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

Systematic Testing of Concurrent Erlang Programs: Some Experiences

Διπλωματική Εργασία

του

Ηλία Τσιτσιμπή

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

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Δεκέμβριος 2013

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

Systematic Testing of Concurrent Erlang Programs: Some Experiences

Διπλωματική Εργασία

του

Ηλία Τσιτσιμπή

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Δεκεμβρίου, 2013.

.....	.....	.....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Άρης Κοζύρης
Αν. Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Δεκέμβριος 2013

.....

Ηλίας Τσιτσιμπής
Διπλωματούχος Ηλεκτρολόγος Μηχανικός
και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στις μέρες μας παρατηρείται μια αυξανόμενη τάση χρησιμοποίησης παράλληλων τεχνικών στον προγραμματισμό. Αυτό εξηγείται τόσο από την αύξηση των επεξεργαστών που περιλαμβάνονται πλέον σε κάθε προσωπικό υπολογιστή, όσο και από την δημιουργία και ανάπτυξη υπηρεσιών Cloud. Ωστόσο ο παράλληλος προγραμματισμός εισάγει μια σειρά από σφάλματα τα οποία δεν υπάρχουν στο σειριακό προγραμματισμό και τα οποία τα συνηθισμένα εργαλεία αποσφαλμάτωσης δεν μπορούν να εντοπίσουν.

Σε αυτή τη διπλωματική μελετάμε τον Concuerror, ένα εργαλείο ελέγχου προγραμμάτων γραμμένων σε Erlang το οποίο μπορεί να χρησιμοποιηθεί για τον εντοπισμό λαθών που εισάγονται από τον ταυτοχρονισμό. Επίσης εξετάζουμε κατά πόσο ένα τέτοιο εργαλείο μπορεί να χρησιμοποιηθεί από μεγάλα projects (χιλιάδες γραμμές κώδικα), τα οποία μπορεί να χρησιμοποιούν αρκετές από τις βιβλιοθήκες της υλοποίησης της Erlang και να υλοποιούν περίπλοκα πρωτόκολλα επικοινωνίας.

Λέξεις Κλειδιά

Erlang, concurrency, software testing, model checking, test-driven development

Abstract

Concurrent programming has become increasingly widely used in the last decade. This can be explained by the increasing number of multiprocessor personal computers and the new trend of Cloud computing. Nevertheless, concurrent programming introduces a number of new errors not seen in sequential programming and which traditional testing tools largely cannot easily detect.

In this thesis we study Concuerror, a testing tool for concurrent Erlang programs, that aims to facilitate the task of detecting and eliminating concurrency-related errors. We also examine how Concuerror can be used in practice to test projects with thousands of lines of code, which may use many system libraries and implement complex communication protocols.

Keywords

Erlang, concurrency, software testing, model checking, test-driven development

Ευχαριστίες

Θα ήθελα να πω ένα μεγάλο ευχαριστώ στον Κωστή Σαγώνα για την διαρκή υποστήριξη και πολύτιμη καθοδήγηση, η οποία συνέβαλε καθοριστικά στη διαμόρφωση αυτής της διπλωματικής εργασίας, καθώς και για την εμπιστοσύνη και σεβασμό τον οποίο μου έδειξε. Επίσης, θα ήθελα να ευχαριστήσω τον Νίκο Παπασπύρου για την πολύ σημαντική βοήθεια την οποία μου προσέφερε.

Χρωστάω ένα μεγάλο ευχαριστώ στους γονείς μου για την υποστήριξη που μου έχουν προσφέρει και την εμπιστοσύνη που έδειξαν σε κάθε επιλογή μου.

Τέλος, θέλω να πω ένα μεγάλο ευχαριστώ σε όλους τους φίλους μου που μου στάθηκαν τα τελευταία χρόνια.

Ηλίας Τσιτσιμπής

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Testing as part of Software Development

Back in the old days, most of the software testing processes used to occur after the requirements had been defined and the coding process had been completed. Unfortunately, this process leads to defects being discovered late in the development cycle and makes them more expensive to fix. This is sometimes called the Defect Cost Increase (DCI) principle in software development [12, p. 98].

As software engineers began to realize the difficulties in software development and the failure of the existing inflexible development models, new ones began to emerge. Most of them, such as Agile [49], often employ test-driven development and place an increased portion of the testing in the hands of the developer, before it reaches a team of testers.

In test-driven development, each new feature begins with writing a test. This test must inevitably fail because it is written before the feature has been implemented. To write a test, the developer must clearly understand the feature's specification and requirements. This is a differentiating feature of test-driven development versus writing unit tests *after* the code is written: it makes the developer focus on the requirements *before* writing the code, a subtle but important difference [11].

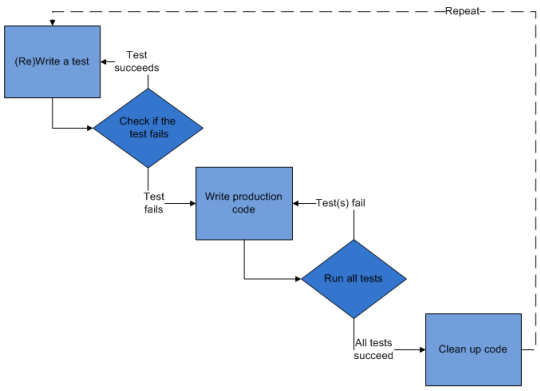

Figure 1.1: Graphical representation of the test-driven development cycle

Groups that use test-driven development tend to rely more and more on automated testing. There are many frameworks to write tests in, and continuous integration software will run tests automatically every time code is checked into a version control system.

While automation cannot reproduce everything a human can do (and all the ways they think of doing it), it can be very useful for regression testing. However, it does require a well-developed test suite in order to be truly useful.

1.2 Testing concurrent programs

Testing concurrent programs is harder than testing sequential ones. This is trivially true: tests for concurrent programs are themselves concurrent programs. But it is also true for another reason: the failure modes for concurrent programs are less predictable and repeatable than for sequential programs. Failures in sequential programs are deterministic; if a sequential program fails with a given set of inputs and initial state, it will fail every time. Failures in concurrent programs, on the other hand, tend to be rare probabilistic events.

Because of this, reproducing failures in concurrent programs can be maddeningly difficult. Not only might the failure be rare, and therefore not manifest itself frequently, but it might not occur at all in certain configurations, so that a bug that happens daily at production might never happen at all in test lab. Further, attempts to debug or monitor the program can introduce timing or synchronization artifacts that prevent the bug from appearing at all. As in Heisenberg's uncertainty principle, observing the state of the system may in fact change it.

So, given all these, how are we supposed to ensure that concurrent programs work properly? A number of techniques have been proposed that aim to alleviate the seemingly grave situation [47].

- Using stress testing [45] one can test the application for robustness, availability and graceful error handling simulating conditions that may occur in production usage. When conducting stress testing for concurrent applications, more focus should be given to testing the robustness and availability of the application. It should be noted though that analyzing the root cause for stress-related concurrency bugs and trying to reproduce stress-related bugs in a consistent manner is extremely difficult.

- Another approach is static analysis techniques [16, 21] that analyze code without actually executing the program. Usually, static analysis is performed by looking at meta-data from a compiled application or annotated source code. To deal with concurrency bugs, static analysis requires a great deal of annotations. Moreover, these annotations need to be correct themselves. Plus, tools that use static analysis tend to generate a lot of false positives and require a significant effort from the user to distinguish between the false positives.

- In another direction, model checking [19] can be used to verify the correctness of a (finite state) concurrent system. Although model checking tools provide strict verification, they often require a translation of the real system into a formal model. This not only leads to increased programming effort and the need to master a new model-specific language, but also introduces a further source of errors, i.e. the ones arising from mistakes in the translation process.

This list provides only a small portion of existing techniques. Other testing techniques, such as random testing or fuzzing [51, 4], property-based testing [22], symbolic execution [36, 7] and concolic testing [48] can be used as well. Many tools have been developed that try to test concurrent programs using one of the aforementioned techniques. Each of these tools targets programs written in a specific programming language.

1.3 Introducing Concuerror

In this thesis we are going to study and further develop *Concuerror*, a testing tool for programs written in *Erlang* (Section 2.1). Concuerror was designed and developed by A. Gotovos, M. Christakis and K. Sagonas [28, 29]. Concuerror promotes the use of TDD in concurrent programming environments and is intended to assist Erlang programmers in writing high quality concurrent software.

The real power of Concuerror lies in the fact that it is user-friendly and automated, being able to use real tests written for the original system, like stress testing tools, as well as provide sound and complete verification at the same time. Concuerror falls into the category of systematic testing (also known as stateless model checking [50]) and it tries to systematically explore process interleaving in concurrent programs.

Concuerror is inspired from CHESS [43], a systematic testing tool for concurrent software developed by Microsoft Research. Concuerror, just like CHESS aims at systematically generating all interleaving sequences of a given test and is able to consistently reproduce an erroneous execution. Some of the optimizations implemented by Concuerror (such as the preemption bounding optimization [42]) have been inspired by CHESS.

This thesis describes the changes that took place to Concuerror (changes that improve both its functionality and its usability) as well as the reasons for these changes. We also experiment with how a real user would use Concuerror in practice as a testing and debugging aid and distill from the process and the experience gained.

1.4 What's next?

The rest of the thesis is organized as follows. Chapter 2 presents a brief introduction to Erlang, particularly its concurrency related aspects, and Concuerror. Chapter 3 is the main chapter of the thesis, where we present the changes that took place to Concuerror as well as the reasons for these changes. In Chapter 4 we evaluate Concuerror against real case scenarios using big and well known Erlang applications as analysis input. Finally, in Chapter 5 we present our concluding remarks and future work.

Chapter 2

Background

2.1 The Erlang Programming Language

Following the establishment of multi-processor computing systems, software developers are shifting their attention towards languages that support and facilitate concurrent and distributed system development and Erlang is one of them. Erlang is one the oldest concurrency-oriented languages, developed by Joe Armstrong in 1986 [8]. It was originally a proprietary language within Ericsson, but was released as open source in 1998. It was designed to support the implementation of fault-tolerant distributed software systems. Erlang was originally used in telephony applications but has gained a lot more popularity recently with many companies using Erlang in their production systems like the CouchDB [1] database, the ejabberd XMPP server [2] and the SimpleDB web service by Amazon [5]. Erlang, together with libraries and the real-time distributed database Mnesia [39], forms the Open Telecom Platform (OTP) collection of libraries which is the implementation almost exclusively used by Erlang developers.

The following subsections present a brief overview of the main features of Erlang with emphasis on its concurrency related aspects. For more detailed information the reader is referred to introductory Erlang textbooks [9, 14] and the official Erlang/OTP online documentation [3].

2.1.1 Basic features

Erlang is a functional language, although not as pure as other popular functional languages (e.g. Haskell). The main language constructs are functions, and single assignment variables, meaning that variables are immutable. Erlang uses eager evaluation and supports pattern-matching, list comprehensions, higher-order functions and closures.

Among the basic Erlang datatypes are integers, floats, binaries and atoms. Atoms are represented by alphanumeric sequences starting with lowercase letter. On the other hand, variables always begin with an uppercase letter or an underscore. Erlang also provides tuples, which contain a fixed number of elements, and lists, which contain a variable number of elements, not necessarily of the same kind.

Erlang is dynamically typed. Erlang code can be written without providing any type information. Nonetheless, type annotations and function specifications have been added

to the language to allow users to provide information that can then be used to check the program for type inconsistencies. The Erlang/OTP distribution provides two tools, named Typer and Dialyzer, that combine information from user-defined annotations and perform a static analysis of the program to detect errors [37, 38].

Erlang source code is organized into modules and functions. The programmer may choose which functions should be exported and be visible outside of the module, while the rest of the functions can only be used inside the module where they are defined. Exported functions are called from outside their module using the syntax `module:function(…)`. In Listing 2.1 is shown a mergesort algorithm and in Listing 2.2 we have a quicksort algorithm both implemented in Erlang.

```
% This is the file 'ms.erl', the module and the filename must match
-module(ms).
% This exports the function 'mergesort' of arity 1
% (1 parameter, no type, no name)
-export([mergesort/1]).

split([H1, H2|T]) ->
    {L1, L2} = split(T),
    {[H1|L1], [H2|L2]};
split(L) -> {L, []}.

merge([], L) -> L;
merge(L, []) -> L;
merge([H1|T1], [H2|_T2] = L2) when H1 < H2 -> [H1|merge(T1, L2)];
merge(L1, [H2|T2]) -> [H2|merge(T2, L1)].

mergesort([]) -> [];
mergesort([_] = L) -> L;
mergesort(L) ->
    {L1, L2} = split(L),
    merge(mergesort(L1), mergesort(L2)).
```

Listing 2.1: A mergesort algorithm implemented in Erlang

```
%% qsort:qsort(List)
%% Sort a list of items
-module(qsort).     % This is the file 'qsort.erl'
% A function 'qsort' with 1 parameter is exported (no type, no name)
-export([qsort/1]).

qsort([]) -> []; % If the list [] is empty, return an empty list
qsort([Pivot|Rest]) ->
    % Compose recursively a list with 'Front' for all elements that
    % should be before 'Pivot' then 'Pivot' then 'Back' for all
    % elements that should be after 'Pivot'
    qsort([Front || Front <- Rest, Front < Pivot])
    ++ [Pivot] ++
    qsort([Back || Back <- Rest, Back >= Pivot]).
```

Listing 2.2: A quicksort algorithm implemented in Erlang

From the above examples we can conclude that:

- Lists are represented by comma separated expressions put inside brackets i.e. `[Elem1, Elem2, …]`. The "cons" operator is written as `[Head|Tail]` and the expression `[Head1, Head2, …, HeadN|Tail]` is equivalent to `[Head1|[Head2|…[HeadN|Tail]…]]`. Tuples are represented by comma separated expressions put inside curly brackets, i.e. `{Elem1, Elem2, …}`. The expression `[Front || Front<-Rest, Front<Pivot]` is a *list comprehension*, meaning "Construct a list of elements *Front* such that *Front* is a member of *Rest*, and *Front* is less than *Pivot*". `++` is the list concatenation operator.

- Guard expressions are introduced using the `when` keyword. The corresponding clause is entered only if the guard expressions is *true*.

- A catchall pattern is represented by an underscore. We also notice that there are some variables with an underscore prefix in their names. These variables have exactly the same functionality as normal variables, but no "unused variable" warnings are emitted for them by the Erlang compiler.

Erlang programs are normally compiled into bytecode and executed by the Erlang virtual machine named *BEAM*. Alternatively, Erlang source files can be compiled to native code using the *HiPE* compiler, which is also included in the Erlang/OTP distribution [35].

2.1.2 Concurrency in Erlang

Erlang main strength is support for concurrency, which was an essential element of the language's initial design, rather than a later addition. The cornerstones of Erlang's concurrency are the extremely lightweight processes it uses, which are neither operating system processes nor operating system threads and are completely managed by the Erlang VM.

Erlang implements the *actor model* of concurrency [31, 30] which means that inter-process communication is done via message passing. Erlang's processes (called *green threads*) are very lightweight and thus one can spawn thousands of them without degrading performance. In a nutshell, every process may spawn new processes, asynchronously send messages to other processes, and receive messages from them.

Inter-process communication works via an asynchronous message passing system: every process has a "mailbox", a queue of messages that have been sent by other processes and not yet consumed. A message may comprise any Erlang term, including primitives (integers, floats, characters, atoms), tuples, lists and functions. Messages are sent using the `send` (`!`) operator and are received using `receive` expressions. A process uses the `receive` primitive to retrieve messages that match desired patterns. A message-handling routine tests messages in turn against each pattern, until one of them matches. When the message is consumed and removed form the mailbox the process resumes execution.

Processes are identified by unique process identifiers (PIDs) and can be globally registered under a unique name represented by an atom. Processes can also be linked to each other, so that when one process crashes, it neatly exits and sends a message to its linked processes which can take action. This way of error handling increases maintainability and reduces complexity of code.

```
-module(ring).

-export([start/2]).

-define(NPROC, 5).

start(TTL, Token) ->
    Fun = fun(_S, N) -> spawn_link(fun() -> loop(N) end) end,
    Next = lists:foldl(Fun, self(), lists:seq(?NPROC, 2, -1)),
    Next ! {TTL, Token},
    loop(Next).

loop(Next) ->
    receive
        {1, Token} ->
            io:format(''~p: Received final token (~p)~n'',
                      [self(), Token]),
            exit(ttl_limit_surpassed);
        {TTL, Token} ->
            io:format(''~p: Received token (~p); transmitting to ~p~n'',
                      [self(), Token, Next]),
            Next ! {TTL - 1, Token},
            loop(Next)
    end.
```

Listing 2.3: A simple concurrent Erlang program

Listing 2.3 demonstrates how easy concurrency is in Erlang. The program creates a ring of processes like the one shown in Figure 2.1, where each process is linked to its two neighbors, and a message (Token) is transmitted in a circular fashion from process to process a finite (TTL) number of times. Again, we can make some remarks on the code:

- **Concurrency primitives**
  The built-in function (BIF) `spawn_link/1` atomically combines the actions of creating a new process and linking to it. All spawn-related functions have to specify what code will be executed by the newly spawned process. In the example, we use a closure for that purpose. The `spawn_link/1` function returns the PID of the spawned process. The BIF `self/0` returns the PID of the calling process. The send operator (`!`) uses the PID of a process to specify the message's destination. Note that any Erlang term can be used as a message.

- **Library functions**
  We have used functions from the `lists` (common list operations) and `io` (I/O and formatting operations) library modules of the Erlang/OTP distribution. Library functions can be called identically to user functions residing in other modules, i.e. using the `module:function(…)` syntax, and are visible without the need to import them.

- **Preprocessor**
  Erlang's preprocessor allows the use of records and macros, which are expanded before the program is compiled. In our example, we used the macro `NPROC` – referenced as `?NPROC` – to define the number of processes in the ring.

- **Tail recursion**
  Tail-recursive functions like `loop/1` are commonly used in Erlang for server-like processes. The Erlang compiler uses tail-call optimization to avoid memory exhaustion due to recursion. Therefore, memory-wise, functions of this form are equivalent to iterative loops in imperative languages.

- **Links and exits**
  To demonstrate how linked processes interact, we use the BIF `exit/1` in the first clause of the receive expression to abnormally terminate with an exception the process that receives the final token. When a process terminates this way, a signal is sent to each of its linked processes forcing them to terminate too. In our example, this results in every process terminating with an exception, given that processes are circularly linked to each other. Note that links are symmetrical, thus the action of process 1 linking to process is equivalent to the action of process 2 linking to process 1. Erlang also provides a way for processes to "catch" exit signals, instead of terminating unconditionally. Having called `process_flag(trap_exit, true)`, a process will not terminate whenever a linked process terminates abnormally, but rather will receive a message of the form `{'EXIT', Pid, Reason}`.

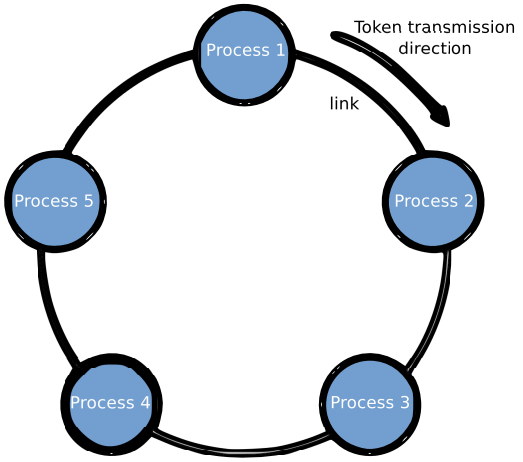

Figure 2.1: The ring of linked processes created by the program of Listing 2.3

Apart from links Erlang also provides *monitors*. Monitors are a special type of link with the difference of being unidirectional, which means that a monitored process does not know anything about being monitored. Similarly to the `'EXIT'` message that is sent by the runtime when a linked process exits, a `'DOWN'` message is sent as soon as a monitored process has exited. In this case, however, the message is sent regardless of the monitoring process' `trap_exit` flag.

This was a brief discussion of Erlang's essentials. More advanced features, like ETS and Dets storage, OTP behaviors, distributed Erlang and hot-swapping, shall not be discussed here. The most important thing to remember for the rest of this thesis, is the notion of Erlang's lightweight processes, that execute their internal actions sequentially, and communicate with each other via message passing.

2.1.3 Concurrency Errors

As we saw, messages between Erlang processes are sent asynchronously using the `!/2` expression, which is a convenient shorthand for the `send/2` function. A process can then consume messages using selective pattern matching in receive expressions, 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 deadlocks.

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) that manipulate data structures at the level of the virtual machine (VM) which are shared between all processes. Some interleaving sequences of calls to theses BIFs, can lead to data races or result in abnormal process exits.

Testing for absence of concurrency errors due to unfortunate process interleavings 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" before it has to yield back to its scheduler which then picks another process to execute. As a result, 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.

Take a look at the code in Listing 2.4. The process running `foo/0` is supposed to spawn a new process and register it under the name `math`. The new process executes `bar/3` with the given arguments and sends the results back to the first process. Among these few lines of code, lingers a concurrency error.

```
1  foo() ->
2      Self = self(),
3      Pid = spawn(fun() -> bar(Self, 42, 5) end),
4      register(math, Pid),
5      receive
6          Result -> Result
7      end.
8
9  bar(Target, X, Y) ->
10     Target ! {result, X + Y}.
```

Listing 2.4: A simple two process example with a bug

What if the newly spawned process running `bar/3` terminates before the first process executes `register/2`? In this case, according to the Erlang/OTP documentation, the `register/2` call will fail and the process will terminate with an exception. The worst part is that this case is very hard to detect using conventional testing. We can try repeatedly running `foo/0`, but it is very unlikely that the above exception will occur because `foo's register/2` almost always precedes `bar's` termination. The problem will likely occur randomly after many hours of running the program under stress.

2.2 Concuerror Overview

Concuerror aims to detect concurrency-related runtime errors like the above abnormal process exit. Concuerror is a tool that, 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 deadlocks.

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 store 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. More concretely, the program undergoes a parse 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 under 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 is very little in Erlang. Such actions are called *preemptive*.

2.2.1 Instrumenter

Concuerror instruments the code of the program under test at the granularity of modules. The translation is source-to-source and processes yield and resume execution at preemption points with a simple `receive` expression as shown in Listing 2.5.

```
1 pause () ->
2     receive scheduler_prompt -> continue end .
```

Listing 2.5: Concuerror's pause function

By calling Concuerror's `pause/0` function, a process blocks on the receive expression until a prompt from the scheduler is received. In this section we will focus on the instrumentation of built-in function calls and receive expressions as it is implemented in the current version of Concuerror (version 0.9). Many of the following concepts will be redesigned to accommodate the *dynamic partial order reduction* optimization technique (see Section 3.4).

Built-In Function Calls

The instrumentation of built-in function calls that interact with the shared state consists in their substitution with calls to appropriate wrapper functions provided by Concuerror.

The interface of a wrapper function is identical to the interface of the BIF it is replacing, i.e. it accepts the same arguments and returns the same values. Internally, all wrapper functions have the same structure: (1) the original BIF is called, (2) the scheduler is notified of the process action, (3) the process yields execution (via a call to `concuerror:pause/0`) until the scheduler prompts it to continue, and (4) when execution resumes, the result of the original BIF call is returned. From the above it is clear that Concuerror chooses to place preemption points *after* any interaction with the shared state to conveniently separate the last such interaction from the process exit. This implies that a context

switch also needs to occur before a newly spawned process starts executing the user code, otherwise the process would only yield execution after executing the first interaction with the shared state.

`receive` Expressions

The instrumentation of `receive` expressions is more complex than that of BIF calls, as receives are language expressions that are more difficult to intercept and whose semantics dictates that processes might block while waiting for a matching message to be received.

Concuerror has to ensure that the process executing the `receive` does not block forever in case no matching message ever arrives. For this, it calls the function `concuerror:receive_check/1`, whose definition is shown in Listing 2.6. More specifically, on line 3 of Listing 2.6, the function argument `F` is used to look for matching messages in the process mailbox without receiving them, i.e. without removing them from the message queue. Note that in the case expression of the function `F` there are additional clauses both for uninstrumented matching messages – send by processes executing uninstrumented code – and for instrumented or uninstrumented messages that do not match. If the mailbox contains a matching message (line 4 of Listing 2.6), then the function `concuerror:receive_check/1` returns and then the message can be consumed by the `receive` expression and the process continues by yielding execution at a preemption point (via a call to `concuerror:receive_notify/2`). If, however, the mailbox contains no matching messages, the process notifies the tool's scheduler that it is blocked and enters a busy-wait loop checking for the arrival of a matching message (line 5). As soon as such a message arrives, the process requests to be unblocked (line 11) and when the scheduler prompts it to continue, the message is received.

```
1  receive_check(F) ->
2      {messages, Mailbox} = process_info(self(), messages),
3      case match(F, Mailbox) of
4          match -> continue ;
5          no_match -> notify_scheduler(block, self()), loop(F)
6      end.
7
8  loop(F) ->
9      {messages, Mailbox} = process_info(self(), messages),
10     case match(F, Mailbox) of
11         match -> notify_scheduler(unblock, self()), pause();
12         no_match -> loop(F)
13     end.
14
15 match(F, []) -> no_match;
16 match(F, [Msg | Msgs]) ->
17     case F(Msg) of
18         match -> match ;
19         no_match -> match(F, Msgs)
20     end.
```

Listing 2.6: The `concuerror:receive_check/1` function

Timeouts

Concuerror's instrumenter eliminates any timeouts or delays in the code under test by setting them to zero. Delaying a process is equivalent to running an interleaving sequence in which that process is executed after some other processes actions have been executed. Therefore, all delays can be set to zero and, still, Concuerror will not miss any interleaving sequence.

2.2.2 Scheduler

The main purpose of Concuerror is to explore the state-space of a concurrent program. To this end, the scheduler has to produce interleaving sequences, carefully control process interleaving for each sequence, and, at the same time, handle and report process actions, including any errors that might be encountered.

For representing interleaving sequences, each process must be identified in a way that is both unique and constant across repeated executions of a test function. For this reason, Concuerror assigns to each process a logical identifier (LID), i.e. a string that uniquely identifies the process in the process tree hierarchy. The LID `P1` is assigned to the initial process of a program. Thereafter, every process' LID consists of the LID of its parent followed by the number of its siblings at the time it was spawned plus one. Thus `P1.1` will be assigned to the first process spawned by `P1`, `P1.2` to the second one and so on. This way the process hierarchy is represented by a tree like the one shown in Figure 2.2.

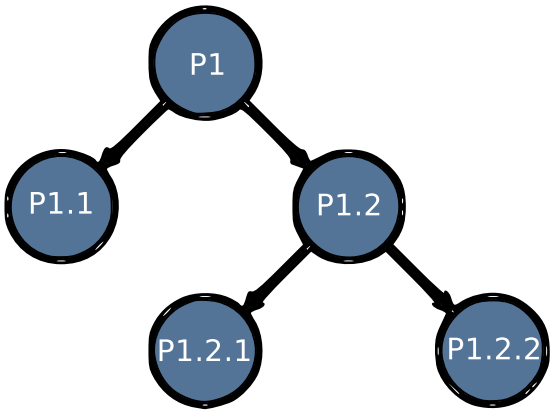

Figure 2.2: Process LID tree

The scheduler needs to keep track of some process related information during each execution. This information is stored in a structure, called scheduler *context* and includes two sets of active and blocked processes, the current interleaving sequence prefix and the currently running process. Active processes are the ones that are paused but ready to be scheduled and blocked processes are the ones that are suspended usually due to a receive.

Concuerror explores the space of valid interleaving sequences in a depth-first way. An iteration of the search consists in running one process at a time to enforce a specific interleaving sequence. At the very beginning of the search procedure, the initial user process (LID = `P1`) is spawned to execute a user defined test function and is paused right before starting its execution. The initial context consists of an active set containing process `P1`, an empty blocked set, no current process and an empty state (no processes run yet). The search begins by calling the driver and passing the initial context. At every preemption point the currently running process is paused and the driver has to determine which process is going to be executed next. Let's take for example that there are two processes in the active set (`P1` and `P1.1`). Under depth-first search, the driver should continue with `P1`, but the other choice has to be stored for future exploration. What is actually stored is the state that would have resulted if process `P1.1` was run instead of `P1` at this point. Such an interleaving sequence, which does not represent a complete program

execution but is a prefix of unexplored states, is called a *partial state*. The execution continues the same way, saving at each step all partial states resulting form choices not taken. When the current execution has finished, i.e. when all processes have terminated normally or an error has occurred, Concuerror records the result of the finished execution and initiates the next one.

The scheduler detects the three types of errors that Concuerror targets as follows:

- **Exceptions**
  Exceptions can be raised by the Erlang runtime at any time and by any process. As long as the process that exits due to an exception is known to Concuerror, the current execution is terminated and the error is logged.

- **Assertion violations**
  Concuerror allows the use of xUnit-style assertions. An assertion violation is essentially a user-defined exception and provides more information about what went wrong at some point of the program. The distinction between *exceptions* and *assertion violations* errors is made depending on the details of the process exit information.

- **Deadlocks**
  The driver reports a deadlock whenever all alive processes are blocked (empty active process set) and at the same time there are suspended processes (non-empty blocked process set) which means that the program under test cannot make progress.

2.2.3 Efficiency Improvements

Concuerror explores the state-space of a program using an exponential time complexity algorithm which suggests that the search becomes quickly infeasible as the number of preemption points increases. To this end, Concuerror uses two techniques that significantly improve the efficiency of the search. The first is a simple partial-order reduction technique that avoids redundant interleaving sequences involving process blocks on `receive` expressions [15]. The second is a heuristic method, called preemption bounding, that bounds the number of allowed context switches and drastically reduces the number of explored interleaving sequences [42].

A newer version of Concuerror developed in parallel with this thesis incorporates a powerful *dynamic partial order reduction* technique, which considerably reduces the number of explored interleaving sequences (Section 3.4).

Chapter 3

Extending Concuerror

Concuerror has been successfully tested and used on a number of small tests that have been created for that sole purpose and the errors were known a priori. We wanted to be able to use Concuerror to test real world projects. We wanted to be able to analyze code bases with hundreds of thousands of lines of code which use complex communication protocols and depend upon many libraries (OTP or not). In order to do so, we introduced a number of improvements over the existing implementation of Concuerror.

3.1 Command Line Interface

The first version of Concuerror was designed to be operated through a graphical user interface (GUI) (Figure 3.1). The GUI allows the user to import Erlang modules and select a test to be executed. After the analysis is complete, information about any errors encountered is displayed. The user may choose to replay some of the erroneous sequences and acquire detailed, action by action interleaving information. The initial developers believed that a command line interface (CLI) was not able to convey information about process interaction in a nice and usable format. Although this may be true in general (GUI is considered more user friendly than CLI) we believe that this is not the case with Concuerror.

Concuerror is a tool designed to be used by programmers. Such users are already familiar with operating a command line interface. As a result the biggest (if not the only) advance of GUI over CLI does not apply in our case. On the other hand command line interfaces offer better scripting and remote access capabilities. Concuerror is a testing tool and as such one must be able to use it in a remote server (testing/development machine) were graphical interface may be not an option, or even automate the test processes through continuous integration software. For the above reasons we decided that we have to implement a command line interface which will offer all the previous functionality.

3.1.1 Analysis Results

Concuerror's analysis results are divided into three parts: the error type (assertion violation, deadlock or exception), the error's short description and the interleaving sequence that led to this error. When one is using the graphical interface, she gets a list of all the

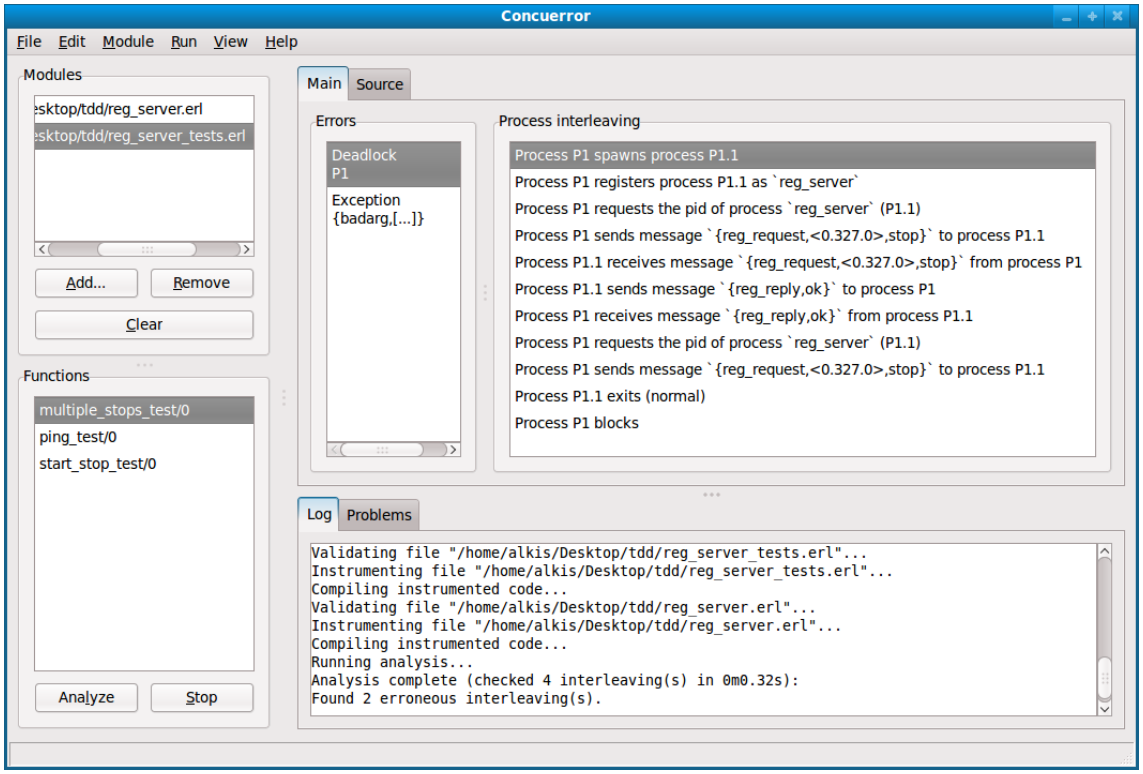

Figure 3.1: The Concuerror GUI

errors Concuerror was able to find (along with a short description) and then when she selects one of them, the corresponding erroneous thread is replayed to produce a detailed list of the interleaving for this error.

As we can see in Figure 3.1, the user has selected two modules to be instrumented by Concuerror, the `reg_server` and the `reg_server_tests`. From them, she has chosen to analyze the `multiple_stops_test/0` function under the `reg_server_tests` module. The analysis has been completed in 0.32 seconds (as we can see in the *Log* tab) after checking 4 interleavings and founding 2 erroneous ones. A short description of these errors can been found under the *Errors* tab (here we have a *Deadlock* and an *Exception*) and the corresponding process interleaving sequence for each error under the *Process Interleaving* tab.

Unfortunately this *point and click* user experience can not be provided by command line environments. Moreover we want to be able to save all the analysis information in a file as Concuerror may be part of an integration testing process.

The first thought was for Concuerror to find all the errors and then for every one of them replay it so we could get full interleaving details. That seemed a bit of excessive so we ended up completely removing the replay functionality. Concuerror now keeps a list of all the interleavings for every error encountered and then it saves the results in a file for the user to review them offline.

Concuerror was also lacking an indicator to show the progress of the analysis. We created a somewhat simple but straightforward progress bar which displays the preemption bound at which the analysis is running as well as the current number of checked interleavings, the

number of errors encountered so far and the elapsed time. The ideal would be for Concuerror to be able to compute the estimated completion time but that proved to be difficult when using algorithms such as the dynamic partial order reduction (see Section 3.4).

3.1.2 Analysis Termination

As we saw in Figure 1.1 most software development cycles usually involve running some tests, finding errors, fixing them and then repeating this process again until no more errors can be found. Concuerror on the other hand performs an exhaustive state space traversal (all interleavings of program threads are checked for property violations). This means that the analysis could take a lot of time to be completed, making Concuerror unsuited for these kinds of development models. In order to adapt to these development models, one should be able to stop the analysis at any time, resolve the errors found so far and repeat. Although this functionality was already provided by Concuerror's graphical user interface, implementing it for the command line interface was a little bit trickier.

Most CLI applications use SIGINT signal (keyboard interrupt) in order to stop execution. We wanted to preserve the same functionality in Concuerror and handle the above signal in order to save the analysis results so far and then exit. Unfortunately, OS signals are handled exclusively by the Erlang VM and do not propagate to the running process so that they can be handled properly. The only viable option is to have a script starting Concuerror and also trapping the SIGINT signal. We chose to use the bash scripting language for simplicity and since all the flag parsing is done inside the Erlang code one can easily opt to use alternative programming languages which would be more portable (such like `C`).

Listing 3.1 shows how we trap the SIGINT signal and instruct Concuerror to stop. In Erlang, a *node* is an executing runtime system which has been given a name, using the command line flag `-name`. Every Concuerror instance, runs in a distinct Erlang node and is assigned a unique name which then will be used to call `concuerror:stop/0` on this node. This function will send the `stop_analysis` message which will cause Concuerror to save the analysis results up to that point and then exit (see Listing 3.2).

3.1.3 Command-line options

With the introduction of the CLI, Concuerror started to use command line options in order to control its operation. The parsing of the flags is done completely using Erlang code by the `concuerror:parse/2` function. Although Concuerror now supports both the usage of a command line and a graphical user interface, most of the options can be given only using the former one. Here we will present some of these options and their usage.

–target

This option is used to specify the function that Concuerror will use as entry point when exploring the interleaving sequences of a test. It can take an arbitrary number of arguments and its functionality depends upon them. If only the module name is given then Concuerror will try and run all Eunit tests for this module (more on this in Section 4.1). Otherwise Concuerror will execute the given function with the specified arguments (if no arguments are given, it is assumed that the function is of arity 0). This option is mandatory and can be specified from inside the GUI too.

```
#!/bin/bash

Date=$(date +%s%N)
Name="Concuerror$Date"
Cookie="ConcuerrorCookie"

trap ctrl_c INT
function ctrl_c() {
    erl -sname ConcuerrorStop -noinput -cookie $Cookie \
        -pa /home/ilias/erlang/Concuerror/ebin \
        -run concuerror stop $Name -run init stop
    wait
}

erl +Bi -smp enable -noinput -sname $Name -cookie $Cookie \
    -pa /home/ilias/erlang/Concuerror/ebin \
    -run concuerror cli -run init stop -- "$@" &
wait $!
```

Listing 3.1: The Concuerror Bash script

```
%% @spec stop() -> ok
%% @doc: Stop the Concuerror analysis
-spec stop() -> ok.
stop() ->
    try ?RP_SCHED ! stop_analysis
    catch
        error:badarg ->
            init:stop()
    end,
    ok.
```

Listing 3.2: The `concuerror:stop/0` function

-D and -I
: These flags are used to define a symbol in the Erlang pre-processor and to add some directories to the directory search list included files.

–keep-tmp-files
: When Concuerror instruments the input files, it performs a module renaming (more on this in Section 3.2). During this processes it creates some temporary files which later removes. With this flag Concuerror will keep these files so that one can later inspect them.

–fail-uninstrumented
: Using this flag, analysis will fail when the program under test tries to call a function of an uninstrumented module. Although Concuerror can handle both instrumented and uninstrumented modules, the more modules one chooses to instrument the more accurate the results will be (see Section 3.2). With `-fail-uninstrumented` the user can inspect these uninstrumented modules and choose to instrument them or ignore them.

```
                          Concuerror
A systematic testing tool for concurrent Erlang programs.
                          Version 0.9

usage: concuerror [<args>]
Arguments:
  -t|--target module      Run eunit tests for this module
  -t|--target module function [args]
                          Specify the function to execute
  -f|--files  modules     Specify the files (modules) to instrument
  -o|--output file        Specify the output file (default results.txt)
  -p|--preb   number|inf  Set preemption bound (default is 2)
  -I          include_dir Pass the include_dir to concuerror
  -D          name=value  Define a macro
  --noprogress            Disable progress bar
  -q|--quiet              Disable logging (implies --noprogress)
  -v                      Verbose [use twice to be more verbose]
  --keep-tmp-files        Retain all intermediate temporary files
  --fail-uninstrumented   Fail if there are uninstrumented modules
  --ignore    modules     It is OK for these modules to be uninstrumented
  --show-output           Allow program under test to print to stdout
  --wait-messages         Wait for uninstrumented messages to arrive
  --app-controller        Start an (instrumented) application controller
  -T|--ignore-timeout bound
                          Treat big after Timeouts as infinity timeouts
  --gui                   Run concuerror with a graphical interface
  --dpor                  Runs the experimental optimal DPOR version
  --help                  Show this help message

Examples:
  concuerror --target foo bar arg1 arg2 --files ''foo.erl'' -o out.txt
  concuerror -DVSN=\''1.0\'' --gui -I./include --files foo.erl --preb inf
```

Listing 3.3: Concuerror's help output

–ignore
: This option is used alongside with `-fail-uninstrumented` and it allows one to choose which modules are ok to be uninstrumented.

–show-output
: By default Concuerror implements an IO server and channels all IO through there by making it the group leader of all newly created processes. This way Concuerror prevents the program under test with writing into the terminal and selectively prints the output only for the erroneous interleavings. This flag disables this behavior.

–wait-messages
: One of the problems that Concuerror has to face (regarding the approach it takes of instrumenting the modules under test) is the inability to know about and correctly handle processes which have been spawned from an uninstrumented part of code. One may try to solve this problem by choosing to instrument as many modules as she can and, as we will see in Section 3.2, instrumenting OTP libraries can help towards this direction. But there are certain processes that currently Concuerror cannot handle and these are the ones initialized by the Erlang VM itself (such as the `init`

process). When a process is sending a message to an uninstrumented one, Concuerror has no means to control when this message will be processed and when or if a reply will be sent back. Therefore Concuerror's analysis becomes non-deterministic. In the best case scenario Concuerror will fail to examine some interleavings but in the worst one it may fail to find some errors or report false positives.

Using `--wait-messages`, Concuerror will wait a pre-configured amount of time (10 msecs) every time an instrumented process tries to send a message to an uninstrumented one. Doing this, Concuerror allows the latter process to receive and handle its messages. Naturally, this slows down the exploration, so this option is to be used only when necessary.

–app-controller

When an Erlang runtime system is started, a number of processes are started as part of the kernel application. One of these processes is the *application controller* process, registered as application_controller. In order for Concuerror to be able to instrument Erlang applications (such as the `ssl` application) it needs to start its own renamed application controller as well as handle the termination of the registered applications afterwards. This is done during the spawn of the function under testing by the `concuerror_rep:start_target/3` function shown in Listing 3.4.

```
%%%----------------------------------------------------------------------
%%% Start analysis target module/function
%%%----------------------------------------------------------------------
-spec start_target(module(), atom(), [term()]) -> ok.
start_target(Mod, Fun, Args) ->
    InstrAppController = ets:member(?NT_OPTIONS, 'app_controller'),
    AppConModule =
        concuerror_instr:check_module_name(application_controller, none, 0),
    AppModule = concuerror_instr:check_module_name(application, none, 0),
    case InstrAppController of
        true ->
            AppConModule:start({application, kernel, []}),
            AppModule:start(kernel),
            AppModule:start(stdlib),
            ok;
        false ->
            ok
    end,
    apply(Mod, Fun, Args),
    case InstrAppController of
        true ->
            lists:foreach(fun ({App, _, _}) -> AppModule:stop(App) end,
                          AppModule:loaded_applications()),
            ok;
        false ->
            ok
    end.
```

Listing 3.4: Start an instrumented application controller

–ignore-timeout

Concuerror by default does not attempt to model or simulate the effect of delays on a program's execution. Delaying a process is equivalent to running an interleaving

sequence in which that process is executed after some other processes' actions have been executed. Therefore, all delays can be set to zero and, still, Concuerror will not miss any interleaving sequence.

That said, we can not ignore that in the presence of delays Concuerror may produce interleaving sequences that are not likely to occur in practice. Moreover, when we instrument OTP modules such as the gen_server module, which use timeouts as a way to detect communication errors, we may end up with too many false positive erroneous interleavings. Using this flag we can provide an upper limit and instruct Concuerror to treat timeouts with values bigger than this limit as they were the atom infinity. This means that Concuerror will never examine interleaving sequences where these timeouts have been executed.

3.2 Instrumentation of Libraries

As we saw in Section 2.2.1, the instrumenter provides some complex hooks for the scheduler, by parse transforming the source code. In doing so, the instrumenter has to be extremely careful not to alter the original program's semantics.

Ideally the instrumenter could transform any module and use it in the analysis. But when it comes to instrumenting modules from the OTP library there are two main problems. The first one is that some of these modules affect the Erlang runtime system itself and so the Erlang Virtual Machine prevents us from reloading them (the `kernel`, `stdlib` and `compiler` directories are considered sticky). The second one is that the very same modules that we may want to instrument are used by Concuerror itself. Of course this can not be done because when Concuerror will try to use one of these instrumented modules it will hang. In order to bypass these problems, Concuerror has to rename all the instrumented modules and assign unique names to them.

Concuerror renames every module during the instrumentation phase (see Listing 3.5). To do so, it first renames and saves the module file under a temp directory and then parses this file in order to change the module attribute that defines the name of the module (which should be the same as the file name minus the extension `.erl`). This step is important as during the preprocessing phase (done by `epp:parse_file`) the predefined macros `?MODULE` and `?FILE` are resolved automatically to the ones specified by the module attribute.

After renaming the modules, we have to change the function calls to these modules to point to the instrumented code. This is done in two steps. At first, we change the function calls statically when we instrument other modules. Then, dynamically, we check if we need to rename any function calls for which the module names were not known at compile time as for example when they are assigned into a variable. In order to decide which module names must be renamed and which not Concuerror uses the function `check_module_name/3`. This function checks a given module name against a set of rules and applies the first rule to match (see Listing 3.6):

1. If the module belongs to Concuerror, that is its name starts with `'concuerror_'`, then don't rename it.

2. If the module has been marked as to be ignored using the `--ignore` flag then don't rename it.

```
rename_module(Module, File) ->
    ModuleStr = atom_to_list(Module),
    NewModuleStr = atom_to_list(new_module_name(Module)),
    TmpDir = ets:lookup_element(?NT_INSTR, ?INSTR_TEMP_DIR, 2),
    NewFile = filename:join(TmpDir, NewModuleStr ++ ''.erl''),
    case file:read_file(File) of
        {ok, Binary} ->
            %% Replace the first occurrence of '-module(Module).'
            Pattern = binary:list_to_bin(
                ''-module('' ++ ModuleStr ++ '').''),
            Replacement = binary:list_to_bin(
                ''-module('' ++ NewModuleStr ++ '').''),
            NewBinary = binary:replace(Binary, Pattern, Replacement),
            %% Count lines of code
            NewLine = binary:list_to_bin(''\n''),
            Lines = length(binary:matches(NewBinary, NewLine)),
            %% Write new file in temp directory
            case file:write_file(NewFile, NewBinary) of
                ok    -> {ok, NewFile, Lines};
                Error -> Error
            end;
        Error ->
            Error
    end.
```

Listing 3.5: Rename modules during the instrumentation phase

3. If the module, function pair is an Erlang built in function (BIF) then don't rename the module.
4. If the module has been instrumented then rename it.
5. If `-fail-uninstrumented` flag has been given, then rename the module. This way the program under test will crash when it will try to call the renamed function (as this function will not exist) and Concuerror will report that there are modules needed by our test that have not been instrumented.
6. If none of the above apply, then don't rename the module.

3.3 Extending Concuerror's Test Suite

In software development, a *test suite* is a collection of test cases that are intended to test a software program and ensure that it has some specified set of behaviors. A test suite often contains detailed instructions or goals for each collection of test cases and information on the system configuration to be used during testing. Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. The main levels during the development process as defined by the *SWEBOK* guide [34] are unit and system testing.

Unit testing, also known as component testing, refers to tests that verify the functionality of a specific section of code, usually at the function level. These types of tests are usually written by developers as they work on code, to ensure that the specific function is working

```
%% ---------------------------
%% Rename a module for the instrumentation.
%% 1. Don't rename 'concuerror_*' modules
%% 2. Don't rename 'ignored' modules
%% 3. Don't rename 'BIFS'.
%% 4. If module is instrumented rename it.
%% 5. If we are in 'fail_uninstrumented' mode rename all modules.
-spec check_module_name(module() | {module(),term()}, atom(), non_neg_integer())
                        -> module() | {module(), term()}.
check_module_name({Module, Term}, Function, Arity) ->
    {check_module_name(Module, Function, Arity), Term};
check_module_name(Module, Function, Arity) ->
    Conc_Module =
        try atom_to_list(Module) of
            ("concuerror_" ++ _Rest) -> true;
            _Other -> false
        catch   %% In case atom_to_list fail, we don't want to rename the module
            error:badarg -> true
        end,
    Rename = (not Conc_Module)
        andalso (not ets:member(?NT_INSTR_IGNORED, Module))
        andalso (not ets:member(?NT_INSTR_BIFS, {Module, Function, Arity}))
        andalso (ets:member(?NT_INSTR_MODS, Module)
            orelse ets:lookup_element(?NT_INSTR, ?FAIL_BB, 2)),
    case Rename of
        true  -> new_module_name(Module);
        false -> Module
    end.

-spec new_module_name(atom() | string()) -> atom().
new_module_name(StrModule) when is_list(StrModule) ->
    %% Check that module is not already renamed.
    case StrModule of
        (?INSTR_PREFIX ++ _OldModule) ->
            %% No need to rename it
            list_to_atom(StrModule);
        _OldModule ->
            list_to_atom(?INSTR_PREFIX ++ StrModule)
    end;
new_module_name(Module) ->
    new_module_name(atom_to_list(Module)).
```

Listing 3.6: The check whether a given function application must be renamed

as expected. System testing on the other hand, tests a completely integrated system to verify that it meets its requirements.

During this thesis Concuerror went through dramatic changes and many of scheduler's and instrumenter's parts were completely rewritten. In order to do so we wanted to be sure that the functionality of the tool itself would not be affected and that the analysis results would be the same. Unfortunately, Concuerror was using only unit tests that could not fully test its capabilities and did not preserve extensive results for feature reference. Therefore, before any alternation could be made, we decided to significantly extend the test suite of Concuerror, load it with enough test results to use as references and gradually extend it as we move on.

Actually, Concuerror's test suite was written from scratch in order to fit exactly in our needs. For each test the user can now specify the preemption bound as well as the analysis algorithm to be used. The results are compared with a set of predefined ones and the user may choose to ignore them or update them respectively. The test suite is written in *python* and has support for executing tests in parallel. Currently there are more than 100 total tests which give rise to more than 400 test cases.

3.4 Dynamic Partial Order Reduction

As we already saw in Section 2.2, Concuerror systematically explores the state space of a concurrent software system by driving its execution via a run-time scheduler. In the context of this approach, partial order reduction seems (so far) to be the most effective technique for reducing the size of the state space of concurrent software systems at the implementation level [26, 52]. Using partial order reduction for model checking software one can initially explore an arbitrary interleaving of the various concurrent processes/threads, and then identify backtracking points where alternative paths in the state space need to be explored by *dynamically* tracking interactions between these processes.

Konstantinos Sagonas and Stavros Aronis are currently researching an optimal dynamic partial order reduction method to be used in Concuerror. The Algorithm is based on the work of Flanagan and Godefroid [23] and has been optimized for the Erlang's process-based, no-shared-state concurrency models. A new scheduler which implements this algorithm has been written for Concuerror and the results so far are more than encouraging with the new scheduler being able to identify the same defects as the old one and dramatically reduce the exploration's state space at the same time.

Chapter 4

Concuerror By Example

In this chapter we will see how Concuerror can be used in practice as a testing and debugging aid. The goal is to experiment with how a real user would use Concuerror and distill from the process and the experience gained. For this we are going to experiment with some well known OTP libraries [46] such as the eunit and the gen_server libraries as well as analyze bigger and more complex Erlang applications such as mochiweb [40].

Instead of writing our own tests for analyzing the mochiweb application we decided to use a subset of the tests that come along with it. Although these tests have not been written for testing concurrency aspects of the system, we wanted to see how easy it is to use the pre-existing ones. Most of Erlang's applications come with a set of unit tests written using the Eunit testing framework. So the first step is to integrate Concuerror with Eunit.

4.1 Run Eunit tests through Concuerror

First, we will create a simple Erlang program involving two processes such as the one shown in Listing 4.1. The `pong/0` function, which is exported and may be called out of the `ping_pong` module, spawns a process that will execute the code of function `ping/1` (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, such errors compromise the robustness of applications.

Let's run Concuerror over this simple example and confirm that it can identify this concurrency error. We invoke Concuerror as shown in Listing 4.2 and the results are shown in Listing 4.3. As we can see, Concuerror correctly reports one erroneous interleaving with error `Exception: badarg` when trying to register the aforementioned process. Inside the `results.txt` file we can find the complete trace that led to the error.

Now we will create another module, named `ping_pong_test` (Listing 4.4) that will contain a simple unit test for our `pong/0` function using eunit. Running this test of course will succeed most of the times and it is very likely that it may never fail and we may never notice

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

Listing 4.1: Simple example program involving two processes and a concurrency error

```
$ ./concuerror --target ping_pong pong --files ping_pong.erl \
               --preb inf -o results.txt --dpor

Instrumenting files... done

Running analysis with preemption bound infinity...

Analysis complete. Checked 2 interleaving(s) in 0m0.01s:
Found 1 erroneous interleaving(s).

Writing output to file results.txt... done
```

Listing 4.2: Analyze ping_pong using Concuerror

```
Checked 2 interleaving(s). 1 errors found.

1
Error type        : Exception
Details           : {badarg,[{erlang,register,[ping_pong,<0.51.0>],[]},
                             {ping_pong,pong,0,[]}]}
  Process P1 spawns process P1.1
  Process P1.1 sends message 'ping' to process P1
  Process P1.1 exits (normal)
  Process P1 registers process P1.1 (dead) as 'ping_pong'
  Process P1 exits (''Exception'')
```

Listing 4.3: Analysis results from ping_pong

the concurrency error. In order to run this test under Concuerror, one has to instrument the `eunit` and `io` modules as a minimal requirement. We need to instrument the eunit module because Concuerror doesn't know how to interpret and correctly run eunit tests. On the other hand, the requirement for the io module is a little bit trickier. Eunit, just like Concuerror, implements its own IO server in order to manage the output messages and display them in the logs. That means that when one is using the io module (which implements an IO client) it sends a message to eunit and waits for response. Therefore Concuerror has to know that a message was send to eunit though the io module and so the io module has to be instrumented. With all that said we can invoke Concuerror to

run this eunit test as shown in Listing 4.5.

```
1 -module(ping_pong_test).
2
3 -include_lib("eunit/include/eunit.hrl").
4
5 pong_test() ->
6     ?assertEqual(ok, ping_pong:pong()).
```

Listing 4.4: Eunit test for the ping_pong:pong/0 function

```
$ ./concuerror --target ping_pong_test -f ping_pong.erl ping_pong_test.erl \
    --wait-messages -T 2000 -p 1 --dpor \
    -I $OTP_PATH/lib/eunit/include \
    -f $OTP_PATH/lib/eunit/src/*.erl $OTP_PATH/lib/stdlib/src/io.erl

Instrumenting files... done

Running analysis with preemption bound 1...

Analysis complete. Checked 10 interleaving(s) in 0m0.10s:
No errors found.

Writing output to file results.txt... done
```

Listing 4.5: Analyze ping_pong_test using Concuerror

4.1.1 Let it crash

But wait! Analysis yields that there are no errors at all. Why did that happen? Well, Concuerror reports two type of errors, *Exceptions* and *Deadlocks*. In order to catch an exception the program under test has to crash. This does not happen when we run a test using Eunit because, the Eunit framework catches all exceptions, reports them and exits gracefully leaving Concuerror to believe that everything went normal. To circumvent that we have to patch Eunit (Listing 4.6), and force it to propagate any exceptions during the test to Concuerror. Repeating the analysis process again, we get the erroneous interleaving.

4.2 Identifying a bug in the gen_server OTP module

During our experiments with the mochiweb library, we stumbled upon a concurrency error in the gen_server OTP module. In fact this error has not been discovered or resolved by the Erlang community yet. In this section we will try to reproduce and analyze the above defect.

Let's begin with a simple example that implements the Generic Server Behavior as shown in Listing 4.7. In this example we have written the very basic functionality of a gen_server and in the function `test_start_stop_twice/0` we start and stop this server twice. By

```
diff --git a/lib/eunit/src/eunit_proc.erl b/lib/eunit/src/eunit_proc.erl
index ec7d93f..8303e05 100644
--- a/lib/eunit/src/eunit_proc.erl
+++ b/lib/eunit/src/eunit_proc.erl
@@ -505,11 +505,13 @@ handle_test(T, St) ->
 run_test(#test{f = F}) ->
-    try eunit_test:run_testfun(F) of
-       {ok, _Value} ->
-           %% just discard the return value
-           ok;
-       {error, Exception} ->
-           {error, Exception}
-    catch
-       throw:WrapperError -> {skipped, WrapperError}
-    end.
+    F(),
+    ok.
+%%    try eunit_test:run_testfun(F) of
+%%     {ok, _Value} ->
+%%         %% just discard the return value
+%%         ok;
+%%     {error, Exception} ->
+%%         {error, Exception}
+%%    catch
+%%     throw:WrapperError -> {skipped, WrapperError}
+%%    end.
```

Listing 4.6: Patch `eunit` to propagate exceptions to Concuerror

calling the function `gen_server:start_link/4`, we spawn and link to a new process, a gen_server.

- The first argument `{local, gsb}` specifies the name. In this case, the gen_server will be locally registered as `gsb`. If the name is omitted, the gen_server is not registered. As we will discuss later on, not registering the server makes this bug disappear.
- The second argument, `?MODULE`, is the name of the callback module, that is the module where the callback functions are located. In this case, the interface functions are located in the same module as the callback functions hence the use of this predefined macro.
- The third argument, `[]`, is a term which is passed as is to the callback function `init`. Here, `init` does not need any input data and ignores the argument.
- The fourth argument, `[]`, is a list of options. Here we leave the defaults.

If name registration succeeds, the new gen_server process calls the callback function `gen_server_bug:init([])`. After that we make a synchronous stop request to the server using the `gen_server:call/2` function. The request is made into a message and sent to the gen_server. When the request is received, the gen_server calls `handle_call(stop, From, State)` which returns `{stop, normal, ok, State}`, at which point the server exits with reason normal.

```
-module(gen_server_bug).
-behaviour(gen_server).

-export([test_start_stop_twice/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
         code_change/3]).

test_start_stop_twice() ->
    ServerName = {local, 'gsb'},
    {ok, Pid1} = gen_server:start_link(ServerName, ?MODULE, [], []),
    gen_server:call(Pid1, stop),
    {ok, Pid2} = gen_server:start_link(ServerName, ?MODULE, [], []),
    gen_server:call(Pid2, stop),
    ok.

%% ====================================================================
%% CallBack Functions
init([]) ->
    {ok, undefined}.

handle_call(stop, _From, State) ->
    {stop, normal, ok, State};
handle_call(_Event, _From, State) ->
    {reply, ok, State}.

handle_cast(_Event, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Listing 4.7: A simple server using the gen_server behavior

At first glance there is nothing wrong with our code. We expect to be able to start our `gen_server` right away after we have stopped it with a synchronous stop call. The next step is to analyze this example using Concuerror. The minimum set of instrumented OTP modules needed are the `gen_server`, the `gen` and the `proc_lib` ones. Invoking Concuerror as shown in the previous examples we encounter an exception. In particular the second `start_link/4` function call returns that the server is already started. The full analysis result when running Concuerror with infinite preemption bound is shown in Listing 4.8.

In order to understand how this exception was raised and why Concuerror thinks that our `gen_server` is *already started* we have to follow the chain of events reported in the results.

Process P1 calls `gen_server:start_link/4` where after checking if a process with name `gsb` exists, it spawns process P1.1 (our gen_server) and blocks. Process P1.1 registers itself as `gsb` and calls `gen_server_bug:init/1` where the initialization of our server is taking place. Afterwards it sends an acknowledgment message back to process P1 and blocks. As we see from this the start of a gen_server is a synchronous process.

```
Checked 5 interleaving(s). 1 errors found.

1
Error type        : Exception
Details           : {{badmatch,{error,{already_started,<0.88.0>}}},
                     [{gen_server_bug,test_start_stop_twice,0,
                                      [{file,"gen_server_bug.erl"},
                                       {line,12}]}]}
  Process P1 requests the pid of unregistered process `gsb` (undefined)
  Process P1 spawns with opts to process P1.1
  Process P1 blocks
  Process P1.1 registers process P1.1 as `gsb`
  Process P1.1 sends message `{ack,<0.88.0>,{...}}` to process P1
  Process P1.1 blocks
  Process P1 receives message `{ack,<0.88.0>,{...}}` from process P1.1
  Process P1 monitors process P1.1
  Process P1 sends message `{'$gen_call',{<0.87.0>,...},stop}` to process P1.1
  Process P1 blocks
  Process P1.1 receives message `{'$gen_call',{<0.87.0>,...},stop}`
               from process P1
  Process P1.1 sends message `{#Ref<0.0.0.1152>,ok}` to process P1
  Process P1 receives message `{#Ref<0.0.0.1152>,ok}` from process P1.1
  Process P1 demonitors process P1.1
  Process P1 requests the pid of process `gsb` (P1.1)
  Process P1 exits ("Exception")
```

Listing 4.8: Analysis results for the gen_server_bug module

Now process P1 tries to stop our server. For this it sends a specially crafted message to process P1.1 with stop as a request. Then it blocks waiting for a reply (we used the `gen_server:call/2` function call so our request ought to be a synchronous one). Our server receives this message and then proceeds with the following:

- Process P1.1 calls `gen_server:decode_msg/8`
- Process P1.1 calls `gen_server:handle_msg/5`
- Process P1.1 calls `gen_server_bug:handle_call/3`
- Process P1.1 calls `gen_server_bug:terminate/2`
- Process P1.1 sends reply to P1
- Process P1.1 exits

So the reply to the process P1 is sent before process P1.1 exits. If process P1 was to try and start the gen_server again it would complain that the server is already started because process P1.1 has not exited yet and it remains registered under the name `gsb`. This is exactly what is happening on this particular interleaving.

In order to solve this race condition, one solution could be for the server (process P1.1) to unregister itself before sending back the reply message. This can be easily implemented inside the gen_server OTP module but providing such a patch here is outside of the scope of this thesis.

4.3 Analyze the MochiWeb library

MochiWeb is an Erlang library for building lightweight HTTP servers, created by Bob Ippolito. Despite not having an official website or narrative documentation, mochiweb is a popular choice to build web services in Erlang. It is very minimal but at the same time uses a lot of OTP applications and libraries. This makes it a perfect candidate for testing how Concuerror copes with real life examples.

We are going to use the latest versions of mochiweb and OTP as of the point of writing this, that is, commit 680dba8 for mochiweb and version R16B01 for OTP. MochiWeb has a set of unit tests written using the Eunit testing framework. These tests are divided into four modules located under the *test* directory inside the source tree. From these four modules, we are going to examine the `mochiweb_tests` one, as it contains tests that provide full coverage over the mochiweb code.

In order to determine the full extent of Concuerror's capabilities over instrumenting OTP modules, we are going to instrument all the modules that are used by mochiweb and not just the minimum set needed to run the tests (as we did with our previous examples). We achieve that by using the `fail-uninstrumented` flag and gradually instrument modules as we move on.

4.3.1 Configuring Concuerror

There is a number of preparations needed to be made before we are able to run Concuerror over such a big project as mochiweb. These are some small modifications in both mochiweb and OTP code bases about things that Concuerror cannot resolve by itself.

- Since mochiweb uses the Eunit framework, the first thing we need to do is to apply our patch to Eunit that will allow Concuerror to catch the exceptions (see Listing 4.6).
- We need to apply a small patch (Listing 4.9) to the `ssl` module of OTP. For some reason, `ssl_manager` module uses the predefined macro `?MODULE` to spawn and register a gen_server but then uses the atom `ssl_manager` to refer to this server. Concuerror's rename functionality (Section 3.2) cannot handle this type of hard coded naming references at the moment and so we have to rename `ssl_manager` by hand.
- MochiWeb depends on four OTP applications for its operation, namely the `crypto`, `asn1`, `public_key` and `ssl` applications. For Concuerror to run correctly we have to instrument at least the last one. This is because the function `ssl:transport_accept/1` called by `mochiweb_socket:accept/1` waits until a connection has been established. Concuerror has to know that this function sleeps otherwise it will report it as a deadlock.

  In order to instrument the `ssl` application we have to tell mochiweb to use the instrumented one, using the patch shown in Listing 4.10 and we have to provide an application resource file for our instrumented `ssl` (Listing 4.11).

```
diff --git a/lib/ssl/src/tls_connection.erl b/lib/ssl/src/tls_connection.erl
index 246fecf..ee72462 100644
--- a/lib/ssl/src/tls_connection.erl
+++ b/lib/ssl/src/tls_connection.erl
@@ -1240,5 +1240,5 @@ ssl_init(SslOpts, Role) ->
 init_manager_name(false) ->
-    put(ssl_manager, ssl_manager);
+    put(ssl_manager, conc__ssl_manager);
 init_manager_name(true) ->
-    put(ssl_manager, ssl_manager_dist).
+    put(ssl_manager, conc__ssl_manager_dist).
```

Listing 4.9: Patch ssl from OTP and rename `ssl_manager` atom

```
diff --git a/src/mochiweb_socket_server.erl b/src/mochiweb_socket_server.erl
index a3d4da3..5f287b9 100644
--- a/src/mochiweb_socket_server.erl
+++ b/src/mochiweb_socket_server.erl
@@ -141,3 +141,3 @@ prep_ssl(true) ->
     ok = mochiweb:ensure_started(public_key),
-    ok = mochiweb:ensure_started(ssl);
+    ok = mochiweb:ensure_started(conc__ssl);
 prep_ssl(false) ->
```

Listing 4.10: Patch `mochiweb` to use the instrumented `ssl` application

```
{application, conc__ssl,
   [{description, ''Erlang/OTP SSL application''},
    {vsn, ''5.3''},
    {registered, [conc__ssl_sup, conc__ssl_manager]},
    {applications, [crypto, public_key, kernel, stdlib]},
    {env, []},
    {mod, {conc__ssl_app, []}}]}.
```

Listing 4.11: Application resource file for the instrumented `ssl` application

4.3.2 Findings

We are now ready to deploy Concuerror over mochiweb. We are doing so by running the script shown in Figure 4.12, where we have try to instrument as many modules as we could leaving aside (using the `ignore` flag) only a few ones. With this we reach the astonishing number of 106727 total lines of instrumented code. The module `mochiweb_tests` contains approximately 20 tests, and each interleaving can take up to 3 minutes to complete. In order to reduce the time needed to run each interleaving we have to break the module into smaller pieces and analyze each test function one by one while commenting out the rest of them.

From our analysis we were able to find two different defects. The first one we already discussed, was the bug in the `gen_server` OTP module. The second one was more or less the same but it was a defect in mochiweb itself. MochiWeb was using the `gen_server:cast/2`

function in order to send the stop request to the server, which is an asynchronous call. This means that if one were to try and start the mochiweb HTTP server again before the stop request could be completed she would have faced with the same problems discussed in Section 4.2. Because of the nature of the `gen_server:cast/2` call this bug is much more likely to occur in real time applications than the one found in the `gen_server` OTP module. This is why the above defect was reported to the mochiweb developer, and a patch fixing it has been merged upstream.

The fact that Concuerror finds and reports the above concurrency errors means that our tools works and it is capable of instrumenting and analyzing modules that heavily depend on OTP libraries. It also means that we are able to test a software end to end and identify bugs not only on the software's code base but also on the libraries it may use. This way the developer can be sure that she uses the libraries the way it meant to be and doesn't get any unexpected defects caused by them. In addition our experiments show that Concuerror is compatible with Eunit and that one can easily use the already existing Eunit tests to analyze her software.

```
#!/bin/bash

OTP_PATH=../otp
CONC_PATH=../Concuerror

$CONC_PATH/concuerror -pa . -t mochiweb_tests -p 1 --dpor \
    -f src/*.erl test/*.erl -I include --wait-messages -T 2000 \
    --fail-uninstrumented --app-controller --keep-tmp-files \
    --ignore crypto crypto_app erl_prim_loader erl_parse \
             code public_key global \
    -I $OTP_PATH/lib/eunit/include \
    -f $OTP_PATH/lib/eunit/src/*.erl \
    -I $OTP_PATH/lib/kernel/include \
    -f $OTP_PATH/lib/kernel/src/inet*.erl \
       $OTP_PATH/lib/kernel/src/error_logger.erl \
       $OTP_PATH/lib/kernel/src/application*.erl \
       $OTP_PATH/lib/kernel/src/gen_tcp.erl \
       $OTP_PATH/lib/kernel/src/os.erl \
       $OTP_PATH/lib/kernel/src/file.erl \
    -I $OTP_PATH/lib/stdlib/include \
    -f $OTP_PATH/lib/stdlib/src/dict.erl \
       $OTP_PATH/lib/stdlib/src/queue.erl \
       $OTP_PATH/lib/stdlib/src/sets.erl \
       $OTP_PATH/lib/stdlib/src/proplists.erl \
       $OTP_PATH/lib/stdlib/src/lists.erl \
       $OTP_PATH/lib/stdlib/src/io*.erl \
       $OTP_PATH/lib/stdlib/src/unicode.erl \
       $OTP_PATH/lib/stdlib/src/math.erl \
       $OTP_PATH/lib/stdlib/src/erl_scan.erl \
       $OTP_PATH/lib/stdlib/src/string.erl \
       $OTP_PATH/lib/stdlib/src/gb_trees.erl \
       $OTP_PATH/lib/stdlib/src/gen*.erl \
       $OTP_PATH/lib/stdlib/src/proc_lib.erl \
       $OTP_PATH/lib/stdlib/src/supervisor*.erl \
       $OTP_PATH/lib/stdlib/src/timer.erl \
       $OTP_PATH/lib/stdlib/src/calendar.erl \
       $OTP_PATH/lib/stdlib/src/sys.erl \
       $OTP_PATH/lib/stdlib/src/filename.erl \
       $OTP_PATH/lib/stdlib/src/filelib.erl \
       $OTP_PATH/lib/stdlib/src/re.erl \
       $OTP_PATH/lib/stdlib/src/erl_posix_msg.erl \
       $OTP_PATH/lib/stdlib/src/random.erl \
       $OTP_PATH/lib/stdlib/src/epp.erl \
    -f $OTP_PATH/lib/syntax_tools/src/erl_syntax.erl \
       $OTP_PATH/lib/compiler/src/compile.erl \
    -I $OTP_PATH/lib/ssl/include \
    -f $OTP_PATH/lib/ssl/src/*.erl \
    -I $OTP_PATH/lib/xmerl/include \
    -D ''VSN=\''4.9.1\'''' \
    -f $OTP_PATH/lib/xmerl/src/xmerl_ucs.erl \
    -I $OTP_PATH/lib/inets/include \
    -I $OTP_PATH/lib/inets/src/inets_app \
    -I $OTP_PATH/lib/inets/src/http_lib \
    -f $OTP_PATH/lib/inets/src/ftp/*.erl \
    -f $OTP_PATH/lib/inets/src/http_client/*.erl \
    -f $OTP_PATH/lib/inets/src/http_lib/*.erl \
    -f $OTP_PATH/lib/inets/src/http_server/*.erl \
    -f $OTP_PATH/lib/inets/src/inets_app/*.erl \
    -f $OTP_PATH/lib/inets/src/tftp/tftp_sup.erl \
    -I $OTP_PATH/lib/kernel/src \
    -f $OTP_PATH/erts/preloaded/src/prim_inet.erl \
       $OTP_PATH/erts/preloaded/src/init.erl \
       $OTP_PATH/erts/preloaded/src/prim_file.erl
```

Listing 4.12: Deploy Concuerror over mochiweb

Chapter 5

Conclusion and future work

In this thesis we have studied Concuerror, a testing tool for Erlang programs that uses stateless model checking techniques for systematically producing process interleaving sequences of a program, after having instrumented its code. We have described the changes that took place to Concuerror (changes that improve both its functionality and its usability) as well as the reasons for theses changes. We also have experimented with how a real user would use Concuerror in practice as a testing and debugging aid. At first we showed how one can integrate Concuerror with the Eunit testing framework to use existing tests for her analysis. Then we moved onto bigger code bases and successfully analyzed the MochiWeb Erlang library. This was the first time Concuerror was used to debug and analyze a well known Erlang library which depends heavily on OTP.

The development of Concuerror is far from over and the tool has a lot of room for improvements. Our tasks for the future include:

- **Auto instrument libraries**
  Currently, instrumenting the code under test and the modules/libraries it uses can be a tedious task. The user will have to use the `-fail-uninstrumented` command line flag to instruct Concuerror to warn about uninstrumented modules and gradually choose to either instrument or ignore them. We would like to make this task more straight forward by having Concuerror to on-the-fly instrument and load one module if it is needed by our code under test.

- **Full compatibility with EUnit**
  Currently, using the Eunit test as an input for Concuerror's analysis requires a patch to be applied to the Eunit OTP library. This happens because Eunit catches all exceptions in order to report them and then exits gracefully leaving Concuerror to believe that everything went normal. We would like to find new ways to integrate Concuerror with Eunit, ones that will allow the end user to use both systems as is (without modifying them).

- **Fair scheduling**
  As we have seen, the Concuerror scheduler is currently unfair, meaning that it could be running a single process for ever, as long as that process never blocks. Using fair scheduling will avoid these situations, and will additionally enable Concuerror to detect livelocks.

- **Extension for multi-node programs**
  Currently, Concuerror is not able to test programs that extend to more than one node. Handling the case of multi-node programs will allow the testing of distributed systems, which are fairly common in Erlang. Additionally, Concuerror could eventually be extended to drop its closed-world hypothesis and test programs that communicate with the outside world (e.g. ports).

- **UI improvements and visualization**
  The Concuerror's UI can be improved in many ways to become more usable. Some directions are project creation and management, test automation, and visualization of process interaction to further simplify the grasping and debugging of concurrency errors.

Chapter 6

Related work

Model checking techniques have been used for years to verify concurrent and distributed systems. More "traditional" model checkers require describing the system to be verified in a special modeling language. An example of this is the SPIN model checker [32], which verifies models expressed in the Promela language [6]. Automatic code translation to the modeling language has been proposed and used in tools like Bandera [20], which translates Java code into one of three modeling languages. Starting with the Verisoft model checker [27], several others, like Java PathFinder [52], CMC [41] and CHESS [43], have been designed to directly verify code written in the original language.

Of the aforementioned model checkers, SPIN, Java PathFinder and CMC deal with capturing the program state and caching visited states. On the other hand, Verisoft and CHESS use a stateless approach and enumerate process or thread interleaving sequences, much like Concuerror does.

Although Erlang is a concurrency-oriented language, there has not been much effort towards concurrency testing and verification. According to a recent survey [44], Dialyzer [37], and Eunit [13] are the mostly used Erlang testing tools. Eunit provides no means of detecting concurrency errors, while Dialyzer has been recently extended to detect some kinds of data races [16] and message passing errors [17] via static analysis. QuickCheck [10], a property-based testing tool for Erlang, has introduced a user-level scheduler named PULSE [18], which is able to detect some concurrency errors via random process interleaving. Besides the random nature of the testing procedure, which provides no correctness guarantees, the user is required to write down desired properties using a special semi-formal notation, which is by itself not a trivial task and, additionally, excludes the use of existing unit tests.

Verification tools for Erlang programs include Huch's abstract interpretation model checker [33], and the McErlang model checker [24, 25]. McErlang uses a stateful exploration approach and allows the parametrization of the algorithms and structures used inside the tool. However, by default processes are only allowed to be preempted at receive expressions, thus the resulting search is very coarse-gained compared to Concuerror. The introduction of finer-gained preemption points requires the manual placement of commands, which is a strenuous task and, at the same time, alters the original code.

Bibliography

[1] CouchDB. https://couchdb.apache.org/.

[2] ejabberd. http://www.process-one.net/en/ejabberd/.

[3] Erlang/OTP online documentation. http://www.erlang.org/doc/.

[4] Fuzz testing of application reliability. http://pages.cs.wisc.edu/~bart/fuzz/fuzz.html.

[5] SimpleDB. https://aws.amazon.com/simpledb/.

[6] Book review: Design and validation of computer protocols by Gerard J. Holzmann (prentice hall, 1991). *SIGCOMM Comput. Commun. Rev.*, 21(2):14–, Apr. 1991. Reviewer-Fredlund, Lars-Åke.

[7] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] J. Armstrong. A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

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

[10] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM.

[11] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[12] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[13] R. Carlsson and M. Rémond. Eunit: a lightweight unit testing framework for Erlang. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 1–1, New York, NY, USA, 2006. ACM.

[14] F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[15] M. Christakis, A. Gotovos, and K. Sagonas. Systematic testing for detecting concurrency errors in Erlang programs. In *ICST*, pages 154–163. IEEE, 2013.

[16] M. Christakis and K. Sagonas. Static detection of race conditions in Erlang. In *Proceedings of the 12th international conference on Practical Aspects of Declarative Languages*, PADL'10, pages 119–133, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] M. Christakis and K. Sagonas. Detection of asynchronous message passing errors using static analysis. In R. Rocha and J. Launchbury, editors, *PADL*, volume 6539 of *Lecture Notes in Computer Science*, pages 5–18. Springer, 2011.

[18] 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 *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 149–160, New York, NY, USA, 2009. ACM.

[19] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, Nov. 2009.

[20] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 439–448, New York, NY, USA, 2000. ACM.

[21] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[22] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, July 1997.

[23] 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*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[24] L.-Å. Fredlund and C. B. Earle. Model checking Erlang programs: the functional approach. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 11–19, New York, NY, USA, 2006. ACM.

[25] L.-Å. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 125–136, New York, NY, USA, 2007. ACM.

[26] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[27] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[28] A. Gotovos. Dynamic systematic testing of concurrent Erlang programs. Master's thesis, Computer Engineering, National Technical University of Athens, June 2011.

[29] A. Gotovos, M. Christakis, and K. Sagonas. Test-driven development of concurrent programs using Concuerror. In K. Rikitake and E. Stenman, editors, *Erlang Workshop*, pages 51–61. ACM, 2011. https://github.com/mariachris/Concuerror/.

[30] C. Hewitt. Actor model for discretionary, adaptive concurrency. *CoRR*, abs/1008.1459, 2010.

[31] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[32] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[33] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, pages 261–272, New York, NY, USA, 1999. ACM.

[34] IEEE Computer Society. *Software Engineering Body of Knowledge (SWEBOK)*. Angela Burgess, EUA, 2004.

[35] E. Johansson, M. Pettersson, and K. Sagonas. A high performance Erlang system. In *Principles and Practice of Declarative Programming*, pages 32–43, 2000.

[36] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[37] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In W.-N. Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2004.

[38] T. Lindahl and K. Sagonas. Typer: a type annotator of Erlang code. In K. Sagonas and J. Armstrong, editors, *Erlang Workshop*, pages 17–25. ACM, 2005.

[39] H. Mattsson, H. Nilsson, and C. Wikström. Mnesia - a distributed robust dbms for telecommunications applications. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, pages 152–163, London, UK, UK, 1998. Springer-Verlag.

[40] MochiWeb. https://github.com/mochi/mochiweb/.

[41] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, Dec. 2002.

[42] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[43] 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 conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[44] T. Nagy and A. Nagyné Víg. Erlang testing and tools survey. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, pages 21–28, New York, NY, USA, 2008. ACM.

[45] W. Nelson. *Accelerated Testing: Statistical Models, Test Plans, and Data Analysis*. John Wiley & Sons, 1990.

[46] OTP. https://github.com/erlang/otp/.

[47] R. V. Patil and B. George. Tools and techniques to identify concurrency issues. *MSDN Magazine*, June 2008.

[48] K. Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM.

[49] J. Shore and Chromatic. *The Art of Agile Development*. O'Reilly Media, 2008.

[50] A. J. H. Simons. A theory of regression testing for behaviourally compatible object types. *Softw. Test. Verif. Reliab.*, 16(3):133–156, Sept. 2006.

[51] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

[52] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 3–, Washington, DC, USA, 2000. IEEE Computer Society.