CHALMERS

Testing Erlang Concurrency with QuickCheck

Master of Science Thesis in Program Software Engineering and Technology

ZICHEN CAO

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, August 2011

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Testing Erlang Concurrency with QuickCheck

Zichen Cao
© Zichen Cao, August 2011.

Examiner: John Hughes

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Supervisor: Nick Smallbone

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2011

Contents

1 Introduction 1
1.1 Preceding Work 1
1.2 Purpose 1
1.3 Scope 1
1.4 Tool 2
1.5 Contribution 2
1.6 Incompleteness 2
2 QuickCheck 3
2.1 Three Steps of Testing 3
2.2 Generator 4
2.3 Symbolic Representation 5
3 QuickCheck State Machine 6
3.1 Symbolic Commond 6
3.2 State 6
3.3 Callback Functions 7
3.4 Property 9
3.5 Parallel Testing 9
4 Model Based Testing 10
5 Testing ETS 11
5.1 State in ets_tests 12
5.1.1 Main State 12
5.1.2 continuation 12
5.1.3 matchcont 13
5.2 Generator in ets_tests 13
5.3 Run tests 14
5.4 Findings 14
5.4.1 ets:tab2list/1 14
5.4.2 Use of ets:safe_fixtable/2 17
5.4.3 Separating ets:match_object/2 18
5.4.4 An error report of ets:insert/2 and ets:delete_all_objects/1 20
5.5 Counter example 21
5.6 eqc:recheck/1 22
6 PULSE 23
6.1 instrument:c/1 23
6.2 ?SCHEDULED 23

7 Testing DETS 24
7.1 State in dets_tests 25
7.1.1 Main state 25
7.1.2 continuation 25
7.2 Generator in dets_tests 25
7.3 Use of PULSE 26
7.4 d_ets 26
7.5 dets:open_file/2 26
7.6 Run tests 26
7.7 Atomicity in DETS 27
7.8 Findings 27
7.8.1 dets:match_object/2 & dets:match_delete/1 27
7.8.2 An error report of dets:insert/2 & dets:delete_all_objects/1 29
7.8.3 d_ets:insert/2 & d_ets:match_delete/2 30
7.8.4 dets:match_object/3 31
7.8.5 dets:match_object/1 32
8 Testing Supervisor 37
8.1 State in supervisor_tests 37
8.1.1 Main State 37
8.1.2 Relation and Unrelaiton 38
8.1.3 Pid and OldPids 38
8.2 Generator in supervisor_tests 39
8.3 Use of PULSE 39
8.4 super_visor 40
8.5 Run tests 40
8.6 Difficulties 40
8.6.1 Restart strategy of Supervisor 40
8.6.2 Restart of Child 41
8.6.3 exit/2 41
8.7 Finding 41
8.7.1 exit/2 with reason shutdown 41
8.7.2 Race condition with exit/2 41
8.7.3 supervisor:terminate_child/2 42
9 Conclusion 44

List of Figures

Figure 3-1 **Process during generation step**........ 8
Figure 3-2 **Process during execution step**........ 9
Figure 4-1 **Model Based Testing** 10
Figure 4-2 **Example of Model Based Testing**........ 10

List of Tables

Table 5-1 **Main State of ets_tests** ... 12
Table 5-2 **continuation in Main State of ets_tests** ... 13
Table 5-3 **Generator in ets_tests** ... 13
Table 5-4 **matchcont in Main State of ets_tests** ... 20
Table 7-1 **Main State of dets_tests** ... 25
Table 7-2 **continuation in Main State of dets_tests** ... 25
Table 7-3 **Generator in dets_tests** ... 26
Table 8-1 **Main State of supervisor_tests** ... 37
Table 8-2 **Generator in supervisor_tests** ... 39

Abstract

Erlang is a functional language developed by Ericsson AB, in which concurrency belongs to the programming language rather than the operating system. It can make parallel programing much easier by modeling the program as several processes running in parallel which interact with each other only via exchanging messages. There is no shared memory in Erlang, due to its advantages on message passing, Erlang have been widely used in development of telecom and internet products. As the importance of usage of` Erlang increases, to ensure the progams operate as they should becames the most significant and challenging task. QuickCheck is a specification-based testing tool produced by the company Quivq AB. The commercial version of QuickCheck can support generating random test cases for Erlang programs. It offers the ability for test programmer to test Erlang functions by specifying their expected operations and results. It is a tool that can liberate the test programmers from heavy work of writing test cases by hand, and reduce the time that the test programmers spend on simplifying the failing test cases after the execution of the test cases.

In the first part of this thesis, we develop specification of main functions in ETS, DETS, by analyzing their operation and results. Then, we test these functions by using QuickCheck state machine. In the second part, with analyzing the shortage of QuickCheck state machine, we modify the property, and test DETS by using PULSE. Meanwhile, main functions in Supervisor are specified and tested as well. Bugs and interesting findings from this project are described in relevant parts as well.

Acknowledgement

This thesis project will not be possible finished, without supports from many people, and I would like to thank them all here

Firstly, I would like to deeply thank professor Dr. John Hughes, who offer the proposal of this thesis project and provided me a great opportunity to work on testing Erlang Concurrency with QuickCheck provided by him.

Secondly, I would like to express my sincere thank to my supervior, Nick Smallbone, who encouraged me to be strategic in the entire processes of the project, provided me a lot of brillent suggestions when I was at a loss of what to do the next step, and solutions when I met difficults through out all the project.

Additionally, I would like to thank my dear family who constantly encourage and support me to pursue my study in Sweden. Thank you for give me enough space and endless love I needed during these years studies.

1 Introduction

Erlang is a language where concurrency belongs to the programming language and not the operating system [1]. It can make the parallel programing much easier by modeling the program as several processes running in parallel which interact with each other only via exchanging messages. There is no shared memory in Erlang, due to which, parallel processes can run without locks, synchronized methods and possibility of shared memory corruption. Erlang program can be made from thousands to millions of extremely lightweight processes that can run on a single processor, a multicore processor, or a network of processor [1].

Due to its advantages on message passing, Erlang have been widely used in development of telecom and internet products such as switchs, email gateways, instant messaging services, semi-structured databases and game server [2]. As the importance of usage of` Erlang increases, to ensure the progams operate as they should becames the most significant and challenging tasks.

1.1 Preceding Work

There were some proceding work have already been done in testing Erlang with QuickCheck. The pure functions and imperative functions in various data-structure models have been tested with QuickCheck [3] by Crystal Chang Din (2009) [2]. The test suites with properties and the bugs found in Erlang/OTP were clear described. There were three bugs found in module digraph and one bug found in module DETS and they had been reported to Erlang Community.

1.2 Purpose

In this thesis project, the tasks focus on testing Erlang concurrancy with using QuickCheck state machine and PULSE, which aim to increase the possibility of the emergence of race condition when a set of processess running simultaneously. For example, testing the functions in module ETS by letting several processes work in the same table at the same time.

1.3 Scope

In this thesis project, there are three parts of tests being included:

The first part is testing the main functions in Module ETS. This module is an interface to the Erlang built-in term storage BIFs [6]. It provides large key-value lookup tables, which provide the ability store very large amount of data in an Erlang runtime system. Meanwhile, it supports constant access time to the data.

The second part is testing the functions in Module DETS with both QuickCheck and PULSE. Similar to ETS, this module is a collection of objects, but provides term storage on file.

The third part is testing the functions in Module Supervisor with both QuickCheck and PULSE. It is a behavior module for implementing a supervisor process to supervise other child processes.

1.4 Tool

In this thesis project, the QuickChick and PULSE will be the testing tools mainly used. In Section 2, we will give a detailed description of QuickCheck State Machine [3] and PULSE [7], as well as how it works. The Erlang Shell will be used as the main tool for running the properties.

1.5 Contribution

After the detailed description of QuickCheck in second section, the contribution of this thesis will be presented. The main functions in modules ETS, DETS, and Supervisor are tested. Meanwhile, the corresponding interesting issues of these functions and the found bugs are presented as well.

1.6 Incompleteness

In this thesis projet, instead of testing all the functions in relevant modules, only the most often used functions are tested, the rest functions omited in this project need to be considered in future work.

During testing ETS and DETS, there are some functions opreate unpredicablely, in terms to the atomicity. These functions can be modified into two separated functions, which can solve the problem. However, due to the time limitation in this project only one function ets:match_object/2 is modified as an example.

2 QuickCheck

QuickCheck is a specification-based testing tool produced by the company Quviq AB, which was found by John Hughes (CEO) and Thomas Arts (CTO). The programs are tested by writing properties in the source code. This tool can support generating random test cases for Erlang and C program, etc. It offers the ability for test programmer to test software by specifying their expected behaviors and results of the tested systems. The generator can generate 100 random test cases each time automatically, when applying QuickCheck. With using eqc:numtests/2, the user can customize the number of test cases as well. Then the random generated test cases will be executed and checked against the specification that ought to hold by the system to accomplish the test efficiently. It is a tool can take test programmers out of the drudgery of the software testing.

The advantages that QuickCheck provides include:

- Liberating the test programmers from heavy work on writing test cases by hand,
- Reducing the time that the test programmers spend on simplifying the failing test cases after the execution of the test cases.

These advantages can reduce the time spent on testing the systems and improve the quality of the tested systems as well.

2.1 Three Steps of Testing

QuickCheck takes the test works quickly from the specification to identified bugs via the steps described below [4]:

- **Step 1:**
  Instead of writing traditional test cases that the system should satisfy, QuickCheck specification will be written by the programmer. It should consist of both properties and generators, which can specify the expected operations of system. And then, the properties can generate random test cases at one time, based on what are specified before. Testing with generated random test cases can make the testing more thorough.

- **Step 2:**
  QuickCheck uses controlled random generation to test the code against the specification [4]. The interface of QuickCheck is simple and powerful, which can ensure that it can generate appropriate complex data and better distribution of test cases.

  The test programmer can easily define the generator by composing simple ones, for instance: defining the generator gen_ttype/0 as below to generate type of ets table in this study,

```
gen_ttype() ->
    oneof([set, ordered_set, bag, duplicate_bag])
```

This generator uses oneof to choose one of the ets table types from the list [set, ordered_set, bag, duplicate_bag]. Then the type generated by this generator will be used when generating ETS table. In this case, the type of ETS table can be will controlled according to the defined needs.

- **Step 3:**
  When the tests fail, and failing cases are found, QuickCheck can automatically simplify the complex failing cases into simplest examples, which can provoke the failure. This feature makes the counter examples much clearer to read and understand which make the analysis of the failure much easier.

2.2 Generator

In QuickCheck, the generators randomly generate data for the test case and have built-in shrinking behavior [5], all the basic generators are defined in *eqc_gen* module, for example:

int ( )	Generate a random integer
bool ( )	Randomly generate *true* or *false*
char ( )	Generate a random character, shrink to a, b or c
list (int ( ))	Generate a list of random length with randomly chosen integers

QuickCheck permits the constants to be used as generators for their own value, and permit tuples, records, and lists containing generators to be used as generators for values of the same form, for example:

```
{ int( ), bool( ) }
```

is a generator for generating random pairs of integer and booleans. This feature enable the programmer to define data generators based on basic generators and generator constructors [5], for example:

```
person( ) -> { name = name ( ),
               gender = oneof ( [male, female] )
               age = choose ( 0, 100 )}.
```

in the code above, *name ( )* is a user defined generator, gender and age use basic generators: *oneof* and *choose.*

Many functions defined in the *eqc_gen* are usually used via macros. In this project there are several macros being frequently used. For example, the generator gen_object/1 is used both in ets and dets tests to generate and restrict the objects further used in tests:

```
gen_object(S) ->
   frequency([
```

```
{1,?LET(ListOfObjects, ?SUCHTHAT(L,
        list(?LET(Tuple,oneof([{char()}|S#state.objects]),
        oneof(tuple_to_list(Tuple)))),L/=[]),list_to_tuple(ListOfObjects))},
{6, oneof([{char()}|S#state.objects])},
{1, ?LET(KeyObject, key(S),
        ?LET(Rest, list(char()), list_to_tuple([KeyObject|Rest])))}])
```

The action of this test programmer defined generator is to generate an object, which is a tuple.

?SUCHTHAT here generate a list of characters, such that the list is not an empty list, and bind this list to L.

?LET here generates a list of characters from ?SUCHTHAT and binds it to ListOfObject, then generate the needed object by converting it to a tuple, using *list_to_tuple/1*.

This generator can be run and generate objects as below:

```
1> eqc_gen:sample(ets_tests:gen_object(S)).
{181,178,254,160}
{188,170,223}
…….      (Six generated objects skip here)
{242}
{86,220,197}
ok
```

2.3 Symbolic Representation

Symbolic representations are used when function calls are some elements of generators. It makes the counter examples readable, further helps the test analyst understand test data much more easily. It is always constructed in a format of a tuple with several elements: {call, module, function, [Args]}. It defines the function been called and the module where it comes from, as well as the arguments that the function use.

For example the symbolic call below

```
{call, ets, insert, [ Tab, Object]}
```

It represents a call of function 'insert' from module 'ets' with passing two parameters 'Tab' and 'Object'. The operation of this function is to insert the Object into an ETS table, name of which is Tab.

3 QuickCheck State Machine

QuickCheck State Machine was produced for testing the functions with side-effect. It generates random sequence of calls, which exercise many more possibilities than is practical manually; then tests against a predefined specification; finally, simplifies the failing sequences automatically.

3.1 Symbolic Commond

In QuickCheck State Machine, test cases are generated in a symbolic form, representing as lists of symbolic commands, which generated by *command/1*. Each of the symbolic commands binds a symbolic variable to the result of a symbolic function call, for example:

{set, {var, 1}, {call, ets, insert, [Table, {97}]}}

This command above sets variable {var, 1} to the result of the symbolic call: *ets:insert/2*. When this test case runs, the symbolic calls are performed, and the symbolic variable will be replaced by the actual value it was set to.

To understand a program effectively often requires thoroughly thinking about the possible histories, and side-effect makes it harder. Symbolic representation can enable the test programmer to display, analyze and save the test case much easier.

3.2 State

The client module defines an initial state at the beginning of the test, and how the state will be changed by each command. For example, if the test case call ets:insert(Tab, ObjectOrObjects) to insert a number of objects into the ETS table, then the state will be a list of all the objects have been inserted. The state is used both the test case generation and its execution. During the test case generation phase, S#state.objects is constructed as a part of symbolic state as:

[{var, 1}, {var, 2}, {var, 3}]

It means that the table contains three variables {var, 1}, {var, 2}, {var, 3}, which are symbolic representations. Then, during the execution phase the corresponding dynamic state will be computed, and in this case, a list of actual objects that generated by generator gen_object/1 are returned. The Dynamic states always have the same structure as their corresponding symbolic states, but with symbolic variables and calls replaced their values, which are three objects in this case.

It is also not necessary to track and store all the information in the test case in the state. Only the information that will be further analyzed needs to be included. For example, when spawning a number of processes, the names and Pids of which might be the necessary information will be used later, rather than their specification, etc. In this case, a state with a list of Pids stored should be enough. Whereas, in the module supervisor_tests.erl in this

project, the name and detailed specification are saved for further use, as well as the Pid of process. Even a list of OldPids is involved to check the restart strategies of supervisor.

3.3 Callback Functions

There are several callback functions being written in this project, which include: initial_state, precondition, command, postcondition, and next_state. These functions will be further called by the property, during the testing. The detailed descriptions of these functions are described as below.

- **initial_state**

  Symbolic state returned when test case starts. It is evaluated to construct the initial dynamic state before the test case is executed. For example, in this study, all the initial states are empty list that used for storing relevant information will be further analyzed.

  ```
  -record(state, {objects, type, continuation}).
  ```

  The code above is an example from module dets_tests.erl in this thesis, used for storing the information of DETS table test case. In the state, *object* is used to store the objects inserted in the table, *type* is used to store the type of the table, *and continuation* is used to save the relevant information of continuation returned by the command:

  ```
  {call, dets, match_object, [TableId, Pattern, Limit]} and
  {call, dets, match_object, [Continuation]}
  ```

  In the beginning, there should be no information stored in the table except the type of the table, which will be generated at the beginning of the test case. Therefore, the state is initialized as

  ```
  #state{objects=[], type=Type, continuation=[]}
  ```

  The Type above in DETS table can be one of the following: set, bag, duplicate_bag.

- **precondition**

  It will return true if the symbolic call is valid to perform in the state. After checking the command generated by the *command* generator during the generation step, *precondition* is used to filter and check the inclusion of these commands. The command can be included in the command sequence, only when the return value is true.

  The following two lines of code illustrate an example codes used in the module *supervise_tests.erl* in this project:

  ```
  precondition (S, {call, supervisor, start_child, [SupRef, ChildSpec]}) ->
          lists:member(SupRef, S#state.supervisor) == true;
  ```

The command {call, supervisor, start_child, [SupRef, ChildSpec]} will be added into the command sequence, only if the SupRef is a member of the *S#state.supervisor* in the state, which means that the type of SupRef should be supervisor, rather than worker.

- **command**

  This function is used to generate symbolic function calls appearing in the next step in the test case. The command generated will be checked first by the function precondition, it will be included into the command sequence, only if the precondition returns true.

- **postcondition**

  The postcondition will check the result of relevant symbolic calls, and the actual value will be passed and checked.

```
postcondition(S, {call, ets, lookup, [_TableId, Key]}, R) ->
    lists:sort(R) = = lists:sort(lookup_list(Key, S#state.objects))
```

  In the example above, the QuickCheck checks the result R of symbolic call

```
{call, ets, lookup, [ TableId, Key]}
```

  If the returned value R is one of the objects from the ETS table, then corresponding *next_state/1* function will be applied to modify the dynamic state.

- **next_state**

  The next_state function is a transition function of QuickCheck State Machine; it is changed twice during both test generation and test execution steps. During the test generation step, if precondition returns true, the state will be updated with symbolic values with the format as {var, 2}. During the execution step, after the postcondition being checked, and if return value are true, the next state will change the state based on its specification. Take the command: {call, ets, insert, [TableId, Object]} as an example, the states will changed as the figure illustrate below:

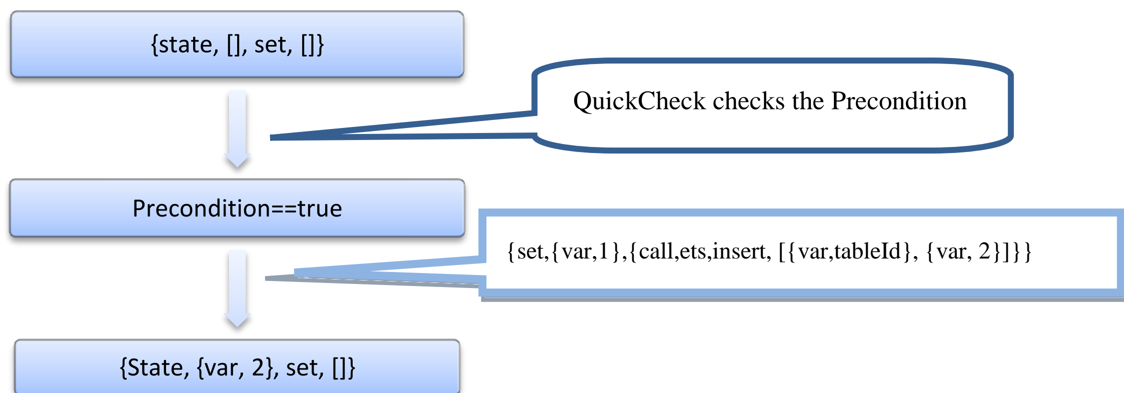

Figure 3-1 **Process during generation step**

During the test case generation step, only symbolic variables will be generated.

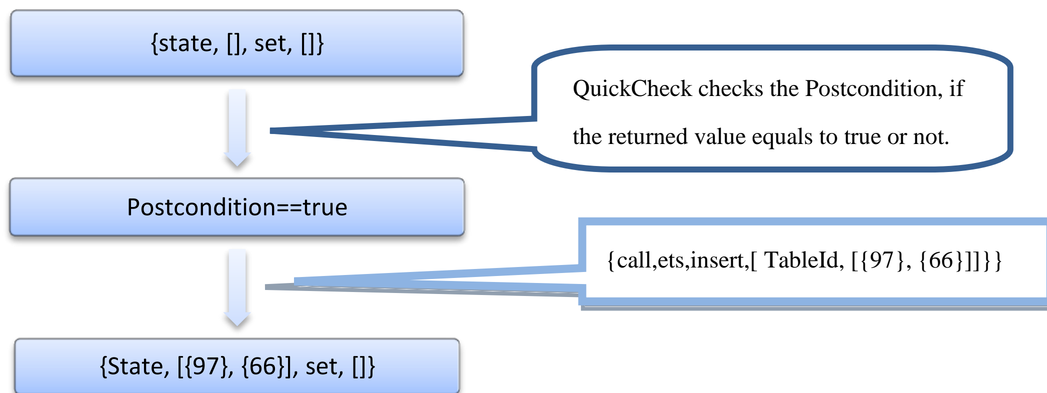

Figure 3-2 **Process during execution step**

During the test case execution step, the real value which generated by generator *gen_object(S)* will be insert into the table, and the state will be modified by corresponding specifications.

3.4 Property

QuickCheck doesn't define any property to test. However, it provides functions to make defining property easier [3]. Within the client defined property, {H, S, Result} will be used to save the result of function *run_commands/2* or *run_parallel_commands/3*. The elements in the tuple refer to *history*, *dynamic_state*, and *reason* respectively.

3.5 Parallel Testing

Initially, QuickCheck only provides the ability to generate test cases, which are executed sequentially. In order to test for race condition, test cases should be generated and executed in parallel. QuickCheck module *eqc_statem* provides new functions *parallel_commands/2* generate a parallel test case consists of a sequential prefix, followed by a list of concurrent tasks, and *run_paralllel_commands/3* to run them. By executing the prefix first in a normal way, QuickCheck then execute the concurrent tasks in newly spawned processes. [3] The parallel testing provides the ability to check the atomicity of functions.

The property of parallel testing is alike in the appearance with it in the sequential testing, only with *commands/2* and *run_command/3* replaced by corresponding parallel versions.

In order to increase the possibility of testing race conditions, which only occur sometimes, ?ALWAYS is used to repeat each test several times to ensure to provoke it.

These functions support parallel testing and make the testing of concurrency of the Erlang feasible, and therefore become the backbone of this thesis project.

Detailed description of how to apply the functions provided by module *eqc_statem* into this study and the consequences of using them will be presented in the following Section 5, 7and 8.

4 Model Based Testing

In order to make the test trust worth, model based testing is used in this thesis. The Figure below visualizes how the concept is applied and works:

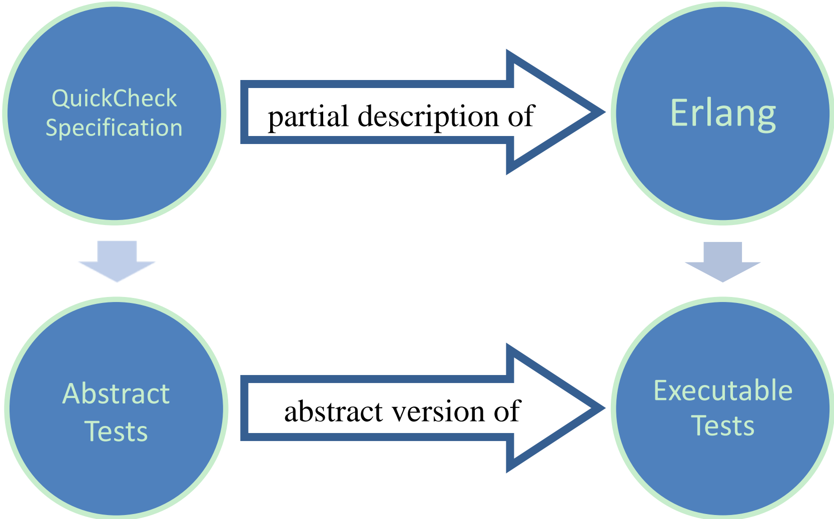

Figure 4-1 **Model Based Testing**

In the QuickCheck specification, the expected operations and results of the specific function are specified by using next_state/1 functions, as well as how to change the data stored for further use and the state to save the relevant data. The results get from real test cases, will be checked by comparing the result with what is specified in the postcondition.

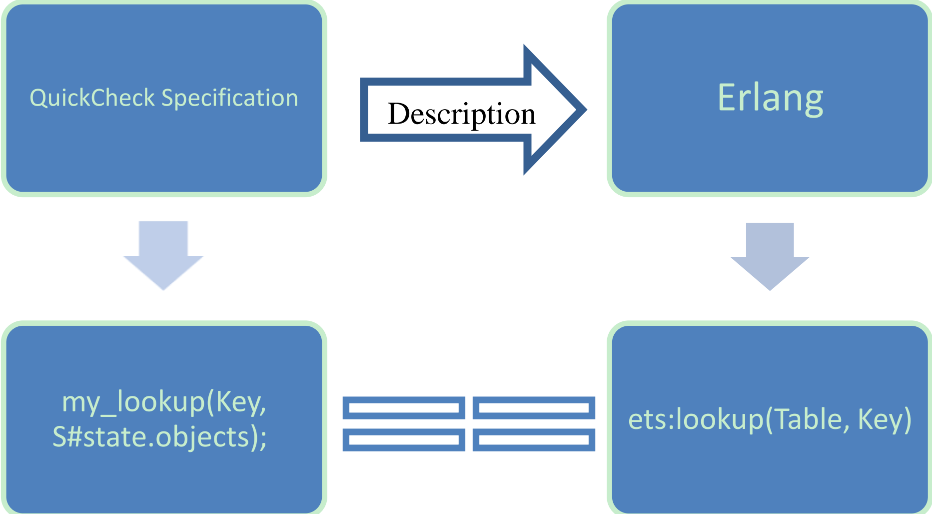

Figure 4-2 **Example of Model Based Testing**

The example in Figure 4-2 illustrates how the theory of Model Based Testing is used in the test case. In the module ets_tests.erl, a part of state S#state.objects is used to save the objects inserted into the ETS table, customized function ets_tests:my_loolup/2 is used to find the objects with the given Key, from the state. The result is compared with the returned value by calling function ets:lookup/2 with comparing operator '=='. The test case will pass, only when the returned value of the checking expression returns true.

5 Testing ETS

The first part of testing in this thesis is testing the concurrency in ETS table. ETS provides large key-value lookup tables, which provide the ability to store very large quantities of data in an Erlang runtime system. Data stored in a ETS table is transient, which means that they will be deleted as soon as the ETS table concerned is disposed of, for example, when the process which creates the ETS table terminates, the table is automatically destroyed, meanwhile, the data stored in the table are deleted as well. [6]

The types of the ETS table can be four different types: *set, ordered_set, bag* and *duplicate-_bag*. A set or ordered_set table can only have one object associated with each key, whereas, a bag or duplicate_bag can have many objects associated with the same key.

The default key of an object in ETS table is the first element in the inserted tuple. In this thesis, by considering the purpose of this thesis is finding race condition, the key position is not changed.

The functions being tested include:

- ets:insert(Tab, ObjectOrObjects) -> true
  Insert an Object of Objects into the table Tab.
- ets:lookup(Tab, Key) –> (Object)
  Return a list of objects from the table Tab, with the same Key.
- ets:delete(Tab, Key) –> true.
  Delete all the objects from the table Tab, with the same Key.
- ets:delete_all_objects(Tab) –> true.
  Delete all of the objects from table Tab
- ets:delete_object(Tab, Object) –> true.
  Delete the Object from the table Tab.
- ets:match_delete(Tab, Pattern) –> true.
  Delete all the objects, which match the Pattern, from the table Tab.
- ets:match_object(Continuation) –> {{Match}, Continuation|'$end_of_table'}.\
  Return a number of object aginest the pattern that specified when generating the Continuation.
- ets:match_object(Tab, Pattern) –> (Object).
  Return all the objects, which match the Pattern, from the table Tab.
- ets:match_object(Tab, Pattern, Limit) –> {{Match}, Continuation|'$end_of_table'}.
  Match the objects in the table Tab against the Pattern, and return a number Limit of them.
- ets:tab2list(Tab) –> (Object)

Return a list of all Objects from the table Tab.

In this thesis, module ***ets_tests.erl*** is used to store all specifications used in ETS test.

An example of operation of functions ets:match_object/3 and ets:match_object/3 is described below:

```
9> Tab=ets:new(test, [set]).
20496
10> ets:insert(Tab, [{a,a,a}, {b,b,b}]).
true
11> {_, C}=ets:match_object(Tab, {'_','_','_'}, 1).
{[{a,a,a}],{20496,113,1,<<>>,[],0}}
12> ets:match_object(C).
{[{b,b,b}],'$end_of_table'}
```

In this example, objects {a,a,a} and {b,b,b} are inserted into the table Tab, when call function ets:match_object/3, the object {a,a,a} is returned against the pattern {'_','_','_'}, according to the Limit 1. Then, function ets:match_object/1 is called, and the object {b,b,b} is continuously returned. '$end_of_table' indicates that there is no object against the given pattern can be continuously returned.

5.1 State in ets_tests

5.1.1 Main State

All the states are saved as lists of information. The state used in ets_tests include four parts: objects, type, continuation, and matchcont with detailed description show as the table below:

Table 5-1 **Main State of ets_tests**

Parts in State	Description
Objects	Save all the objects stored in ETS table, which are all tuples in this thesis project.
Type	Save the type of ETS table generated in corresponding test case.
Continuation	Save all the relevant information of continuation returned by functions: ets:match_object/1 and ets:match_object/3.
Matchcont	Save the continuation and relevant information used in ets_tests:start_match_object/2, and ets_tests:finish_match_object/2 (Which encapsulate the function ets:match_object/2)

5.1.2 continuation

The continuation is a list of tuples stored relevant data returned by calling function ets:match_object/1 and ets:match_object/3, the state includes five elements that can be further used in the dynamic state.

The reason to have MustReturnList and MayReturnList is that during the period between calling the function ets:match_object/3 and ets:match_object/1, the objects in ETS table might be changed by calling other functions, which will affect the result of ets:match_object/1. It's hard to predict which object will be returned, especially when running test case in parallel. Since the uncertainty of the results, MustReturnList and MayReturnList are used to restrict the specificaiton and expected outcomes of operation of function.

Table 5-2 **continuation in Main State of ets_tests**

Elements in continuation	Description
{Continuation, RemainMatchList}	Save the continuation returned by functions: ets:match_object/1 and ets:match_object/3, and a list of objects which match the pattern, but haven't been returned.
MustReturnList	A list of objects that match the corresponding Pattern but haven't returned, it will be changed during the test case, the objects in this list are the ones must be returned.
Pattern	The Pattern generate with the format {list('_')}
Limit	The number used to restrict the number of returned objects in each call of functions ets:match_object/1 and ets:match_object/3
MayReturnList	The objects which may be returned by ets:match_object/1

5.1.3 matchcont

In order to save the relevant information of modified function ets:match_obejct/2, the state matchcont is used in this project. The detailed description can be found in Section 5.4.3.

5.2 Generator in ets_tests

The generators used in testing ETS include:

Table 5-3 **Generator in ets_tests**

Generator	Description
gen_type/0	Generate one of the type: set, ordered_set, bag, and duplicate_bag for ETS table.
gen_object/1	Generate a list of tuples with random numbers of element as an object can be inserted into and delete from ETS table.
gen_key/1	Generate a key can be used in testing function ets:lookup/2 and ets:delete/2.
gen_pattern/0	Generate a pattern can be used in testing functions: ets:match_delete/2 and ets:match_object2 and ets:match_object/3.
gen_limit/0	Generate a random number can be used for restrict the number of return results of ets:match_object/3.
gen_mcont/1	Generate (pick up) a continuation which can be use in testing the function: ets:match_object/2.

In generators gen_object/1, gen_key/1 and gen_mcont/1, state is passed as parameters. The reason for using state as parameters is to refine the generated testing data, make the test case more meaningful. For example, when testing function ets:delete_object/2, it is more meaningful to delete an object that already existed in the table, rather than new object. If most of the object generated by generator gen_object/1 are not the objects already stored in the table, is will make the test case useless and meaningless, it is harder for QuickCheck to find if the function operates properly. Similarly, when testing ets:delete/2, it will be time consuming to generate a key which doesn't associate with any existed objects.

Meanwhile, frequency/1 is used to make a weighted choice between the generators in its argument, such that the probability of choosing each generator is proportional to the weight paired with it. [3] The statement of frequency/1 is illustrated as below:

```
frequency( list({integer(), gen()})) -> gen()
```

With using of frequency/1, the generator can generate more meaningful test data rather than generating random data which is not in the table.

5.3 Run tests

When the specification of functions finished, the module, where the specification and property saved, needs to be compiled before running the tests.

The module is compiled by calling:

```
1> c(ets_tests).
{ok,ets_tests}
```

The property in module ets_tests is tested by calling:

```
2> eqc:quickcheck(ets_tests:prop_ets_test()).
....................................................................................................
OK, passed 100 tests
true
```

5.4 Findings

5.4.1 ets:tab2list/1

When running the test cases, the error report returned and showed that the result of ets:tab2list/1 could be affected by the functions which involve insert or delete operations in parallel processes. The expected returned result after calling this function should be a list of all of the objects in the table. This indicated that the operation of functions ets:tab2list/1 should be guaranteed to be atomic. This error report was a hint for the problem that

ets:tab2list/1 can be interrupted by other functions running in a parallel process. The error report returned by running the test case and detailed description of it list as below.

```
14> eqc:quickcheck(ets_tests:prop_ets_test()).
..................................................................................................Failed! After 96 tests, and 4 repetitions.
( The counter example before shrinking is skipped here.)
Shrinking....................................(38 times)
duplicate_bag
{[{init,{state,[],duplicate_bag,[]}},
 {set,{var,6},{call,ets,tab2list,[{var,tableId}]}},
 {set,{var,7},{call,ets,delete_all_objects,[{var,tableId}]}},
 {set,{var,8},
     {call,ets,insert,
          [{var,tableId},
           [{97}, {97},  {97}, {97}, {97},
            {97,97,97,97,97,97,97,97},
            {97,97,97,97,97,97}]]}}],
 [[{set,{var,9},
      {call,ets,match_object,
           [{var,tableId},{'_','_','_','_','_','_','_','_'}]}},
  {set,{var,13},{call,ets,tab2list,[{var,tableId}]}}],
 [{set,{var,14},
      {call,ets,insert,
           [{var,tableId},
            [{214},{119},{119},{204,92,245,167,179,4,223,67,110}]]}}]]}
[{{set,{var,6},{call,ets,tab2list,[132653083]}},[]},
 {{set,{var,7},{call,ets,delete_all_objects,[132653083]}},true},
 {{set,{var,8},
     {call,ets,insert,
          [132653083,
           [{97}, {97}, {97}, {97}, {97},
            {97,97,97,97,97,97,97,97},
            {97,97,97,97,97,97}]]}},
 true}]
[[{set,{var,9},
     {call,ets,match_object,[132653083,{'_','_','_','_','_','_','_','_'}]}},      ←Function Call
     [{97,97,97,97,97,97,97,97}]},        ←Result
 {set,{var,13},
```

```
        {call,ets,tab2list,[132653083]},
        [{204,92,245,167,179,4,223,67,110},
         {97}, {97}, {97}, {97}, {97},
         {97,97,97,97,97,97,97,97},
         {97,97,97,97,97,97}]}],
   [{set,{var,14},
        {call,ets,insert,
            [132653083,
             [{214},{119},{119},{204,92,245,167,179,4,223,67,110}]]},
        true}]]
  no_possible_interleaving
  false
```

The codes in the color of Green are in the first process, and the codes in Orange are in the second process.

This error report showed that the function ets:tab2list/1 and ets:insert/2 were interrupted mutually. When inserting a list of objects, the entire operation should be guaranteed to be atomic and isolated.

In the report, after shrinking, the Shell got the simplified counter example, the type of the ETS table was duplicate bag, the Id of table was '132653083'. Quickcheck generated two processes running in parallel:

- One process generated symbolic calls of ets:match_object/2 and ets:tab2list/1, the results were displayed as well:

```
[[{set,{var,9},
     {call,ets,match_object,[132653083,{'_','_','_','_','_','_','_','_'}]]},
     [{97,97,97,97,97,97,97,97}]]},
 {set,{var,13},
     {call,ets,tab2list,[132653083]]},
     [{204,92,245,167,179,4,223,67,110}, {97}, {97}, {97}, {97}, {97},
      {97,97,97,97,97,97,97,97}, {97,97,97,97,97,97}]}],
```

- The other process generated symbolic call of ets:insert/2.

```
[{set,{var,14},
     {call,ets,insert, [132653083,
          [{214},{119},{119},{204,92,245,167,179,4,223,67,110}]]},
     true}]
```

By analyzing the results of these two parallel running processes, it's easy to find when calling ets:tab2list/1, a list of tuples which included {204,92,245,167,179,4,223,67,110} was returned,

It's easy to see that this object was inserted by ets:insert/2 which was in the first process in parallel. There were three more objects {214}, {119}, {119} being inserted at the same time. Therefore, more objects in the table should be returned by ets:tab2list/1.

This error report indicated that either the function ets:tab2list/1 or ets:insert/2 didn't behave atomically.

A similar error report was found involving functions ets:tab2list/1 and ets:delete_all_object/1. The report presented these two functions located in parallel processes can interrupt mutually, which indicated that either ets:tab2list/1 or ets:delete_all_object/1 didn't behave atomically. Since the second counter example was much similar with one presented before, it's omitted in this part.

5.4.2 Use of ets:safe_fixtable/2

In order to avoid getting this error report all the time, ets:safe_fixtable/2 [6] was used to encapsulate function ets:tab2lists/1 into a custom defined function ets_tests:my_tab2list/1, to ensure that the table was fixed until the process terminates. Which might make the function ets:tab2list/1 behave atomically. But even ets:safe_fixtable/2 was used, we still continuously get the error reports. One of them after shrinking is chosen as an example as below:

```
12> eqc:quickcheck(ets_tests:prop_ets_test()).
..........................................................Failed! After 70 tests.
        ( The counter example before shrinking is skipped here.)
Shrinking.......................................................................................................
.......................................................(192 times)
false
false
bag
{[[{init,{state,[],bag,[],[]}},
  {set,{var,15},
     {call,ets,insert,
          [{var,tableId},[{97},{58},{204,204},{249,97,97,97,97}]]}},
  {set,{var,16},
     {call,ets,insert,
          [{var,tableId},
           [{249}, {204}, {204,97,97,97,97,97,97,97,97,97,97},
            {97,97,97,97,97}, {197}, {145,97,97,97,97,97,97,97,97,97,97}, {77},
            {58,97,97,97,97,97,97,97,97}]]}},
  {set,{var,17},{call,ets_tests,my_tab2list,[{var,tableId}]}}],
 [[{set,{var,37},{call,ets_tests,my_tab2list,[{var,tableId}]}}],
```

```
 [{set,{var,34},{call,ets,delete_all_objects,[{var,tableId}]}}]]}
[[{set,{var,15},
    {call,ets,insert,[-125616102,[{97},{58},{204,204},{249,97,97,97,97}]]}},
 true},
 {{set,{var,16},
    {call,ets,insert,
       [-125616102,
        [{249}, {204}, {204,97,97,97,97,97,97,97,97,97,97}, {97,97,97,97,97}, {197},
         {145,97,97,97,97,97,97,97,97,97,97}, {77}, {58,97,97,97,97,97,97,97,97}]]}},
 true},
 {{set,{var,17},{call,ets_tests,my_tab2list,[-125616102]}},
 [{145,97,97,97,97,97,97,97,97,97,97},
   {204,204},{204},{204,97,97,97,97,97,97,97,97,97,97},{77}, {97}, {97,97,97,97,97},
  {197}, {249,97,97,97,97}, {249}, {58},
  {58,97,97,97,97,97,97,97,97}]}]
[[{{set,{var,37},{call,ets_tests,my_tab2list,[-125616102]}},
  [{204,204}, {204}, {204,97,97,97,97,97,97,97,97,97,97}, {77}, {97}, {97,97,97,97,97},
{197}, {249,97,97,97,97}, {249}, {58}, {58,97,97,97,97,97,97,97,97}]}],
 [{{set,{var,34},{call,ets,delete_all_objects,[-125616102]}},true}]]
no_possible_interleaving
false
```

In this counter example, QuickCheck generated two processes running in parallel:

- One process called ets:insert/1 and ets:tab2list/1.
- The other process called ets:tab2list/1 only.

The result of the ets:tab2list/1 in the second process included 11 objects only with the object {145,97,97,97,97,97,97,97,97,97,97} lost, comparing with the result of function ets:tab2list/1 called by the first process.

This counter example indicated that even the function ets:safe_fixtable/2 was applied into guarantee the atomicity of function ets:tab2list/1, the operation of it was still affected by functions running in a parallel process.

5.4.3 Separating ets:match_object/2

When running the test cases in ETS testing, one of the reports most often returned was about ets:match_object/2 cannot behave atomically. According to the documentation of Erlang [6], the atomicity of this function cannot be guaranteed, which make the error report similar all the time. In order to avoid continuously getting the similar error reports, this function was modified. Instead of have a single function, two functions ets_tests:start_match_object/2 and

ets_tests:finish_match_object/1 were used to represent the operation of function ets:match_object/2. The detailed description of these functions shows as below:

```
start_match_object(Tab, Pattern) –>
        MCont=list_to_atom(io_lib:write(make_ref())),
        ReturnList=ets:match_object(Tab, Pattern),
        {MCont, ReturnList}.
finish_match_object({_MCont, ReturnList}) –>
        ReturnList.
```

The reason of separating function ets:match_object/2 into two functions was to allow the function to not operate atomically, so that functions in other processes running in parallel can affect the result of it. This is modeled by having the functions executing between the functions start_match_object/2 and finish_match_object/2. Therefore, the tests can pass without interfere of this issue.

The parameters of ets_tests:start_match_object/2 were as the same as ets:match_object/2, the results of it were saved in mcont in the state. This function generated a tuple with two elements:

- One was the unique reference generated by calling make_ref/0, this reference would be used as a key to find the corresponding information in the state, which can be used by function ets_tests:finish_match_object/1.
- The other one was a list of objects which match the pattern, which was returned by calling the function ets:match_object/2.

The result of the function start_match_object/2 can be affect by functions running in parallel.

The parameter of the ets_tests:finish_match_object/1 was the result returned by the function ets_tests:start_match_object/2. The result of this function was to pick up the list of objects returned by ets_tests:start_match_object/2.

In the state, the list matchcont was used to store all the relevant data of these two functions. It was a list of tuples which include five elements that can be further used in the dynamic state of function ets_tests:start_match_object/2 and ets_tests:finish_match_object/2. The reason for separate the function ets:match_object/2 into two separated functions was that ets:match_object/2 didn't behave atomically, which affected the tests.

The fourth element was Undefined. During running the test cases, the data stored in it will be 'undefined'. The data stored in this part was never used in tests. The reason to have this part in matchcont was to make the matchcont have as the same construction as continuation in the state, so that relevant function for continuation can be reused for matchcont as well.

Table 5-4 **matchcont in Main State of ets_tests**

Elements in matchcont	Description
{MCont, ReturnList}	▪ Mcont is the continuation generate by make_ref/0 to specify the key point where ets_tests:start_match_object/2 is called. ▪ ReturnList is used to store the result returned by ets:match_object/2 which is encapsulated in the function ets_tests:start_match_object/2.
MustReturnList	A list of objects that match the corresponding Pattern and must be returned.
Pattern	The Pattern generate with the format {list('_')}
Undefined	The information saved in this part is 'undefined'
MayReturnList	The objects which may be returned.

After this modification, the functions were able to be interrupted by the functions called by other processes, further avoiding returning similar meaningless error reports, which improved the efficiency of the test case.

5.4.4 An error report of ets:insert/2 and ets:delete_all_objects/1

The error report which can indicate that one of bugs in the operation of ETS table returned by running the test cases when testing ETS table is list as below:

```
8> eqc:quickcheck(ets_tests:prop_ets_test()).
.........................................................Failed! After 67 tests.
       ( The counter example before shrinking is skipped here.)
Shrinking........................................................(65 times)
false
false
duplicate_bag
{[{init,{state,[],duplicate_bag,[],[]}},
  {set,{var,1},{call,ets,match_object,[{var,tableId},{'_'},1]}}],
 [[{set,{var,2},{call,ets,insert,[{var,tableId},[{154},{152},{92}]]}},
   {set,{var,4},{call,ets,match_object,[{var,tableId},{'_'},2]}}],
  [{set,{var,3},{call,ets,delete_all_objects,[{var,tableId}]}},
   {set,{var,7},{call,ets,insert,[{var,tableId},{157,24,7,24,11,80}]}},
   {set,{var,8},{call,ets_tests,my_tab2list,[{var,tableId}]}},
   {set,{var,10},{call,ets,match_delete,[{var,tableId},{'_'}]}}]]}
[{{set,{var,1},{call,ets,match_object,[26992666,{'_'},1]}},'$end_of_table'}]
[[{{set,{var,2},{call,ets,insert,[26992666,[{154},{152},{92}]]}},true},
```

```
  {{set,{var,4},{call,ets,match_object,[26992666,{'_'},2]}},
   {[{152},{92}],{26992666,87,2,<<>>,[],0}}}],
 [{{set,{var,3},{call,ets,delete_all_objects,[26992666]}},true},
  {{set,{var,7},{call,ets,insert,[26992666,{157,24,7,24,11,80}]}},true},
  {{set,{var,8},{call,ets_tests,my_tab2list,[26992666]}},
   [{154},{157,24,7,24,11,80}]},
  {{set,{var,10},{call,ets,match_delete,[26992666,{'_'}]}},true}]]
no_possible_interleaving
false
```

According to documentation of Erlang [6], ets:insert/2 and ets:delete_all_objects/1 should behave atomically.

The report illustrated as above was a counter example simplified by shrinking. From the report, it was apparent to see that even the second process called function ets:delete_all_objects(Tab) to empty the table, in the result of ets_tests:my_tab2list, the tuple {154} was still returned. Firstly, we thought that it indicated the operation of ets:delete_all_objects/1 was influenced by calling ets:insert(Tab, [{154},{152},{92}]) in the other process. The result can be due to either the function ets:insert/2 or ets:delete_all_objects/1 didn't behave atomically. But after analyzing this error report more, we found that was more likely that the problem was ets:tab2list/1 couldn't behave atomically. When function ets:tab2list/2 in the second process was running in parallel with the function ets:insert/2 in the first process, it also could be a case that ets:tab2list/2 only picked up part of the inserted objects from the first process.

5.5 Counter example

The counter examples found in the previous section was stored in a separated module: counter_bug.erl, in order to make it reusable for rechecking afterwards. The example below is a description of how to store the counter example in this module:

```
bug1()–>
    [duplicate_bag,
    {[{init,{state,[],duplicate_bag,[],[]}},
      {set,{var,1},{call,ets,match_object,[{var,tableId},{'_'},1]}}],
     [[{set,{var,2},{call,ets,insert,[{var,tableId},[{154},{152},{92}]]}},
       {set,{var,4},{call,ets,match_object,[{var,tableId},{'_'},2]}}],
      [{set,{var,3},{call,ets,delete_all_objects,[{var,tableId}]}},
       {set,{var,7},{call,ets,insert,[{var,tableId},{157,24,7,24,11,80}]}},
       {set,{var,8},{call,ets_tests,my_tab2list,[{var,tableId}]}},
      {set,{var,10},{call,ets,match_delete,[{var,tableId},{'_'}]}}
       ]]}].
```

5.6 eqc:recheck/1

The function eqc:recheck/1 was used to test the property with the same random number seed as the last failing call of eqc:quickcheck/1[3]. If the property was as the same as the one used when the failing call was generated, the same test case would be gererated, and eqc:recheck would repeat the test and shrinking phase. The example below was a rechecking the bug found in last section.

```
36> eqc:quickcheck(ets_tests:prop_ets_test(counter_bug:bug1())).
.................Failed! After 18 tests.
[{{set,{var,1},{call,ets,match_object,[4800539,{'_'},1]}},'$end_of_table'}]
[[{{set,{var,2},{call,ets,insert,[4800539,[{154},{152},{92}]]}},true},
  {{set,{var,4},{call,ets,match_object,[4800539,{'_'},2]}},
   {[{152},{92}],{4800539,87,2,<<>>,[],0}}}],
 [{{set,{var,3},{call,ets,delete_all_objects,[4800539]}},true},
  {{set,{var,7},{call,ets,insert,[4800539,{157,24,7,24,11,80}]}},true},
  {{set,{var,8},{call,ets_tests,my_tab2list,[4800539]}},
   [{154},{157,24,7,24,11,80}]},
  {{set,{var,10},{call,ets,match_delete,[4800539,{'_'}]}},true}]]
no_possible_interleaving
false
```

In this counter example, it was clear to find that the function ets:insert/2 still interrupted with ets:delete_all_objects/1, and the last symbolic call:

```
{set,{var,10},{call,ets,match_delete,[{var,tableId},{'_'}]}}
```

seemed didn't make sense. Due to this idea, the corresponding code in module counter_bug.erl was commented out and the test was rerun.

```
73> eqc:quickcheck(ets_tests:prop_ets_test(counter_bug:bug1())).
.................................Failed! After 36 tests.
  (Counter example skipped here.)
```

As the report above, the test case passes all the time, which indicated that in this counter example, the symbolic call ets:match_delete/1 did interfered the test failure for some reason, but the reason of how this call affecting the failure was not clear to describe.

6 PULSE

PULSE, short for ProTest User-Level Scheduler for Erlang, is a randomizing scheduler for Erlang, which can be used to find race conditions in concurrent Erlang code. [7]

With the use of QuickCheck, even the failure of simple test case can be found, the reason of why it fails is hard to tell. The normal next step is to rerun the test with Erlang's tracing features. But when the bug is caused by race condition, turning the tracing on is likely change the timing property, which may interfere with the test failure. With the thought of repeat the test case as many times as we like, PULSE is introduced and implemented. PULSE can control the execution of designated Erlang processes and records a trace of all relevant events, which makes it's possible to analyze how the race condition is provoked [8].

PULSE is used to increase the possibility of finding concurrency errors, as well as improving the analysis of these errors by offering the ability to rerun the same scheduling deterministically.

6.1 instrument:c/1

In order to use PULSE, one needs to instrument the code under test, by compiling it with a special flag instrument:c/1. After that, the concurrent processes can be run by PULSE.

6.2 ?SCHEDULED

In this thesis project, the research version of PULSE is used, with few special flag, when using it. The macro ?SCHEDULED is used, which runs a piece of code under PULSE. Meanwhile, the property is changed to use this macro.

7 Testing DETS

The second part of this thesis is the tests in DETS. The module dets provides term storage on file. [6] Just as ETS table, DETS table is also a collection of Erlang tuples, which are called objects. However, there a distinguishing difference that the data stored in ETS table is transient, whereas, data stored in DETS table is persistent and should survive an entire system crash.

Meanwhile, unlike that ETS have four different types, DETS only have three, which includes: set, bag, duplicate_bag. The type of ordered_set is not implemented in DETS.

DETS table must be opened before they can be updated or read, and also, it should be properly closed when finish. If not, DETS will automatically repair the table, which may take a long time if the table is large. When several Erlang process open the same DETS table, they will share the table, in which case, the race condition more likely arise. The table is then properly closed when all the processes have terminated.

Since the operations perform by DETS are disk operations, the functions of DETS are much slower than corresponding ETS functions.

The functions being tested include:

- dets:insert(Tab, ObjectOrObjects) –> ok.
  Insert an Object of Objects into the table Tab.
- dets:insert_new(Tab, Object) –> Bool.
  Insert an Object into the table Tab, if the key of Object is not existed in the table.
- dets:lookup(Tab, Key) –> (Object).
  Return a list of objects from the table Tab, with the same Key.
- dets:delete(Tab, Key) –> ok.
  Delete the objects with the same Key from the table Tab.
- dets:delete_all_objects(Tab) –> ok.
  Delete all of objects from table Tab
- dets:delete_object(Tab, Object) –> ok.
  Delete the Object from the table Tab.
- dets:match_delete(Tab, Pattern) –> ok.
  Delete all the objects against Pattern, from the table Tab.
- dets:match_object(Continuation) –> {{Match}, Continuation|'$end_of_table'}.
  Return a number of object aginest the pattern that specified when generating the Continuation.
- dets:match_object(Tab, Pattern) –> (Object).
  Return all the objects, which match the Pattern, from the table Tab.

- dets:match_object(Tab, Pattern, Limit) –> {{Match}, Continuation|'$end_of_table'}.
  Match the objects in the table Tab against the Pattern, and return a number Limit of he matched objects.

7.1 State in dets_tests

7.1.1 Main state

The state used in dets_tests include three parts: objects, type, continuation, with detailed description show as the table below:

Table 7-1 **Main State of dets_tests**

Parts in State	Description
Objects	Save all the object stored in DETS table, which are all tuples in this thesis.
Type	Save the type of DETS table generated in the test case.
Continuation	Save all the relevant information of continuation returned by functions: dets:match_object/1 and dets:match_object/3.

7.1.2 continuation

Similar as module ets_tests.erl, there is a continuation part in dets_tests.erl. The continuation include five elements is described in the table below.

Table 7-2 **continuation in Main State of dets_tests**

Elements in continuation	Description
{Continuation, RemainMatchList}	Save the continuation returned by functions: dets:match_object/1 and dets:match_object/3, and a list of objects which match the pattern, but haven't been returned.
MustReturnList	A list of objects that match the corresponding Pattern but haven't returned, it will be changed during the test case, the objects in this list are the ones must be returned.
Pattern	The Pattern generate with the format {list('_')}
Limit	The number used to restrict the number of keys in returned objects in each call of functions dets:match_object/1 and dets:match_object/3
MayReturnList	The objects which may be returned by dets:match_object/1

7.2 Generator in dets_tests

The generators used in testing DETS are described as below. In generators gen_object/1 and gen_key/1, state are passed as parameters as well.

Table 7-3 **Generator in dets_tests**

Generator	Description
gen_type/0	Generate one of the type: set, ordered_set, bag, and duplicate_bag for DETS table.
gen_object/1	Generate a tuple with random numbers of element as an object can be inserted into and delete from DETS table.
gen_key/1	Generate a key can be used in testing function dets:lookup/2 and dets:delete/2.
gen_pattern/0	Generate a pattern can be used in testing functions: dets:match_delete/2 and dets:match_object2 and dets:match_object/3.
gen_limit/0	Generate a random number can be used for restrict the number of return results of dets:match_object/3.

7.3 Use of PULSE

After both sequence and parallel testing of the functions in DETS, there was no bug found. It may due to the limits of Erlang virtual machine (VM), which runs processes for relatively long time-slices, in order to minimize the time spent on context switching, but as a result, it is unlikely to provoke race conditions in small tests. Therefore, PULSE was applied into testing DETS. With the ability of controlling the execution of designated Erlang processes and records a trace of all relevant events, and furthermore, taking control over all sources of non-determinism in Erlang programs, and instead taking those scheduling decisions randomly, it was more likely that race conditions can be provoked and concurrent bugs can be found.

7.4 d_ets

In order to use PULSE, an instrument version of dets was used and the name of the module was changed to d_ets. Even the name of module has been changed, all the functions are exactly as the same as the ones in dets. In this paper, in order to ensure the consistency and understandability, only dets will be used to descript relevant findings, even it may presents as d_ets in the error report.

7.5 dets:open_file/2

According to the documentation of DETS, if two processes open the same table by giving the same name and arguments, then the table will be used by these two processes. If one user closes the table, it will still remain open until the second process closes the table. [6]

In order to make different process working fine in the same DETS table simultaneously, dets:open_file/2 was used at the beginning of list of commands in each process.

7.6 Run tests

When the specification of functions finished, the module, where the specification and property saved, needs to be compiled before running the tests.

The module is compiled by calling:

```
1> c(dets_tests).
{ok,dets_tests}
```

Before running the tests, instrument:c/1 is used to instrument all the relevant modules.

The property in module dets_tests is tested by calling:

```
3> eqc:quickcheck(dets_tests:prop_dets_test()).
....................................................................................................
OK, passed 100 tests
true
```

7.7 Atomicity in DETS

In the testing of ETS, there are several functions being found that cannot behave atomically, for example: ets:match_object/2, ets:match_delete/2, etc. The problems of similar function in DETS table are suppose to present as well, since they have the similar interface.

According to the documentation of DETS, there is not explicit explanation describing any function in DETS should behave atomically, but the operation in DETS is done in one single process, so the behavior of functions shouldn't be interrupted by other functions, which might indicate that the atomicity should be somehow guaranteed by DETS.

7.8 Findings

7.8.1 dets:match_object/2 & dets:match_delete/1

During the testing, counter example was similar to what found in ETS testing. The counter example presents below can explain the problems in running DETS tests in parallel.

```
Shrinking............................(29 times)
duplicate_bag
    (The symbolic commands generated before running actual tests are skipped here.)
{propdict_keeper,<0.17796.7>,#Ref<0.0.0.245790>}
[{{set,{var,1},{call,d_ets,match_object,[{var,tableId},{'_'}]}},[]},
 {{set,{var,2},{call,d_ets,match_object,[{var,tableId},{'_'},1]}},
  '$end_of_table'},
 {{set,{var,3}, {call,dets_tests,my_match_object,
            [{call,erlang,element,
                 [1, {call,erlang,element,
                       [1, {{{call,dets_tests,my_element,[{var,2}]},[]},
                         [],{'_'}, 3,[]}]}]}]}},
  '$end_of_table'},
```

```
{{set,{var,4},{call,d_ets,match_object,[{var,tableId},{'_'},1]}},
 '$end_of_table'},
{{set,{var,5},{call,d_ets,match_object,[{var,tableId},{'_'},1]}},
 '$end_of_table'},
{{set,{var,6},{call,d_ets,match_delete,[{var,tableId},{'_'}]}},ok},
{{set,{var,7},{call,d_ets,lookup,[{var,tableId},97]}},[]},
{{set,{var,8},{call,d_ets,delete_object,[{var,tableId},{97}]}},ok}]
[[{{call,d_ets,open_file,[test,[{type,duplicate_bag}]]},{ok,test}},
 {{call,d_ets,delete_all_objects,[test]},ok},
 {{call,d_ets,insert_new,[test,{97}]},true},
 {{call,d_ets,insert,[test,[{12}]]},ok},
 {{call,d_ets,match_delete,[test,{'_'}]},ok}],
 [{{call,d_ets,open_file,[test,[{type,duplicate_bag}]]},{ok,test}},
 {{call,d_ets,insert,[test,[]]},ok},
 {{call,d_ets,delete,[test,97]},ok},
 {{call,d_ets,insert,[test,[{12},{98}]]},ok},
 {{call,d_ets,match_object,[test,{'_'}]},[{98}]}]]
no_possible_interleaving
false
```

As the counter example presents above, after generating a list of sequential test cases, PULSE generated a pair of processes in parallel:

- One process generated symbolic calls of dets:open_file/2, dets:insert_new/2, dets:insert/2, and dets:match_delete/2 the results were displayed as well:

```
[{{call,d_ets,open_file,[test,[{type,duplicate_bag}]]},{ok,test}},
 {{call,d_ets,delete_all_objects,[test]},ok},
 {{call,d_ets,insert_new,[test,{97}]},true},
 {{call,d_ets,insert,[test,[{12}]]},ok},
 {{call,d_ets,match_delete,[test,{'_'}]},ok}],
```

- The other process generated symbolic calls dets:open_file/2, dets:insert/2, dets:delete/2, and dets:match_object/2, the results were returned and displayed as well:

```
[{{call,d_ets,open_file,[test,[{type,duplicate_bag}]]},{ok,test}},
 {{call,d_ets,insert,[test,[]]},ok},
 {{call,d_ets,delete,[test,97]},ok},
 {{call,d_ets,insert,[test,[{12},{98}]]},ok},
 {{call,d_ets,match_object,[test,{'_'}]},[{98}]}]]
```

In the second process, there was only one object {98} was returned by dets:match_object/2, with losing object {12}. These two objects were inserted by function dets:insert/2, which was in the same process, at the same time. It was apparent that functions in the second process were interrupted by the functions in the first process, which was more likely function dets:match_delete/2 affected the operation of dets:match_object/2 in this counter example.

7.8.2 An error report of dets:insert/2 & dets:delete_all_objects/1

The problem below indicated that either function dets:insert/2 or dets:delete_all_objects/1 didn't operate atomically:

```
Shrinking................................(34 times)
bag
{[{init,{state,[],bag,[]}},
    (The symbolic commands generated before running actual tests are skipped here.)
{propdict_keeper,<0.4204.10>,#Ref<0.0.1.80189>}
[{{set,{var,1},{call,d_ets,delete,[{var,tableId},97]}},ok},
 {{set,{var,2},{call,d_ets,insert,[{var,tableId},{97}]}},ok},
 {{set,{var,3},{call,d_ets,match_object,[{var,tableId},{'_'},1]}},
  {[{97}],{dets_cont,object,1,eof,<<>>,test,<<>>}}},
 {{set,{var,4},
      {call,dets_tests,my_match_object,
           [{call,erlang,element,
                [1, {call,erlang,element,
                      [1, {{{call,dets_tests,my_element,[{var,3}]},[]},
                       [],{'_'}, 26,[]}]}]}]}}, '$end_of_table'},
 {{set,{var,7},{call,d_ets,insert,[{var,tableId},[{54}]]}},ok}]
{[[{{call,d_ets,open_file,[test,[{type,bag}]]},{ok,test}},
  {{call,d_ets,match_delete,[test,{'_','_'}]},ok},
  {{call,d_ets,insert,[test,[{9},{54}]]},ok}],
 [{{call,d_ets,open_file,[test,[{type,bag}]]},{ok,test}},
  {{call,d_ets,delete_all_objects,[test]},ok},
  {{call,d_ets,insert,[test,{45}]},ok},
  {{call,d_ets,match_object,[test,{'_'}]},[{45},{9}]}]]}
no_possible_interleaving
false
```

In this counter example, it's clear to see that there were two processes running in parallel, after a list of sequential commands:

- One process called symbolic calls of dets:open_file/2, dets:match_delete/2 and dets:insert/2, the results were displayed as well:

```
[{{call,d_ets,open_file,[test,[{type,bag}]]},{ok,test}},
 {{call,d_ets,match_delete,[test,{'_','_'}]},ok},
 {{call,d_ets,insert,[test,[{9},{54}]]},ok}],
```

- The other process called symbolic calls dets:open_file/2, dets:delete_all_object/1, dets:insert/2, and dets:match_object/2, the results were returned and displayed as well:

```
[{{call,d_ets,open_file,[test,[{type,bag}]]},{ok,test}},
 {{call,d_ets,delete_all_objects,[test]},ok},
 {{call,d_ets,insert,[test,{45}]},ok},
 {{call,d_ets,match_object,[test,{'_'}]},[{45},{9}]}]
```

At the beginning, we thought that function dets:insert/2 in the first process was interrupted by the function dets:delete_all_objects/1 in the second process. Therefore, only one object {54} was deleted, which indicated that either the function dets:insert/2 or dets:delete_all_object/1 cannot behave atomically.

After more analysis, we found it might also can be due to dets:match_object/2 was interrupted by the functions dets:insert/2, and just pick part of the objects inserted.

7.8.3 d_ets:insert/2 & d_ets:match_delete/2

The counter example below indicated that function d_ets:insert/2 or d_ets:match_delete/2 didn't behave atomically.

```
Shrinking........................(25 times)
set
[[{init,{state,[],set,[]}},
   (The symbolic commands generated before running actual tests are skipped here.)
{propdict_keeper,<0.13182.5>,#Ref<0.0.0.187967>}
[[{set,{var,1},{call,d_ets,insert,[{var,tableId},[]]}},ok},
 {{set,{var,2},{call,d_ets,match_object,[{var,tableId},{'_'}]}},[]},
 {{set,{var,3},{call,d_ets,match_object,[{var,tableId},{'_'}]}},[]},
 {{set,{var,4},{call,d_ets,delete,[{var,tableId},97]}},ok},
 {{set,{var,14},{call,d_ets,insert_new,[{var,tableId},{161,241}]}},true}]
{[[{{call,d_ets,open_file,[test,[{type,set}]]},{ok,test}},
  {{call,d_ets,insert,[test,{211}]},ok},
  {{call,d_ets,match_delete,[test,{'_'}]},ok}],
 [[{{call,d_ets,open_file,[test,[{type,set}]]},{ok,test}},
  {{call,d_ets,insert,[test,[{211},{97}]]},ok},
```

```
 {{call,d_ets,delete_object,[test,{211,97}]},ok},
 {{call,d_ets,match_object,[test,{'_'}]},[{97}]}]}
no_possible_interleaving
false
```

In this counter example, it's clear to see that there were two processes running in parallel, after a list of sequential commands:

- One process called symbolic calls of dets:open_file/2, dets:insert/2 and dets:match_delete/2, the results were displayed as well:

```
[{{call,d_ets,open_file,[test,[{type,set}]]},{ok,test}},
 {{call,d_ets,insert,[test,{211}]},ok},
 {{call,d_ets,match_delete,[test,{'_'}]},ok}],
```

- The other process called symbolic calls dets:open_file/2, dets:insert/2, dets:delete_object/2 and dets:match_object/2, the results were displayed as well:

```
[{{call,d_ets,open_file,[test,[{type,set}]]},{ok,test}},
 {{call,d_ets,insert,[test,[{211},{97}]]},ok},
 {{call,d_ets,delete_object,[test,{211,97}]},ok},
 {{call,d_ets,match_object,[test,{'_'}]},[{97}]}]
```

It's obvious that in the second process, the function dets:match_object should returned two objects {211} and {97}, whereas, only {97} was returned. It was due to the insert function in the second process was interrupted by the function dets:match_delete/2 in the first process.

7.8.4 dets:match_object/3

In DETS table, after calling dets:match_object/3, a list of objects matching with the pattern and a continuation match are returned. If there is no object matching with the pattern, '$end_of_table' will be returned. The continuation returned can be further used by calling dets:match_object/1. The parameters used by this function include: TableId, Pattern and a Number. [6]

This function have as the same interface as ets:match_object/3, but with slight differences in defining the third parameter. In ets:match_object/3, the Limit is used to restrict the length of the returned list, which means the Limit is refers to the number of objects in the returned list. Whereas in the dets:match_object/3 the Number is used to restrict the number of Keys presented in the returned list. Two examples below can be a clearer description:

Example 1: ets:match_object/3

```
1> Tab=ets:new(table, [duplicate_bag]).
```

```
16400
2> ets:insert(Tab, [{a, b}, {a,c}, {a, d}]).
true
3> ets:match_object(Tab, {'_', '_'}, 1).
{[{a,d}], {16400,113,1,<<>>,[{a,b},{a,c}],2}}
```

Example 2: dets:match_object/3

```
13> {ok, Tab}=dets:open_file(table, {type, duplicate_bag}).
{ok,table}
14> dets:insert(Tab, [{a, b}, {a,c}, {a, d}]).
ok
15> dets:match_object(Tab, {'_', '_'}, 1).
{[{a,b},{a,c},{a,d}],  {dets_cont,object,1,eof,<<>>,table,<0.51.0>,<<>>}}
```

According to these two example above, it is clear to see that, same objects {a,b},{a,c},{a,d} are insert in both ETS table and DETS table, with the same parameters (Tab, {'_', '_'}, 1) for functions ets:match_object/3 and dets:match_object/3 are called, the distinct values are returned. In the ETS table, only one object is returned, whereas, in the DETS table, three objects with the same key are returned.

7.8.5 dets:match_object/1

7.8.5.1 Difference from ets:match_object/1

The function dets:match_object/1 uses the Continuation which is returned by calling function dets:match_object/1 or previous dets:match_object/3 as a parameter, returns a list of objects stored in a DETS table that match the given pattern. [6]

At the beginning of testing DETS, a similar specification as corresponding functions in testing ETS was used, the test cases crashed frequently. After reading and checking the error reports returned, another distinction between ETS and DETS was found. In ETS, '$end_of_table' was allowed to used as the parameter for ets:match_object/1, whereas, it was not the case in DETS.

```
16> ets:match_object('$end_of_table').
'$end_of_table'

17> dets:match_object('$end_of_table').
** exception error: bad argument
   in function  dets:match_object/1
      called as dets:match_object('$end_of_table')
```

These two examples above illustrated how ETS and DETS behave distinctively. When using '$end_of_table' in ets:match_object/1, the call was perceived as valid, and another '$end_of_table' was returned. But when passing '$end_of_table' as the parameter in dets:match_object/1, an exception was thrown with the error reason 'bad argument'.

7.8.5.2 dets_tests:my_match_object/1

In order to solve the problem found in last section that '$end_of_table' cannot be passed as parameter in dets:match_object/1, and make the tests functioning, one custom function was used and encapsulated this function.

```
my_match_object(Cont) ->
    case Cont of
        '$end_of_table' -> '$end_of_table';
        _->?DETS:match_object(Cont)
    end.
```

As the code presented above, the function dets:match_object/1 will be called, only when the parameter was not '$end_of_table'.

7.8.5.3 Uncertainty of returned value

Besides of the interesting points stated above, there was another interesting point we found, which was the uncertainty of the returned value of function dets:match_object/1.

According to the documentation of DETS, this function is supposed to return a non-empty list of some objects stored in a table match a given pattern in some unspecified order [6]. Even the order is unspecified, there should be an order anyway, and therefore, the returned value of this function is supposed to be a list of objects which haven't been returned.

But the error reports in this project presented that the objects had been returned before would be returned again for some reason. The counter examples below presented how the uncertainty arises.

Counter example 1:

During running parallel testing with PULSE, the counter example was found that function dets:match_object/1 returned the objects which had already been returned.

```
Shrinking.........................................................................(76 times)
duplicate_bag
{[{init,{state,[],duplicate_bag,[]}},
   (The symbolic commands generated before running actual tests are skipped here.)
{propdict_keeper,<0.261.2>,#Ref<0.0.0.125845>}
[{{set,{var,1},{call,d_ets,insert_new,[{var,tableId},{97}]}},true},
```

```
{{set,{var,2},{call,d_ets,insert,[{var,tableId},{150}]}},ok},
{{set,{var,5},{call,d_ets,insert_new,[{var,tableId},{97}]}},false},
{{set,{var,7},{call,d_ets,delete_all_objects,[{var,tableId}]}},ok},
{{set,{var,17},
    {call,d_ets,insert,
        [{var,tableId},[{18},{40,97,97,97,97,97,97,97,97}]]}},
 ok},
{{set,{var,18},{call,d_ets,insert_new,[{var,tableId},{97}]}},true}]
{[{{call,d_ets,open_file,[test,[{type,duplicate_bag}]]},{ok,test}},
 {{call,d_ets,insert_new,[test,{18,144,148,66,186,243,233,8}]},false},
 {{call,d_ets,match_object,[test,{'_','_','_','_','_','_','_','_','_'},1]},
  {[{40,97,97,97,97,97,97,97,97}],
   {dets_cont,object,1,<<>>,{5624,5624,<<>>},test,<<>>}}},
 {{call,dets_tests,my_match_object,
      [{dets_cont,object,1,<<>>,{5624,5624,<<>>},test,<<>>}]},
  {[{40,97,97,97,97,97,97,97,97}],{dets_cont,object,1,eof,<<>>,test,<<>>}}}],
[{{call,d_ets,open_file,[test,[{type,duplicate_bag}]]},{ok,test}},
 {{call,d_ets,insert,[test,[{40}]]},ok},
 {{call,d_ets,match_object,[test,{'_','_','_','_','_','_','_'},1]},
  '$end_of_table'},
 {{call,d_ets,match_delete,[test,{'_','_','_','_','_','_','_'}]},ok}]}
no_possible_interleaving
false
```

In the first process, there were two commands came one after the other, the first command involved function dets:match_object/3. This function returned {40,97,97,97,97,97,97,97,97} as the result and {dets_cont,object,1,<<>>,{5624,5624,<<>>},test,<<>>} as the continuation.

In the second command, the function dets_tests:my_match_object/1, which actually behaved as the same as dets:match_object/1, used the continuation returned from previous command as the parameter. The expected returned value should be '$end_of_table', whereas this function returned the tuple {40,97,97,97,97,97,97,97,97} as the result again, and meanwhile returned {dets_cont,object,1,eof,<<>>,test,<<>>} as the continuation.

In this counter example, {40,97,97,97,97,97,97,97,97} was returned twice, which was strange. It presented the problem that the returned value of dets:match_object/1 was hard to specify.

Counter example 2:

In order to check whether this problem was provoked due to use of PULSE, and if this error message was only reported when running the parallel tests, the sequential test cases were run. As the result, the similar error reports were returned as well, below is an example:

```
Shrinking.................(18 times)
duplicate_bag
{[{init,{state,[],duplicate_bag,[]}},
   (The symbolic commands generated before running actual tests are skipped here.)
{propdict_keeper,<0.17629.4>,#Ref<0.0.0.172272>}
[{{set,{var,16},
     {call,d_ets,insert,
         [{var,tableId},[{94,87,18,146},{97},{84},{84,177,254,226,17}]]}},
  ok},
 {{set,{var,22},{call,d_ets,match_object,[{var,tableId},{'_'},3]}},
  {[{84},{97}],
   {dets_cont,object,3,
          <<0,0,0,27,18,52,86,120,0,0,0,19,0,0,0,15,131,104,4,97,94,97,87,
            97,18,97,146>>, {5528,5560,<<>>}, test,<<>>}}},
 {{set,{var,23},{call,d_ets,insert,[{var,tableId},{84,254,86,18,87}]}},ok},
 {{set,{var,27},{call,d_ets,match_object,[{var,tableId},{'_'},1]}},
  {[{97}],
   {dets_cont,object,1,
          <<0,0,0,27,18,52,86,120,0,0,0,19,0,0,0,15,131,104,4,97,94,97,87,
            97,18,97,146,0,0,0,0,0,0,0,0,0,55,18,52,86,120,0,0,0,47,0,0,0,9,
            131,104,1,97,84,0,0,0,17,131,104,5,97,84,97,177,97,254,97,226,
            97,17,0,0,0,17,131,104,5,97,84,97,254,97,86,97,18,97,87>>,
          {5528,5624,<<>>}, test,<<>>}}},
 {{set,{var,24},{call,d_ets,match_object,[{var,tableId},{'_'},1]}},
  {[{97}],
   {dets_cont,object,1,
          <<0,0,0,27,18,52,86,120,0,0,0,19,0,0,0,15,131,104,4,97,94,97,87,
            97,18,97,146,0,0,0,0,0,0,0,0,0,55,18,52,86,120,0,0,0,47,0,0,0,9,
            131,104,1,97,84,0,0,0,17,131,104,5,97,84,97,177,97,254,97,226,
            97,17,0,0,0,17,131,104,5,97,84,97,254,97,86,97,18,97,87>>,
          {5528,5624,<<>>}, test,<<>>}}},
 {{set,{var,30},
     {call,dets_tests,my_match_object,
         [{call,erlang,element,
              [1, {call,erlang,element,
```

```
                        [1, {{{call,dets_tests,my_element,[{var,22}]},[]},
                         [],{'_'}, 4,[]}}]}]}}},
  {[{84}],{dets_cont,object,3,eof,<<>>,test,<<>>}}}]
 {[],[]}
 {postcondition,false}
 false
```

In this sequential test case, variable {var, 22} was used to store the result returned by dets:match_object/3. This function returned two tuples {84}, {97}, and a continuation, which was a long continuation.

After several sequential commands, variable {var, 30} was used to save the result returned by function dets_test:my_match_object/1, which behaved as the same as dets:match_object/1. The continuation saved in {var, 22} was used as the parameter. The object {84} was returned as the result once again, meanwhile new continuation {dets_cont,object,3,eof,<<>>,test,<<>>} was returned.

In this counter example, the object {84} was returned twice. The unexpected operation of function dets:match_object/1 presented that the returned value of this function was hard to specify, and also this problem was not provoked by running parallel tests.

8 Testing Supervisor

A supervisor tree is tree of processes where upper processes (supervisor) in the tree monitor the lower child processes. [1] The type of the child process can be either supervisor or worker. The child process can monitor a lower process, only when the type is supervisor.
One of the main work that supervisor does is ensure that their child processes are alive during the running time, avoid they being killed due to some exception. [6]

The functions being tested include:

- supervisor:start_child(SupRef, ChildSpec) –> {ok, ChildSpec}
  Dynamically add child proceese to a supervisor SupRef.
- supervisor:terminate_child(SupRef, Id) –> ok
  Terminate a child process Id supervised by supervisor SupRef.
- supervisor:delete_child(SupRef, Id) –> ok
  Delete a child Id, from supervisor SupRef.
- supervisor:restart_child(SupRef, Id) –> ok
  Restart a terminated child process supervised by supervisor SupRef.
- supervisor:which_children(SupRef) –> [ChildSpec]
  Return information about all the children specifications and child processes supervised by the supervisor SupRef.

In this thesis project, module ***supervisor_tests.erl*** is used to store all specifications used in supervisor test.

8.1 State in supervisor_tests

8.1.1 Main State

All the states are saved as lists of information. The state used in supervisor_tests include five parts: relation, unrelation, supervisor, children and latestSupervisor. With detailed descriptions show as the table below:

Table 8-1 **Main State of supervisor_tests**

Parts in State	Description
relation	Stores the pairs of name of supervisor and Ids of their children, which are generated after calling functions supervisor:start_child/2 or restart_child. EX: {Sup, Id, Pid, OldPids}
unrelation	Stores the pairs of name of supervisor and Ids of their children which are terminated, and can be used by function supervisor:restart_child/2. EX: {Sup, Id, undefined, OldPids }
supervisor	Stores the name of processes whose type is supervisor

children	Stores the ChildSpec that generated by generator gen_childspec/0 when call the function supervisor:start_child/2
latestSupervisor	Stores the last modified supervisor, which can be used by function supevisor:which_children/1

8.1.2 Relation and Unrelaiton

Relation in the state is a list of tuple/4 in the format { Sup, Id, Pid, OldPids}.

- Sup: the name of the supervisor, which supervises child process Id.
- Id: the name of the child which is supervised by Sup.
- Pid: is the current pid of alive child process.
- OldPids: is a list of pids that child process had before using the current pid.

The reason to save Pid and OldPids in the state of relation is to check the restart strategy of the supervisor, detailed descriptions can be found in Section 8.6.1, as well as of the restart method of child, detailed descriptions can be found in Section 8.6.2.

When the child is restarted the pid of child process should be changed, and the current Pid will be compared with old pids in list OldPids. If Pid is not a member of list OldPids, it indicates that the child process is restart properly.

Unrelation in the state is also a list of tuple/4 in the formation {Sup, Id, undefined, OldPids}. It is used for providing information for the function supervisor:restart_child/2. All of the data are as the same as them in relaiton, except the third element is replaced by 'undefined', since when the child process is terminated, the pid of process will be specified as undefined.

8.1.3 Pid and OldPids

In the Relation and Unrelation parts of state, besides of containing Sup and Id, there are two other elements saved, which are: Pid and OldPid. As the statement in last section, the reason of including these two parts is to save proper data in the state for testing both the restart strategy of supervisor and restart method of child. The example below describes how it works, assuming that there are two processes Id1 and Id2, with the same supervisor Sup, using one_for_one restart strategy, and Id1 should be restarted when terminated.

The state before calling exit/2 is:

```
        #state.relation=[{Sup, Id1, Pid1, [ ]}, {Sup, Id2, Pid2, []}]
1> exit(Id1, kill) –> ok
        #state.relation= [{Sup, Id1, unknown, [Pid1]}, {Sup, Id2, Pid2, []}]
2> supervisor:which_children(Sup) –> [{Id1, Pid3, _, _}, {Id2, Pid2, _, _}]
        #state.relation= [{Sup, Id1, Pid3, [Pid1]}, {Sup, Id2, Pid2, []}]
```

In the initial state, Sup is the supervisor process, Id1 and Id2 are the child processes, Pid1 and Pid2 are the pids of these child processes, there is no old pid in the state, which are empty lists.

After calling exit/2 firstly in <1>, the child process Id1 is killed, and is restarted by its suepervisor Sup afterwards. Since we cannot know what is the new pid after restarting, the pid in the state is replaced by 'unknown' temporarily. Meanwhile, Pid1 is moved into the corresponding list OldPids. Whereas since the restart strategy of Sup is one_for_one, Id2 shouldn't be affected.

When calling supervisor:which_children/1 in second call <2>, the pids of Id1 and Id2 are returned as Pid3 and Pid2. Pid3 is compared with Pid1 in the OldPids, if they are different, it indicates that Pid3 is the current pid of child process Id1, after it is restarted. Then, Pid3 will replace the 'unknown' in the state. If they are equal, it means the child process Id1 wasn't killed by the function exit/2, then postcondition will fail. If Pid3 is 'undefined', it means that Id1 is not restart properly by its supervisor.

According to the restart strategy of the supervisor Sup, when killing Id1, child process Id2 shouldn't be restarted. If the pid returned by the function supervisor:which_children for Id2 is still Pid2, it indicates that the child process Id1 is restarted in a proper way, and Id2 isn't affected. Otherwise, it means that the child process Id2 is restarted as well, which indicates the problem that the child processes are restarted in an unexpected way, and the postcondition will fail as well.

8.2 Generator in supervisor_tests

The generators used in testing supervisor include:

Table 8-2 **Generator in supervisor_tests**

Generator	Description
gen_childSpec/0	Generate child specifications that used when starting a child process.
gen_id/0	Generate a unique name used as the Id of the child process.
gen_restart/0	Choose one of the restart types from permanent, temporary and transient for the child process.
gen_shutDown/0	Generate a shutdown type of child process.
gen_type/0	Choose a type for the child process from either supervisor or worker.
gen_restart_strategy/0	Generate a restart strategy for supervisor process.
gen_latestSupervisor/1	Pick up one supervisor from the state, which can be used in function supervisor:which_children/1 afterwards.

8.3 Use of PULSE

Same as DETS, PULSE is used for testing the main functions in Supervisor. With the advantage of ability of controlling the execution with considering the time property of PULSE, it is more likely that race conditions can be provoked and corresponding concurrent bugs can be found.

8.4 super_visor

In order to use PULSE, an instrument version of supervisor was used and the name of the module was changed to super_visor. All of the functions remain the same, just with name changed. In this section, in order to ensure the consistency and understandability, only supervisor will be used to descript relevant findings, even it may presents as super_visor in the error report.

8.5 Run tests

When the specification of functions finished, the module, where the specification and property saved, needs to be compiled before running the tests.

The module is compiled by calling:

```
1> c(supervisor_tests).
{ok, supervisor _tests}
```

Before running the tests, instrument:c/1 is used as well to instrument all the relevant modules.

The property in module supervisor_tests is tested by calling:

```
3> eqc:quickcheck(supervisor_tests:prop_ supervisor_test()).
....................................................................................................
OK, passed 100 tests
true
```

8.6 Difficulties

8.6.1 Restart strategy of Supervisor

A supervisor tree is a tree of processes, and the upper supervisor which monitoring the lower supervisors and workers, and restart these lower processes when they fail. A supervisor can have one of the following four restart strategies: [6]

- **one_for_one:** if one child process terminates and should be restarted, only this child process should be restart.
- **one_for_all:** if one child process terminates and should be restarted, all other child processes should be terminated and restarted.
- **rest_for_one:** if one child process terminates and should be restarted, the child processes after this terminated child process in the start order should be terminated and restarted.
- **simple_one_for_one:** when a child process are dynamically added instances of the same process type, i.e. running the same code.

In this thesis project, only the first three restart strategies are included into the consideration, since when the simple_one_for_one is used as the restart strategy, the list of child

specifications must be a list with one child specification only, which is not what we use in this project.

8.6.2 Restart of Child

When child process fails, the restart part in the specification will control if this child process should be restarted or not, there are three kinds of restart method for child processes:

- **permanent:** A permanent child process should be always restarted.
- **temporary:** A temporary child process should never be restarted.
- **transient:** A transient child process should be restarted only when the child process is terminated abnormally, i.e. another exit reason than normal.

8.6.3 exit/2

In order to check the restart strategy of supervisor and the restart method of child process, function exit/2 is used to terminate child process. When a process is restarted by its supervisor, only the pid of process will be changed, with all the information in the specification reserved. Meanwhile, the processes which are supervised by the same supervisor process will be affected.

8.7 Finding

8.7.1 exit/2 with reason shutdown

Function exit/2 contains two parameters, one is the pid of process which is going to be killed, and the other is the reason used when kill the process. When the process is a supervisor, the reason can be one of kill and shutdown, whereas worker process in this thesis project can be killed with reason normal as well.

According to the documentation of supervisor, the child process can have one of the restart methods: permanent, temporary and transient, detailed description is in the Section 8.6.2. Only the reason normal is mentioned, which is that "a transient child process should be restarted only if it terminates abnormally. i.e. with another reason than normal". It doesn't mention the exit reason 'shutdown' at all.

During the tests, it is found that when terminate a child process with the reason shutdown, the child process will be terminated, even the restart type that the child process is transient, which is contradictory to the documentation.

Meanwhile, it is found as well that when terminate a child with type of supervisor with the reason shutdown, the child will still be alive and hold the same pid as before, which is not accordant to the documentation.

8.7.2 Race condition with exit/2

An error report was found during testing supervisor, which is illustrated as below:

```
3> eqc:quickcheck(supervisor_tests:prop_supervisor_test()).
Licence for Chalmers reserved until {{2011,6,6},{14,17,45}}
....Failed! After 5 tests.
```

(The symbolic commands generated before running the real test cases are skipped here)

```
[{{set, {var,1}, {call,supervisor,start_child,  ['#Ref<0.0.0.691>',
        {'#Ref<0.0.0.692>', {supervisor,start_link, {local,'#Ref<0.0.0.692>'},
        supervisor_process,one_for_all]}, temporary,infinity,supervisor,
        [supervisor_process]}]}},
   {ok,<0.444.0>}},
 {{set,{var,3},{call,supervisor_tests,erlang_exit,['#Ref<0.0.0.692>',kill]}},
   ok},
 {{set, {var,5}, {call,supervisor,delete_child,['#Ref<0.0.0.691>','#Ref<0.0.0.692>']}},
   {error,running}}]
{[],[]}
{postcondition,false}
false
```

In this example, the function exit/2 tried to kill the child process '#Ref<0.0.0.692>', and returned ok as the result, which indicated this child process was successfully killed. But when the supervisor tried to delete the child afterwards, error report showed that the child process was still running.

The reason might be that the supervisor hadn't yet handled the exit message, when running function supervisor:delete_child/2. If we insert a delay after calling exit/2, the problem can be fixed.

8.7.3 supervisor:terminate_child/2

According to the documentation of function supervisor:terminate_child/2, it specifies this function can terminate the child process corresponding to the child specification by Id of the child process, but the child specification is still kept by the supervisor. It means that the child process may be later restarted by the supervisor [6]. During the testing we found that when the restart type of child process is temporary, error could happen when trying to restart the child. The example below can describe what this problem looks like:

```
14> eqc:quickcheck(supervisor_tests:prop_supervisor_test()).
........Failed! After 9 tests.
Shrinking..(2 times)
```

(The symbolic commands generated before running the real test cases are skipped here)

```
 [{{set, {var,1}, {call,supervisor,start_child,  ['#Ref<0.0.0.1882>',
{'#Ref<0.0.0.1883>', {supervisor_process,start,['#Ref<0.0.0.1883>']},
temporary,1,worker, [supervisor_process]}]}},
```

```
 {ok,<0.2212.0>}},
{{set, {var,2}, {call,supervisor,terminate_child, ['#Ref<0.0.0.1882>','#Ref<0.0.0.1883>']}},
ok},
{{set, {var,3}, {call,supervisor,restart_child,['#Ref<0.0.0.1882>','#Ref<0.0.0.1883>']}},
{error, {'EXIT', {badarg,
[{erlang,apply,[supervisor_process,start,undefined]},
{supervisor,do_start_child,2},
{supervisor,handle_call,3},
{gen_server,handle_msg,5},
{proc_lib,init_p_do_apply,3}]}}}}]
{[],[]}
{postcondition,false}
falsex
```

The supervisor '#Ref<0.0.0.1882>', started a child with the Id {'#Ref<0.0.0.1883>'. Then it terminates this child process, and returned ok, which mean the child process had been killed. But when the supervisor tried to restart the child process, an error with ‘badarg’ appeared. This was contradictory to the documentation of supervisor:terminate_child/2.

This error was found from the version R14B02. In version R14B01, it was actually possible to restart the child process with the restart type as temporary. In version R14B03, the temporary child process were deleted, when it was terminated, but the transient child process would not be deleted even when it exits with the reason ‘normal’, therefore, the behavior ofs transient child process and temporary child process were inconsistent.

9 Conclusion

In this study, parallel tests for three different modules in Erlang, which are ets, dets and supervisor, have been conducted, with both QuickCheck and PULSE. Due to the ability of generating random processes running in parallel, the possibility of finding the concurrent bugs was improved.

During this project, we got many difficulties and we spent a lot of time on fixing them. There were several functions couldn't guarantee the atomicity of their operation, due to which, when running tests in parallel, it was hard to predict the expected results of these function. It was annoying of getting this error messages all the time. We have tried different possible ways to solve this problem. Finally, the solution of separating functions into two functions, and writing specifications for each of them was proved as the effective way to fix this problem and made the entire test process more efficient.

When testing main functions from different modules, many unexpected and interesting findings, which were against the specification, were got. Since the documentation of these functions were not very clear, even we could imagine that the returned results were not what the user want, we could only make sure that few of them were real bugs.

The found bugs will be reported to Erlang programming team.

Bibliography

[1] Armstrong, J. (2007). *Programming Erlang.* United States of America: The Progmatic Programmers.

[2] Crystal Chang Din. (2009). *Testing Erlang-OTP with QuickCheck.* Göteborg: Chalmers University and Technology.

[3] Quviq AB. (2010). *QuickCheck 1.201* . Quviq.

[4] Quviq AB. (2010). *QuickCheck Flyer.*

http://www.quviq.com/documents/QuviqFlyer.pdf

[5] Quviq AB. (2008). *QuickCheck for Erlang Users.* Stockholm: Quviq.

[6] Ericsson AB. (2009). *STDLIB Reference Manual.* Retrieved 2009, from Erlang/OTP R14B03: http://www.erlang.org/doc/

[7] Quviq AB. (2009). *PULSE 1.24.1.*

[8] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In 14th ACM SIGPLAN international conference on Functional programming (ICFP), pages 149-160. ACM, 2009.

Appendix A. Auxiliary Functions

Auxiliary Function for ETS Tests:

ETS Auxiliary Functions	Description
my_lookup(Key, List)	Return a list of objects with the same Key, from the List.
list_match_object(Pattern, List)	Return a list of objects which match with the given Pattern, from the List.
check_inclusion(List1, List2)	Check if every object in list List1 is in List2 as well.
my_element/1	Elicit the continuation from given parameter, which is a symbolic returned value from symbolic call ets:match_object/1 and ets:match_object/3.
my_deleteobject(Object, List)	Delete Object from list List.
my_keyfind(Cont, List)	Find the relevant information from the state with continuation Cont as the key.
same_find(List1, List2)	Return a list of the object included in both List1 and List2.
insert_cont(Object, L)	▪ Insert Object into the continuation part in the state, which are relevant to function ets:match_object/1 and ets:match_object/3, and saved as list L. ▪ Insert Object into the matchcont part in the state, which are relevant to function ets:match_object/2, and saved as list L.
delete_key_cont(Key, L)	Delete objects with same key from the continuation part in the state, which are relevant to function ets:match_object/1 and ets:match_object/3, saved as list L.
delete_object_cont(Object, L)	Delete Objects from the continuation part in the state, which are relevant to function ets:match_object/1 and ets:match_object/3, saved as list L.
delete_pattern_cont(Pattern, L)	Delete objects match the given Pattern from the continuation part in the state, which are relevant to function ets:match_object/1 and ets:match_object/3, saved as a list L.
delete_key_mcont(Key, L)	Delete objects with same key from the matchcont part in the state, which are relevant to function ets:match_object/2, saved as list L.
delete_object_mcont(Object, L)	Delete Objects from the matchcont part in the state, which are relevant to function ets:match_object/2, saved as list L.
delete_pattern_mcont(Pattern, L)	Delete objects match the given Pattern from the matchcont part in the state, which are relevant to function ets:match_object/2, saved as list L.
my_tab2list(Tab)	Fix function ets:tab2list/1 with function ets:safe_fixtable/2. Return a list of all the objects stored in the table, Tab.
start_match_object(Tab, Pattern)	Return a tuple of a unique reference and a list match with given Pattern from the table Tab.
finish_match_object({MCont, ReturnList})	Extract the ReturnList from the tuple which is returned by function start_match_object/2.

Auxiliary Functions for DETS Tests:

DETS Auxiliary Functions	Description
my_lookup(Key, List)	Return a list of objects with the same Key, from the List.
list_match_object(Pattern, List)	Return a list of objects which match with the given Pattern, from the List.
check_inclusion(List1, List2)	Check if every object in list List1 is in List2 as well.
my_element/1	Elicit the continuation from given parameter, which is a symbolic returned value from symbolic call dets:match_object/1 and dets:match_object/3.
my_deleteobject(Object, List)	Delete Object from list List.
my_keyfind(Cont, List)	Find the relevant information from the state with continuation Cont as the key.
same_find(List1, List2)	Return a list of the object included in both List1 and List2.
insert_cont(Object, L)	Insert Object into the continuation part in the state, which are relevant to function dets:match_object/1 and dets:match_object/3, and saved as list L.
delete_key_cont(Key, L)	Delete objects with same key from the continuation part in the state, which are relevant to function dets:match_object/1 and dets:match_object/3, saved as list L.
delete_object_cont(Object, L)	Delete Objects from the continuation part in the state, which are relevant to function dets:match_object/1 and dets:match_object/3, saved as list L.
delete_pattern_cont(Pattern, L)	Delete objects match the given Pattern from the continuation part in the state, which are relevant to function dets:match_object/1 and dets:match_object/3, saved as a list L.
my_match_object(Cont)	Fix function dets:match_object(Cont) by avoiding using '$end_of_table' as the parameter.
compile/0	Instrument relevant functions will be used in the DETS tests.

Auxiliary Function for Supervisor Tests:

Supervisor Auxiliary Functions	Description
my_terminate([Id], Relation, List)	Return a pair of two lists, which include both the changed relation in the state after terminate the child process Id by calling supervisor:terminate_child/2, and a list of the names of all relevant terminated child processes.
next_which_children([SupRef], Relation)	Return the changed relation in the state, after calling the function supervisor:which_children/1, with the parameter SupRef. Since it needs to use the state, Relation is passed as parameter as well.
post_which_children([SupRef], Relation, Unrelation)	Checking the postcondition of the result returned by function supervisor:which_children/1, with the parameter SupRef. Since it needs to check the state, Relation and Unrelation are passed as parameters as well.
next_state_exit(S, [Id], Reason)	Provide corresponding changes of state, after calling function exit/2, with the Id of process and Reason as parameters. Since this function needs to change the state, the state S is passed as one of the parameters as well.
exit_restart ([Id], Relation)	Return the changed relation in the state, after calling the function exit/1, with the parameter Id. Since this function needs to change the state, Relation is passed as one of the parameters as well.
start_after(Id, Relation, Children, L)	Return a list of name of child processes which is started after the child process Id. Since this function needs to use relation and children in the state, Relation and Children are passed as parameters, as well as an empty list L as an accumulator to collect the results.
my_whichChildren(SupRef, Relation)	Return a list of name of child processes which are supervised by SupRef
my_exit(Id, Reason)	Custemrized function of exit/2 for terminate child processes with the type child.
erlang_exit(Id, Reason)	Custemrized function of exit/2 for terminate child processes with the type supevisor.
compile/0	Instrument relevant functions will be used in the Supervisor tests.

Appendix B. Complete Code

http://www.cse.chalmers.se/~nicsma/zichen_code.zip