RELEASE

SEVENTH FRAMEWORK PROGRAMME

ICT-287510
RELEASE
A High-Level Paradigm for Reliable Large-Scale Server Software
A Specific Targeted Research Project (STReP)

D5.3 (WP5): Systematic Testing and Debugging Tools

Due date of deliverable: June 30, 2014
Actual submission date: June 30, 2014

Start date of project: 1st October 2011 Duration: 36 months

Lead contractor: University of Kent Revision: 0.1 (30th June 2014)

Purpose: To describe techniques and tools developed within RELEASE for the systematic testing and debugging of Erlang applications.

Results: The main results presented in this deliverable are:

- an overview of the refactoring assistance provided within Wrangler to guide the application of refactorings to particular systems based on monitoring data and its analysis;
- the architecture and the implementation technology of Concuerror, a systematic testing tool for concurrent Erlang programs.

Both tools are publicly available and have been presented to the Erlang programming community on various occasions during the lifetime of the project.

Conclusion: The deliverable has provided tool support for the development of Erlang systems with a focus on SD Erlang, and Erlang concurrency.

Project funded under the European Community Framework 7 Programme (2011-14)			
Dissemination Level			
PU	Public		✳
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Systematic Testing and Debugging Tools

Simon J. Thompson <s.j.thompson@kent.ac.uk>
Stavros Aronis <stavros.aronis@it.uu.se>
Huiqing Li <H.Li@kent.ac.uk>
Konstantinos Sagonas <kostis@it.uu.se>

Contents

1 Executive Summary

This document presents the third deliverable of Work Package 5 (WP5) of the RELEASE project. WP5 is concerned with developing tools that support the effective deployment and development of Erlang/OTP software for massively parallel systems.

More specifically, this report presents:

- in Section 3, refactoring assistance within Wrangler designed to guide the application of refactorings to particular systems based on monitoring data and its analysis; and
- in Section 4, the architecture and main aspects of the implementation technology of Concuerror, a tool for the systematic testing of concurrent Erlang applications.

We have enhanced Percept2 and Wrangler to establish a link between 'bad smells' related to concurrency and distribution within (SD) Erlang, as well as refactorings for eliminating them. To minimize the amount of code change to user's applications when a refactoring is applied, we try to provide users with ready-to-use s_group patterns and library functions, as well as providing an enhanced version of the Wrangler program analysis API for users to write their own code inspection functions.

We have designed and implemented a systematic testing tool for Erlang programs, called Concuerror, which, given a set of user-supplied tests, is able to detect concurrency errors in the programs or verify their absence, something that is not possible to do with unit or random testing. In the process of developing the tool, we have developed and proposed two new novel algorithms for dynamic partial order reduction. One of them in particular, is the first such algorthm which is provably optimal and significantly outperforms and extends the state-of-the-art in the area. Both algorithms as well as various other options that can be controlled by the user by are implemented in the tool and briefly presented in this document.

2 Introduction

The main goal of the RELEASE project is to investigate extensions of the Erlang language and improve aspects of its implementation technology in order to increase the performance of Erlang applications and allow them to achieve better scalability when run on big multicores or clusters of multicore machines. Work Package 5 (WP5) of RELEASE aims to build tools to support the effective deployment and development of Erlang/OTP software for massively parallel systems. These tools include augmenting the Wrangler refactoring system with a refactoring assistance that guides users on how to apply the refactorings to systems on the basis of offline monitoring data, and techniques and tools to support the systematic testing of Erlang programs by effectively detecting time-dependent concurrency bugs. The lead site of WP5 is University of Kent. The tasks of WP5 pertaining to this deliverable are:

Task 5.4: "*... develop within Wrangler a refactoring assistant to suggest how the refactorings can be applied in practice to a given system, informed by the analysis of offline and online monitoring data for that system.*"

Task 5.5: "*... develop techniques that, given a testsuite, will find and reproduce Heisenbugs (e.g. deadlocks, livelocks and data races) in concurrent Erlang programs, will effectively explore the large state space of possible interleavings and provide quantified coverage guarantees.*"

The standard Erlang distribution comes with a set of tools that provide complementary facilities for tracing, monitoring, profiling and debugging, and an overview of these tools was given in the first deliverable for this work package, D5.1, and revisited briefly in the second, D5.2. These tools are used together to support complementary aspects of program development and maintenance.

In delivering the work required in Work Package 5, the RELEASE team has followed a twin-track approach.

- Where that is appropriate we have chosen to enhance existing tools – such as Percept (to Percept2) and Wrangler – rather than to duplicate existing functionality in new tools. In particular we have enhanced Percept to deal explicitly with concurrent systems running on parallel and distributed platforms, and added to the built-in code inspection functions and refactorings of Wrangler, rather than developing a separate refactoring tool.

- When these is no overlap with an existing tool, we have developed *new* tools – such as Concuerror and Devo – which can be adopted alongside existing tools.

Towards fulfilling these tasks, this deliverable (D5.3) presents techniques and tools for the systematic testing and debugging of concurrent Erlang programs that support the effective deployment of Erlang applications and identify errors in them, focussing on errors related to concurrency.

The main body of this document describes:

- the design and implementation of the refactoring assistance provided within Wrangler to guide the application of refactorings to particular systems based on monitoring data and its analysis (Section 3) ; and
- the architecture and main aspects of the implementation technology of Concuerror, a publicly available tool for the systematic testing of concurrent Erlang applications (Section 4).

The report ends with a brief section with some concluding remarks.

The work for this deliverable has been done mainly by researchers from the University of Kent (UNIKENT), Uppsala University (UU), and the Institute of Communication and Computer Systems (ICCS). The breakdown of the work was that the UNIKENT team developed the refactoring assistance within Wrangler, while the UU team (mainly) and researchers at ICCS developed Concuerror, the systematic testing tool for finding concurrency errors in Erlang programs.

3 Refactoring Assistance

We aim to provide tool support for guiding the application of refactorings to particular systems based on profiling data and its analysis. To support this iterative profiling and refactoring process, we have extended Percept2 and Wrangler to establish a link between bad concurrency- and distribution-related 'smells', as well as refactorings for eliminating them. To minimize the amount of code change to user's applications when a refactoring is applied, we try to provide users with ready-to-use s_group patterns and library functions. Wrangler provides a program analysis API [14] for users to write their own code inspection functions, and this has been further enhanced with program slicing support for understanding expression-level dependency within the scope of a function clause.

Percept2 [15] has been used in various case studies to improve application performance, these applications include Percept2 itself, Wrangler's similar code detection algorithm and program rendering algorithm, as well as SD-Orbit which is the SD Erlang implementation of the Orbit application. Our use experience of Percept2 has motivated a number of extensions which could facilitate the identification of bottlenecks and the locating of relevant source code. We cover the main extensions in the next two subsections.

3.1 Concurrency-related and distribution-related 'bad smells'

We have identified a number of common 'bad smells' that should attract a user's attention while browsing through the profiling data. These include:

Insufficient parallelisation/scheduler under-utilisation. In order to make full use of the multicore resource, an Erlang application should have enough processes to keep all the schedulers busy to avoid (as much as possible) any schedulers being inactive. Percept2 provides users with not only a histogram of the number of active schedulers, but also statistical data about the utilisation of each individual scheduler. A low utilisation rate indicates inadequate parallelism and potential for performance improvement. Insufficient parallelism could be fixed by

spawning more processes to do the job in most cases. Depending on the concrete application, there are different ways to refactor an application in order to introduce parallelism, such as shifting part of the computation of an existing process to one or more new processes if the new process and the parent process can be run in parallel, duplicating a server process, etc.

Over parallelism. In contrast to insufficient parallelism, where there are not enough runnable processes to keep all the schedulers busy, over parallelism refers to the situation when the ratio between the number of runnable processes and the number of schedulers available is too high. A very high ratio indicates a large number of unfinished processes, which could potentially be a problem if the amount of memory used by each process is large. For the detection of over parallelisation, Percept2 provides a histogram of the ratio between the number of runnable processes and the number of schedulers available. Applications with too much parallelism could be refactored to either reduce the number of processes spawned or to introduce a mechanism to control the runnability of processes.

Processes with heavy load. Given a time interval T, we say that a process is heavily loaded over the period of T if the ratio between its accumulated runtime, R say, and T is above a threshold specified. If the situation of over-loaded processes and scheduler under-utilisation coexist, then those processes with heavy load are candidates where potential parallelism could be introduced. Those processes should be refactored if possible to shift part of its computation to other processes.

Large messages. Due to the copying of data between processes when a message is sent from one process to another, large message passing is considered as a 'bad smell'. To detect large messages, Percept2 provides users with not only the process communication graph with filtering support, but also the querying facility for reporting the top N largest messages sent. One possibility to remove large message passing is to shift the computation between processes so that the original message can be computed in the receiver process, another possibility is to allow the sharing of data between processes by using ETS tables or database. This kind of refactoring process is very much application dependent.

Large process entry arguments. Due to the same data-copying reason, large process entry arguments are also undesirable. Again Percept2 allows the user to query the top N processes ordered by the size of their entry arguments. Similar to the handling of large message passing, same strategies can be used to avoid this 'bad smell'.

Very short-lived processes. By short-lived processes, we refer to those processes that have a very short accumulated runtime or lifetime. This kind of processes may not improve application scalability if the overhead of setting up a process and waiting for a reply is greater than the benefit of using parallel processes to do the job. Depending on the functionality of these processes, if might be possible to replace the process with sequential function calls, or to spawn a smaller number of such processes with each doing more computation.

Processes that receive/send a very large number of messages. This might not be a 'bad smell' given the fact that message passing is how processes are suppose to communicate with each other, but depending on the workload of the receiver processes, in some scenarios a process receiving a large number of messages might also be an overloaded process. So this is another sign to watch out when browsing the profiling data.

Large cluster of distributed nodes. With the current model of distributed Erlang, the larger the number of nodes in a cluster, the more expensive it becomes for each node to periodically check the liveness of connections, and the longer it takes to get the replications of global names updated. This limits the scalability of distributed Erlang when the number of nodes in an Erlang cluster goes into the hundreds. Percept2 provides users with the functionality to visualise the overall communication graph, hence the connectivity of nodes, in a distributed system.

Figure 1: Report page added to Percept2

One approach to reducing the size of node clusters of a distributed Erlang system is to use *s_groups*, which is one of the major outcomes of WP3 of this project. With *s_groups*, nodes within the same cluster have transitive connection, and share the same global name space, whereas nodes from different *s_groups* can communicate via a gateway node that serves as a bridge between two *s_groups*. Wrangler's support for introducing *s_groups* to an Erlang application is illustrated in Section 3.3.

To support the identification of these 'bad smells', we have extended Percept2 with a new *Report* page, which shows the scheduler utilisation rate, the histogram of the ratio between the number of runnable processes and the number of available schedulers, as well as the top N, whose value is input by the user, processes sorted by a particular criterion, such as runtime, garbage collection time, blocking time, etc. Figure 1 shows part of such a report page.

3.2 Navigation through source code in Percept2

Something that a user would like to do while browsing through the profiling data is to look at the source code in order to better understand the profiling data and/or the code itself. In previous versions of Percept2, the user had to open a text editor, browse through the code and locate the function of interest manually. To make the navigation through source code easier, we have extended Percept2 to link function entries, i.e. those shown in the process, or function, information page, to their definitions in the source code. In order to make source code available, the user needs to tell Percept2 the location of the source code; this can be done either by calling the command:

```
percept2:load_code(PathsToSourceCode)
```

after the Percept2 web server has been started, or by specifying the source code location when starting the Percept2 web server, e.g.

```
percept2:start_webserver(PortNumber, {src, PathsToSourceCode}).
```

As an example, Figure 2 shows the information page of `sim_code_v0:sim_code_detection_1/6`, and Figure 3 shows the function definition.

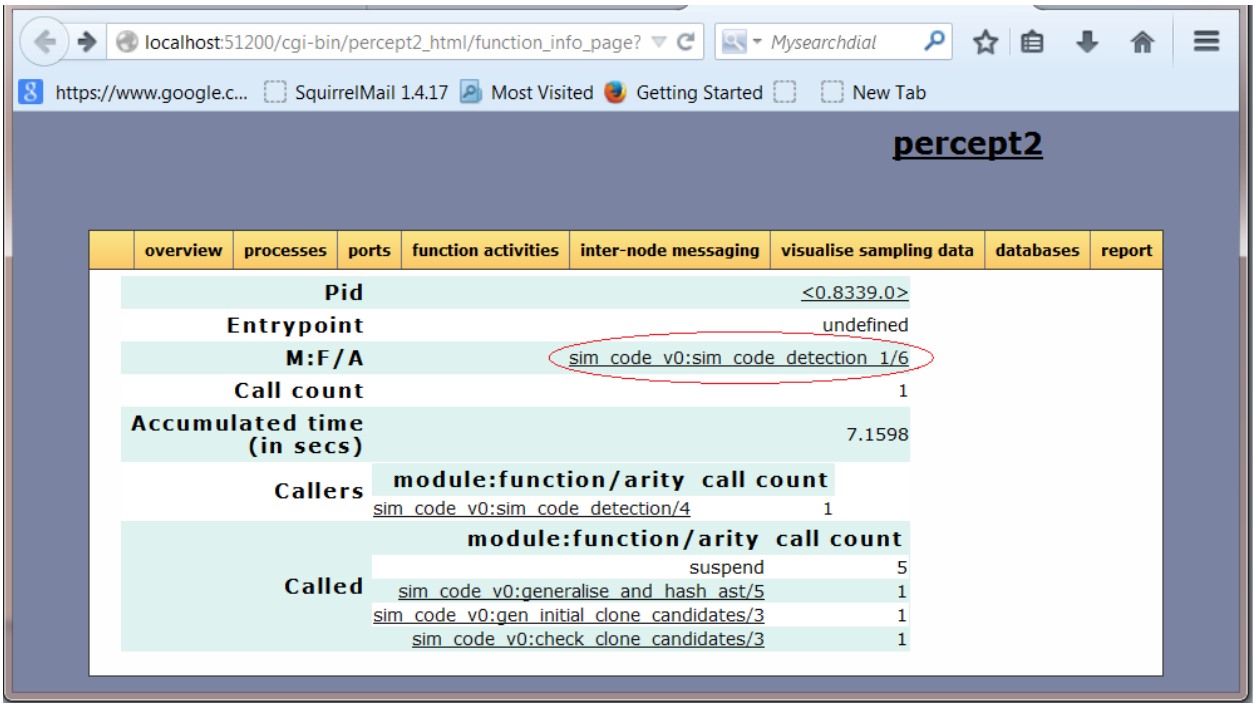

Figure 2: Function information page

Figure 3: Source code visualisation

3.3 Reusable Generic s_group Patterns

Orbit (https://github.com/amirghaffari/Orbit) is a distributed Erlang application used as a benchmark case study by the RELEASE project. Given a space X, a list of generators $f\_1, ..., f\_n : X \to X$, and an initial vertex $x\_0 : X$, the goal of Orbit is to compute the least subset Orb of X such that Orb contains x_0 and is closed under all generators.

The distributed Orbit implementation consists of a master node and a collection of worker nodes. A master process in the master node spawns worker processes to worker nodes, starts the computation and collects statistics from the worker processes after the completion of the computation. The master node itself is also a worker node, therefore hosts a number of worker processes as well. Processes from different nodes have the freedom to communicate with each other directly, as a result, all the nodes participating the computation form a single cluster. The bigger the cluster size, the more overhead incurred to keep the connectivity between nodes. As a case study, the research

group from Glasgow University has rewritten the distributed Orbit implementation to make use of *s_groups*, so that multiple node clusters are formed, each of a smaller size. We refer the original distributed implementation of Orbit as D-Orbit, and this new implementation as SD-Orbit.

While the performance comparison between SD-Orbit and D-Orbit looks promising, the re-implementation involves heavy rewriting of the original implementation, due to the deep coupling of s_group manipulation and the application logic. This kind of rewriting requires deep understanding of the application under consideration. Being application-dependent makes it extremely hard to automate the rewriting process, and the amount of effort involved to incorporate s_groups into existing applications may prohibit people from doing so.

For this reason, we have examined another approach to incorporating s_groups into distributed Erlang systems. This approach aims to minimize the changes needed to the original implementation as much as possible, and ideally make it possible to automate the migration using a refactoring tool like Wrangler. To illustrate our approach, again we use D-Orbit as an example.

The key idea of our approach is *reusable generic s_group patterns*. An *s_group pattern* implements a particular way of grouping Erlang nodes into certain cluster structure, and is application independent. Regardless of the final cluster structure to be constructed, an s_group pattern implements a number of things, and these include:

- Functions for setting up the s_group structure according to the pattern specified.
- Functions for spawning gateway processes which are in charge of the message relaying from one s_group to another. There can be more than one gateway process in each gateway node in order to avoid the situation where a gateway process becomes the bottleneck of the system because of the large amount of traffic passing trough it. Routing information is stored in each gateway node so that a gateway process knows where to forward each message it receives.
- s_group-specific `send` and `spawn` functions. An s_group `send` function behaves exactly the same as the built-in `send` function if the message receiving process belongs to a node that shares the same s_group with the sending process; otherwise if the target process belongs to a different s_group, the message is going to be sent to a gateway process first, then relayed to the target process. In a similar way, spawning a new process in another s_group should also go through a gateway node.

The s_group pattern we have implemented so far is to create a central gateway s_group structure, as shown by the example in Figure 4, where a central gateway s_group is used to connect together a number of s_groups each of which consists of a number of worker nodes and the gateway node that belongs to the central gateway s_group. The example shown in Figure 4 has five s_groups. The s_group in the middle is the gateway group, which overlaps with each of the remaining four s_groups. Nodes within the same s_group are transitively connected to each other, however in order to send a message to a process in a different s_group, the message will have to hop through some gateway nodes before reaching the target process.

With this pre-implemented s_group pattern, we were able to refactor the D-Orbit implementation into another SD-Orbit implementation with very little effort. In fact, the new implementation can be run either in the original distributed way or in the SD Erlang way depending on the value of the macro `SDOrbit`. This version of `SD-Orbit` implementation is available from `https://github.com/RefactoringTools/SD-Orbit`.

Altogether, five places in the D-Orbit implementation were refactored. The most complex change involves adding code for setting up the s_group structure before running the Orbit computation. This refactoring is shown in Figure 5. As it shows, depending on the value of the macro `SDOrbit`, the application can be run with, or without, s_groups. If s_groups are to be used, the size of each s_group needs to be specified. Given all the available nodes, 16 say, and the maximal size of each s_group, 4 say, the function `central_grouping:create_s_groups/3` creates the s_group structure as shown in Figure 4.

The remaining four changes involve replacing the uses of `spawn` with `central_grouping:spawn` and the uses of `!` with `central_grouping:send`. All these changes are fairly trivial and easy

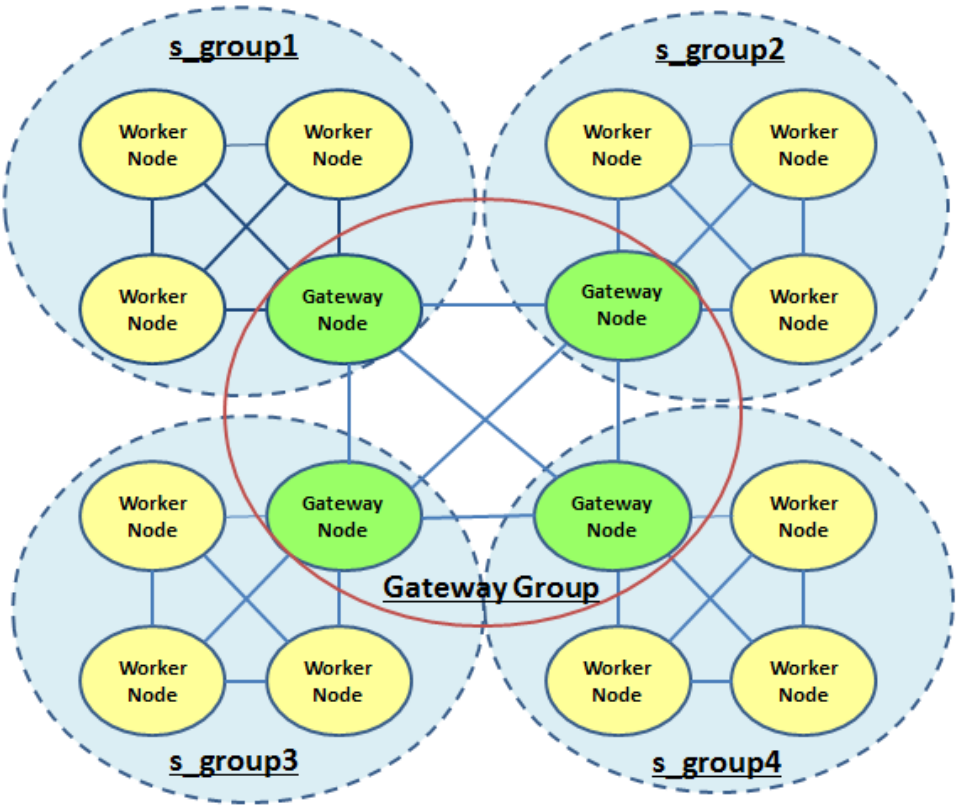

Figure 4: A central gateway s_group structure

```
case ?SDOrbit of
    true -> %% run SDorbit
        GroupSize = 4, %% Number of nodes in each s_group
         case central_grouping:create_s_groups(Nodes, GroupSize) of
            {ok, WorkNodes, _GateWayNodes} ->
                bench:dist(G, N, P, WorkerNodes);
            {error, Reason} ->
                {error, Reason}
        end;
    false -> %% run distributed Orbit.
        bench:dist(G, N, P, Nodes)
end.
```

Figure 5: Code Change 1: from S-Orbit to SD-Orbit

to automate using Wrangler's rule-based transformation API [14]. For instance, in one case we refactored the `send` expression:

```
Pid ! {vertex, X, Slot, K}
```

to:

```
case ?SDOrbit of
  false ->
    Pid ! {vertex, X, Slot, K};
  true ->
    central_grouping:s_group_send(Pid, {vertex, X, Slot, K})
end
```

The rule for this kind of transformation can be written as:

```
?RULE(?T("Pid@!Expr@"),
      ?ToAST("case ?SDOrbit of
                false ->
                  _This@;
                true ->
                  central_grouping:s_group_send(Pid@, Expr@)
              end"),
       true).
```

Recall that the transformation rules in the API have the form

```
?RULE(<OLD>,<NEW>,<COND>)
```

where `<OLD>` is to match the code to be replaced, `<NEW>` is the replacement code, and `<COND>` is the pre-condition on the rule (trivially `true` here). The `<OLD>` pattern its typically a template which is a fragment in Erlang concrete syntax, with meta-variables (ending in ‘’); the meta- variables can be used in the generated result, and in the condition.

So, in this particular case, instead of manually replacing every use of `spawn/send`, we only need to write a simple refactoring with a number of transformation rules, then apply this refactoring to the whole application.

In summary, the use of pre-defined s_group patterns has the following advantages:

- it separates the s_group manipulation from the application logic;
- it reduces the amount of changes needed to the application code, and makes it possible to automate;
- it takes the burden of understanding how s_group works away from users; and finally
- it allows a more fair performance comparison between an s_group implementation and a non s_group implementation.

3.4 Extensions to Wrangler

Based on the concurrency/distribution-related ‘bad smells’ that Percept2 is able to report, we identified a number of refactorings for eliminating those ‘smells’. The kinds of refactorings that we have identified include:

- Refactorings for introducing parallelisation. This kind of refactorings aim to improve the utilisation of available computing resources, and/or to alleviate processes with heavy load.
- Refactorings for reducing parallelisation. These refactorings aim to reduce the number of runnable processes by controlling either the number of processes being spawned or the runnability of processes spawned.
- Refactorings for introducing data sharing, therefore to avoid very large message passing between processes.
- Refactorings for introducing s_groups in order to reduce the size of node clusters.
- Refactorings for turning the use of ETS tables to Mnesia database. The rationale behind this kind of refactorings is to allow transactional control over the access of data shared by multiple processes.

So far, our effort has been on refactorings for introducing parallelism and for introducing s_groups. Our approach to introducing s_groups has been reported in Section 3.3, so in this section we focus on Wrangler’s support for introducing parallelism. Apart from these refactorings, we also report a new component added to Wrangler, that performs *program slicing* at function level. This slicing

component is used to implement some of the refactorings, but it can also be used on its own by programmers for code inspection purpose.

Work in the ParaPhrase project has also examined refactorings in this area, where they have concentrated thus far on identifying 'skeletons' in the code, and parallelising these. We see this as relatively straightforward once the code is in the right format, but here we also look at the – more profound – steps required to put to code into recognisable form before skeletons can be identified. In doing this we use the results of tracing with Percept2 to identify suitable loci for refactoring.

3.4.1 Support for Introducing Parallelism

Parallel map and foreach List processing is one of the most obvious cases where parallelism could be introduced. In particular, a number of sequential list processing operations provided by Erlang's `lists` library, such as `lists:map/2`, `lists:foreach/2`, etc, are perfect candidates for a parallelised implementation. So far this kind of parallel list processing operations are not supported by the `lists` library yet.

There are a number of things one needs to consider when implementing a parallel list processing operation, such as the size of each work unit, the number of parallel processes, the handling of process failure, etc. So instead of letting users write their own parallel list operations, a general purpose library is much more preferred.

In order to assist the use of parallel list processing operations, we have added to Wrangler a small library, called `para_lib`, which provides parallel implementation of *map* and *foreach*. There are also some other open source parallel lists processing libraries available, such as http://code.google.com/p/plists/source/browse/trunk/src/plists.erl, so users have the choice of which library to use.

The transformation from an explicit use of sequential `map/foreach` to the use of their parallel counterparts is very straightforward, even manual refactoring would not be a problem. However a `map/foreach` operation could also be implemented differently using recursive functions, list comprehensions, etc., as shown by the examples in Figure 6. Identifying this kind of implicit `map/foreach` usages can be done using Wrangler's API for code inspection, and a refactoring that turns an implicit `map/foreach` to an explicit `map/foreach` can also be specified using Wrangler's rule-based transformation API. As a matter of fact, Wrangler has been used by the **Paraphrase** project (paraphrase-ict.eu) to do this kind of transformations.

```
mdd([], _, _, _ST, Acc) ->
    lists:reverse(Acc);
mdd([J|Rest], Durations, Deadlines, Scheduled_Time, Acc) ->
    This_Deadline = element(J, Deadlines),
    This_Duration = element(J, Durations),
    Finish_Time = Scheduled_Time + This_Duration,
    MDD = max(Finish_Time, This_Deadline),
    mdd(Rest, Durations, Deadlines, Scheduled_Time, [{J,1.0/MDD}|Acc]).
```

(a) Example 1: 'map' using recursive function

```
[generalise_and_hash_file_ast(File, Threshold, ASTPid, true, SearchPaths, TabWidth)
       ||File <-Files]
```

(b) Example 2: 'map' using list comprehension

Figure 6: Implicit map operations

Introducing a process to compute a task. If the computation of two non-trivial tasks do not depend on each other, then they can be executed in parallel. The *Introduce a New Process*

refactoring implemented in Wrangler can be used to spawn a new process to execute a task in parallel with its parent process. The computation result of the new process is sent back to the parent process, which will then consume it when needed. As an example, Figure 8 shows the refactoring result of introducing a new process to do the first part of the computation of the code shown in Figure 7. In order not to block other computations that do not depend on the result returned by the new process, the `receive` expression is placed immediately before the point where the result is needed.

```
readImage({FileName, FileName2, Output}) ->
    {ok, _Img=#erl_image{format=F1, pixmaps=[PM]}} =
        erl_img:load(FileName),
    #erl_pixmap{pixels=Cols} = PM,
    R = lists:map(fun({_A,B}) -> B end, Cols),

    {ok, _Img2=#erl_image{format=F2, pixmaps=[PM2]}} = erl_img:load(FileName2),
    #erl_pixmap{pixels=Cols2} = PM2,
    R2 = lists:map(fun({_A2,B2}) -> B2 end, Cols2),

    {R, R2, F1, F2, Output}.
```

Figure 7: Refactoring: *Introduce a new process* (code before refactoring)

```
readImage({FileName, FileName2, Output}) ->
    Self = self(),
    Pid = spawn_link(fun () ->
                             {ok, _Img=#erl_image{format=F1, pixmaps=[PM]}} =
                                 erl_img:load(FileName),
                             #erl_pixmap{pixels=Cols} = PM,
                             R = lists:map(fun({_A,B}) -> B end, Cols),
                             Self ! {self(), {R, F1}}
                     end),
    {ok, _Img2=#erl_image{format=F2, pixmaps=[PM2]}} = erl_img:load(FileName2),
    #erl_pixmap{pixels=Cols2} = PM2,
    R2 = lists:map(fun({_A2,B2}) -> B2 end, Cols2),

    receive {Pid, {R, F1}} -> {R, F1} end,

    {R, R2, F1, F2, Output}.
```

Figure 8: Refactoring: *Introduce a New Process* (code after a refactoring)

Introducing a worker processes to `handle_call` Among those processes with heavy load, there can be `gen_servers` stuck with request messages. It is a good practice to check the `handle_call` function and see if any clause implementation may be divided into two parts: one that must be executed on the main `gen_server` process because it affects the state and another that does not affect the server state and therefore may be executed in a worker process spawned for it. For instance, the `handle_call` clause (code borrowed from RabbitMQ) shown in Figure 9 can be refactored to the code shown in Figure 10 using Wrangler.

Parallelise a tail-recursive function While some tail-recursive list processing functions can be refactored to an explicit map operation, many cannot due to data dependencies. For instance, the example shown in Figure 11 does a recursion over the list `Nodes` while accumulating results to the accumulator variable `Acc`. Each recursive call processes a number of elements in `Nodes`,

```
handle_call(which_children, _From, State) ->
     Resp = lists:map(fun(#child{pid = ?restarting(_), name = Name,
                                 child_type = ChildType, modules = Mods}) ->
                              {Name, restarting, ChildType, Mods};
                         (#child{pid = Pid, name = Name,
                                  child_type = ChildType, modules = Mods}) ->
                              {Name, Pid, ChildType, Mods}
                      end,
                      State#state.children),
     {reply, Resp, State};
```

Figure 9: Refactoring: Introduce a Worker Process to `handle_call` (code before refactoring).

```
handle_call(which_children, _From, State) ->
    proc_lib:spawn_link(
      fun () ->
          Res = try
                    Resp =
                        lists:map(fun(#child{pid = ?restarting(_), name = Name,
                                             child_type = ChildType, modules = Mods}) ->
                                          {Name, restarting, ChildType, Mods};
                                     (#child{pid = Pid, name = Name,
                                             child_type = ChildType, modules = Mods}) ->
                                          {Name, Pid, ChildType, Mods}
                                  end,
                                  State#state.children)
                catch throw:Error -> {throw, Error}
                end,
          gen_server:reply(From, Res)
      end),
    {no_reply, State};
```

Figure 10: Refactoring: Introduce a Worker Process to `handle_call` (code after refactoring).

the values of `NumGroup` and `Counter` have a dependency on their values in the previous recursion. Suppose the computation of `NewGroup` is rather expensive, and there is a need for performance improvement, then simply spawning a new process to do the computation, as shown in the example in Figure 7 and Figure 8, would not help in this case. In order to handle this kind of situation, we examined a subset of direct tail-recursive functions that meet certain constraints, and this led to a new prototype refactoring implemented in Wrangler.

This refactoring takes a function definition as input, and carries out a sequence of analysis to:

- check that the function is a direct tail-recursive function, and the recursive call appears as the last expression of the function clause body. The current implementation of this refactoring requires that there is only one recursive function clause; this restriction will be lifted in the future.
- decide which parameter is an accumulator parameter if there is one. We say that a parameter is an accumulator if its value depends on the value of other parameters, but not the other way around.
- identify the parts of computation that influence the value of accumulators only, and to
- identify the parts of computation that influence the value of non-accumulator parameters.

Take the function shown in Figure 11 as an example, Wrangler's static analysis would decide that this function is a direct tail-recursive function in which its fourth parameter, `Acc`, is an accumulator,

whereas all the other parameters are recursion control parameters. Using program slicing techniques, it then partitions the second function clause body into three sections: lines 5 and 6, which are needed for the computation of those recursion control parameters, line 7, which influences, but does not depend on the accumulator, and line 8, which influences but also depends on the accumulator.

```
1. do_grouping(Nodes, _Size, 1, Counter, Acc) ->
2.     {ok, add_new_group(make_group(Nodes, Counter), Acc)};
3.
4. do_grouping(Nodes, Size, NumGroup, Counter, Acc) ->
5.     Group = lists:sublist(Nodes, Size),
6.     Remain = lists:subtract(Nodes, Group),
7.     NewGroup = make_group(Group, Counter),
8.     NewAcc = add_new_group(NewGroup, Acc),
9.     do_grouping(Remain, Size, NumGroup-1, Counter+1, NewAcc).
```

Figure 11: An example tail-recursive list processing function

The above analysis result is then used to construct a parallel implementation of the function. Figure 12 and Figure 13 show such an implementation generated by Wrangler's refactoring support. With this new implementation, the `do_grouping/5` function is defined to spawn a number of worker processes according to the number of schedulers available, and another process which is in charge of dispatching tasks to worker processes and collecting results from them. This task dispatching process uses indices to track each job dispatched, and also ensures the correct order of the results received from worker processes. In this implementation, a worker process with the shortest message queue is selected, however this could be changed by the user.

3.4.2 Support for Program Slicing

Program slicing is a general technique of program analysis for extracting the part of a program, also called the *slice*, that influences or is influenced by a given point of interest, i.e. the *slicing criterion*. Static program slicing is generally based on program dependency including both control dependency and data dependency.

Backward intra-function slicing is used by some of the refactorings described here to extract the program slice that influences a particular variable/expression. Slicing is a useful program analysis tool on its own, and can be used by programmers in their daily practice, so we have exposed the backward slicing functionality to end-users under the `Wrangler -> Inspector` menu.

More complex code inspections can be built using Wranger's code inspection API and slicing. For example, it is a good practice to reduce the amount of computation done by a server process as much as possible to avoid it being overloaded, so if part of the computation that handles a received message only depends on the message received, it might be a good idea to shift that part of the computation to the client process. This kind of code fragments can be found using backward slicing by checking if a slice depends on any server data.

3.5 Further Extension to Erlang's Built-in Tracing

One of the factors that limit the scalability of tracing/profiling tools is the vast amount of data generated/collected, hence reducing the data generated/collected could potentially improve the scalability of tracing/profiling tools. For this purpose, we have carried out a number of experimental extensions to the Erlang built-in trace, and some of the earlier extensions were reported in D5.1. Our latest extension is to support the tracing of inter-node only message sending.

Message 'send' events can be traced in Erlang, hence information about message passing between nodes can be obtained by tracing process-level message passing. However, the current Erlang implementation does not allow users to specify a condition so that only message sending between processes from different nodes are traced. This might result in a huge amount of trace data being

```
do_grouping(Nodes, Size, NumGroup,Counter, Acc) ->
    Parent = self(),
    %% spawn a number of worker processes.
    Workers = [spawn(fun() ->
                             do_grouping_worker_loop(Parent)
                     end)
               || _ <- lists:seq(1, erlang:system_info(schedulers))],
    %% spawn the process for dispatching tasks to worker
    %% processes and collecting result from them.
    Pid = spawn_link(
            fun() ->
                do_grouping_dispatch_and_collect_loop(Parent, Acc, Workers, 0, 0)
            end),
    %% send the computation request.
    Pid ! {Nodes, Size, NumGroup, Counter},
    %% collect final result, and stop worker processes.
    receive
        {Pid, Acc} ->
            [P ! stop || P <- Workers],
            Acc
    end.

do_grouping_dispatch_and_collect_loop(Parent, Acc, Workers, RecvIndex, CurIndex) ->
   receive
      %% base case, and all the results have been received.
      {Nodes,_Size, 1, Counter} when RecvIndex == CurIndex ->
            Parent ! {self(), {ok, add_new_group(make_group(Nodes, Counter), Acc)}};
      %% base case, but still waiting for most results to come.
      {Nodes, Size, 1, Counter} when RecvIndex < CurIndex ->
           self() ! {Nodes, Size, 1, Counter},
           do_grouping_dispatch_and_collect_loop(
             Parent, Acc, Workers, RecvIndex, CurIndex);
      {Nodes, Size, NumGroup, Counter} ->
           Group = lists:sublist(Nodes, Size),
           Remain = lists:subtract(Nodes, Group),
           %% select a worker process.
           Pid = oneof(Workers),
           %% send a job to the worker process.
           Pid ! {self(), Group, Size, Counter},
           %% send to itself the remaining job
           self() ! {Remain, Size, NumGroup-1, Counter+1},
           do_grouping_dispatch_and_collect_loop(
              Parent, Acc, Workers, RecvIndex, CurIndex+1);
       {{worker, _Pid}, RecvIndex, NewGroup} ->
           %% receive result from a worker process.
           NewAcc = add_new_group(NewGroup, Acc),
           do_grouping_dispatch_and_collect_loop(
              Parent, NewAcc, Workers, RecvIndex+1, CurIndex)
   end.
```

Figure 12: A parallel implementation of the `do_grouping` function (part 1)

generated, only to be thrown away later. For this reason, we have added one more flag to Erlang's built-in trace, called '`remote_send`'. When this flag is enabled, a message sending event is only traced if the sender and receiver processes are from different nodes. For example, the command

```
erlang:trace(all, true, [remote_send])
```

tells the Erlang VM to enable the tracing of inter-node message sends for all (both existing and

```
do_grouping_worker_loop(Parent) ->
     receive
         {Group, Size, Counter, Index} ->
             NewGroup = make_group(Group, Counter),
             Parent ! {Index, NewGroup},
             do_grouping_worker_loop(Parent);
         stop ->
             ok
     end.

oneof(Workers) ->
    ProcInfo = [{Pid, process_info(Pid, message_queue_len)} || Pid <- Workers],
    [{Pid, _}|_] = lists:keysort(2, ProcInfo),
    Pid.
```

Figure 13: A parallel implementation of the `do_grouping` function (part 2)

new) processes.

4 Systematic Testing Using Concuerror

Verification and testing of concurrent programs is difficult in any language, since one must consider all the different ways in which processes/threads can interact. Erlang programs are no different, even though processes in Erlang have only very specific and well defined ways to interact. Since the scheduler of the Erlang Virtual Machine is beyond the control of the programmer, unit testing and random testing are not guaranteed to detect all the errors that can occur due to specific schedulings of the processes. Concuerror is a tool designed to systematically test an Erlang program using a set of user-supplied tests and detect all concurrency errors that are exposed by these tests or verify their absence.

4.1 Concurrency Errors in Erlang

Erlang's concurrency model Erlang is a programming language based on the actor model of concurrency. In Erlang, actors are realized by language-level processes that, by default, share no memory and communicate with each other via asynchronous message passing. Erlang processes are very lightweight as they are implemented by the runtime system of the language rather than by OS threads, and typical applications often create several thousands of them. Processes get created using the `spawn` family of functions. A `spawn` call creates a new process P having its own private memory area (stack, heap and mailbox) and returns a *process identifier* (PID) for it. Optionally, P can be `link`ed to another process, typically its parent, or `register`ed under a specific name in a global table, so that other processes can refer to P using its name instead of its PID when sending messages to it. Messages are sent *asynchronously* using the `!/2` expression, which takes two arguments and is a convenient shorthand for the `send/2` function. A process can then consume messages using selective pattern matching in `receive` expressions[1], which are *blocking* operations in case a process mailbox does not contain a matching message. Of course, blocking the execution of a process until a specific kind of message from another process arrives can lead to processes which are stuck, and often to deadlocks.

Stuck processes and deadlocks, however, are not the only kinds of concurrency errors that are possible in Erlang. Although the majority of memory that programs access is process-local, the language comes with various built-in functions (BIFs), implemented in C, that manipulate

[1]The general form of receive expressions is `receive...after`. What comes after the `after` keyword is a *timeout* value: either an integer or the special value `infinity`, in which case the behavior is that of a `receive` expression without an `after` clause.

```
1  -module(ping_pong).
2  -export([pong/0]).
3
4  pong() ->
5    Self = self(),
6    register(ping_pong, spawn(fun () -> ping(Self) end)),
7    receive ping -> ok end.
8
9  ping(PongPID) ->
10   PongPID ! ping.
```

Figure 14: Simple example program involving two processes and a concurrency error.

data structures at the level of the Virtual Machine (VM) which are shared between all processes. Interleaving sequences of calls to these BIFs can lead to data races or result in abnormal process exits. The latter may in turn result in abnormal termination of other processes. Testing for absence of concurrency errors due to unfortunate process interleaving is complicated by the fact that many errors are hard to come across and expose by conventional unit testing. Part of the difficulty lies in that the scheduling of processes is done by the Erlang VM and is mostly deterministic. It is currently based on the notion of *reduction steps*: roughly, each process gets to execute for a certain number of "reductions" (currently 2,000 function calls) before it has to yield back to its scheduler which then picks another process to execute. (A process also yields if it gets blocked on a `receive`.) Consequently, multiple runs of the same unit test are most likely to exhibit the same behavior with respect to process interleaving as such tests are too small for scheduling non-determinism to take effect.

Example A simple Erlang program involving two processes is shown in Figure 14. The `pong/0` function, which is exported and may be called from outside the `ping_pong` module, spawns a process that will execute the code of function `ping/1` (line 6). The spawned process, which is registered under the same name as the module (line 6), sends a `ping` message to the parent process (line 10), which, in turn, is expected to receive this message and return `ok` (line 7). This code has a concurrency error. Its execution will raise a runtime exception if the spawned process terminates before the parent process attempts to register its PID, which would not exist after the process terminates. As a result of this exception, the process executing function `pong/0` will crash and exit abnormally. This error is so subtle that many Erlang programmers are not even aware of its possibility. Still, errors such as this compromise the robustness of applications.

The core of the problem in this example is that the sequence of calls that `spawn`s the new process and `register`s its PID needs to run atomically. In the current implementation of Erlang/OTP, the probability of the parent process being scheduled out between these two consecutive calls is small[2]. However, even if the calls were further apart, which would increase the likelihood that the process running the code is scheduled out somewhere in between, the error cannot easily be provoked with unit testing because the scheduler of the Erlang VM is deterministic. To expose it, one would have to abandon unit for system testing or employ a randomized scheduler like PULSE [2]. In any case, both styles of testing would have to rely on luck to provoke and reproduce the error. In contrast, Concuerror, the systematic testing tool we developed to detect this kind of concurrency errors, is able to find it immediately. Before we describe Concuerror, let us briefly introduce the area of systematic testing and review its technology.

[2]To be precise the probability that reductions are exhausted at the `spawn` call is $1/2,000$.

4.2 Systematic Testing

Model checking addresses the problem of scheduling non-determinism by systematically exploring all the states that can be reached when executing a given program and verifying that each state satisfies a given property. To achieve this, the program and the property need to be formulated in some precise mathematical language. However, applying traditional model checking to realistic programs is difficult, since it requires to capture and store a large number of global states of the program. *Systematic testing* (also known as *stateless model checking* [8]) avoids this problem by exploring the state space of the program without explicitly storing global states. A special run-time scheduler drives the program execution, making decisions on scheduling whenever such decisions may affect the interaction between processes. Stateless model checking has been successfully implemented in tools, such as VeriSoft [9] and CHESS [18]. While stateless model checking is applicable to realistic programs, it still suffers from combinatorial explosion, as the number of possible interleavings grows exponentially with the length of program execution. There are several approaches that limit the number of explored interleavings, such as depth-bounding and context bounding [17]. Among them, *partial order reduction* (POR) [3, 7, 19, 23] stands out, as it provides full coverage of all behaviors that can occur in *any* interleaving, even though it explores only a representative subset. Partial order reduction is based on the observation that two interleavings can be regarded as equivalent if one can be obtained from the other by swapping adjacent, non-conflicting (independent) execution steps. In each such equivalence class (called a *Mazurkiewicz trace* [16]), partial order reduction explores at least one interleaving. This is sufficient for checking most interesting safety properties, including race freedom, absence of global deadlocks, and absence of assertion violations [3, 7, 23].

Concuerror employs both a *dynamic partial order reduction (DPOR)* algorithm and a bounding technique to increase its efficiency. Both are described in Section 4.5.

4.3 Concuerror in a Nutshell

To detect concurrency-related runtime errors such as the one described in Figure 14, Concuerror, given a program and its test suite, systematically explores process interleaving and presents detailed interleaving information on any errors that occur during the execution of the tests. In addition to abnormal process exits, Concuerror detects assertion violations and stuck processes.

Method overview To detect these kinds of errors, Concuerror effectively explores all interleaving sequences of the processes that participate in a test execution using a *stateless search strategy*, i.e. a search strategy that does not capture the shared state of the program. Specifically, recording an interleaving sequence involves storing information only about context switches, while enforcing the execution of all such sequences consists in efficiently controlling when the participating processes yield or resume execution.

The delegation of control over process execution from the Erlang scheduler to Concuerror is achieved through source-to-source instrumentation of the program under test, followed by its execution in a totally unmodified Erlang/OTP runtime system. (The alternative would have been to modify the runtime system, but that approach would run the risk of slightly altering the runtime semantics, besides being more difficult to implement and maintain across different versions of Erlang/OTP.) More concretely, the program undergoes a transformation that inserts *preemption points* in the code, i.e. points where a context switch is allowed to occur, without altering its semantics. In practice, a context switch may occur at any function call during the execution of a process in the Erlang VM. However, to avoid generating redundant interleaving sequences that lead to the same shared state, instrumentation in Concuerror inserts preemption points only at process actions that interact with (i.e. inspect or update) this shared state, which are very few in Erlang. We call such actions *preemptive*. As a result, as long as the semantics of the program under test is not altered and preemption points are inserted at all preemptive actions, our approach is both sound and complete in terms of exploring all valid interleaving sequences of the program and detecting all the concurrency errors it targets.

```
1 -module(test).
2 -export([pong_test/0]).
3
4 pong_test() ->
5   ok = ping_pong:pong().
```

(a) Test module for the example in Figure 14

```
1: P: P.1 = erlang:spawn(...)
2: P.1: ping = P ! ping in ping_pong.erl line 10
3: Message (ping) from P.1 reaches P
4: P.1: exits normally
5: P: Exception badarg raised by:
     erlang:register(ping_pong, P.1)
   in ping_pong.erl line 6
6: P: exits abnormally ({badarg,...})
```

(b) Erroneous interleaving sequence as reported by Concuerror

Figure 15: Erroneous interleaving sequence found by running the test in Concuerror.

Using Concuerror Specifying the Erlang module that contains the test function is typically enough to initiate a test for finding concurrency errors. Concuerror uses the standard Erlang code path to find *on-the-fly* any modules that are reached by the test and automatically instruments and reloads their code. Note that Concuerror can run already existing test functions and requires no modifications to them or to the program under test. In addition, since its testing approach is systematic, programmers need not spawn huge numbers of processes in their tests to increase the chances of detecting any concurrency errors—the minimum number of processes that is necessary to cause the error will do.

As an example, let us assume that the user has compiled module `ping_pong` of Figure 14 and a module containing a test for this program shown in Figure 15a. This test simply checks whether the return value of `ping_pong:pong/0` is `ok`. Here, pattern matching is used to check an assertion, but the effect would be equivalent if the test used assertions, such as those provided by EUnit. The user then specifies that Concuerror can systematically explore all process interleavings of this test by calling the function `test:pong_test/0`:

```
concuerror -m test -t pong_test
```

As a first step, Concuerror's *instrumenter* automatically transforms the test module to insert preemption points into the code and also to calls to functions of other modules that have not yet been instrumented. In our example the call to `ping_pong:pong/0` will be instrumented and the test will begin. Control flow immediately passes to the `ping_pong` module, which is also instrumented and loaded, adding preemption points at all process interactions with the shared state, i.e. process creation (`spawn/1` on line 6), process registration (`register/2` on line 6) and message passing (`receive` on line 7 and `!/2` on line 10). Even though in this simple example preemption points are inserted at most process actions, in bigger programs the actions that interact with the shared state constitute only a small portion of the code, thus allowing our approach to handle large programs effectively.

Execution continues now, stopping at every preemption point and recording the effects of the relevant operations. After all processes have finished executing in this first run, more interleavings are explored, where they are possible. This is the responsibility of the tool's *scheduler*. As expected, an error will be reported in one of the interleavings, shown in Figure 15b.

Using this information, the user can iteratively apply code changes and replay the erroneous interleaving sequence to observe how program execution is affected. If no errors are reported, the program is indeed free from the kinds of concurrency errors detected by the tool. In this case, Concuerror functions not only as a testing, but also as a verification tool.

Concuerror's report Concuerror reports any errors it finds in a textual report like the one shown in Figure 16. This report contains the options that have been used, any messages that have been printed during the analysis (Tips, Info, Warnings, Errors, etc.) and a detailed log for every erroneous interleaving. This log contains highlighted information for every process that crashed (reason and stack trace) and every process that is stuck (in which `receive` statement it is stuck). By default, Concuerror will replace Erlang process identifiers (PIDs) with symbolic names: the first

```
################################################################################
Concuerror started with options:
  [{after_timeout,infinity},
   {assume_racing,true},
   {delay_bound,infinity},
   {depth_bound,5000},
   {entry_point,{test,pong_test,[]}},
   {files,["ping_pong.erl","test.erl"]},
   {ignore_error,[]},
   {ignore_first_crash,false},
   {instant_delivery,false},
   {non_racing_system,[]},
   {optimal,true},
   {print_depth,20},
   {scheduling,round_robin},
   {show_races,true},
   {strict_scheduling,false},
   {symbolic_names,true},
   {timeout,1000},
   {treat_as_normal,[]}]
################################################################################
Erroneous interleaving 1:
* At step 6 process P exited abnormally
    Reason:
      {badarg,[{erlang,register,[ping_pong,P.1],[6,{file,"ping_pong.erl"}]},
               {ping_pong,pong,0,[{file,"ping_pong.erl"},{line,6}]},
               {test,pong_test,0,[{file,"test.erl"},{line,5}]}]}
    Stacktrace:
      [{erlang,register,[ping_pong,P.1],[6,{file,"ping_pong.erl"}]},
       {ping_pong,pong,0,[{file,"ping_pong.erl"},{line,6}]},
       {test,pong_test,0,[{file,"test.erl"},{line,5}]}]
--------------------------------------------------------------------------------
Interleaving info:
   1: P: P.1 = erlang:spawn(...)
   2: P.1: ping = P ! ping in ping_pong.erl line 10
   3: Message (ping) from P.1 reaches P
   4: P.1: exits normally
   5: P: Exception badarg raised by: erlang:register(ping_pong, P.1)
    in ping_pong.erl line 6
   6: P: exits abnormally ({badarg,...})
################################################################################
Info:
--------------------------------------------------------------------------------
Instrumented ping_pong
Instrumented test
Instrumented io_lib
Instrumented erlang
################################################################################
Race Pairs:
--------------------------------------------------------------------------------
You can disable race pair messages with '--show_races false'
* A) P: true = erlang:register(ping_pong, P.1)
    in ping_pong.erl line 6
  vs B) P.1: exits normally
################################################################################
Done! (Exit status: completed)
  Summary: 1 errors, 2/2 interleavings explored
```

Figure 16: Sample report generated by Concuerror.

process has the symbolic name P and every subsequent process is named after its parent, with a suffix denoting the order in which it was spawned: e.g. P's first child is $P.1$ and $P.3$'s second child is $P.3.2$. Raw PIDs can be shown if the option `--symbolic false` is used.

As shown, the generated report also contains information about every pair of events that is racing and necessitates the exploration of additional interleavings.

Concuerror can also produce a graph of the explored interleavings, with the `--graph` option. Such a graph can be seen in Figure 17. The graph contains info about successful and failing interleavings, normal and abnormal process exits and racing instructions.

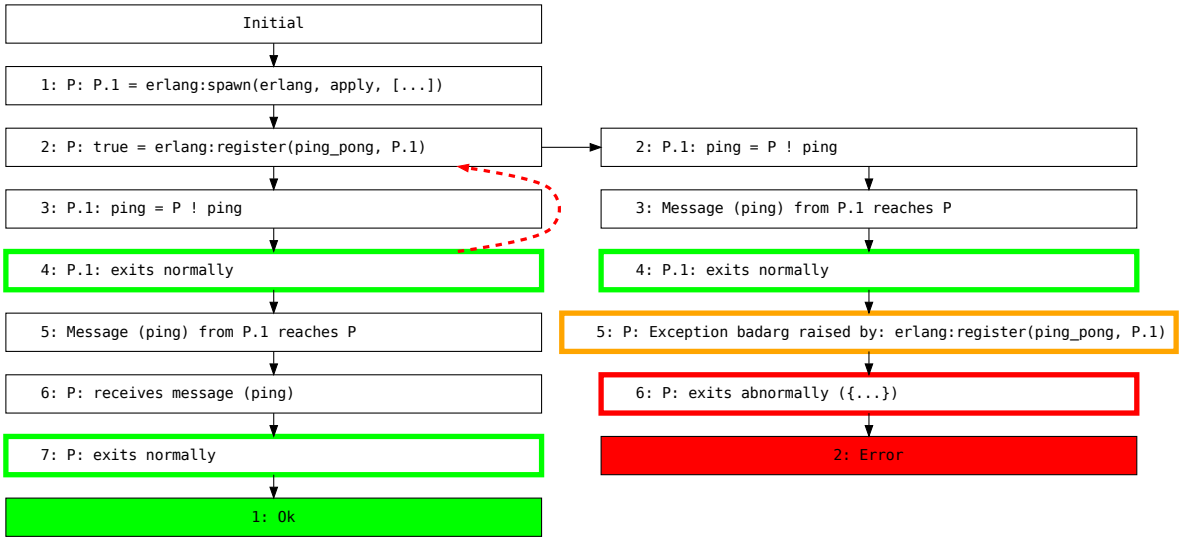

Figure 17: Graphical representation of the interleavings of the example as produced by Concuerror.

4.4 Implementation Technology

We now briefly describe the implementation of the main components of Concuerror: in particular, how programs are instrumented, Concuerror's scheduler, and the options that the user can employ to handle some aspects of Erlang programs that are tricky to handle without help from the programmer.

Program instrumentation In order to explore interleavings in an efficient way, Concuerror is designed to consider the actual dependencies between Erlang built-in operations. There is no reason to consider two schedulings that differ only in the order of execution of instructions that are not in a race; more on that in Section 4.5. To facilitate reasoning about possible races, Concuerror needs to be aware of such dependencies. The built-ins that are currently included are summed up in Figure 18, but this list is constantly extended and is planned to eventually include all operations that can depend on others, when executed by different processes. The list also excludes operations that are never racing.

The instrumentation is such that only processes that are involved in a test are affected. This is controlled by a special value that is added in their process dictionary and checked by instrumented code. Other processes in the system can therefore bypass the instrumentation.

The instrumentation also ensures that calls to function in other modules are intercepted and the respective modules are instrumented before a process executes their code.

Process scheduling Concuerror's scheduler serves multiple purposes. It keeps track of all the operations that have happened in a particular interleaving in order to generate a detailed report of the events that have lead to a potential error and also plan additional interleavings. It also ensures that only one process is running at any time. The instrumentation enables precisely these features: it sends a message to the scheduler every time an instrumented operation is detected and waits a message from the scheduler before continuing.

Options that control the exploration There are a number of options that can be used to focus testing on parts of a program that are more interesting than others.

`--depth_bound` : Sets a limit on the length of an interleaving. This limit is useful e.g. in order to avoid infinite runs when a program can run forever in a particular scheduling. It also catches explorations that are too long and may lead to impractically big number of interleavings.

```
                                erlang:process_info/2
erlang:!/2                      erlang:put/2                ets:first/1
erlang:cancel_timer/1           erlang:read_timer/1         ets:give_away/3
erlang:date/0                   erlang:register/2           ets:info/1,2
erlang:demonitor/1,2            erlang:send/2,3             ets:insert/2
erlang:exit/2                   erlang:send_after/3         ets:insert_new/2
erlang:get_stacktrace/0         erlang:spawn/3              ets:lookup/2
erlang:group_leader/0,2         erlang:spawn_link/3         ets:lookup_element/3
erlang:halt/0,1                 erlang:spawn_opt/1          ets:match/2
erlang:is_process_alive/1       erlang:start_timer/3        ets:match_object/2
erlang:link/1                   erlang:time/0               ets:member/2
erlang:make_ref/0               erlang:unlink/1             ets:new/2
erlang:monitor/2                erlang:unregister/1         ets:next/2
erlang:now/0                    erlang:whereis/1            ets:select/2,3
erlang:processes/0              ets:delete/1                ets:select_delete/2
erlang:process_flag/2           ets:delete/2                os:getenv/1
```

Figure 18: Explicitly supported built-ins

`--after_timeout` : A common pattern in Erlang is for a process to send a message as a request to another process and wait for a reply. Erlang libraries often set a timeout for the reply, so that the calling process does not get stuck. Concuerror has to assume that this timeout can be reached, as there are indeed interleavings where the reply might take too long to arrive. This behaviour may not be interesting for a particular test, so this option can be used to effectively treat timeouts higher than a specific value as infinite.

`--instant_delivery` : Erlang's semantics separate the event of sending a message from its delivery to the corresponding process. Concuerror is by default faithful to this behaviour. However, in the current implementation of Erlang/OTP, when both processes are running on the same node messages are delivered instantly. This option enables exactly this behaviour.

`--non_racing_system` : If multiple processes under Concuerror send messages to a system process, then the order in which these messages are delivered might not be important.

Even with these options however, the number of process interleavings that Concuerror has to explore is enormous. This number can be significantly reduced by partial order reduction methods, and dynamic partial order reduction techniques in particular.

4.5 DPOR Algorithms

Existing partial order reduction approaches are essentially based on two techniques, both of which reduce the set of process steps that are explored at each preemption point:

- The *persistent set* technique, that explores only a provably sufficient subset of the enabled processes. This set is called a *persistent set* [7] (variations are *stubborn sets* [23] and *ample sets* [3]).
- The *sleep set* technique [7], that maintains information about the past exploration in a so-called *sleep set*, which contains processes whose exploration would be provably redundant.

These two techniques are independent and complementary, and can be combined to obtain increased reduction.

The construction of persistent sets is based on information about possible future conflicts between threads. Early approaches analyzed such conflicts statically, leading to over-approximations

and therefore limiting the achievable reduction. *Dynamic Partial Order Reduction* [5] improves the precision by recording actually occurring conflicts during the exploration and using this information to construct persistent sets on-the-fly, "by need". DPOR guarantees the exploration of at least one interleaving in each Mazurkiewicz trace when the explored state space is acyclic and finite. This is the case in stateless model checking in which only executions of bounded length are analyzed [5, 9, 18].

Since DPOR is excellently suited as a reduction technique, several variants, improvements, and adaptations for different computation models have appeared [5, 13, 20–22]. The obtained reduction can, however, vary significantly depending on several factors, e.g. the order in which processes are explored at each point of scheduling. For a particular implementation of DPOR (with sleep sets) [11], up to an order of magnitude of difference in the number of explored interleavings has been observed, when different strategies are used. For specific communication models, specialized algorithms can achieve better reduction [22]. Heuristics for choosing which next process to explore have also been investigated without conclusive results [12].

The above variation in obtained reduction can be explained as follows: In DPOR, the combination of persistent set and sleep set techniques guarantees to explore at least one complete interleaving in each Mazurkiewicz trace. Moreover, it has already been proven that the use of sleep sets is sufficient to prevent the complete exploration of two different but equivalent interleavings [10]. At first sight, this seems to imply that sleep sets can give optimal reduction. What it actually implies, however, is that when the algorithm tries an interleaving which is equivalent to an already explored one, the exploration will begin but it will be blocked sooner or later by the sleep sets in what we call a *sleep-set blocked* exploration. When only sleep sets are used for reduction, the exploration effort will include an arbitrary number of sleep-set blocked explorations. It is here where persistent sets enter the picture, and limit the number of initiated explorations. Computation of smaller persistent sets, leads to fewer sleep-set blocked explorations.

In view of these variations, a fundamental challenge has been to develop an *optimal* DPOR algorithm that: (i) always explores the minimum number of interleavings, regardless of scheduling decisions, (ii) can be efficiently implemented and (iii) is applicable to a variety of computation models, including communication via shared variables and message passing.

Optimal DPOR We developed a new DPOR algorithm, called *optimal-DPOR* [1], which is provably optimal in that it always explores exactly one interleaving per Mazurkiewicz trace, and never initiates any sleep set-blocked exploration. Our optimal algorithm is based on a new theoretical foundation for partial order reduction, in which persistent sets are replaced by a novel class of sets, called *source sets*. Source sets are often smaller than persistent sets and are provably minimal, in the sense that the set of explored processes from some scheduling point must be a source set in order to guarantee exploration of all Mazurkiewicz traces. When a minimal persistent set contains more elements than the corresponding source set, the additional elements will always initiate sleep-set blocked explorations.

Concuerror includes optimal-DPOR by default, as well as a slightly weaker variant, *source-DPOR* which can be easily combined with existing bounding techniques. Whether the optimal or the weaker algorithm is used can be controlled with the `--optimal` option of Concuerror.

Performance on two "standard" benchmarks We compared the reduction achieved by both source- and optimal-DPOR on the two benchmarks from the DPOR paper [5]: filesystem and indexer. These are benchmarks that have been used to evaluate another DPOR variant (DPOR-CR [20]) and a technique based on unfoldings [11]. Both benchmarks are parametric on the number of threads they use. For filesystem we used 14, 16, 18 and 19 threads. For indexer we used 12 and 15 threads.

Table 1 shows the number of traces that the algorithms explore as well as the time it takes to explore them. It is clear that our algorithms, which in these benchmarks explore the same (optimal) number of interleavings, beat 'classic' DPOR with sleep sets, by a margin that becomes wider as the number of threads increases.

	Traces Explored			Time		
Benchmark	classic	source	optimal	classic	source	optimal
filesystem(14)	4	2	2	0.54s	0.36s	0.35s
filesystem(16)	64	8	8	8.13s	1.82s	1.78s
filesystem(18)	1024	32	32	2m11s	8.52s	8.86s
filesystem(19)	4096	64	64	8m33s	18.62s	19.57s
indexer(12)	78	8	8	0.74s	0.11s	0.10s
indexer(15)	341832	4096	4096	56m20s	50.24s	52.35s

Table 1: Performance of DPOR algorithms on two benchmarks.

	Traces Explored			Time		
Benchmark	classic	source	optimal	classic	source	optimal
readers(2)	5	4	4	0.02s	0.02s	0.02s
readers(8)	3281	256	256	13.98s	1.31s	1.29s
readers(13)	797162	8192	8192	86m 7s	1m26s	1m26s
lastzero(5)	241	79	64	1.08s	0.38s	0.32s
lastzero(10)	53198	7204	3328	4m47s	45.21s	27.61s
lastzero(15)	9378091	302587	147456	1539m11s	55m 4s	30m13s

Table 2: Performance of DPOR algorithms on more benchmarks.

Performance on two synthetic benchmarks Next we compare the algorithms on two synthetic benchmarks that expose differences between them. The results, for 2, 8 and 13 readers are shown in Table 2. For 'classic' DPOR the number of explored traces is $O(3^{\mathtt{N}})$ here, while source- and optimal-DPOR only explore $2^{\mathtt{N}}$ traces. Both numbers are exponential in `N` but, as can be seen in the table, for e.g. $\mathtt{N} = 13$ source- and optimal-DPOR finish in about one and a half minute, while the DPOR algorithm with the sleep set extension [6] explores two orders of magnitude more (mostly sleep-set blocked) traces and needs almost one and a half hours to complete.

The second benchmark is the lastzero(`N`) program whose pseudocode is shown in Figure 19. Its `N`+1 threads operate on an array of `N`+1 elements which are all initially zero. In this program, thread 0 searches the array for the zero element with the highest index, while the other `N` threads read one of the array elements and update the next one. The final state of the program is uniquely defined by the values of `i` and `array[1..N]`. What happens here is that thread 0 has control flow that depends on data that is exposed to races and represents a case when *source-DPOR* may encounter sleep-set blocking, that the *optimal-DPOR* algorithm avoids. As can be seen in Table 2, *source-DPOR* explores about twice as many traces than *optimal-DPOR* and, naturally, even if it uses a cheaper test, takes almost twice as much time to complete.

Performance on real programs Finally, we evaluate the algorithms on four Erlang applications. The programs are: (i) dialyzer: a parallel static code analyzer included in the Erlang distribution;

Variables: `int array[0..N] := {0,0,...,0}, i;`

Thread 0**:** `for (i := N; array[i] != 0; i--);`

Thread j ($j \in 1..\mathtt{N}$)**:** `array[`j`] := array[`j`-1] + 1;`

Figure 19: The pseudocode of the `lastzero(N)` benchmark.

	Traces Explored			Time		
Benchmark	classic	source	optimal	classic	source	optimal
dialyzer	12436	3600	3600	14m46s	5m17s	5m46s
gproc	14080	8328	8104	3m 3s	1m45s	1m57s
poolboy	6018	3120	2680	3m 2s	1m28s	1m20s
rushhour	793375	536118	528984	145m19s	101m55s	105m41s

Table 3: Performance of DPOR algorithms on four real programs.

	filesystem(19)	indexer(15)	gproc	rushhour
classic	92.98	245.32	557.31	24.01
source	66.07	165.23	480.96	24.01
optimal	76.17	174.60	481.07	31.07

Table 4: Memory consumption (in MB) for selected benchmarks.

(ii) gproc: an extended process dictionary (iii) poolboy: a worker pool factory[3]; and (iv) rushhour: a program that uses processes and ETS tables to solve the Rush Hour puzzle in parallel. The last program, rushhour, is complex but self-contained (917 lines of code). The first three programs, besides their code, call many modules from the Erlang libraries, which Concuerror also instruments. The total number of lines of instrumented code for testing the first three programs is 44596, 9446 and 79732, respectively.

Table 3 shows the results. Here, the performance differences are not as profound as in synthetic benchmarks. Still, some general conclusions can be drawn: (1) Both source- and optimal-DPOR explore less traces than 'classic' (from 50% up to 3.5 times fewer) and require less time to do so (from 42% up to 2.65 times faster). (2) Even in real programs, the number of sleep-set blocked explorations is significant. (3) Regarding the number of traces explored, *source-DPOR* is quite close to optimal, but manages to completely avoid sleep-set blocked executions in only one program (in dialyzer). (4) *Source-DPOR* is faster overall, but only slightly so compared to *optimal-DPOR* even though it uses a cheaper test. In fact, its maximal performance difference percentage-wise from *optimal-DPOR* is a bit less than 10% (in dialyzer again).

Although we do not include a full set of memory consumption measurements, we mention that all algorithms have very similar, and quite low, memory needs. Table 4 shows numbers for gproc, the real program which requires most memory, and for all benchmarks where the difference between source and optimal is more than one MB.

4.6 Bounding

Even with optimal-DPOR, the number of interleavings can be too high. Moreover, some of these interleavings might be really unlikely to happen in actual runs of a program, as the scheduler will e.g. not preempt a process that is able to continue executing too soon after enabling it. It is therefore beneficial, especially when trying to detect a specific bug, to focus the exploration on "simpler" interleavings.

Concuerror currently supports a technique called *delay bounding* which enforces a deterministic round-robin scheduling of processes, and only allows a limited number of "deviations" from it [4]. This technique has not yet been modified to be compatible with optimal-DPOR and enforces the use of the weaker source-DPOR algorithm instead. It is enabled with the `--delay_bounding` option.

[3] https://github.com/uwiger/gproc and https://github.com/devinus/poolboy

4.7 More Information about Concuerror

Concuerror's website Concuerror has a public website, hosted at http://concuerror.com which is directly linked to the public repository containing its code. We provide links to all the publications related to Concuerror, tutorials for its use, and other related material. Rather than duplicating that information here, we refer the interested reader to Concuerror's website instead.

General options Finally we describe Concuerror's general command line options.

`--module` The module containing the main test function.

`--test` The name of the 0-arity function that starts the test. [default: `test`]

`--output` File where Concuerror shall write the analysis results. [default: `concuerror_report.txt`]

`--help` Display Concuerror's usage information.

`--version` Display version information about Concuerror.

`--pa, --pz` Add directory at the front/end of Erlang's code path. These options enable Concuerror to locate modules that are not included in the default code path.

`--file` Explicitly load a file (.beam or .erl). (A .erl file should not require any command line compile options.)

`--verbosity` Sets the verbosity level. Concuerror prints diagnostic messages in stderr.

`--quiet` Do not write anything to standard error. Equivalent to `-v 0`.

`--print_depth` Specifies the maximum depth for any terms printed in the log (behaves just as the extra argument of `~W` and `~P` argument of `io:format/3`. If the user wants more info about a particular piece of data, a possibility is to use `erlang:display/1` and check the standard output section instead. [default: 20]

5 Concluding Remarks

In this deliverable we have described enhanced refactoring assistance to support concurrency and distribution related transformations in Wrangler and Percept2, as well as the Concuerror tool designed to find concurrency errors in Erlang programs using systematic testing.

In Section 3 we described the new extensions to Percept2 and Wrangler that aim to support the iterative process of profiling and refactoring. Unlike traditional structural refactorings, many performance-driven refactorings are application-dependent, hence hard to automate; nevertheless we tried to automate the application-independent refactorings, provide re-useable patterns, library functions and code analysis support to facilitate the refactoring process.

In the future, we will further enhance Wrangler to support other concurrency/distribution-related refactorings, such as turning the use of ETS tables to Mnesia database, introducing data sharing to avoid large message passing, etc. Wrangler has basic support for side-effect analysis, but it is rather coarse. We would also like to refine this analysis and provide finer information about side-effects.

Acknowledgments

We thank Maria Christakis and Alkis Gotovos for their work in the initial version of Concuerror and Ilias Tsitsimpis for his help with extending and testing that version of the tool. Although Concuerror's current design differs significantly from the initial version of the tool, the experiences learned have been invaluable and have heavily influnced the design of Concuerror's current architecture.

Change Log

Version	Date	Comments
0.1	30/6/2014	First Version Submitted to the Commission Services

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 373–384, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535845. URL http://doi.acm.org/10.1145/2535838.2535845.

[2] K. Claessen, M. Pałka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 149–160, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. URL http://doi.acm.org/10.1145/1596550.1596574.

[3] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[4] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 411–422, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926432. URL http://doi.acm.org/10.1145/1926385.1926432.

[5] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 110–121, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. URL http://doi.acm.org/10.1145/1040305.1040315.

[6] C. Flanagan and P. Godefroid. Addendum to *Dynamic partial-order reduction for model checking software*, 2005. Available at http://research.microsoft.com/en-us/um/people/pg/.

[7] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. PhD thesis, University of Liège, 1996. Also, volume 1032 of LNCS, Springer.

[8] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-853-3. URL http://doi.acm.org/10.1145/263699.263717.

[9] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005. URL http://dx.doi.org/10.1007/s10703-005-1489-x.

[10] P. Godefroid, G. J. Holzmann, and D. Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3):227–241, 1995. URL http://dx.doi.org/10.1007/BF01384077.

[11] K. Kähkönen, O. Saarikivi, and K. Heljanko. Using unfoldings in automated testing of multithreaded programs. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 150–159, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2. URL http://doi.acm.org/10.1145/2351676.2351698.

[12] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In D. S. Rosenblum and G. Taentzer, editors, *Fundamental Approaches to Software Engineering*, volume 6013 of *LNCS*, pages 308–322. Springer, 2010. ISBN 978-3-642-12028-2. doi: 10.1007/978-3-642-12029-9_22. URL http://dx.doi.org/10.1007/978-3-642-12029-9_22.

[13] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Software Eng.*, 32(6):382–403, 2006. URL http://doi.ieeecomputersociety.org/10.1109/TSE.2006.56.

[14] H. Li and S. Thompson. A User-extensible Refactoring Tool for Erlang Programs. Technical Report 4-11, School of Computing, Univ. of Kent, UK, 2011.

[15] H. Li and S. Thompson. Multicore Profiling for Erlang Programs Using Percept2. In *Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop*, pages 33–42, Boston, USA, September 2013.

[16] A. W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, volume 255 of *LNCS*, pages 279–324. Springer, 1986. ISBN 3-540-17906-2. URL http://dx.doi.org/10.1007/3-540-17906-2_30.

[17] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. URL http://doi.acm.org/10.1145/1250734.1250785.

[18] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280. USENIX Association, 2008. ISBN 978-1-931971-65-2. URL http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf.

[19] D. Peled. All from one, one for all, on model-checking using representatives. In *Computer Aided Verification*, volume 697 of *LNCS*, pages 409–423. Springer, 1993. ISBN 3-540-56922-7. URL http://dx.doi.org/10.1007/3-540-56922-7_34.

[20] O. Saarikivi, K. Kähkönen, and K. Heljanko. Improving dynamic partial order reductions for concolic testing. In *Proceedings of the 12th International Conference on Application of Concurrency to System Design*, pages 132–141. IEEE, 2012. URL http://doi.ieeecomputersociety.org/10.1109/ACSD.2012.18.

[21] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multithreaded programs. In *Hardware and Software, Verification and Testing, Second International Haifa Verification Conference, Revised Selected Papers*, volume 4383 of *LNCS*, pages 166–182. Springer, 2006. ISBN 978-3-540-70888-9. URL http://dx.doi.org/10.1007/978-3-540-70889-6_13.

[22] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference*, volume 7273 of *LNCS*, pages 219–234. Springer, 2012. ISBN 978-3-642-30792-8. URL http://dx.doi.org/10.1007/978-3-642-30793-5.

[23] A. Valmari. Stubborn sets for reduced state space generation. In *Applications and Theory of Petri Nets*, volume 483 of *LNCS*, pages 491–515. Springer, 1989. ISBN 3-540-53863-1. URL http://dx.doi.org/10.1007/3-540-53863-1_36.