moreover, we will include synchronous `play_tennis/1` calls to the ping-pong players, to test that such calls do not influence the players' interaction with the master. In our case, this is quite obvious. But when testing, for example, the interaction of processes in a big supervision tree, we cannot be sure about the possible side-effects of each operation.

On the other hand, it is important to keep the complexity of our model at a reasonable level. Otherwise, it's quite probable to make errors in the state machine specification. For each different feature we would like to test, defining a simple state machine that concentrates on the operations related to that feature will usually reveal any inconsistencies between the model and the actual system behaviour. These inconsistencies will be reflected in the results of the selected API calls.

In Listing 5.3 we specify the model state representation and the callback used to initialize this state. Then, in Listings 5.4 – 5.7, we define the abstract state machine that will be used to test the ping-pong system. The callback functions that should be defined are `command/1`, `next_state/3`, `precondition/2` and `postcondition/3`. We should note, at this point, that we do not need to explicitly define a `name()` generator. Valid names for the player processes are generated from the type declaration for `name()`, as specified in Listing 5.3.

We can notice that the model state contains a dictionary data structure. Therefore, when running a command sequence, the `History` and `State` fields returned by `proper_statem:run_commands/2` will also contain dictionaries. Upon failure, the dictionaries will be printed out based on their internal representation, and this is something we would like to avoid. We decide to deal with this issue by including some pretty-printing

```erlang
-type name() :: 'bob' | 'alice' | 'joe' | 'mary'.

-record(state, {players = []          :: [name()],
                scores  = dict:new() :: dict()}).

initial_state() -> #state{}.
```

Listing 5.3: Model state representation

```erlang
command(#state{players = []}) ->
    {call,?MASTER,add_player,[name()]};
command(S) ->
    oneof([{call,?MASTER,add_player,[name()]},
           {call,?MASTER,remove_player,[name(S)]},
           {call,?MASTER,get_score,[name(S)]},
           {call,?PLAYER,play_ping_pong,[name(S)]},
           {call,?PLAYER,play_tennis,[name(S)]}]).

name(S) -> elements(S#state.players).
```

Listing 5.4: The operations to be tested

```erlang
next_state(S, _V, {call,_,add_player,[Name]}) ->
    case lists:member(Name, S#state.players) of
        false ->
            S#state{players = [Name|S#state.players],
                    scores  = dict:store(Name, 0, S#state.scores)};
        true ->
            S
    end;
next_state(S, _V, {call,_,remove_player,[Name]}) ->
    S#state{players = lists:delete(Name, S#state.players),
            scores  = dict:erase(Name, S#state.scores)};
next_state(S = #state{scores = Sc}, _V, {call,_,play_ping_pong,[Name]}) ->
    S#state{scores = dict:update_counter(Name, 1, Sc)};
next_state(S, _, _) ->
    S.
```

Listing 5.5: State transitions

```erlang
precondition(S, {call,_,remove_player,[Name]}) ->
    lists:member(Name, S#state.players);
precondition(S, {call,_,get_score,[Name]}) ->
    lists:member(Name, S#state.players);
precondition(S, {call,_,play_ping_pong,[Name]}) ->
    lists:member(Name, S#state.players);
precondition(S, {call,_,play_tennis,[Name]}) ->
    lists:member(Name, S#state.players);
precondition(_, _) ->
    true.
```

Listing 5.6: Preconditions

```
postcondition(_S, {call,_,add_player,[_Name]}, Result) ->
    Result =:= ok;
postcondition(_S, {call,_,remove_player,[Name]}, Result) ->
    Result =:= {removed, Name};
postcondition(S, {call,_,get_score,[Name]}, Result) ->
    Result =:= dict:fetch(Name, S#state.scores);
postcondition(_S, {call,_,play_ping_pong,[_Name]}, Result) ->
    Result =:= ok;
postcondition(_S, {call,_,play_tennis,[_Name]}, Result) ->
    Result =:= maybe_later.
```

Listing 5.7: Postconditions

functions in the property, so as to output more informative debugging information. The pretty-printing functions and the property to test the ping-pong system are specified in Listing 5.8.

```
pretty_history(History) ->
    [{pretty_state(State),Result} || {State,Result} <- History].

pretty_state(#state{scores = Scores} = S) ->
    S#state{scores = dict:to_list(Scores)}.

prop_ping_pong_works() ->
    ?FORALL(Cmds, commands(?MODULE),
            ?TRAPEXIT(
                begin
                    ?MASTER:start_link(),
                    {History,State,Result} = run_commands(?MODULE, Cmds),
                    ?MASTER:stop(),
                    ?WHENFAIL(
                        io:format("History: ~w\nState: ~w\nRes: ~w\n",
                                  [pretty_history(History),
                                   pretty_state(State), Result]),
                        aggregate(command_names(Cmds), Result =:= ok))
                end)).
```

Listing 5.8: Property with pretty-printing functions

Having successfully tested the stand-alone behaviour of the master, we expect the property to pass the tests. However, as we can see in Listing 5.9, the property fails along with error reports on the server crashing!

This happens because the asynchronous `play_ping_pong/1` operation introduces non-determinism in the order in which messages are received by the server. Here we can see yet another benefit of property based testing: it helps to increase our understanding about process interaction in the system under test.

Fixing the postcondition of `get_score/1` so as to achieve deterministic results is quite simple in this case. This is described in Listing 5.10.

```
7> proper:quickcheck(ping_pong_statem:prop_ping_pong_works()).
.............
=ERROR REPORT==== 30-May-2011::02:09:56 ===
<...error description...>
.
=ERROR REPORT==== 30-May-2011::02:09:56 ===
<...error description...>
..........!
Failed: After 25 test(s).
[{set,{var,1},{call,ping_pong,add_player,[alice]}},
 {set,{var,2},{call,ping_pong,play_ping_pong,[alice]}},
 {set,{var,3},{call,ping_pong,play_tennis,[alice]}},
 {set,{var,4},{call,ping_pong,play_tennis,[alice]}},
 {set,{var,5},{call,ping_pong,remove_player,[alice]}},
 {set,{var,6},{call,ping_pong,add_player,[mary]}},
 {set,{var,7},{call,ping_pong,play_ping_pong,[mary]}},
 {set,{var,8},{call,ping_pong,get_score,[mary]}},
 {set,{var,9},{call,ping_pong,add_player,[john]}},
 {set,{var,10},{call,ping_pong,play_tennis,[john]}},
 {set,{var,11},{call,ping_pong,add_player,[alice]}},
 {set,{var,12},{call,ping_pong,add_player,[bob]}},
 {set,{var,13},{call,ping_pong,play_tennis,[john]}}]
History: [{{state,[],[]},ok},{{state,[alice],[{alice,0}]},ok},
          {{state,[alice],[{alice,1}]},maybe_later},
          {{state,[alice],[{alice,1}]},maybe_later},
          {{state,[alice],[{alice,1}]},{removed,alice}},
          {{state,[],[]},ok},{{state,[mary],[{mary,0}]},ok},
          {{state,[mary],[{mary,1}]},0}]
State: {state,[mary],[{mary,1}]}
    Result: {postcondition,false}

Shrinking ........(8 time(s))
[{set,{var,6},{call,ping_pong,add_player,[mary]}},
 {set,{var,7},{call,ping_pong,play_ping_pong,[mary]}},
 {set,{var,8},{call,ping_pong,get_score,[mary]}}]
History: [{{state,[],[]},ok},{{state,[mary],[{mary,0}]},ok},
          {{state,[mary],[{mary,1}]},0}]
State: {state,[mary],[{mary,1}]}
Result: {postcondition,false}
false
```

Listing 5.9: Testing the property

```
postcondition(S, {call,_,get_score,[Name]}, Result) ->
    Result =< proplists:get_value(Name, S#state.scores);
```

Listing 5.10: Fixing the postcondition

The error reports, however, are triggered by a not-so-evident bug in the code. They are occassionaly produced when stopping the server, because of an attempt to get and subsequently kill the pid associated with a name that is actually not present in the process registry. Let us re-examine the code that is executed when stopping the server. We present it in Listing 5.11.

```
terminate(_Reason, Dict) ->
    Players = dict:fetch_keys(Dict),
    lists:foreach(fun (Name) -> exit(whereis(Name), kill) end, Players).
```

Listing 5.11: Clean-up code upon termination

The exception raised suggests that there exist some names which are stored in the server's internal dictionary, but are not associated with any (process) pid. But where do these names come from? To get the answer we have to take a look at how 'ping' messages are handled by the server. This is described in Listing 5.12.

```
handle_call({ping, FromName}, _From, Dict) ->
    {reply, pong, dict:update_counter(FromName, 1, Dict)};
```

Listing 5.12: Handling ping messages

It suggests that incoming 'ping' messages associated with names not present in the server's dictionary are actually inserted in the dictionary. When we perform an asynchronous `play_ping_pong/1` request to a player, there is a chance that this player might be removed before her 'ping' message is received by the master. In this case, when the master eventually receives the 'ping' message, the name of the removed player will be added to the dictionary, despite not being associated with any process. Having spotted the bug, we can easily fix it, as described in Listing 5.13.

```
handle_call({ping, FromName}, _From, Dict) ->
    case dict:is_key(FromName, Dict) of
        true ->
            {reply, pong, dict:update_counter(FromName, 1, Dict)};
        false ->
            {reply, {removed, FromName}, Dict}
    end;
```

Listing 5.13: Fixing the bug

And now the property successfully passes the tests, as we can see in Listing 5.14.

```
11> proper:quickcheck(ping_pong_statem:prop_ping_pong_works()).
<...1000 dots...>
OK: Passed 1000 test(s).

34% {ping_pong,add_player,1}
16% {ping_pong,remove_player,1}
16% {ping_pong,play_ping_pong,1}
16% {ping_pong,play_tennis,1}
16% {ping_pong,get_score,1}
true
```

Listing 5.14: Test case distribution of 1000 successful tests

5.2 PropEr FSM tutorial

In this tutorial, we will use PropEr to test the finite state machine implementation that we introduced in Chapter 2. The finite state machine describes the daily habbits of a strange creature that feeds on cheese, lettuce and grapes, but never eats the same kind of food on two consecutive days.

As a first example, we would like to test that the creature never runs out of food in the storage. The property for this test is specified in Listing 5.15.

```
prop_doesnt_run_out_of_supplies() ->
    ?FORALL(Cmds, proper_fsm:commands(?MODULE),
            begin
                start(cheese_day), %% could also be grapes_day or lettuce_day,
                                   %% but the same kind of day should be used
                                   %% to initialize the model state
                {History, State, Result} =
                    proper_fsm:run_commands(?MODULE, Cmds),
                stop(),
                ?WHENFAIL(io:format("History: ~w\nState: ~w\nResult: ~w\n",
                                    [History, State, Result]),
                          Result =:= ok)
            end).
```

Listing 5.15: Property for the ‘creature’ finite state machine

5.2.1 Defining the PropEr finite state machine

Let us start on a *cheese_ day* with the default portions of each kind of food available in the storage, since the initial state of our model should coincide with the initial state of the system under test. This is described in Listing 5.16.

In our specification, we will define a separate callback function for each state in the state diagram (i.e. `cheese_day/1`, `lettuce_day/1`, `grapes_day/1`). Each of these callbacks should specify a list of possible transitions from that state, as described in Listing 5.17. As we mentioned before, the state callbacks take as argument the state data. In our case, the state data is a record containing the quantity of cheese, lettuce and grapes in the food storage.

```
-type quantity() :: non_neg_integer().

-record(storage, {cheese  = 5 :: quantity(),
                  lettuce = 5 :: quantity(),
                  grapes  = 5 :: quantity()}).

initial_state() -> cheese_day.

initial_state_data() -> #storage{}.
```

Listing 5.16: Initializing the model state

```
cheese_day(_S) ->
    store_transition() ++ eat_transition() ++
        [{grapes_day,  {call,?MODULE,new_day,[grapes]}},
         {lettuce_day, {call,?MODULE,new_day,[lettuce]}}].

lettuce_day(_S) ->
    store_transition() ++ eat_transition() ++
        [{grapes_day, {call,?MODULE,new_day,[grapes]}},
         {cheese_day, {call,?MODULE,new_day,[cheese]}}].

grapes_day(_S) ->
    store_transition() ++ eat_transition() ++
        [{lettuce_day, {call,?MODULE,new_day,[lettuce]}},
         {cheese_day,  {call,?MODULE,new_day,[cheese]}}].
```

Listing 5.17: Callbacks for the states of the finite state machine

A `store_transition` is triggered every time the creature buys some food, whereas an `eat_transition` is triggered every time it is hungry. These are depicted in Listing 5.18. Both of these transitions do not change the current state of the FSM and this is specified by having `history` as the target state. In order to specify a `store_transition`, we also need generators for `food()` and `store_quantity()`. The `food()` generator is automatically derived from the corresponding type declaration.

```
-type food() :: 'cheese' | 'lettuce' | 'grapes'.

store_transition() ->
    [{history, {call,?MODULE,buy,[food(), store_quantity()]}}].

eat_transition() ->
    [{history, {call,?MODULE,hungry,[]}}].

store_quantity() ->
        range(1, 4).
```

Listing 5.18: Store and eat transitions

Initially, we do not impose any preconditions. However, postconditions are needed, as specified in Listing 5.19.

```
precondition(_From, _Target, _StateData, {call,_,_,_}) ->
    true.

postcondition(cheese_day, _, S, {call,_,hungry,[]}, Result) ->
    Cheese = S#storage.cheese,
    Cheese > 0 andalso Result =:= {cheese_left, Cheese};
postcondition(lettuce_day, _, S, {call,_,hungry,[]}, Result) ->
    Lettuce = S#storage.lettuce,
    Lettuce > 0 andalso Result =:= {lettuce_left, Lettuce};
postcondition(grapes_day, _, S, {call,_,hungry,[]}, Result) ->
    Grapes = S#storage.grapes,
    Grapes > 0 andalso Result =:= {grapes_left, Grapes};
postcondition(_From, _Target, _StateData, {call,_,_,_}, Result) ->
    Result =:= ok.
```

Listing 5.19: Preconditions and postconditions

The set of callback functions corresponding to the states of the FSM update part of the model state, since they specify the target state of a transition. In order to update the *state data*, we have to define a separate callback function, as described in Listing 5.20.

```
next_state_data(_, _, S, _, {call,_,buy,[Food, Quantity]}) ->
    case Food of
        cheese ->
            S#state{cheese = S#state.cheese + Quantity};
        lettuce ->
            S#state{lettuce = S#state.lettuce + Quantity};
        grapes ->
            S#state{grapes = S#state.grapes + Quantity}
    end;
next_state_data(Today, _, S, _, {call,_,hungry,[]}) ->
    case Today of
        cheese_day ->
            S#state{cheese = S#state.cheese - 1};
        lettuce_day ->
            S#state{lettuce = S#state.lettuce - 1};
        grapes_day ->
            S#state{grapes = S#state.grapes - 1}
    end;
next_state_data(_From, _Target, StateData, _Result, {call,_,_,_}) ->
    StateData.
```

Listing 5.20: Updating the state data

5.2.2 PropEr in action

Let us run the first tests on our property. It states that the creature never runs out of food in the storage.

```
5> proper:quickcheck(food_fsm:prop_doesnt_run_out_of_supplies()).

Error: The transition from "cheese_day" state triggered by
{food_fsm,new_day,1} call leads to multiple target states.
Use the precondition/5 callback to specify which target state should be chosen.
** exception error: too_many_targets
```

Listing 5.21: First attempt to test the property

Well, we didn't see that coming! It seems that some part of our specification is not proper enough.

PropEr allows more than one transitions to be triggered by the same symbolic call and lead to different target states. As in the case in Listing 5.22.

```
cheese_day(_S) ->
    [{grapes_day,  {call,?MODULE,new_day,[grapes]}},
     {lettuce_day, {call,?MODULE,new_day,[lettuce]}}].
```

Listing 5.22: Selected transitions from a cheese_day

However, the precondition callback may return true for at most one of these target states. Otherwise, PropEr will not be able to detect which transition was chosen and an exception will be raised. In our case, we have to specify the preconditions described in Listing 5.23 to associate each possible argument of `new_day/1` with the correct target state.

```
precondition(Day, Day, _, {call,_,new_day,_}) ->
    false;
precondition(_, grapes_day, _, {call,_,new_day,[grapes]}) ->
    true;
precondition(_, cheese_day, _, {call,_,new_day,[cheese]}) ->
    true;
precondition(_, lettuce_day, _, {call,_,new_day,[lettuce]}) ->
    true;
precondition(_, _, _, {call,_,new_day,_}) ->
    false;
precondition(_, _, _, {call,_,_,_}) ->
    true.
```

Listing 5.23: Preconditions

If we run the test again, finally we get some results. It seems that there is always some food in the storage. But... wait a minute! We cannot be sure of what was tested unless we have a look at the test case distribution. Listing 5.24 describes how to instrument the property so as to collect statistics about how often each transition is tested. The resulting test case distribution is depicted in Listing 5.25.

```
prop_doesnt_run_out_of_supplies() ->
    ?FORALL(Cmds, proper_fsm:commands(?MODULE),
            begin
                start(cheese_day), %% could also be grapes_day or lettuce_day,
                                   %% but the same kind of day should be used
                                   %% to initialize the model state
                {History, State, Result} =
                    proper_fsm:run_commands(?MODULE, Cmds),
                stop(),
                ?WHENFAIL(io:format(''History: ~w\nState: ~w\nResult: ~w\n'',
                                    [History, State, Result]),
                          aggregate(zip(proper_fsm:state_names(History),
                                        command_names(Cmds)),
                                   Result =:= ok))
            end).
```

Listing 5.24: Collecting statistics on test case distribution

It seems that the creature doesn't get hungry very often in our tests. Thus, our previous conclusion about everlasting food supplies cannot be really trusted. We can instruct PropEr to choose `hungry/0` calls more often by assigning weights to transitions, as described in Listing 5.26.

The result of running the tests with weighted transitions are presented in Listing 5.27.

As we can see now, in case of non-stop eating the creature eventually runs out of food. What is more, the content of the `State` variable reveals that the quantity of available food starts getting negative values. We will correct the code to prevent this from happening, as described in Listing 5.28.

```
12> proper:quickcheck(food_fsm:prop_doesnt_run_out_of_supplies()).
...........................................................................
..........................
OK: Passed 100 test(s).

20% {cheese_day,{food_fsm,new_day,1}}
14% {grapes_day,{food_fsm,new_day,1}}
14% {lettuce_day,{food_fsm,new_day,1}}
 9% {cheese_day,{food_fsm,hungry,0}}
 9% {cheese_day,{food_fsm,buy,2}}
 8% {lettuce_day,{food_fsm,buy,2}}
 8% {grapes_day,{food_fsm,buy,2}}
 7% {grapes_day,{food_fsm,hungry,0}}
 7% {lettuce_day,{food_fsm,hungry,0}}
true
```

Listing 5.25: How often each transition is tested?

```
weight(_Today, _Tomorrow, {call,_,new_day,_}) -> 1;
weight(_Today, _Today, {call,_,hungry,_}) -> 3;
weight(_Today, _Today, {call,_,buy,_}) -> 2.
```

Listing 5.26: Assigning weights to transitions

```
15> proper:quickcheck(food_fsm:prop_doesnt_run_out_of_supplies()).
..........................!
Failed: After 27 test(s).
<... 13 commands ...>
History: <... long execution history ...>
State: {grapes_day,{storage,4,8,-1}}
Result: {postcondition,false}

Shrinking .....(5 time(s))
[{set,{var,4},{call,food_fsm,hungry,[]}},
 {set,{var,8},{call,food_fsm,hungry,[]}},
 {set,{var,9},{call,food_fsm,hungry,[]}},
 {set,{var,11},{call,food_fsm,hungry,[]}},
 {set,{var,12},{call,food_fsm,hungry,[]}},
 {set,{var,13},{call,food_fsm,hungry,[]}}]
History: [{{cheese_day,{storage,5,5,5}},{cheese_left,5}},
          {{cheese_day,{storage,4,5,5}},{cheese_left,4}},
          {{cheese_day,{storage,3,5,5}},{cheese_left,3}},
          {{cheese_day,{storage,2,5,5}},{cheese_left,2}},
          {{cheese_day,{storage,1,5,5}},{cheese_left,1}},
          {{cheese_day,{storage,0,5,5}},{cheese_left,0}}]
State: {cheese_day,{storage,-1,5,5}}
Result: {postcondition,false}
false
```

Listing 5.27: Running the test with weighted transitions

```
cheese_day(eat, Caller, #storage{cheese = Cheese} = S) ->
    gen_fsm:reply(Caller, {cheese_left, Cheese}),
    case Cheese > 0 of
        true ->
            {next_state, cheese_day, S#storage{cheese = Cheese - 1}};
        false ->
            {next_state, cheese_day, S}
    end.

lettuce_day(eat, Caller, #storage{lettuce = Lettuce} = S) ->
    gen_fsm:reply(Caller, {lettuce_left, Lettuce}),
    case Lettuce > 0 of
        true ->
            {next_state, lettuce_day, S#storage{lettuce = Lettuce - 1}};
        false ->
            {next_state, lettuce_day, S}
    end.

grapes_day(eat, Caller, #storage{grapes = Grapes} = S) ->
    gen_fsm:reply(Caller, {grapes_left, Grapes}),
    case Grapes > 0 of
        true ->
            {next_state, grapes_day, S#storage{grapes = Grapes - 1}};
        false ->
            {next_state, grapes_day, S}
    end.
```

Listing 5.28: Correcting our code

Let us now assume that the creature is wise enough to take care of buying food before it gets hungry. This can be modeled by adding the precondition specified in Listing 5.29. Moreover, we can instruct PropEr to choose an 'eat_transition' only when there is food available, as described in Listing 5.30.

```
precondition(Today, _, S, {call,_,hungry,[]}) ->
    case Today of
        cheese_day ->
            S#storage.cheese > 0;
        lettuce_day ->
            S#storage.lettuce > 0;
        grapes_day ->
            S#storage.grapes > 0
    end;
```

Listing 5.29: Additional precondition

```
cheese_day(S) ->
    store_transition() ++ eat_transition(S#storage.cheese) ++
        [{grapes_day,  {call,?MODULE,new_day,[grapes]}},
         {lettuce_day, {call,?MODULE,new_day,[lettuce]}}].

lettuce_day(S) ->
    store_transition() ++ eat_transition(S#storage.lettuce) ++
        [{grapes_day, {call,?MODULE,new_day,[grapes]}},
         {cheese_day, {call,?MODULE,new_day,[cheese]}}].

grapes_day(S) ->
    store_transition() ++ eat_transition(S#storage.lettuce) ++
        [{lettuce_day, {call,?MODULE,new_day,[lettuce]}},
         {cheese_day,  {call,?MODULE,new_day,[cheese]}}].

eat_transition(Food_left) ->
    [{history, {call,?MODULE,hungry,[]}} || Food_left > 0].
```

Listing 5.30: Introducing conditional transitions

The property now successfully passes 1000 tests, as we can see in Listing 5.31.

```
18> proper:quickcheck(food_fsm:prop_never_run_out_of_supplies(), 1000).
<...1000 dots...>
OK: Passed 1000 test(s).

19% {cheese_day,{food_fsm,hungry,0}}
13% {cheese_day,{food_fsm,buy,2}}
13% {cheese_day,{food_fsm,new_day,1}}
11% {lettuce_day,{food_fsm,hungry,0}}
11% {grapes_day,{food_fsm,hungry,0}}
 8% {lettuce_day,{food_fsm,new_day,1}}
 8% {grapes_day,{food_fsm,new_day,1}}
 7% {lettuce_day,{food_fsm,buy,2}}
 7% {grapes_day,{food_fsm,buy,2}}
true
```

Listing 5.31: Test case distribution of 1000 successful tests

Chapter 6

Conclusion

6.1 Concluding remarks

In this thesis we have presented our implementation of two library modules, proper_statem and proper_fsm, written on top of the core PropEr system. Our contribution lies in providing open-source support for random model-based testing of stateful reactive Erlang systems. We have also provided detailed tutorials on the use of the new 'stateful testing' subsystem of PropEr. Although it is quite early to draw conclusions about the effectiveness and robustness of our implementation, the first signs seem quite promising. PropEr (including our extensions) was properly announced[1] in Erlang Factory London 2011. In another talk given at that same conference, PropEr was included among the 'cool tools' for testing Erlang programs, especially for detecting failing edge-case scenarios.[2] PropEr is currently used as a testing tool in open-source projects, including the AMQP messaging protocol implementation RabbitMQ and Mochiweb, an Erlang library for building HTTP servers.

6.2 Future work

In the future, we aim to provide a tighter integration of PropEr with other testing tools for Erlang. One of these tools is Concuerror [10], a testing tool for concurrent Erlang programs. Concuerror relies on systematically exploring process interleaving to reveal concurrency-related errors. To be able to force desired interleaving sequences, Concuerror comes with a custom scheduler that takes care of controlling the order in which the commands of the various processes are executed. We believe that a proper integration of our tool with Concuerror would allow greater control over scheduling of parallel commands. Therefore, parallel statem testing with PropEr could be effectively performed in a fully automated way.

Providing control over scheduling of parallel commands seems a prerequisite, so that our tool can be used in practice to detect race conditions. Aside from this more pressing concern, we would like to experiment with the number of concurrent processes used for

[1]`http://proper.softlab.ntua.gr/talks/`
[2]`http://etrepum.github.com/erl_testing_2011/`

parallel testing or provide additional ways of generating parallel test cases. Since constraints can be easily moved from preconditions to postconditions when switching from positive to negative testing, in some cases it might be preferable to eliminate preconditions from state machine specifications used for parallel testing. Such an approach is interesting, as it would also eliminate the need to produce all possible command interleavings at generation time.

Bibliography

[1] J. Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, Department of Microelectronics and Information Technology, The Royal Institute of Technology Stockholm, December 2003.

[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* The Pragmatic Bookshelf, 2007.

[3] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, pages 2–10. ACM, 2006.

[4] K. Beck. *Test Driven Development: By Example.* Addison-Wesley Professional, 2002.

[5] J. Boberg. Early Fault Detection with Model-Based Testing. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 9–20. ACM, 2008.

[6] F. Cesarini and S. Thompson. *Erlang Programming.* O'Reilly Media, 2009.

[7] K. Claessen and J. Hughes. QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 268–279. ACM, 2000.

[8] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-Based Testing in Practice. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE)*, page 285. ACM, May 1999.

[9] I. K. El-Far and J. A. Whittaker. Model-based Software Testing. *Encyclopedia on Software Engineering*, 2001.

[10] A. Gotovos, M. Christakis, and K. Sagonas. Test-Driven Development of Concurrent Programs using Concuerror, June 2011. To appear in the 2011 ACM SIGPLAN Workshop on Erlang.

[11] J. Hughes. QuickCheck Testing for Fun and Profit. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*, pages 1–32. PADL, 2007.

[12] K. Klaessen, M. Palka, N. Smallbone, J. Huges, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and Pulse. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 149–160. ACM, 2009.

[13] M. Papadakis. Automatic random testing of function properties from specifications. Master's thesis, Department of Computer Science, School of Electrical and Computer Engineering, National Technical University of Athens, October 2010.

[14] M. Papadakis, E. Arvaniti, and K. Sagonas. PropEr: an open-source, Quickcheck-inspired property based tool for Erlang. `http://proper.softlab.ntua.gr`.

[15] M. Papadakis and K. Sagonas. A PropEr Integration of Types and Function Specifications with Property-Based Testing, June 2011. To appear in the 2011 ACM SIGPLAN Workshop on Erlang.

[16] K. K. Thorup. Triq: Trifork QuickCheck. `http://krestenkrab.github.com/triq/`.

[17] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-Based Testing. Working paper, April 2006.

[18] U. Wiger. Erlang Programming for Multi-core, March 2009. Slides presented in QCON London 2009 conference.