FP7 Project ICT-2011-317820 PROWESS

PROWESS: Property Based Testing of Web services

D4.4 Stress testing time dependent software

Lars-Åke Fredlund (UPM)

Abstract

EU-ICT Specific targeted research project (STREP) ICT-2011-317820
Deliverable D4.4 (WP4)

The Prowess project aims to provide the European software industry with efficient and effective testing tools and techniques for web services and internet applications. In this deliverable, we provide a public prototype for stress testing time dependent software.

The prototype is composed of a library, `pulse_time`*, to be used together with the QuickCheck random testing tool [AHJW06] in testing time-dependent Erlang software (e.g., programs containing explicit timers) more efficiently, by permitting precise control over the scheduling of real-time tasks.*

Technically the work has been achieved by adding, to an open source version of the PULSE [CPS+09] user-level scheduler for Erlang programs, a set of new schedulers focusing on real-time behaviour, with the needed infrastructure. One such scheduler, for instance, enables timeouts to occur more frequently (compared to normal schedulings), and thus potentially permits to observe time-dependent program bugs earlier in testing.

The `pulse_time` *library is available under a BSD licence.*

Keyword list: PULSE, Real-Time, Property-based testing, Erlang, Scheduling

WP4 Property based testing of non-functional requirements
Nature: Prototype
Contractual date of delivery: 31/01/2014
Reviewed by: ESL, SP
Weblinks: www.prowess-project.eu

Document ID: Prowess/2013/D4.4/v1.0
Dissemination: PU
Actual date of delivery: 31/01/2014
Updated delivery:

PROWESS Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number ICT-2011-317820.

University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello St.
Sheffield S1 4DP
UK
Tel: +44 114 222 1930
Fax: +44 114 222 1810
Contact person: John Derrick
E-mail: J.Derrick@dcs.shef.ac.uk

University of Canterbury
The Registry Canterbury,
Canterbury,
Kent, CT2 7NZ
UK
Tel: +44 1227 823820
Fax: +44 1227 762811
Contact person: Prof Simon Thompson
E-mail: S.J.Thompson@kent.ac.uk

Chalmers Tekniska Hoegskola Aktiebolag
412 96
Gotetborg,
Sweden
Tel: +46 707563760
Contact person: Prof John Hughes
E-mail rjmh@chalmers.se

Universidad Politécnica de Madrid
Dpto. LSIIS, Facultad de Informática, UPM.
Campus de Montegancedo s/n, 28660 Boadilla
del Monte, Spain
Tel: +34 913366903
Fax: +34 913363571
Contact person: Lars-Ake Fredlund
E-mail: lfredlund@fi.upm.es

Universidade da Coruña
Departamento de Computación (Lab 4.1)
Facultade de Informática
Campus de Elviña S/N 15071 A Coruña
Spain
Tel: +34 981 167 000 ext. 1359
Fax: +34 981 167 160
Contact person: Dr. Laura M. Castro
E-mail: lcastro@udc.es

Quviq AB
Bergshamravagen 4
433 60 Savedalen
Sweden
Tel: +46 70 4388567
Contact Person: Thomas Arts
Email Thomas.arts@quviq.com

Erlang Solutions LTD
New Loom House
101 Back Church Lane
London E1 1LU
UK
Tel: +44 2074561020
Contact person: Torben Hoffmann
E-mail: torben.hoffmann@erlang-solutions.com

Interoud Innovation SL
Avenida De Alfonso Molina 53B
15006 A Coruna
Spain
Tel: +34 981173344
Contact person: Oscar Sacristan
E-mail: oscar.sacristan@interoud.com

SP Sveriges Tekniska Forskningsinstitut AB
Brinellgatan 4
Boras
501 15 Sweden,
Tel: +46 105165359
Contact person: Jonny Vinter
E-mail: jonny.vinter@sp.se

Executive Summary

This document reports on the activities of **Task 4.3** ("Testing time dependent software") of Work Package 4 ("Property based testing of non-functional requirements"), and as such documents **Deliverable 4.4** ("Stress testing time dependent software"), a *prototype* described below.

The results presented in this document are:

1. An open source prototype library, `pulse_time`, to be used in conjunction with QuickCheck in testing time-dependent software by permitting precise control over the scheduling of real-time tasks. Formally the deliverable is the prototype itself.
2. In this document we in addition document the research issues underlying the design of the library, i.e., we describe the technical challenges, explain the detailed design, and overview the experimental results obtained so far with respect to the prototype library.

We also report on our plans and research activities to follow the efforts presented in this document.

Finally, the interactions of these results with the research activities in other work packages are discussed, too, including how this tool integrates with other PROWESS tools (e.g., QuickCheck).

Contents

1 Introduction

Property-based testing relies on the fact that software developers specify the behaviour of their system in one or more properties. From these properties, test cases are automatically generated. However, even if properties are well chosen, and good test cases are derived from these properties, for concurrent and/or time-dependent software checking an individual test case remains a difficult task.

If the system under test is a concurrent one, there are typically a huge number of system schedulings, and testing should explore a substantial number of them. Moreover, if the system also depends on the completion times of subtasks (e.g., *when* subtasks complete, instead of just in *what order* subtasks complete) testing becomes even more difficult. Moreover, a significant challenge is to obtain *repeatable* test executions, i.e., when a bug is found, to be able find out exactly in which order tasks were run, and to be able to later re-run the same execution.

In task 4.3 we create a set of new types of schedulers for Erlang programs, which focus on timed behaviour. One such scheduler, for instance, attempts to prioritise program timeouts. That is, the scheduler will delay some program actions sufficiently so that timeouts become more probable than under normal program execution. The idea is to explore timeout behaviour systematically during testing; normally it is quite difficult to do so as it requires interfering substantially in the execution of a program. Thus, by developing the new scheduler, together with other time-dependent schedulers, we aim to find time dependent bugs in Erlang programs earlier in the development process, something which is notoriously difficult to do using normal testing practices. Moreover, once a bug is found, the test leading to the error condition can be repeated in a deterministic and efficient manner. The technical approach taken is to add a set of new schedulers to the PULSE [CPS+09] user scheduler; the result is a open source version of PULSE, `pulse_time`, which constitutes deliverable D4.4. Since PULSE is well integrated with QuickCheck, the use of these new schedulers for checking programs is straightforward.

In the following we describe the `pulse_time` library, and discuss the technical work underpinning the implementation. Moreover, we analyse how our contributions go beyond the state of the art, and discuss items for future work. We conclude the document by discussing what is to follow from these results, both within WP4 itself, and in interaction with other work packages in the project.

2 Research results

In this section we present the details of our work on developing a prototype for real-time stress testing of Erlang programs, `pulse_time`. The main results are:

- a prototype tool, `pulse_time`, which provides fine control of the (real-time) scheduling of Erlang programs, and permits to explore various scheduling options highly interesting for testing, e.g., a program is infinitely fast (compared to timers), a program is infinitely slow, and a program is realistically fast. Moreover, execution of tests under these schedulers are repeatable and efficient.
- this accompanying document which describes in further detail the design of the tool, and which contains examples that illustrate its use in testing with QuickCheck.

2.1 Technical Work

In the following we will first briefly describe the main constructs for time dependent code in Erlang. Next we proceed by investigating the state-of-the art with regards to testing time dependent software in Erlang. Then, we describe how the PULSE scheduler was modified to provide a set of new schedulers, focusing on timed behaviour. We illustrate the use of the new schedulers using two examples; a more complete evaluation of the `pulse_time` prototype will be conducted in pilot studies.

2.1.1 Handling Time in Erlang

Here we describe the main language features which concern timing, i.e., the receive statement and timestamps.

The receive Statement. The basic mechanism for handling time dependent behaviour in Erlang is the timeout clause of a receive statement:

```
receive
  Pat1 when Guard1 -> Expr1;
  ...
  PatN when GuardN -> ExprN
after Deadline -> TimeoutExpr
end
```

The intuitive semantics of the receive statement is as follows. If a message matches a pattern `PatI` ($1 \leq$ `I` $\leq$ `N`), and the guard `GuardI` (which may contain variables bound by the match) evaluates to true, and moreover no earlier pattern `PatJ` ($1 \leq$ `J` $<$ `I`) matches, or the guard `GuardJ` does not evaluate to true, the message is removed from the mailbox and evaluation continues with expression `ExprI` under the matching binding.

Concretely, the oldest message in the process mailbox is first matched against the clauses according to the above procedure. If no pattern and guard match this message, the same sequence of tests continues with the second oldest message, and so on. If no message matches, the process waits for the reception of a matching message for *at least* `Deadline` milliseconds, until it times out and starts executing the expression `TimeoutExpr`.

A zero deadline corresponds to the case when, if no matching message is in the mailbox, the timeout can happen at once. The special atom **`infinity`** may also be used as a time deadline, signifying waiting forever without timing out (which is equivalent to omitting the **`after`** clause).

Timestamps. The API calls `now()` and `os:timestamp()` return the time elapsed since 00:00 GMT, January 1, 1970 as a tuple `{MegaSeconds,Seconds,MicroSeconds}`.

2.1.2 Handling Time in existing Erlang tools

In the following we consider the transitions (computation steps) an Erlang program may take. A *timed transition* correspond to an Erlang process executing a **`receive`** statement, which contains a *non-zero* timeout (in the **`after`** clause), and such that no message is receivable and the process eventually times out, and starts executing the expression in the timeout clause. Any other computation step that the program might take is considered a *non-timed transition*.

In the following we consider how tools (runtime systems, PULSE) that execute Erlang programs choose between executing timed and non-timed transitions when both are enabled in a program state, and how two (or more) timed transitions with different timeouts are prioritised.

In the case of the normal Erlang runtime system, which is used by default by QuickCheck, there is no particular priority assigned to either executing a timed or a non-timed transitions[1]. However, from the point-of-view of testing concurrent and timed code there are several drawbacks with using the scheduler provided by the runtime system. First, it is too deterministic. In a given situation, the runtime system will always choose the same process to schedule. In testing, instead, one wants to explore all possible schedulings, to prepare a program for running under quite different environmental conditions. In practice the result is that repeating the same test execution many times with QuickCheck (e.g., using the `ALWAYS` macro) is less effective than one could hope. Moreover, if a bug is found, the runtime system does not provide enough information to be able to repeat, or even inspect, in every relevant detail, the test run that lead to the bug.

For this reason (lack of randomness, and lack of repeatability) the PULSE[CPS+09] user level scheduler was developed. PULSE treats the choice of which process to schedule in a completely random way, thus ensuring that repeating test runs has a high chance of exploring new program behaviour. However, using PULSE has a number of disadvantages too. First, programs must be instrumented. The PULSE scheduler is implemented as a normal Erlang process, outside the runtime system, and code that runs under the PULSE scheduler must ask permission to the scheduler before executing a side effect. The requirements on instrumentation make it more difficult to apply PULSE, especially if the system under test relies on libraries that for some reason cannot be instrumented. Deliverable D1.4 (month 29) is intended to address the difficulty of testing using PULSE in the presence of non-instrumented code.

The PULSE scheduler treats timed tasks different from the normal Erlang scheduler. Timed tasks are considered *infinitely slow* compared to non-timed tasks, so that a timeout with a non-zero deadline will never happen unless there is no non-timed task to run. A timed task competing with another timed task will complete in strictly numeric order, i.e., a task with timeout value T_1, competing with a task with timeout value T_2, will timeout only if $T_1 \leq T_2$. Such a semantics is sometimes very useful for testing, and for instance the McErlang [FS07] model checker of Erlang programs also implements this semantics. In this deliverable we generalise the handling of time in PULSE, while preserving the essential property that test runs are repeatable and easily inspected.

[1] at least not without inspecting in detail the implementation of the runtime system

2.1.3 Handling Time in `pulse_time`

The PULSE tool does not keep a dedicated system time (or clock), instead it relies on the normal Erlang runtime to keep the time. In `pulse_time` we instead explicitly keep track of the current time (a number representing the number of microseconds elapsed since 00:00 GMT, January 1, 1970) as part of the state of the revised scheduler. The reason for this decision is to obtain consistent clock values. Consider the program below:

```
Clock1 = now(),
receive
after 1000 -> ok
end,
Clock2 = now()
```

Clearly in a "time-consistent" execution the value of `Clock2` compared to `Clock1` should have increased with at least one second. If we wish to preserve this property, without keeping an explicit system clock, we would have to delay the execution of the above code fragment with up to a second. However, this is clearly undesirable in many situations, as it would severely slow down testing.

Thus, the decisions `pulse_time` has to take, is

- when and how to advance the system time,
- what priorities (if any) should govern the execution of timed and nontimed transitions,
- which API calls to instrument in addition to the ones already instrumented by PULSE (e.g., at least the calls to the `now()` function).

The decisions are controlled by a number of parameters to a test run[2]:

- `timeIncrement` – the maximum cost (time) in microseconds of executing a non-timed transition, which may be zero. The system time is advanced with this value. In addition, system time is advanced upon executing the timeout clause of a receive statement, with the fragment of time left until timing out[3].
- `timeoutInterval` – defines a time interval measured in microseconds, starting with zero and ending in the parameter value, during which timed actions (and possibly non-timed actions) may occur. If a non-timed transition is enabled, no timer with a deadline *after* the time interval may time out first. Intuitively this corresponds to modelling an execution environment where an enabled non-timed transition t is sufficiently fast so that a timer too far in the future (outside the interval) can never be executed before t.
- `timeoutPrecedence` – a boolean value which if true signifies that timed transitions have priority over non-timed transitions if both a timed transition and a non-timed transition is executable in the time interval defined by the `timeoutInterval` parameter. If false, both timed and non-timed transitions are executable in the defined time interval.
- `timeoutJitter` – defines a time interval measured in microseconds, starting with zero and ending in the parameter value, such that if there are two (or more) timeout tasks T_1 and T_2 (with associated deadlines) and $abs(T2 - T1) \leq$ `timeoutJitter` the choice of whether to execute T_1 or T_2 is made randomly without considering their exact deadlines.

Let us illustrate the parameters with an example. Consider the Erlang program below:

[2] these parameters are provided as arguments to the function `pulse_time_scheduler:start/2` which initiates a test run

[3] and potentially some added time too, to model nondeterministic execution

```
Parent = self(),
PidA = spawn(fun () ->
               receive
                 X -> Parent!X
               after 10 -> Parent!a
               end
             end),
PidB = spawn(fun () ->
               PidA!b
             end),
receive Y -> Y end.
```

In the example the "main" process spawns two new processes, and waits to receive a message. The first processes spawned, here referred to as `PidA`, waits to receive a message and forward it to the main process, but if no such message is received within 10 milliseconds, it times out and instead sends the message `a` to the main process. The second process, referred to as `PidB`, sends the message `b` to the first process.

We use the PULSE functionality of producing a figure depicting the scheduling of processes, and sending of messages, to illustrate the role of the above parameters, as shown in figure 1:

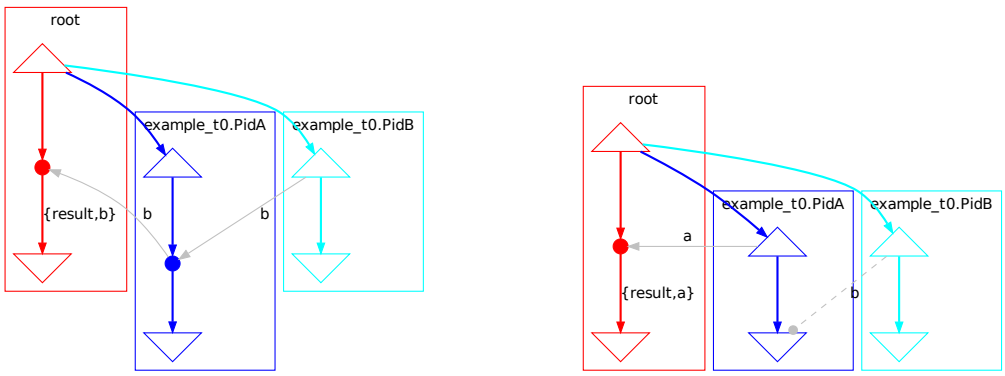

Figure 1: *Left:* `timeoutInterval=0`*, right:* `timeoutInterval=20000,timeoutPrecedence=true`

The figure to the left corresponds to setting `timeoutInterval` to zero, and consequently, `Pid` sends a `b` message that is received by `PidA` which forwards it to the main process. In the figure to the right, in contrast, we have used parameters `timeoutInterval=20000` and `timeoutPrecedence=true` which corresponds to a time interval of 20 milliseconds and letting timed actions have precedence. We see, as a consequence, that although the message `b` is sent by process `PidB` here too, it never gets received at `PidA` which instead times out (since timed actions have priority in the time interval between 0 and 20 milliseconds), and after the timeout sends the message `a` to the main process. The case when `timeoutPrecedence` is false, corresponds to a random choice between first delivering the message from `PidB` or timing out, and as a consequence both the left and right diagrams in the figure correspond to permitted schedulings, and the `pulse_time` tool will randomly choose one such scheduling.

Setting `timeoutPrecedence` to true is a somewhat coarse operation that give priority to *all* timers. There are situations when to test a system one wants to treat different timers in a different manner; we shall see such an example in section 2.1.7 below. To permit this, code can be annotated with the *maximum* time that should be spent in a receive statement until the corresponding timeout clause is activated. Consider a concrete example:

```
receive
  a -> ok
after 10 -> timed_out
end.
```

In Erlang there is no guarantee for when the timeout clause in the above statement will be activated, even if the `a` message never arrives (except the waiting time is *at least* 10 milliseconds).

The `pulse_time` understands specifications of the maximum waiting for such a receive statement, indicated by a call to the function `pulse_time_scheduler:max_wait_time(N)`, where `N` is the maximum waiting time in milliseconds.

Consider the example below:

```
  Parent = self(),
  PidA = spawn(fun () ->
                   pulse_time_scheduler:max_wait_time(N),
                   receive
                      X -> Parent!X
                   after 10 -> Parent!a
                   end
               end),
  PidB = spawn(fun () ->
                   receive
                   after 20 -> ok
                   end,
                   PidA!b
               end),
  receive Y -> Y end.
```

The example is similar to the previous one, except that we specify a maximum waiting time for process `PidA` (a parameter `N`), and process `PidB` unconditionally waits at least 20 milliseconds before sending message `b`.

Figure 2 below depicts the result of running the above program with parameters `timeoutInterval=`**`infinity`**, `timeoutPrecedence=false` (the default), and `timeoutJitter=false` (the default) in the left graph, and `timeoutJitter=15000` (15 milliseconds) in the right graph. Moreover the maximum waiting time (parameter `N`) for both graphs is **`infinity`**.

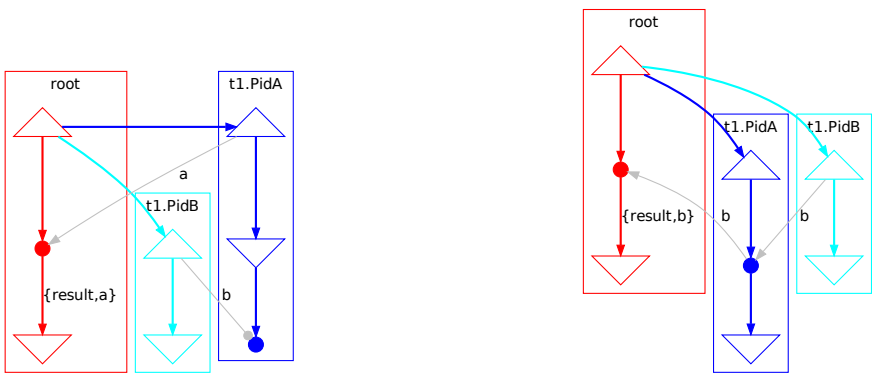

Figure 2: *Left:* `timeoutJitter=false`, *right:* `timeoutJitter=15000`

When the `timeoutJitter` parameter is set to `false`, the situation depicted in the left graph, then process `PidA` will always time out before `PidB` and as a consequence the message `a` will always be delivered to the main process. The right figure depicts one possible execution when `timeoutJitter=15000` (15 milliseconds), where before `PidA` times out, the message `b` arrives and is forwarded to the main process. This is permitted since the jitter specification permits the second timeout clause to "overtake" the first one, in an interval of 15 milliseconds after the first clause becomes enabled. Note that this is permitted, but not mandatory. The right figure remains a valid execution for the whole system when `timeoutJitter` is 15 milliseconds.

Given the same set of parameters as in the right figure, i.e., `timeoutJitter` set to 15 milliseconds, and in addition specifying the maximum waiting time for `PidA` to a value smaller than the second deadline (e.g., `N=20`) then the only valid execution is represented by the left graph (since `PidB` can never time out before `PidA` *must* time out due to the specification of a maximum waiting time).

2.1.4 High-Level Time Parameters

We can reinterpret some of the usual options for prioritising or not prioritising timed transitions as combinations of the above "low-level" parameters. Concretely the following "high-level" time parameter specifications are admitted by `pulse_time`:

Name	Definition	Explanation
`infinitely_fast`	`timeoutInterval=0`	always giving priority to non-timed transitions
`infinitely_slow`	`timeoutInterval=0` & `timeoutPrecedence=true`	always giving priority to timed transitions
`time_random`	`timeoutInterval=0` & `timeoutPrecedence=false`	no priority to either timed or non-timed transitions
{`quite_slow,N`**}**	`timeoutInterval=N` & `timeoutPrecedence=true`	giving priority to timed transitions in the interval 0..N
{`quite_slow_random,N`**}**	`timeoutInterval=N` & `timeoutPrecedence=false`	permitting timed transitions in the interval 0..N

2.1.5 Installing `pulse_time`

The easiest way of using the `pulse_time` library is by installing it as a normal Erlang library. A good alternative is to add the directory containing the installation to the environment variable (under UNIX) `ERL_LIBS`. Note that this verson of `pulse_time` was compiled using Erlang R16B02; running it under an earlier version of Erlang may cause problems. For source code access, please send an email to `lfredlund`fi.upm.es@.

In the following we assume that the contents of the file `pulse_time_binary.zip` has been unzipped, and that the current working directory has been set to that directory. To accomplish this under Linux:

```
$ unzip -x pulse_time_binary.zip
$ cd pulse_time_version_1.0
```

To install `pulse_time` in the normal Erlang directory structure (we assume that Erlang is already installed) type the line `pulse_time_install:install().` to the Erlang shell. Erlang should be started from the directory that contains the file `INSTALLATION`. As an example:

```
> erl

Erlang R16B02 (erts-5.10.3) [source] [64-bit] [smp:12:12] ...

Eshell V5.10.3  (abort with ^G)
1> pulse_time_install:install().
Installation program for pulse_time.

This will install 1.0
in the directory ...
Proceed? y
pulse_time is installed successfully.
ok
```

Depending on the details of the installation of Erlang it is probably necessary to run erlang with super-user privileges (under Unix/Linux) or administrator privileges (Windows) for the installation to succeed.

After restarting Erlang, the status of the `pulse_time` installation can be verified by typing `pulse_time_version:version().` in the Erlang shell.

2.1.6 API

The main functions for interacting with the `pulse_time` tool are located in the modules `pulse_time_scheduler` (for running an instrumented module under `pulse_time`) and `pulse_time_instrument` (for instrumenting a module contained in a file):

- `pulse_time_instrument:c(Files::[string()]) ->any()` – instruments the list of files provided as argument.
- `pulse_time_scheduler:start(Options::[option()], Fun::fun(() ->any()))` – runs the function in the second argument `Fun`, with options (e.g., time options such as e.g. `is_infinitely_fast`) provided in the first argument `Options`.

2.1.7 Example 1: Fischer's mutual exclusion algorithm

To evaluate the expressiveness of `pulse_time` we use Fischer's mutual exclusion algorithm [GM99]. The idea of the algorithm is to use timers to ensure that two (or more) processes cannot both be in a mutual exclusion region (of the program). To communicate between the processes trying to enter their critical regions a shared variable is used.

Fig. 3 contains an implementation of the algorithm in Erlang. The Erlang `ets` tables (a shared memory) is used to implement reading and writing to the shared variable; see the functions `read` and `write` (lines 60-61). The entering of the critical region of a process `Id` is similarly indicated by writing to the shared memory (line 37). To check for correctness, the implementation writes such an entry after acquiring the mutex, sleeps for 1 millisecond, and then reads from the mutex again (lines 39-41). If the protocol functions correctly, the value read should be the written one, otherwise the process reading the value will raise a runtime exception (causing QuickCheck to notice the failure).

Crucial for the correctness of the algorithm is picking the correct values for the timers in line 27 (the `T` parameter) and line 22 (the `D` parameter). We check mutual exclusion in a range of experiments characterised by the parameters of the `start(N,D,T)` function: `N` is the number of

```
1  -module(fischer).
2  -compile(export_all).
3
4  start(N,D,T) ->
5    start(N,D,T,infinity).
6
7  start(N,D,T,MB) ->
8    ets:new(mutex_table,[named_table,public]),
9    ets:insert(mutex_table,{variable,0}),
10   lists:foreach
11     (fun (Id) -> spawn_link(fun () -> idle(Id,D,T,MB) end) end,
12      lists:seq(1,N)),
13   receive X -> X end.
14
15 idle(Id,D,T,MB) ->
16   case read() of
17     0 -> set(Id,D,T,MB);
18     _ -> sleep(1), idle(Id,D,T,MB)
19   end.
20
21 set(Id,D,T,MB) ->
22   max_sleep(1,D),
23   setting(Id,D,T,MB).
24
25 setting(Id,D,T,MB) ->
26   write(Id),
27   sleep(T),
28   testing(Id,D,T,MB).
29
30 testing(Id,D,T,MB) ->
31   case read() of
32     Id -> mutex(Id,D,T,MB);
33     _ -> idle(Id,D,T,MB)
34   end.
35
36 mutex(Id,D,T,MB) ->
37   ets:insert(mutex_table,{mutex,{enter,Id}}),
38   sleep(1),
39   case ets:lookup(mutex_table,mutex) of
40     [{mutex,{enter,Id}}] -> ok
41   end,
42   write(0),
43   ets:insert(mutex_table,{mutex,void}),
44   if
45     MB>1 -> idle(Id,D,T,MB-1);
46     true -> ok
47   end.
48
49 sleep(Milliseconds) ->
50   receive
51   after Milliseconds -> ok
52   end.
53
54 max_sleep(SleepMin,SleepMax) ->
55   scheduler:max_wait_time(SleepMax),
56   receive
57   after SleepMin -> ok
58   end.
59
60 read() -> [{variable,Value}] = ets:lookup(mutex_table,variable), Value.
61 write(Value) -> ets:insert(mutex_table,{variable,Value}).
```


Figure 3: *Fischer's mutual exclusion algorithm coded in Erlang*

processes, `D` is the *maximum* time to wait until writing to the shared variable, and `T` is the *minimum* time to wait until reading from the shared variable. In this experiment we will assume that non-timeout transitions takes zero microseconds to execute (i.e., `timeIncrement=0`). Moreover we assume that internal actions are infinitely fast compared to timeouts, and that timers can release in any order, irrespective of their values (still, of course, respecting maximum waiting times).

```
1  fischer_prop() ->
2    SchedParms =
3      [infinitely_fast,
4       {timeParms,[{timeoutJitter,infinity}]},
5      {verbose,false},
6      {eventLog,false}],
7    ?FORALL
8       ({N,D,T},
9        {pos_nat(),pos_nat(),pos_nat()},
10       begin
11         io:format("Trying␣configuration␣~p,~p,~p~n",[N,D,T]),
12         Result =
13           pulse_time_scheduler:start
14             (SchedParms,
15              fun () -> fischer:start(N,D,T,?NUM_WRITES) end),
16         some_process_is_alive(Result)
17       end).
18
19 test() ->
20   pulse_time_instrument:c(["fischer"]),
21   eqc:quickcheck(eqc:on_test(fun teardown/2, ?ALWAYS(10,fischer_prop()))).
22
23 teardown(_,_) ->
24   ets:delete(mutex_table).
25
26 pos_nat() ->
27   ?SUCHTHAT(X,nat(),X=/=0).
28
29 some_process_is_alive(Result) ->
30   case lists:keyfind(live,1,Result) of
31     {live,Pids} -> length(Pids)>0;
32     Other -> false
33 end
```

Figure 4: *The QuickCheck code for testing Fischer's algorithm*

In figure 4 we show the QuickCheck testing code for running the example. The main function is `test()`, in lines 18–20, which first instruments the `fischer` module, and then invokes QuickCheck on the property defined in lines 1–16. The function `teardown` is responsible for cleaning up state after a test run, e.g., to delete the shared memory table.

The `fischer_prop` property first selects parameters `N,D,T` as small random positive natural numbers, and proceeds to run the example by invoking the pulse scheduler with the specified time parameters on the function `fischer:start`. To obtain finite executions we have added a new parameter, `?NUM_WRITES`, to the implementation of Fischer's algorithm which puts a maximum bound (20 in the example below) on the number of times a critical region can be entered by each process. When the scheduler terminates, we inspect the result and signal testing success (no error detected) if there are still alive processes (but which are deadlocked), and failure otherwise. The `fischer_prop` QuickCheck property is repeated 10 times (using the `?ALWAYS` construct) in

order to take into account the non-deterministic behaviour of the program under test.

```
$ erl -pa ebin -pa examples/ebin
Eshell V5.10.3  (abort with ^G)
1> fischer_qc:test().
Starting Quviq QuickCheck version 1.30.2
   (compiled at {{2013,11,18},{14,34,14}})
Licence for University UPM Madrid reserved until {{2014,1,27},{17,29,19}}
Trying configuration 3,2,1
*** Error: mutex should be held by 2 but status is [{mutex,{enter,1}}]

=ERROR REPORT==== 27-Jan-2014::16:35:14 ===
Error in process <0.159.0> with exit value: {{nocatch,bad},[{fischer,mutex,4,[]}]}

Failed! After 1 tests.
{3,2,1}
Trying configuration 2,2,1
*** Error: mutex should be held by 1 but status is [{mutex,void}]

=ERROR REPORT==== 27-Jan-2014::16:35:14 ===
Error in process <0.173.0> with exit value: {{nocatch,bad},[{fischer,mutex,4,[]}]}

Shrinking.Trying configuration 1,2,1
Trying configuration 2,1,1
(1 times)
{2,2,1}
false
```

Figure 5: *A typical QuickCheck testing run for Fischer's algorithm*

Figure 5 depicts a typical testing run. As expected we see that values where the `D` parameter is less than or equal to the `T` parameter produces an error. QuickCheck first generates values `3,2,2`, which fails, and then shrinks the example to `2,2,1`, again failing. Next QuickCheck shrinks to `1,2,1`, which does not fail.

2.1.8 Example 2: the lift example

As a high-level example of how by selecting the right treatment of timeouts we can debug programs we focus next on a larger program, a control system that manages a set of elevators. This example has been used throughout an Erlang/OTP literacy course, and makes use of several OTP components: the supervisor component, the generic server (`gen_server`) component, the generic finite state machine (`gen_fsm`) component and the generic event (`gen_event`) component. The code contained several errors and the students were guided to observe the erroneous behaviour and to correct it step by step. There is also an simplistic interactive graphic simulator that displays the floors, the elevator, and the buttons that control the elevator. To operate the elevator two types of commands are available: pressing the button calling for the elevator at a floor, or the floor button inside an elevator to go to a certain floor. Here we will only test a very limited part of the elevator functionality directly relating to timed behaviour.

In figure 6 we shows the simple graphic simulator in action, controlling two elevators working three floors.

The elevator example consists of 13 Erlang modules, around 1670 lines of Erlang code and with, for example, two elevators, the maximum number of running process will be 10. The complete source code for the example is included in the D4.3 `pulse_time` prototype. The set-up of

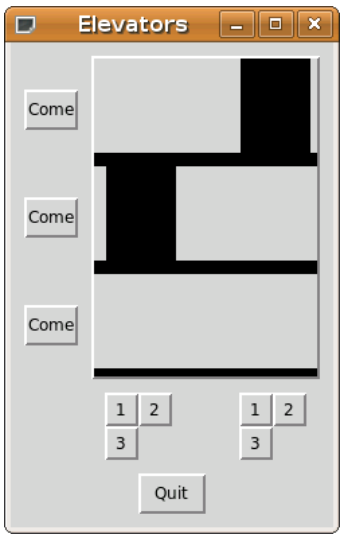

Figure 6: *The graphical simulator with two elevators and three floors*

an elevator system with two elevators is illustrated in Fig. 7. The simulation supervisor process (`sim_sup`) is responsible for spawning the graphics supervisor process (`g_sup`), and the supervisor process of the elevator system (`system_sup`). The latter is responsible for spawning the elevator scheduler process, the process that generates a system event any time something significant occurs in the system (`sys_event`), and the supervisor process for the elevators. The `e_graphic` process is the control process for the graphical (simulated) elevator and the `elevator` process is the control process of each elevator.

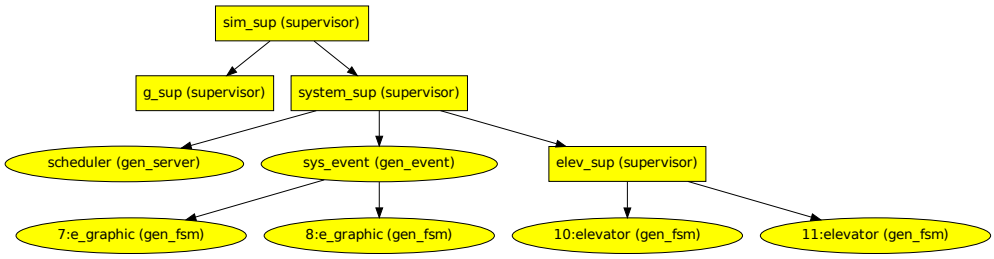

Figure 7: *Processes in the elevator example*

To start the elevator system we should call the function `sim_sup:start_link(InitFloor,NumFloors,NumElevators)` where `InitFloor` is the floor where all elevators start, `NumFloors` is the number of floors, and `NumElevators` is the number of elevators. For testing, the number of of floors and elevators are selected randomly, and the initial floor too.

To issue commands to the system the function `lift_scheduler:f_button_pressed` is called, signifying an order for the elevator to go to a randomly generated floor (`f_button` is the button available at each floor for asking the elevator to come). As in the case of Fischer's algorithm a runtime exception represents a testing error.

If the elevator system is started by making a call to `sim_sup:start_link`, and then, after waiting for 1ms, send a command using `lift_scheduler:f_button_pressed`, one would expect the system to either ignore the command, or queue it, until the whole quite complex elevator control system has properly booted. Unfortunately this is not the case. Below the experiences of testing the code with different high-level parameters are summarised:

High-level property	result
`infinitely_fast`	no error found
`infinitely_slow`	error found
`time_random`	error found
`{quite_slow,250}` and `{timeIncrement,5}`	no error found
`{quite_slow,5000}` and `{timeIncrement,50}`	error found

The `infinitely_fast` schedulers, as expected finds no error as it prioritises non-timed transitions. Both the `time_random` and `infinitely_slow` schedulers quickly found the error, as they make it very likely that such timing errors are properly tested. The `{quite_slow,250}` scheduler represents a quite "realistic execution", where timers behave more or less as in a normal running system (and thus no error is detected). However, by tailoring the parameters (`{quite_slow,5000}` to make timeouts more likely to happen early, the error can be found.

It turns out that the error is due to a race condition: the system receives the 'press button'-command before it has been fully initialised. This is a bug in the elevator code, and similar errors are likely to be found in many other control systems.

2.2 Contributions beyond the state of art

As was discussed in section 2.1.2, existing Erlang-based tools (PULSE with QuickCheck, and QuickCheck standalone) for testing concurrent software do not focus on finding timing related errors. In contrast an extension to the McErlang model checker [EF12] does permit model checking of timed Erlang programs. However, model checking remains a difficult technique to apply in practice, due to e.g. state explosion during checking, or the difficulty of abstracting a program into a verifiable model. The Concuerror tool [CGS13], which combines testing and model checking techniques, treats timeouts as non-deterministic choices. An extension of PULSE reported in [AHN+11], termed "procrastination", focuses on steering a test run (i.e., guiding the scheduler) towards finding likely race conditions in a program due to use of shared resources, but does not address timing issues. One could interpret e.g. the design of the "infinitely slow" scheduler presented in this deliverable as doing something similar, e.g. steering the scheduler towards a likely bug condition due to timing issues.

Tools for uncovering concurrency errors in other programming languages and platforms such as e.g. Java and C++ are becoming common; Chess [MQB+08] by Microsoft is the inspiration for the Concuerror tool. For Java the RaceFuzzer [Sen08] and AssetFuzzer [LCC10] tools is related to PULSE "procrastination", in that in a first state testing runs are collected, and then analysed to produce new schedules deemed likely to provoke concurrency errors.

With regards to real-time verification, the Uppaal [LPY97] tool is the currently most well-known model checker for real-time systems. It provides an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays etc.).

In this deliverable, in contrast, we have extended PULSE with a set of new schedulers that focus on timed behaviour, including efficient schedulers whose behaviour changes depending on the *values* of timers. The result framework permits us to more systematically search for *timing bugs during testing*, and their utility has been illustrated in examples.

Moreover, the work presented in this deliverable complements the efforts cited above. There is nothing that prevents a QuickCheck user presented with a testing problem from using our schedulers for a portion of the total testing time, and regular QuickCheck, or QuickCheck combined with regular PULSE (with or without procrastination), for other portions of the total testing time.

2.3 Future work

The extension to PULSE realised by the `pulse_time` library will be thoroughly evaluated in the context of the pilot study task, as described and reported in D6.3.

We can foresee that some case studies will require a more finely grained approach to handling timers. For instance, we might want to prioritise timeout for certain parts (processes, modules) of a program only, and let other parts of the program execute normally.

Finally, in deliverable D1.4 we plan to address the problems of applying PULSE to non-monolithic systems, that is, systems where only part of the code can be instrumented for use with PULSE (or `pulse_time`). This is the case, for instance, for systems partly composed of Erlang code and with other parts written in e.g. C or Java.

3 Relevance to PROWESS

3.1 Relevance to project objectives

The common intent of all WP4 tasks is to apply property-based testing not only to the verification of functional properties, as was predominantly the case in the past, but also to the verification of non-functional properties (with regards to real-time concerns, load testing, fault injection, etc).

The Erlang programming language, and its libraries, have become a very important programming platform for developing the "back ends" that ensure reliability and 24/7 availability of internet services. As an example, the distributed, fault tolerant and highly performing database Riak has become a cornerstone in the back ends of many internet servers, and is written predominantly in Erlang. The focus of the present deliverable is precisely to help verify such types of software, written in Erlang, and whose correctness is critical for the successful deployment of internet services.

Clearly thus the present work on extending the existing user level scheduler PULSE with new features to more directly address the testing of real-time properties, classically considered non-functional properties, is highly relevant to the stated goals of WP4 (and the project in general).

3.2 Relation to other work packages

The PULSE tool is also used in other work packages, perhaps principally in WP1, and our extension will benefit that work package too. As future work, it remains to address the problem of making PULSE (and indeed `pulse_time`), and QuickCheck in general, more applicable for the testing of heterogeneous code, i.e., code written partly in Erlang and partly in e.g. C; this will be reported in deliverable D1.4. Moreover `pulse_time` will be thoroughly evaluated in the context of the pilot study task, as described and reported in D6.3.

3.3 Integration with other PROWESS tools

The `pulse_time` is straightforward to use with other PROWESS tools, e.g., QuickCheck, as demonstrated in the examples included in this deliverable documentation. The installation of `pulse_time` follows the techniques adapted for QuickCheck installation; an Erlang beam file ("pulse_time_install.beam") is distributed that when invoked automatically installs `pulse_time` in the right location (see section 2.1.5 for further instructions). Then, to use `pulse_time` one can simply call its two main functions to instrument, and to execute the program under test (see API documentation for further details) from any Erlang program, including QuickCheck. Moreover, it is possible to PULSE have installed at the same time as `pulse_time`, no conflicts will occur.

References

[AHJW06] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM.

[AHN+11] Thomas Arts, John Hughes, Ulf Norell, Nicholas Smallbone, and Hans Svensson. Accelerating race condition detection through procrastination. In Kenji Rikitake and Erik Stenman, editors, *Erlang Workshop*, pages 14–22. ACM, 2011.

[CGS13] Maria Christakis, Alkis Gotovos, and Konstantinos F. Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *ICST*, pages 154–163. IEEE, 2013.

[CPS+09] 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 *Proceeding of the 14th ACM SIGPLAN Int. conf. on Functional programming (ICFP)*, pages 149–160, 2009.

[EF12] Clara Benac Earle and Lars-Åke Fredlund. Verification of timed Erlang programs using McErlang. In Holger Giese and Grigore Rosu, editors, *FMOODS/FORTE - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012*, volume 7273 of *Lecture Notes in Computer Science*, pages 251–267. Springer, 2012.

[FS07] L-Å. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *Proceeding of the 12th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 125–136, Freiburg, Germany, 2007. ACM.

[GM99] Eli Gafni and Michael Mitzenmacher. Analysis of timing-based mutual exclusion with random times. In *In Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 13–21. ACM Press, 1999.

[LCC10] Zhifeng Lai, S. C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 235–244, New York, NY, USA, 2010. ACM.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2), 1997.

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[Sen08] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.