Diploma Thesis by Alkis Gotovos
Supervisor: Konstantinos Sagonas
Associate Professor, N.T.U.A.

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Dynamic Systematic Testing of Concurrent Erlang Programs

Athens, June 2011

Diploma Thesis by Alkis Gotovos
Supervisor: Konstantinos Sagonas
Associate Professor, N.T.U.A.

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Dynamic Systematic Testing of Concurrent Erlang Programs

Approved by the three-member examining committee on June 14th, 2011.

.....	.....	.....
Kostas Kontogiannis	Nikolaos Papaspyrou	Konstantinos Sagonas
Associate Professor, N.T.U.A.	Assistant Professor, N.T.U.A.	Associate Professor, N.T.U.A.

Athens, June 2011

Alkis Gotovos
Graduate Electrical and Computer Engineer, N.T.U.A.

Simplicity is the final achievement

—Frédéric Chopin

Abstract

Concurrent programming has become increasingly popular in the last decade; yet, it is notoriously error-prone. Even worse, traditional testing tools are largely inadequate when it comes to checking concurrent code. In this thesis we introduce Concuerror, a testing tool for concurrent Erlang programs, that aims to facilitate the task of detecing and eliminating concurrency-related errors. Concuerror relies on systematically exploring process interleaving to reveal such errors. We describe the infrastructure that was developed to support this task, as well as the techniques that were used to make Concuerror more efficient. We also present an extended example of using Concuerror in practice for test-driven development.

Keywords

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

Acknowledgements

First, I would like to thank my supervisor, Kostis Sagonas, not only for his guidance and support during the preparation of this diploma thesis, but also because he has been a decisive factor in my choosing to major in Computer Science.

I would also like to thank doctoral student Maria Christakis for her help in developing Concuerror, for our long insightful talks regarding this project, and, most importantly, for our friendship that has come out of working together for the past year.

Online resources

The source code of the tool developed as part of this thesis can be found at the online github repository `http://www.github.com/mariachris/Concuerror` and is distributed under the simplified BSD licence.

Contact

I can be contacted at `alkisg@softlab.ntua.gr`—any questions or comments are welcome.

Contents

List of Figures

List of Listings

1 Introduction

"If it's not tested, it's broken."

➢ Says Bruce Eckel, while debating whether "strong testing" can replace "strong typing"; it can very successfully indeed, he concludes [34, pp. 67-77].

Back in the old days, testing was part of the verification and validation stage in a software product's lifecycle, following the "real" coding part of implementing the software's logic. Unfortunately, software engineers and programmers alike have since realized that designing and implementing a (well) working software system is much harder than first thought to be. Partitioning software development into distinct stages and using lots of paperwork didn't seem to do the job. Besides, practice has shown that not only do most defects end up costing more than it would have cost to prevent them, but additionally, the later a defect is found, the more expensive it is to fix. This is sometimes called the Defect Cost Increase (DCI) principle in software development [4, p.98].

The realization of the difficulties in software development and the failure of the existing unflexible development models to rise to the challenge and produce high quality software at a reasonable cost, gave rise to the *Agile Software Development* school. Agile development processes advocate flexibility and promote, among others, practices of short iterations and incremental design [33].

From the perspective of modern methodologies, testing is nowadays thought of as an integral part of software design and implementation, rather than a sterile verification stage in the software development process.

Tests are an effective tool for specifying and communicating software requirements and design elements. Unlike vague prose and abstract figures, tests provide a concrete means for storing and conveying up-to-date project information. That way, tests may also be used as software documentation. Moreover, tests can serve as a practical way to measure and report development progress. Finally, maybe the most important aspect of testing is the resulting effect on the programmer's mentality. Tests offer immediate feedback on programming decisions and boost the programmer's confidence when she is faced with complex programming challenges. The deconstruction of difficult programming problems into an organized procedure of writing tests and making them pass, increases productivity, facilitates the production of high quality code and, as a result, dramatically decreases the time spent on debugging (see Figure 1.1).

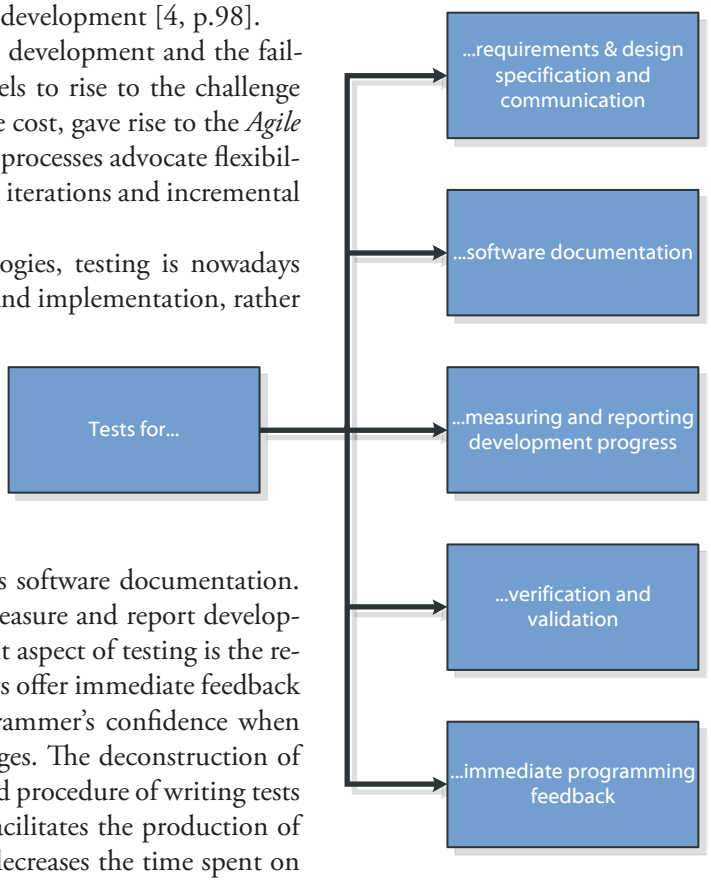


Figure 1.1: The multifold role of testing in modern software development

The importance of testing is especially stressed by supporters of *Extreme*

Programming (XP), a combination of agile principles, values and practices, originally proposed by Kent Beck [4]. In particular, the practice of *Test-Driven Development* (TDD) suggests that tests should be the driving force in software creation.

Test-Driven Development

Test-Driven Development (TDD) [3] is a software development practice that suggests writing tests before writing code. TDD is incorporated into the main practices of Extreme Programming (XP), but—like most XP practices—can also be used on its own. According to the father of TDD, Kent Beck, the goal is *clean code that works*. To that end, he suggests a coding workflow of small iterations, each of them consisting of three steps:

- Write a small test that does not work.
- Make the test work as quickly as possible.
- Refactor to clean up and remove duplication.

Using TDD the programmer decomposes hard problems into small steps and, that way, the development progress can be monitored test by test. Reportedly, this results in reduced anxiety due to problem complexity and higher quality code.

Oh, the horror!

Writing tests for concurrent programs presents a special challenge, namely that of taking into account the effects of usually complex interactions between various processes or threads that participate in a program's execution. As a consequence, testing and debugging in concurrent programming environments is notoriously difficult. Most of us have heard of (or, if lucky enough, been faced with) horror stories of subtle concurrency bugs, also known as *Heisenbugs* in the programming jargon, nigh impossible to figure out, while eerily vanishing from sight when printing calls or debuggers are brought into the hunt.

Concurrency is mentioned as one of the topics that TDD cannot handle, because "[s]ubtle concurrency problems can't be reliably duplicated by running the code" [3, p.xii]. Even worse, using traditional testing techniques for concurrent programs can provide the programmer with a false sense of security. While all tests are passing when run under "mild" conditions, concurrency bugs might be lurking under the hood. Usually, these bugs are caused by intricate process or thread interleaving scenarios and will manifest themselves under random circumstances—more so when the system is put under heavy load, i.e. conditions where a lot of programs and processes are being run in parallel and a high percentage of system resources are being used. Furthermore, it can be a long time after the system's development is completed before such a bug surfaces in practice, in which case an extraordinary amount of cost and effort has to be put into finding and fixing the problem (recall the DCI principle).

Is concurrency destined to bring chaos and despair upon the unfortunate programmer who attempts to harness its power? Maybe! Nonetheless, there have been proposed a number of techniques that aim to alleviate the seemingly grave situation.

- Instead of waiting for the system to break down under heavy load in production usage, why not simulate these conditions to test the system under development? This is the purpose of *stress testing*. Stress tests can be devised to simulate real usage conditions or even extreme scenarios that would rarely arise in practice. However, the random nature of stress testing limits its ability to provide guaranties about the system's behavior.

- In another direction, *static analysis* [7, 13] can be used to detect concurrency related problems. Static analysis tools use compile-time information to reveal bugs without having to execute the program. They are typically simple to use and provide easily understandable results that are directly related to the program's source code. Unfortunately, information at compile time is more often than not insufficient to discover subtle concurrency bugs, especially so in dynamically typed languages. At the same time, aggressive analyses suffer from a high percentage of confusing false positives.

- *Model checking* [10] is a third way to go. It is usually based on creating a formal model of the system and subsequently using search algorithms to explore the model's states. 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. In addition, properties verified by model checking tools apply to the system model and not the original system itself. The worst part, though, is that model checking suffers from the so called *combinatorial explosion* problem, which leads to very slow analyses and large memory consumption.

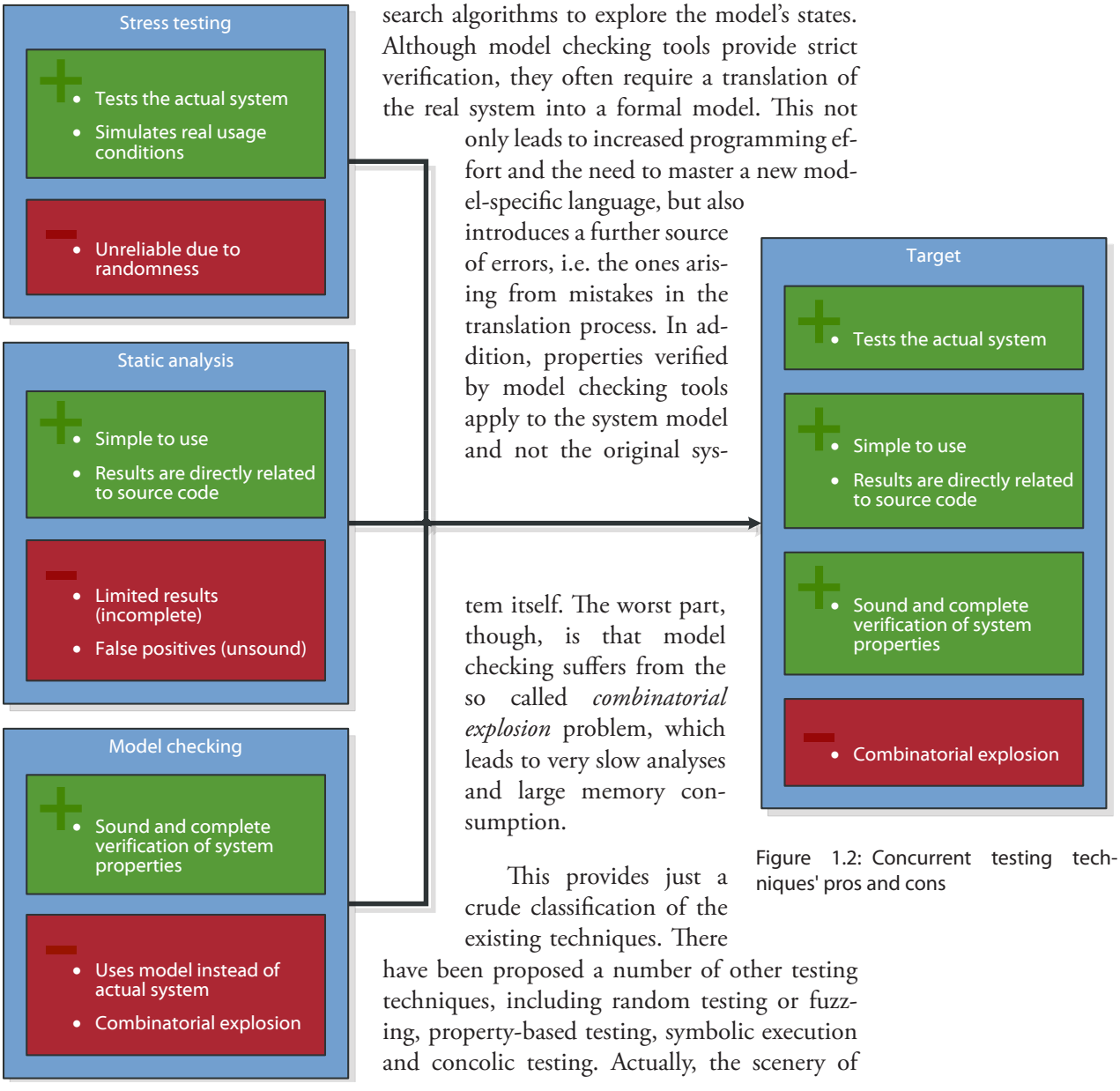


Figure 1.2: Concurrent testing techniques' pros and cons

This provides just a crude classification of the existing techniques. There have been proposed a number of other testing techniques, including random testing or fuzzing, property-based testing, symbolic execution and concolic testing. Actually, the scenery of

concurrent testing and debugging is quite complex. Several different variations and combinations of algorithms, techniques and tools have been used over the years depending on the language, operating system and hardware of choice, as well as the exact problem at hand.

Introducing Concuerror

➢ Concuerror is pronounced /ˈkɒŋkərər/, like conqueror.

This thesis is about the design and development of Concuerror, a testing tool for concurrent *Erlang* programs. Erlang is a general purpose, functional, dynamically typed, concurrency oriented language [1, 6]. Concuerror offers reliable testing for concurrent Erlang programs, promotes the use of TDD in concurrent programming environments and is intended to assist Erlang programmers in writing high quality concurrent software. Concuerror is designed to be user-friendly and automated, like static analysis tools, use real tests written for the original system, like stress testing tools, and provide sound and complete verification, like model checking tools (see Figure 1.2). Of the three aforementioned categories, Concuerror is more closely related to model checking, because it makes use of model checking techniques to systematically explore process interleaving in concurrent programs. Nonetheless, testing in Concuerror is performed on the actual software system using existing tests, rather than on an artificial system model.

We drew our inspiration from CHESS [30], a systematic testing tool for concurrent software developed by Microsoft Research. CHESS aims at systematically generating all interleaving sequences of a given test and is able to consistently reproduce an erroneous execution. It is capable of testing multithreaded programs written in Win32, .NET or Singularity.

Although in some aspects our endeavour is very similar to CHESS, we will see in the following chapters that Erlang's process-based, no-shared-state concurrency model is fundamentally different from the multithreaded shared-memory model of the Win32 and .NET platforms.

Contributions

The contributions of this thesis are summarized as follows:

- **Infrastracture**

  A custom instrumentation and scheduling mechanism that allows to control the interleaving of processes during the execution of an Erlang program.

- **Algorithms**

  The use of iterative context bounding and the incorporation of a custom blocking avoidance scheme.

- **Tool**

  An integrated testing tool that provides an interactive and easy-to-use environment for testing and debugging Erlang programs.

What's next?

Chapter 2 presents a brief introduction to Erlang, particularly its concurrency related aspects. Readers familiar with Erlang can skip to Chapter 3, where we discuss the high-level goals of our project and present an overview of Concuerror's components.

2 Erlang: A Short Tour

Following the establishment of multi-processor computing systems, the newest trends of distributed programming and cloud computing has software developers shifting their attention towards languages that support and facilitate concurrent and distributed system development. Among others, languages like Scala, Clojure and Go, have become increasingly popular in recent years.

Erlang is one of the oldest concurrency-oriented languages, developed in 1986 by Joe Armstrong and Ericsson, and released as open-source in 1998. It was specifically designed to support the implementation of fault-tolerant distributed software systems. Erlang was originally used in telephony applications and more recently has been used in backends of distributed applications, like the CouchDB database, the ejabberd XMPP server and the SimpleDB web service by Amazon. As of June 2011, the Erlang/OTP implementation is the one almost exclusively used by Erlang developers.

➢ The most recent Erlang/OTP release is R14B02 (March 16th, 2011). More information can be found on the official site of Erlang/OTP at www.erlang.org.

The following sections 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 [1, 6] and the official Erlang/OTP online documentation.

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, almost everything is an expression and single assignment is used for 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, the latter being similar to enumerations in C-like languages. Atoms are represented by alphanumeric sequences starting with a lowercase letter, while Erlang variables always begin with an uppercase letter or an underscore. Other than that, Erlang 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 source code is mainly organized into modules and functions. Only exported functions are visible outside a 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(...)`. A module that exports a function for sorting a list is shown in Listing 2.1. In this case, only function `mergesort/1` is visible outside the module and has to be called as `ms:mergesort(...)`. Some remarks on the code:

➢ Erlang functions are commonly written as their name, followed by a slash and their arity. This is done to distinguish between foo/1 and foo/2, which are seperate funtions.

- **Lists and tuples**

  Lists are represented by comma seperated 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 seperated expressions put inside curly brackets, i.e. `{Elem1, Elem2, ...}`.

- **Guards**

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

- **Catchall**

  A catchall pattern is represented by an underscore. Alternatively, variables with an underscore prefix can be used as catchalls. These variables have exactly the same functionality as normal variables, but no "unused variable" warnings are emitted for them by the Erlang compiler.

From the code in Listing 2.1 it is evident that 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 static program analysis to detect errors [25, 26].

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 [23].

Listing 2.1: Mergesort in Erlang

```
-module(ms).

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

Concurrency

The feature that makes Erlang special is its powerful concurrency mechanism, 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 completely managed by the Erlang VM and are not mapped to operating system processes or threads.

Erlang processes have (almost) no shared state and can be created by the thousands or even millions. The language follows the *actor model*, which means that inter-process communication is done via message passing. In a nutshell, every process may create—or *spawn*—new processes, asynchronously send messages to other process, and receive messages from them.

Every process executes its code sequentially and uses a queue—or *mailbox*—to store incoming messages. 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 fails, processes that are linked to it are also terminated.

Messages are sent using the send (`!`) operator and are received using `receive` expressions. The latter are used to pattern match messages in the process mailbox and follow a program path depending on the match. If no matching message is available, `receive` expressions block until such a message arrives. Erlang provides several built-in functions (BIFs) for basic operations, including concurrency primitives, like spawning a new process (`spawn/1`), registering it under a name (`register/2`) or linking it to another process (`link/1`).

Listing 2.2 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 neighbours, and a message (`Token`) is trasmitted 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 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 this 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.

- **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 2 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

➢ In Erlang, closures are also referred to as funs and are written as `fun(Args) -> Body end`.

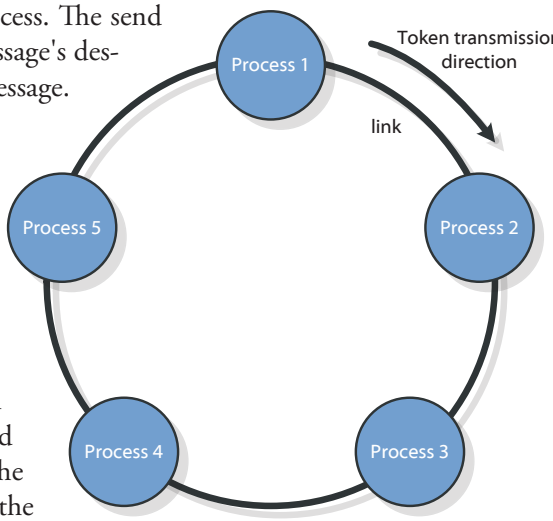


Figure 2.1: The ring of linked processes created by the program of Listing 2.2.

Listing 2.2: A simple concurrent Erlang program

```
-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.
```

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}`.

- **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.

➢ Both lists and io are part of the Erlang/OTP standard library (stdlib).

- **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.

➢ Records are data structures with a fixed number of fields that are accessed by name.

- **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.

Apart from links Erlang also provides *monitors*. Monitors are similar to links, but are not symmetrical, which means that a monitored process does not known 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 do not share memory, execute their internal actions sequentially, and communicate with each other via message passing.

3 Concuerror Overview

What's the idea?

Even in Erlang with its concurrency-made-easy mindset, writing programs with multi-process interactions is extremely error prone. On top of this, traditional testing is usually not able to expose concurrency related bugs.

Take a look at the code in Listing 3.1. 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 result back to the first process. A common error among novice Erlang programmers lingers among these few lines of code.

Listing 3.1: A simple two process example with a bug

```
foo() ->
  Self = self(),
  Pid = spawn(fun() -> bar(Self, 42, 5) end),
  register(math, Pid),
  receive
    Result -> Result
  end.

bar(Target, X, Y) ->
  Target ! {result, X + Y}.
```

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 tests. We can try repeatedly running `foo/0`, but it is very unlikely that the above exception will occur. Because the processes are scheduled in parallel, `foo`'s `register/2` almost always precedes `bar`'s termination. The problem will most likely occur randomly after many hours of running the program under stress (hopefully as a result of stress testing, not production use).

Of the many ways the two processes can be interleaved, only the one described above leads to an error (see Figure 3.1). If we had a way to systematically run the program in every possible process interleaving, we would be able to detect the error with absolute certainty, without the need for hours of stress testing. The analysis done by Concuerror is based upon this simple idea.

Erlang provides several "combination-functions" (`spawn_link`, `spawn_monitor`) to

avoid problems similar to the above—no `spawn_register` function is available though. In addition, the Erlang/OTP distribution comes with higher level generic libraries that aim to reduce the need for writing low level concurrent code from scratch and the risk of doing so.

➢ in Erlang/OTP these libraries are called behaviors and provide abstract implementations of common concurrency patterns, like server-client (gen_server), finite state machine (gen_fsm) and event handler (gen_event).

Even so, as most languages out there, Erlang is rarely used "the optimal way" or "as recommended by experts". Moreover, even well structured code often contains process interactions which become overly complex to follow or reason about, even for moderate size codebases. In such cases, a tool capable of systematically deconstructing process interactions can be invaluable, both as a verification mechanism (make sure nothing is wrong) and as a debugging aid (figure out what is wrong).

Along with exceptions, like the one in the previous example, Concuerror can also detect assertion violations and deadlocks. Assertions are commonly used in testing as a means of comparing expected values of expressions to actual ones. From Concuerror's point of view an assertion violation is essentially a user-defined exception that is raised when an assertion fails. Deadlocks occur when all participating processes of a program are blocked. As we will see in the next chapter, the detection of deadlocks in Concuerror is pretty much straightforward.

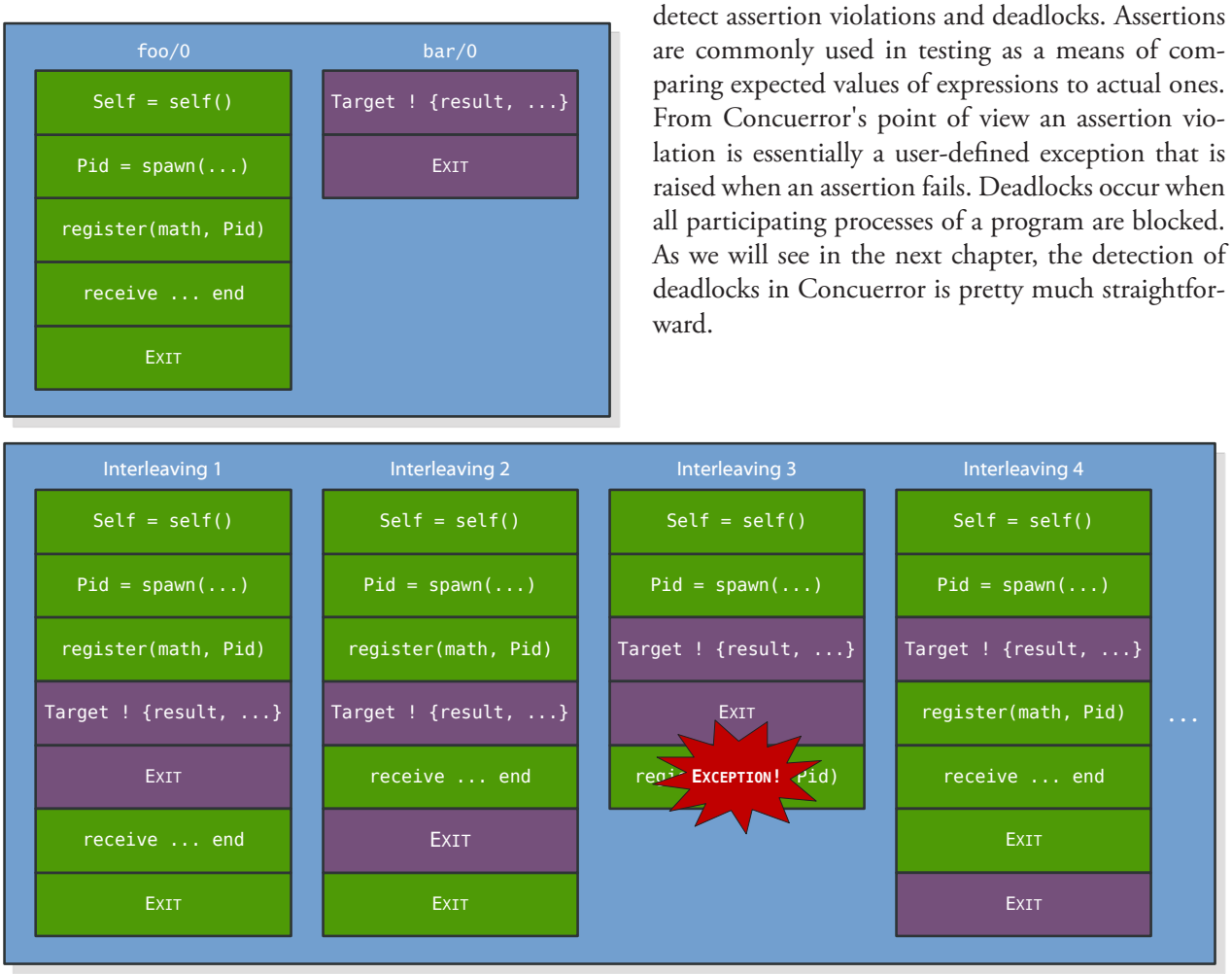


Figure 3.1: Only one way of interleaving the processes of Listing 3.1 leads to an error

Goals

The ability of a software tool to assist the developer in his task is of prime importance and should be the main factor in determining the design guidelines. Some high-level properties that would be desirable for Concuerror from a user's perspective are presented below.

Soundness

Concuerror should be able to detect every error that is possible to occur in practice during the execution of a program. To this end, Concuerror should be able to produce every possible non-redundant interleaving sequence of a program. It is not our intention to formally prove the soundness of our analysis. However, informally speaking, Concuerror should produce as many interleaving sequences as possible.

No false positives

Concuerror should produce no false positives, i.e. reported problems that do not actually exist. Given that every reported problem by Concuerror corresponds to a specific process interleaving, this property should be easy to satisfy.

User code perturbation

Users should be able to analyze their code as is and use existing tests with minimal modifications—ideally none.

Interleaving logging and replay

Erroneous interleaving sequences should be presented in detail, so as to give enough information to the user about the process interaction that caused the problem. Furthermore, the user should be able to replay an erroneous interleaving sequence, that is, execute the program with exactly the same process interleaving as the one that caused the error.

Selective instrumentation

The user should be able to choose the portion of the source code that is going to be analyzed. This enables a layered approach to software verification.

Reasonable time and resources

To be useful in practice, a testing tool has to be quick and lightweight. Time and memory consumption of Concuerror largely depend on the complexity of the program under test. That said, Concuerror should have minimal overhead in executing each individual interleaving sequence and, additionally, avoid producing redundant sequences.

The following sections present a high-level description of Concuerror's components and their functionality. We will delve into more structural and algorithmic detail in subsequent chapters.

Scheduler

Recall that our ultimate goal is to execute every possible interleaving sequence of a multiprocess program and detect sequences that lead to runtime errors. As a prerequisite, we need to have control over the order in which processes are interleaved. Under the scheduler of the Erlang VM, process interleaving is pretty much random.

To be able to force a desired interleaving sequence, we have created a component

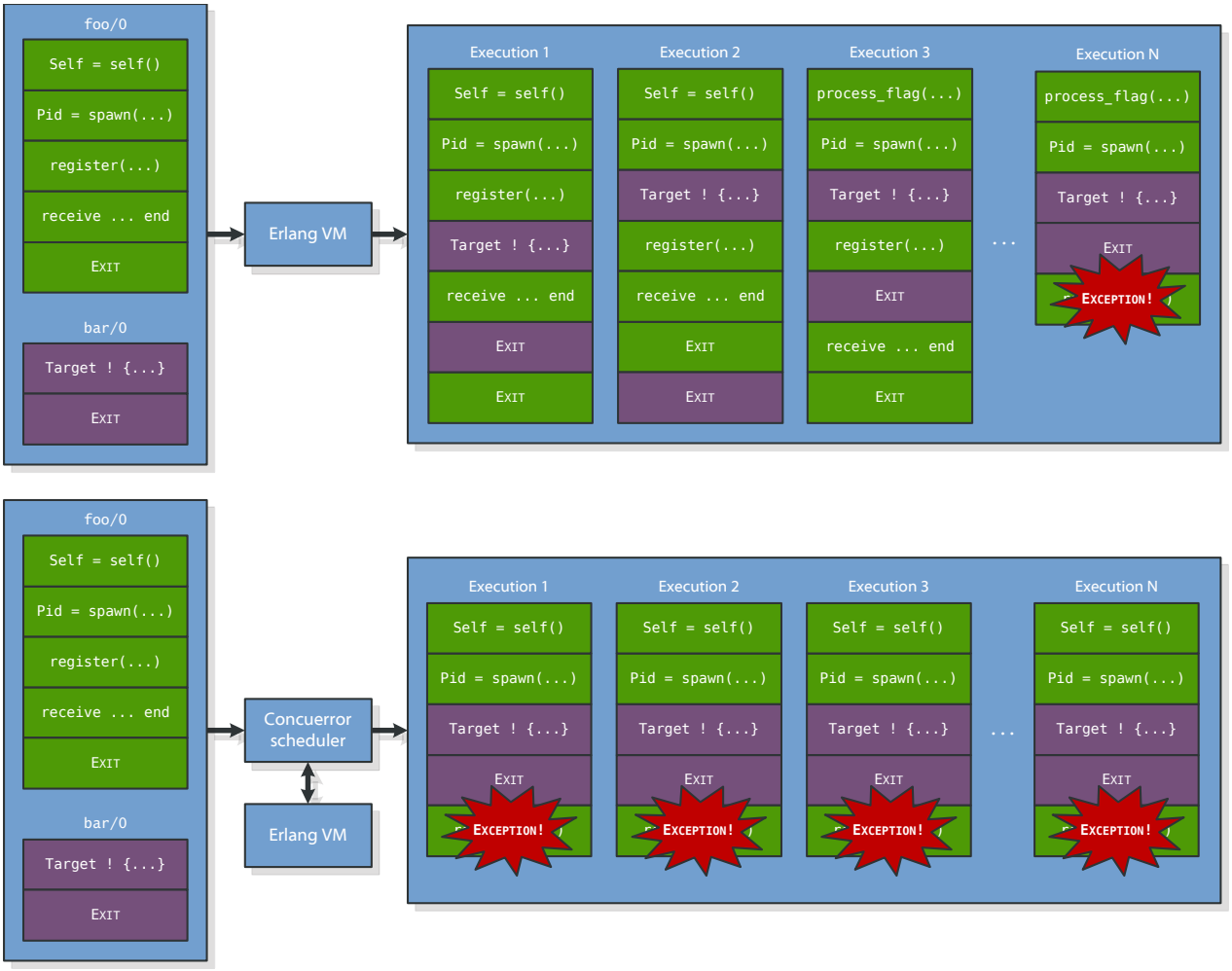


Figure 3.2: Erlang VM scheduler vs Concuerror scheduler in a single core system

that takes care of controlling the order in which the commands of the various processes are executed. We call this component the Concuerror *scheduler*.

Imagine running the example of Listing 3.1 on a single core system. When repeatedly run on the Erlang VM, the program produces various random interleaving sequences. If we are lucky enough, the erroneous sequence will eventually come up. The Concuerror scheduler, however, can replay any desired sequence by carefully controlling the interleaving of the participating processes (see Figure 3.2).

The situation is similar when running the program on a multicore system, because for every possible multicore program execution there exists an equivalent sequential process interleaving sequence. The Concuerror scheduler always runs one process at a time, thus simulating the equivalent sequential interleaving sequence of an execution.

By controlling process execution, the Concuerror scheduler is able to force a specific interleaving sequence. To additionally create *every* possible sequence, the scheduler has to utilize a search algorithm. We will see in Chapter 4 that using a classic algorithm, like depth-first search, does not suffice. Instead, we will have to employ more sophisticated search strategies along with some heuristics to achieve the desired results.

➢ The equivalence of multicore executions and sequential process interleaving sequences is meant with respect to the "observable behavior" of a program and is only valid as long as the program under test contains no data races. For the time being, Concuerror's functionality is limited to analyzing single-node programs in order to avoid some races due to Erlang's fairly complex distributed semantics.

Instrumenter

In our previous presentation of the Concuerror scheduler, we did not discuss at which

point of a program's execution the scheduler is allowed to pause the running process and switch to another process. Such points in the program flow are called *preemption points* and the action of pausing one process and running another is called a *preemption*.

A naive approach would allow a preemption at each program statement. However, such an approach would produce a huge number of redundant sequences. There is no point in interleaving commands with no side-effects, because they only affect the process that executes them. Therefore, we place preemption points only at commands that in some way interact with the outside world with respect to the running process. That way, the source code is effectively divided into chunks, with each of them containing only one command with side-effects. Erlang's shared-nothing actor model provides a big advantage at this point compared to languages with shared memory models. In Erlang, only a limited number of side-effecting expressions and function calls need to be considered as preemption points, whereas in a language like C++, every access point of every global variable would need to be taken into account.

➢ Even to determine if a variable is global would require a sharing and alias analysis of the source code. No such analysis is needed in Concuerror.

Combinatorial Explosion

Suppose that we have a program of 3 processes, each of them executing 10 commands. If we place preemption points at each command of each process, there is a total of 5.6×10^{12} interleaving sequences—a huge number for such a small program! However, let us assume that only 3 commands of each process have side-effects. In that case, there are only 1680 interleaving sequences to be explored. In fact, for n processes of k commands, the number of sequences is greater than $(n!)^k$, which is exponential in both n and k [27].

This problem arises in many algorithms that compute combinations of entities and is commonly called *combinatorial explosion*. Concurrent verification tools and model checkers suffer from combinatorial explosion, because they are required to compute interleaving sequences, i.e. combinations of computational steps, much like Concuerror does.

In this chapter we have presented the simplest method for mitigating the combinatorial explosion problem, namely inserting preemption points only at side-effecting commands. In later chapters we will see more involved methods for reducing the number of interleaving sequences produced.

What constitutes the placement of a preemption point? When the currently running process reaches a preemption point, it should pause its execution, inform the Concuerror scheduler about the event and wait for a prompt to continue. There are two approaches for achieving this effect (see Figure 3.3).

The first approach involves intercepting user code function calls at runtime and redirecting them to custom wrapper functions, which implement the above functionality. CHESS uses this approach [30]. In Erlang, send and receive operations are syntactic constructs, so intercepting function calls is not enough. Furthermore, there seems to be no way to intercept function calls and send/receive operations without messing with the Erlang VM, which we would like to avoid in favor of simplicity.

The second approach involves using a custom parse transformer on the user code, in order to create an instrumented version that contains additional code for pausing/resuming the processes and communicating with the scheduler. We have adopted this approach in Concuerror and the component that implements this functionality is called

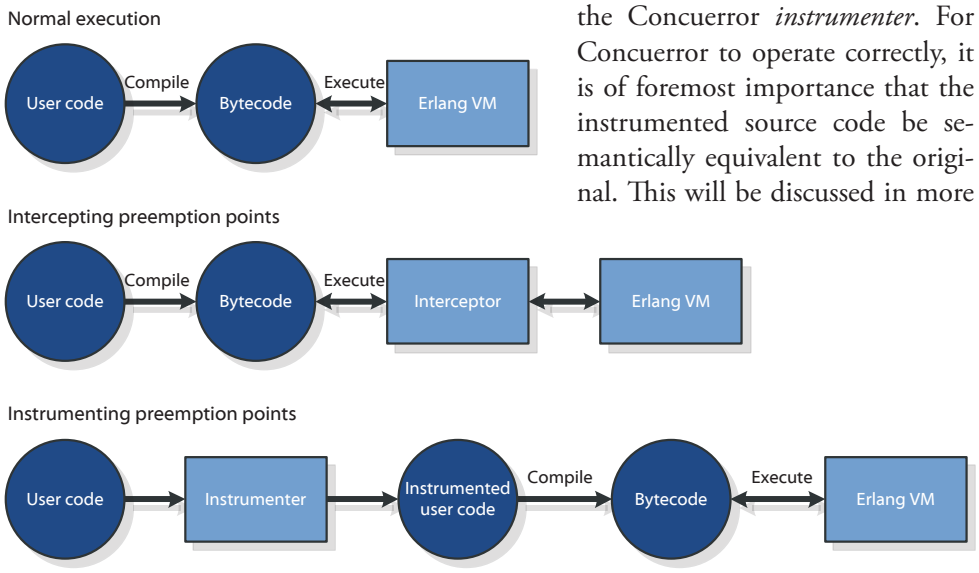


Figure 3.3: Two approaches for inserting preemption points

the Concuerror *instrumenter*. For Concuerror to operate correctly, it is of foremost importance that the instrumented source code be semantically equivalent to the original. This will be discussed in more detail in Chapter 5.

User Interface

Concuerror users will probably have to deal with complex concurrency bugs. In order to understand where the problem lies, let alone fix it, the user needs detailed information about process interaction. A command line interface is not able to convey such information in a nice and usable format. This is why we designed a graphical interface, which we call the Concuerror GUI.

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. Moreover, the user may choose to replay some of the erroneous sequences and acquire detailed, action by action, interleaving information. That way the GUI assists the user in visualizing the program flow and spotting where exactly the problem occurs.

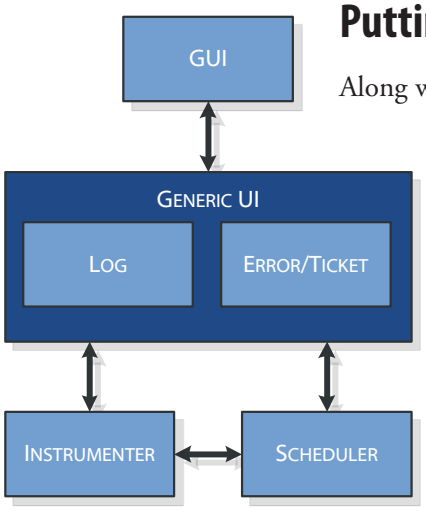


Figure 3.4: High-level Concuerror architecture

Putting everything together

Along with the major components described in the previous sections, Concuerror uses additional components for complementary tasks, like logging (the *Log* component) or error reporting and replay (the *Error* and *Ticket* components). A high-level diagram of the major Concuerror components and their interaction can be seen in Figure 3.4.

In a nutshell, the workflow when using Concuerror in practice is shown in Figure 3.5 and can be described as follows: The user opens the GUI, imports some Erlang modules and picks a test to run. As a first step, the instrumenter applies a parse transformation to the imported modules and then compiles the

transformed code. Subsequently, the scheduler executes all possible interleaving sequences of the test using the transformed modules and reports any errors encountered. The user can then choose to replay an erroneous execution, which means that the scheduler runs the corresponding interleaving sequence and records detailed information about the processes' actions, which is displayed in the GUI. Using this information, the user can apply code changes and replay the erroneous interleaving to see how the program execution is affected. This procedure can be repeated as needed.

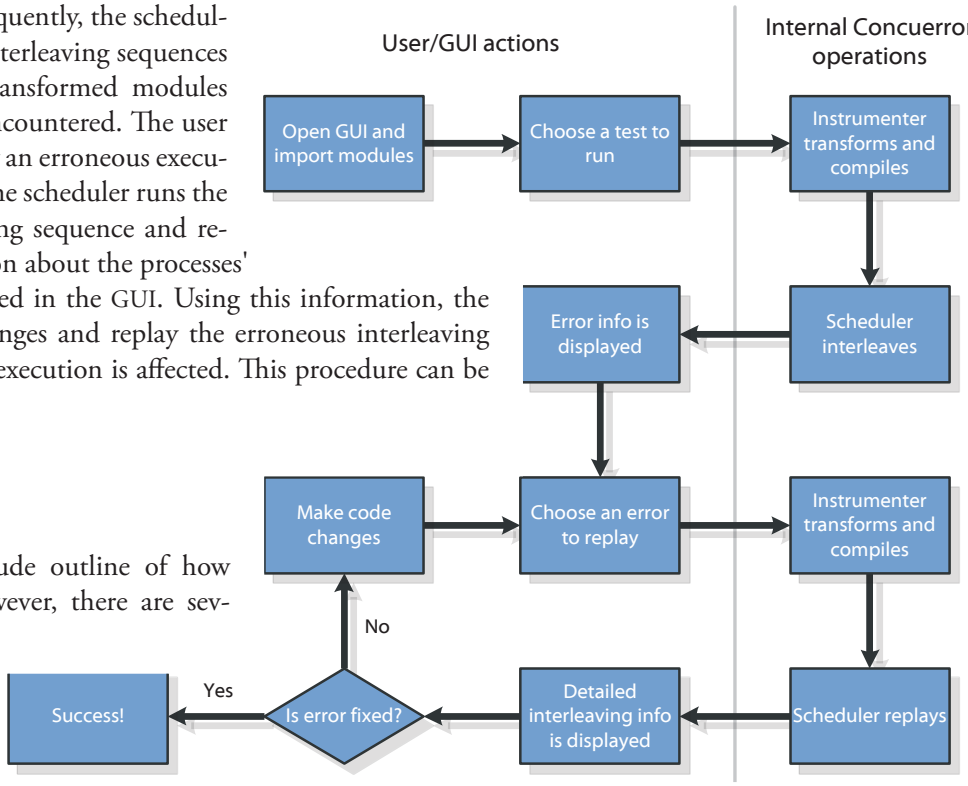


Figure 3.5: Typical workflow when using Concuerror

What's next?

We have seen a crude outline of how Concuerror works. However, there are several features that have not been explained in detail. Chapter 4 provides an in-depth discussion of algorithms and techniques used in the scheduler, while Chapter 5 presents the rather tedious job performed by the instrumenter.

4 The 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. It's like being a policeman in a 4-way junction trying to control rush hour traffic, while at the same time looking for the bank robbers amongst the cars. This is not an easy job for the scheduler—any policeman will tell you so.

State or no state?

The state of a concurrent program consists of at least a program counter for each process or thread and the memory contents that can be accessed by any process or thread. Concuerror should be able to visit every possible program state and report any state that is erroneous according to some user-defined criteria.

Caching visited program states during state-space exploration can provide a huge efficiency benefit. On the other hand, it is extremely difficult to capture the program state of a program written in a modern programming language. This is why Concuerror was designed to be *stateless*, i.e. no detailed information about program state is being held.

Still, to be able to search, that is, reach all states, and replay, that is, reach a specific state again and again, we need a representation of program states. For that purpose, Concuerror uses a trace of the interleaving sequence from the initial state to another state. The mapping of traces to program states is "onto", but not "one-to-one". This is due to the fact that every trace represents a unique program state, namely the one that the program reaches when the processes are interleaved as shown in the trace, but more than one traces can lead to the same program state.

We have seen in Chapter 2 that Erlang uses process identifiers (PIDs) to uniquely distinguish between processes. Unfortunately, the PID assigned to a process is not the same between executions of the same program. This means that a trace of PIDs cannot be used as a state representation. Concuerror uses another way for identifying processes according to their hierarchical place in the program execution. Each process is assigned a *logical identifier* (LID), which we will denote by the letter "P", followed by a sequence of numbers separated by periods. 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 4.1

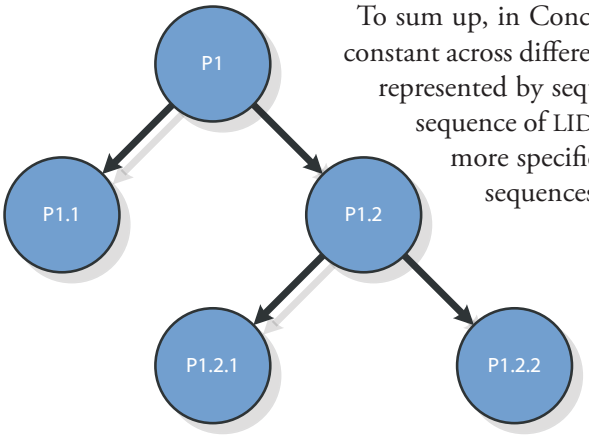


Figure 4.1: Process LID tree

To sum up, in Concuerror processes are represented by unique LIDs, which are constant across different executions of the same program, while program states are represented by sequences of LIDs. For the rest of this thesis, we will refer to a sequence of LIDs as a *state*, although, as mentioned above, a LID sequence is more specific than a program state in the sense that more than one LID sequences can lead to the same program state.

Context and handlers

The scheduler needs to keep track of some process related information during each execution. We call this information the scheduler *context*. The context contains a set of paused processes that are ready to be scheduled, called the *active set*, a set of processes that are suspended (usually due to a `receive`), called the *blocked set*, the currently running process and the current state of the program, as well as other less important information.

Instrumented user code delegates actions with side effects to the scheduler. There are two stages in handling such an instrumented action: At the first stage, the user process enters a wrapper function, where it typically executes a modified version of the original call, informs the scheduler about this action and pauses its execution. This functionality is implemented by the *wrapper* subcomponent. At the second stage, the scheduler logs the action and updates the context accordingly. This is accomplished by the scheduler's *action handler* subcomponent.

➢ The wrapper is discussed in Chapter 5, because it is closely related to the instrumenter.

For example, an instrumented `spawn/1` call executes a wrapper function which takes care of spawning the new process and pausing it right before it starts executing. Furthermore, the wrapper function reports the action to the scheduler and also pauses the execution of the user process that called `spawn/1`. As soon as the action handler is informed, it creates a LID for the newly spawned process, adds the process to the active set and logs the spawn action.

In short, the wrapper works in conjunction with the instrumenter and handles the user side of actions with side-effects, while the action handler takes care of the scheduler side of these actions.

Getting to the core

The core components of the scheduler are the ones that perform the actual search in state-space and control the process execution order. The part of the scheduler that keeps track of the states to be visited and records error information is called the *interleaver*. The part that controls the execution of each individual interleaving sequence, during both search and replay, is called the *driver*. The combined operation of these components determines the search strategy of the scheduler.

Figure 4.2 shows the scheduler, including its subcomponents, and its interaction with some of the other Concuerror components. Notice that the wrapper acts as a kind of middleware between the scheduler and the instrumenter.

Let us see how a simple depth-first search would work. 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. As long as there is only one active process, there is no decision to be made and the state at each point is just a sequence of P1s. However, the situation changes when a spawn call is encountered.

After the spawn call has been handled, there are two processes in the active set (P1 and P1.1), therefore the driver has to make a choice. 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. We call such an interleaving sequence, which does not represent a completed program execution but is a prefix of unexplored states, a *partial state*. The execution continues the same way, saving at each step all partial states resulting from choices not taken. When the current execution has finished, i.e. all processes have terminated normally or an error has occured, the interleaver takes control.

The interleaver records the result of the finished execution and initiates the next one. Depth-first search requires that the last partial state saved in the previous run be replayed up to the last process in the state. This is equivalent to using a stack for storing partial states. The search must then continue from this point on. After the driver has replayed and finished the rest of the execution of the partial state, the search goes on in the same manner until there are no more partial states to be explored.

When replaying a (partial) state, we say that the driver is in *replay mode*, otherwise we say it is in *search mode*. The only difference when being in replay mode is that at each step the process to be executed next is predetermined, that is, no choice has to be made by the driver.

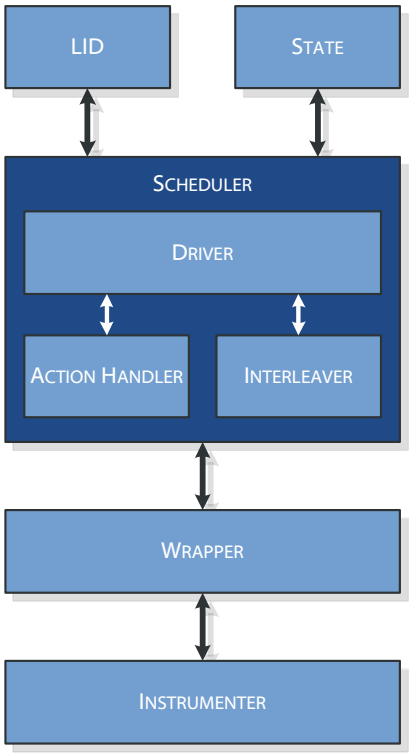


Figure 4.2: Scheduler subcomponent structure

If you have carefully followed the above algorithm, at least one question should have arisen by now. Take a look at Listing 4.1 and let us make some Q&A remarks.

Listing 4.1: Can you spot the potential problem here?

```
foo() ->
  spawn(fun baz/1),
  bar().

bar() ->
  %% do stuff
  bar().

baz() ->
  halt().
```

Q: In what order are partial states inserted into the stack at each step?

A: The order is actually not important here, except for a small detail. In the previous description of the algorithm we inserted states by inverse lexicographic ordering of their last process' LID. The small detail is that any chosen order suffices as long as it produces all interleaving sequences, or equivalently, *as long as the algorithm terminates*. In Listing 4.1 process P1 loops and process P1.1 is supposed to halt the Erlang system. However, when inserting partial states in the way we did above, the first interleaving sequence will have infinite length, because the execution of P1.1 will always be left for a next execution. In general, programs containing potential *livelocks* are vulnerable to

non-termination when using DFS. Using round-robin scheduling solves this problem, but is not compatible with a technique that will be presented later in this chapter. Therefore, we will assume for now that an execution is simply aborted if it exceeds a user-defined time limit.

Q: How are receive expressions handled?

A: After any blocking call, including `receive` expressions, the process has to be moved from the active to the blocked set. Furthermore, it should remain there until it is able to continue its execution. In the case of a `receive` expression, a process can continue as soon as a matching message is available in its mailbox. However, one may notice that the action of blocking has no effect on the program's execution, thus interleaving sequences that contain process blocks are actually redundant. In the next section we present a technique for reducing redundancy due to blocking processes.

Q: What happens when a process exits?

A: Obviously the process has to be removed from the active set. However, there is also a tricky part. We saw that when a process exits in Erlang, it can trigger messages to be sent to other processes that are linked to it. Therefore, process exits are in fact actions with side-effects and should be handled in the same way as any other action of that kind. Fortunately enough, as we will see in Chapter 5, the way that the instrumenter inserts preemption points ensures that process exits are handled correctly.

Figure 4.3 shows step by step the analysis of the program in Listing 4.2 using the simple DFS algorithm described above. The aforementioned inverse lexicographic order is used for inserting partial states into the stack. Notice the redundant sequences caused by the `receive` expression in process P1 and the handling of process exits as separate side-effecting actions.

Listing 4.2: Another simple two process example

```
foo() ->
  register(foo, self()),
  spawn(fun bar/0),
  receive
    bar -> ok
  end.

bar() ->
  foo ! bar,
  ok.
```

Avoiding blocks

We previously noticed that processes blocking on `receive` expressions produce redundant interleaving sequences. In the example of Figure 4.3 six sequences were produced, but only three of them are actually needed. The three sequences with a P1-P1-P1 prefix, i.e. the sequences where process P1 blocks on a `receive`, are one-to-one semantically

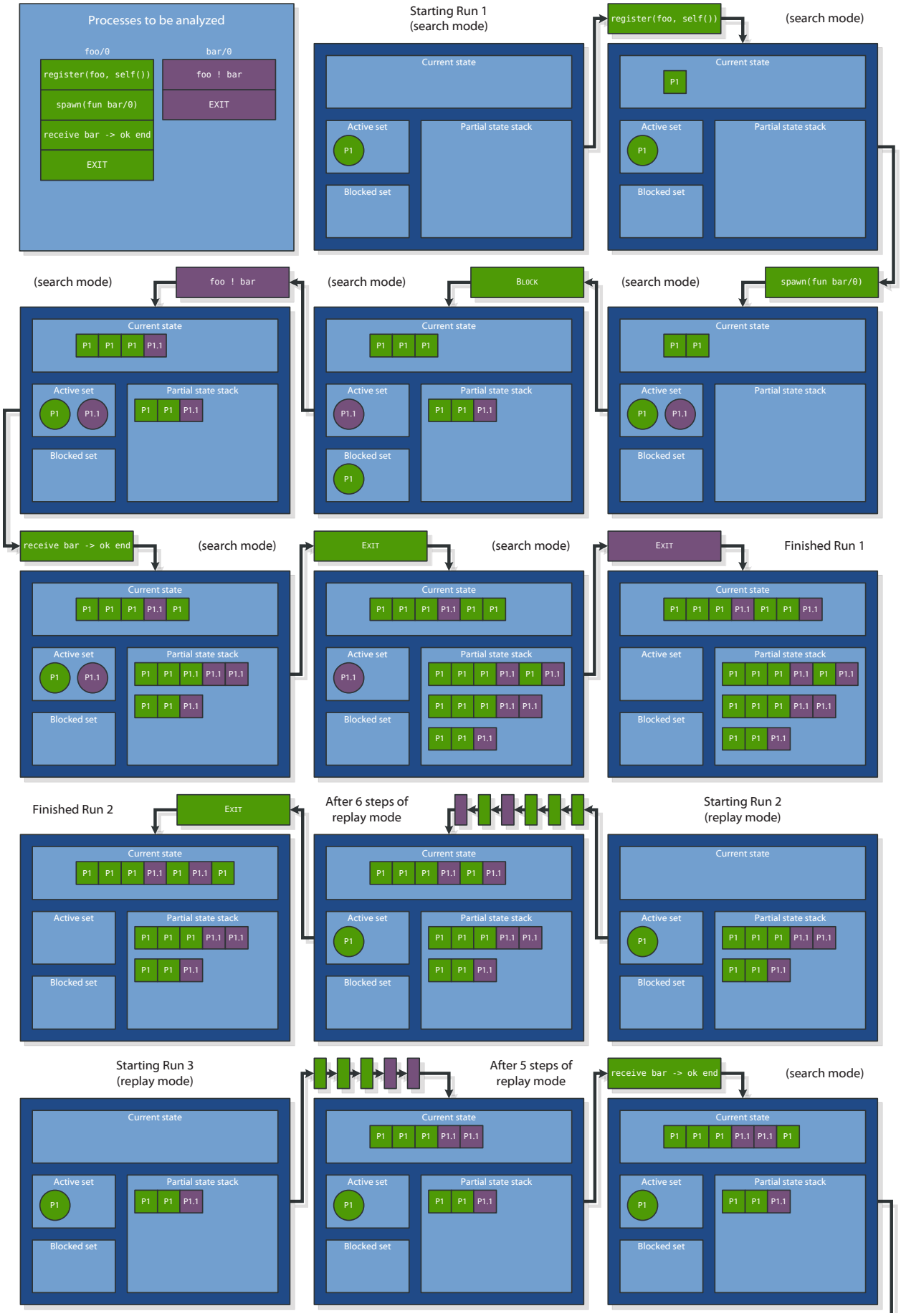


Figure 4.3: Testing the example of Listing 4.2 using a simple depth-first search (cont'd on the next page)

Figure 4.3 (cont'd)

equivalent to the three sequences without blocks.

In fact, any interleaving sequence containing a blocking action on a `receive` is redundant, because it is equivalent to the sequence obtained by removing the blocking action. The production of redundant sequences is both time and memory consuming, as well as confusing for the potential user. We would like Concuerror to report each error once and only once.

To deal with this problem the scheduler uses a technique we call *blocking avoidance*. The main idea is simple: Whenever a blocking action is encountered, the scheduler moves the current process to the blocked set, ignores the action itself and chooses another process from the active set to run. Thereafter, the execution continues normally. There are two points that require some special attention:

➢ Blocking actions can be safely ignored, because they are without side-effects.

- **Instrumentation support**

  As in the case of the simple DFS, upon the arrival of a matching message for a blocked process, the latter has to be moved to the active set and remain paused until the scheduler prompts it to continue. Additionally, blocking avoidance requires that each process be able to determine whether it will block on a `receive`, before executing the actual statement. Both tasks call for some complex instrumentation of `receive` expressions, which will be discussed in further detail in Chapter 5.

- **Aborting executions**

  By using blocking avoidance we can guarantee that during replay mode no action in a partial state can ever be a block, except for the last one. For the last action in a partial state it is not possible to know beforehand whether it is a blocking `receive` or not. However, in case it is indeed a blocking `receive`, we can safely abort the execution of the partial state without adding anything to the state stack. This is valid, because all alternatives have either been already executed or have been stored in the stack for later execution. Additionally, the execution is certainly redundant because of the last action being a block.

Blocking avoidance effectively prunes state-space branches that contain process blocks and results in a more efficient analysis, especially in the case of programs with intense message passing. In Figure 4.4 the program of Listing 4.2 is analyzed again, this time using blocking avoidance. The analysis is clearly shorter than before and no redundant interleaving sequences are produced anymore.

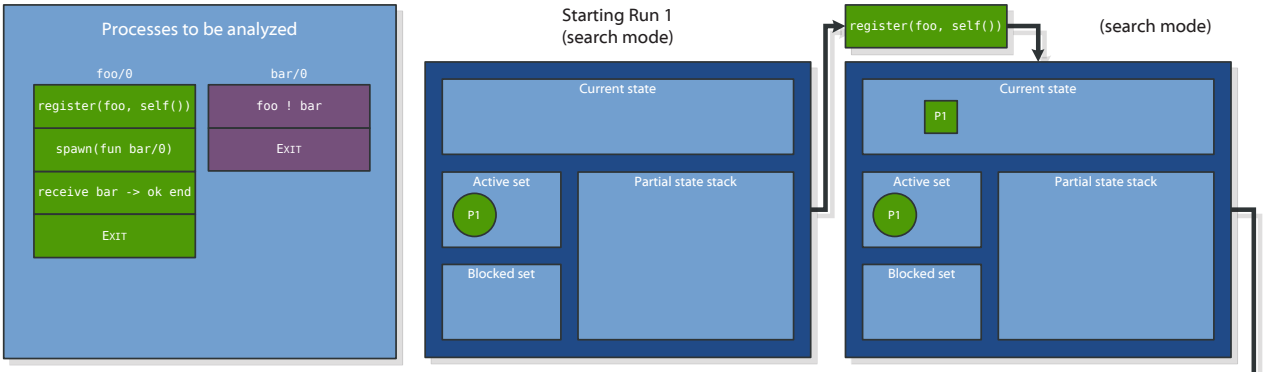


Figure 4.4: Testing the example of Listing 4.2 using depth-first search with blocking avoidance (cont'd on the next page)

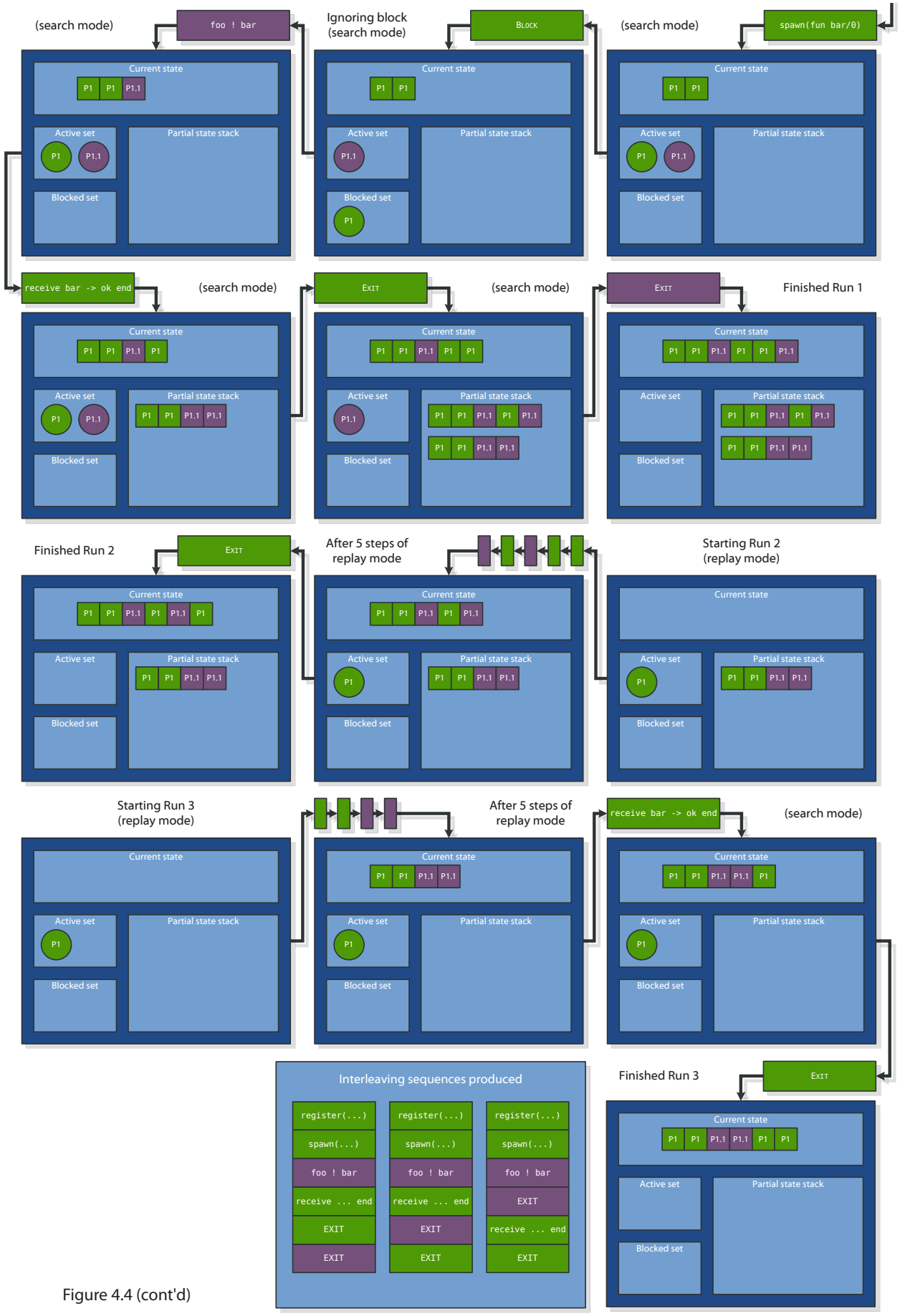


Figure 4.4 (cont'd)

The battle for efficiency

At this stage, the Concuerror scheduler possesses its basic functionality, yet the problem of combinatorial explosion has not been dealt with. Even for programs of moderate size there is a huge number of interleaving sequences (see p.15), for which there is no hope to be explored using reasonable time and memory resources.

➢ Lines of code are not a good metric in this case, because a small program might contain a large number of preemption points and vice versa.

One of the traditional heuristics used in depth-first algorithms is iterative deepening. A limit is set on the depth of the search, i.e. the number of actions with side-effects encountered during execution, and an execution is terminated when surpassing it. The limit is increased until time or memory resources are exhausted.

Musuvathi and Qadeer have proposed a different heuristic called iterative context-bounding, or *preemption bounding* in short, which is used by CHESS [27]. We utilize preemption bounding in Concuerror to limit the number of interleaving sequences explored, because of its advantages over iterative depth-bounding.

As a first step towards using preemption bounding, a distinction has to be made between *preemptions*, i.e. context switches forced upon by the scheduler and *non-preemptive context switches*, i.e. context switches required by the program itself. A process blocking on a `receive` forces a non-preemptive context switch, because another process *has* to be scheduled next. On the other hand, a preemption happens every time the execution of a *still active* process is paused and another process is executed instead (see Figure 4.5).

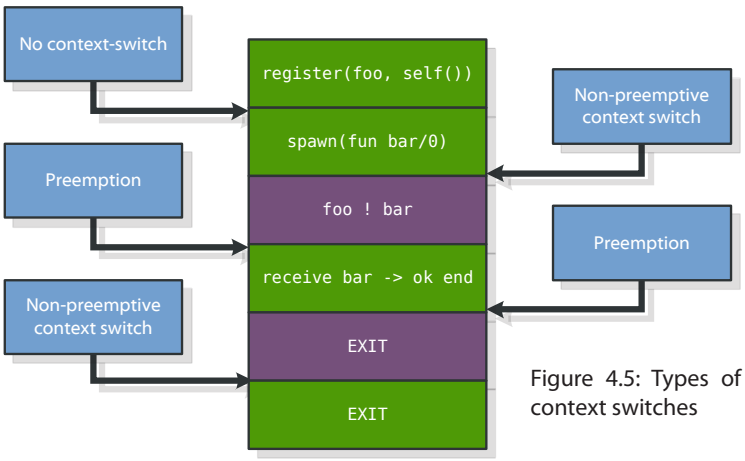


Figure 4.5: Types of context switches

Preemption bounding is based on the idea of limiting the number of preemptions allowed. Non-preemptive context switches cannot be controlled and, therefore, are allowed to happen without constraints. Setting a *preemption bound* equal to c means that on each execution at most c preemptions are allowed. The number of interleaving sequences remains exponential in the number of processes and the number of non-preemptive context switches. However, it is polynomial in the number of side-effecting actions. In addition, it is exponential in the preemption bound c, which can be chosen to be sufficiently smaller than the total number of side-effecting actions.

As a consequence, we can produce a small percentage of the total interleaving sequences by controlling the preemption bound. This compromises the soundness of the analysis, but does so in an elegant way, because the produced sequences possess some particularly attractive properties:

- **No depth limit**

  Preemption bounding offers unrestricted depth of execution. Even for a preemption bound of zero, the program is executed from start to finish (in the case of a terminating program). Contrast this with depth-limited search, which may never execute parts of the program that require a big number of steps to be reached.

- **"Simplest" error explanation**

  By starting with a preemption bound of zero and gradually increasing it, any potential error will be exposed by an interleaving sequence containing the smallest possible number of preemptions. Typically, sequences with less preemptions are easier to understand, thus preemption bounding provides in a way the simplest

explanation for an error.

- **Useful metric**

  Having checked a program up to a preemption bound c guarantees that a potential error would need more than c preemptions to be exposed. This provides a useful verification metric as well as an insightful estimate about the complexity of any remaining errors in the program.

- **Few preemptions are enough**

  Many common concurrency-related errors can be exposed by placing a few context switches in the right places. Moreover, interleaving sequences with few preemptions account for most of the program states, while sequences containing a large number of preemptions are highly redundant and represent only a small portion of the state-space. In addition to being intuitive, both of the above hypotheses are supported by experimental results [27, pp. 452-453].

All in all, favoring interleaving sequences with few preemptions but unlimited depth is intuitive and practical, while it provides a more balanced exploration than depth-limited search, which completely explores sequences up to a limited depth and completely omits deeper ones.

Concuerror uses a modified version of the original preemption bounding algorithm to incorporate the blocking avoidance technique presented in the previous section. The key idea of the original algorithm is to use two stacks for storing partial states. The first stack, called the *current* partial state stack, is used to store partial states that contain an equal number of preemptions to the current preemption bound. The second stack, called the *next* partial state stack, is used to store partial states that require one more preemption than the current bound. Thus, only partial states from the current stack may be retrieved for execution. When the current stack becomes empty, all interleaving sequences containing a number of preemptions up to the current bound have been produced. At this point, the search continues by increasing the preemption bound by one and swapping the current and next stacks.

For a given preemption bound, the driver retrieves partial states from the current stack, replays them and continues the search from there on. The procedure is similar to the simple DFS, save for the fact that now the driver has to select whether a partial state will be inserted into the current or the next stack. The choice depends on the kind of context switch that is happening between the two last actions of the partial state to be inserted. At each step of an execution the driver has to check if the process that executed the previous action is still active. If this is the case, the process has to continue its execution, because running another process in its place would constitute a preemption. The

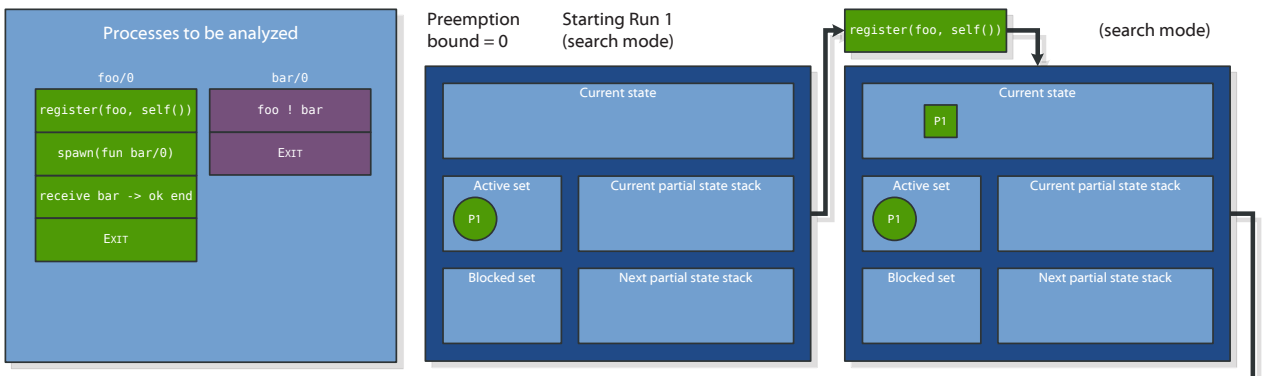


Figure 4.6: Testing the example of Listing 4.2 using preemption bounding with blocking avoidance (cont'd on the next page)

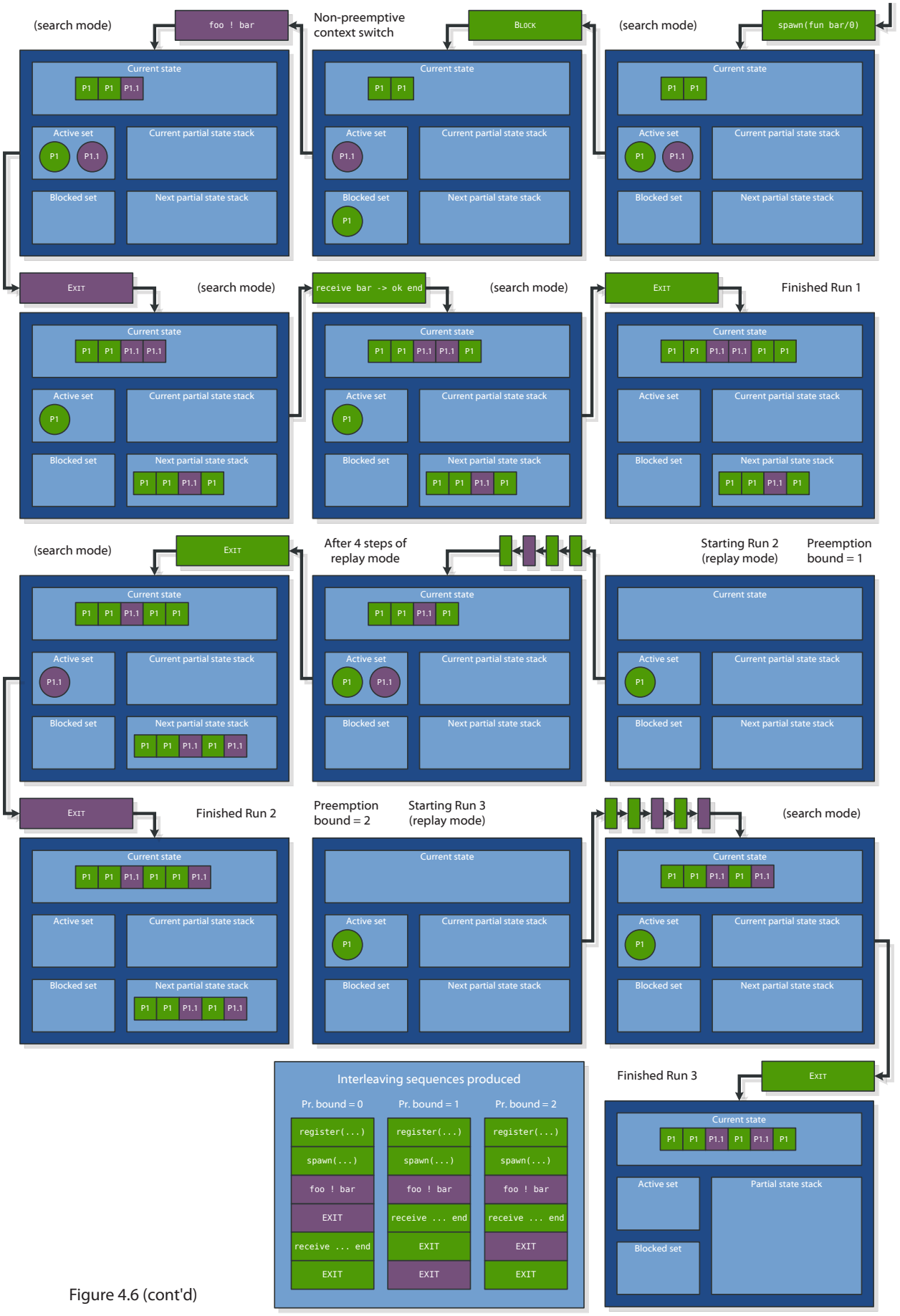


Figure 4.6 (cont'd)

partial state that corresponds to the preemption is inserted into the next stack and shall be explored as soon as the preemption bound is increased. On the other hand, if the previous process is not active anymore, every possible partial state is inserted into the current stack, because a non-preemptive context switch is happening.

In Figure 4.6 the combination of preemption bounding and blocking avoidance is used to analyze the program of Listing 4.2. Constraining the value of the preemption bound to 0, 1 or 2, we get 1, 2 or 3 interleaving sequences respectively. Note that the largest possible number of preemptions for this program is 2. Consequently, having a preemption bound greater or equal to 2 is in this case equivalent to having no preemption bound at all. Lastly, it should be clear that the algorithm terminates as soon as it reaches the largest preemption bound possible, at which point every possible interleaving sequence will have been produced.

Detecting and replaying errors

The ultimate goal of producing interleaving sequences is to find the ones that produce some error. The driver is alert at each step of every execution for any deviation from normal execution that might occur. As we mentioned in Chapter 3, Concuerror was designed to detect the same kinds of errors as conventional testing tools, with the addition of deadlocks which are obviously not detectable when using traditional testing. Potential errors can be classified 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.

- **Deadlocks**

  The driver reports a deadlock whenever a state with an empty active process set and a non-empty blocked process set is reached. Although deadlocks are not especially common in message-passing languages like Erlang, it is fairly simple to detect them in Concuerror. Note that in Concuerror any program state where one or more processes are blocked on a `receive` expression and no other process is available for scheduling is considered a deadlock.

Each detected error is represented by a structure that we call *replay ticket*. A replay ticket contains all information needed to replay a specific interleaving sequence that leads to an error and, that way, enables individual replay of erroneous executions. This information includes (at least) the function to run, its module and its arguments, as well as the state to be replayed.

5 The Instrumenter

The instrumenter is the unsung hero of Concuerror. None of the scheduler's achievements would be possible without properly instrumented user code. By parse transforming source code, which by itself is a particularly tedious task, the instrumenter has to provide some complex hooks for the scheduler, while at the same time it has to be extremely careful not to alter the original program's semantics.

In the rest of this chapter we will discuss the operation of the instrumenter in conjuction with the wrapper subcomponent that handles instrumented functions, because the two components are working closely together (see Figure 4.2).

Instrument what?

Instrumentation in Concuerror is done at module level. We already know that preemption points need only be placed at side-effecting actions. In Erlang there are several side-effecting BIFs and library functions, in addition to the built-in send (`!`) and `receive` expressions.

A send expression is equivalent to the `send/2` BIF and does not need to be considered separately. In addition, the instrumentation of the majority of function calls is roughly the same, thus it is not necessary at this point to specify exactly which functions are instrumented. Therefore, we shall mainly distinguish between function call instrumentation and `receive` instrumentation.

General considerations

The first matter that arises is whether the preemption point, i.e. the point at which a process' execution may be interrupted, is going to be placed before or after the corresponding side-effecting action. Recall that the action of a process exit may have side-effects and needs to be handled separately from any preceding actions. Placing preemption points before side-effecting actions, makes it difficult to separate a process exit from the last side-effecting action of the same process. This is why Concuerror places preemption points *after* side-effecting actions, which results in process exits being naturally separated from the last action. This also means that every newly spawned process has to be paused immediately, because otherwise it would not pause until after having executed its first side-effecting action.

A second subtle point has to do with variable names. In the course of instrumentation, user patterns are used as arguments to wrapper function calls. Therefore, any underscores present in the original code have to be replaced by new underscored vari-

Instrumented Functions

At the time of this writing, Concuerror instruments a number of BIFs, including `demonitor/1`, `demonitor/2`, `halt/1`, `halt/2`, `link/1`, `monitor/2`, `process_flag/2`, `register/2`, `spawn/1`, `spawn/3`, `spawn_link/1`, `spawn_link/3`, `spawn_monitor/1`, `spawn_monitor/3`, `spawn_opt/3`, `spawn_opt/4`, `unlink/1`, `unregister/1` and `whereis/1`.

No Erlang/OTP library function is instrumented yet. This means that if a program under test contains a call to a library function with side-effects, there are two options. The default option for now is to ignore the call, which can sometimes lead to problems. The other option is to include the library itself in the analysis, i.e. have Concuerror instrument the source code of the library together with the user source code.

Instrumenting library code can be good and bad. The good part is that Concuerror has more fine-grained control over process interleaving and may detect more subtle concurrency errors. The bad part is that the library code contributes to the complexity of the analysis and, additionally, it is difficult for the user to understand an interleaving sequence that includes actions performed by the library. Moreover, it is not sensible to spend time and effort testing the library code every time a user program is tested. In Chapter 8 we briefly discuss how this matter should be resolved.

ables, which are equivalent and can be referred to or passed as arguments. In some cases like the above, we will have to use new variable names which do not clash with already defined ones. These will be referred to as *fresh variables*.

Lastly, it should be noted that all instrumentation is done in a depth-first way, in order to handle nested side-effecting actions.

The simple case ...

We will be starting with the instrumentation of function calls, which is considerably simpler than the instrumentation of `receive` expressions. The actual instrumentation of a function call is as simple as replacing it with a call to a wrapper function provided by Concuerror. Wrapper functions are written to accept identical arguments to the original functions. For example, `spawn(fun() -> ok end)` is replaced with `wrapper:rep_spawn(fun() -> ok end)` and `Pid ! Msg` is replaced with

Listing 5.1: Uninstrumented code

```
foo() ->
  register(loop, spawn(fun() -> ok end)),
  loop ! loop ! ahoi.
```

Listing 5.2: Instrumented function calls and send expressions

```
foo() ->
  wrapper:rep_register(loop, wrapper:rep_spawn(fun() -> ok end)),
  wrapper:rep_send(loop, wrapper:rep_send(loop, ahoi)).
```

`wrapper:rep_send(Pid, Msg)`. For example, after instrumenting function calls and send expressions, the code of Listing 5.1 is trasformed to that of Listing 5.2.

Internally, wrapper functions follow a generic four-step structure:

- **Call the original function**

  The original function or a slightly modified version of it is called and the return value is saved.

- **Notify the scheduler**

  The scheduler is notified of the action, including any necessary information, like the name of the function called, the arguments provided or its return value.

- **Pause**

  The user process pauses its execution until the scheduler prompts it to continue.

  ➢ This is where the actual preemption point in the user program lies.

- **Return**

  When the execution continues, the return value of the original function, which was saved in the first step, is returned.

As an example, take a look at the wrapper for `spawn/1`, which is shown in Listing 5.3. We have talked before about the need to pause a newly spawned process just before it begins executing the user function it is intended to. This is accomplished here by calling `pause/0` right after the new process is spawned. The rest of the steps should be pretty obvious.

➢ Our function pause/0 is essentially a receive expression; something like

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

Listing 5.3: Wrapper function for spawn/1

```
rep_spawn(Fun) ->
  Result = spawn(fun() -> pause(), Fun() end),
  notify_scheduler(spawn, Result),
  pause(),
  Result.
```

The send wrapper shown in Listing 5.4 is more interesting. When analyzing a program, there are usually a lot of messages being passed around. Some of them originate in instrumented user processes, while others come from uninstrumented processes or the Erlang runtime itself. In Concuerror, we want to provide the user with as much information as possible about process interactions. To that end, additional information is added to messages sent by instrumented user processes (e.g. the sender's pid), hence the use of that strange `INSTR_MSG` macro instead of the original message. Of course, this has to be supported by the `receive` instrumentation as well. The rest of the wrapper function should again be easy to understand.

Other wrapper functions are very similar to the above, with possibly minor changes if some special handling is needed, depending on the original function's semantics.

Listing 5.4: Wrapper function for send (!) and send/2

```
rep_send(Pid, Msg) ->
  Pid ! ?INSTR_MSG(Msg),
  notify_scheduler(send, {Pid, Msg}),
  pause(),
  Msg.
```

. . . and the hard one!

There are some properties of `receive` expressions that make them particularly hard to instrument. First, a built-in language expression, rather than a function call, is used for receiving messages. Unlike function calls, there are no arguments to be passed to wrapper functions in this case. To convey information to some wrapper function we are going to need some complex syntax transforms. Second, receives are blocking actions and require different handling from most actions that are non-blocking. Last, `receive` expressions may contain timeouts, which can normally not be handled by a tool like Concuerror. However, timeouts have to be ignored in an elegant way that does not alter the program's semantics and at the same time does not miss interleaving sequences that could occur in practice.

We will present the instrumentation of `receive` expressions in an incremental way, starting from the simplest idea and gradually proceeding to more complex ones.

Version 1

In the previous section we mentioned that messages are instrumented to carry additional information. This means that patterns in `receive` expressions have to be altered so that instrumented messages are matched correctly. Other than that, a preemption point has to be inserted immediately after a message has been received. We can place a function call to a wrapper function as the first thing to be executed after the pattern has been matched.

Following these simple ideas, the `receive` expression of Listing 5.4 would be transformed to that of Listing 5.5. For the sake of simplicity we will assume that instrumented messages are tuples consisting of a unique atom, the sender's pid and the original message. The wrapper function should just notify the scheduler and pause the process, as shown in Listing 5.6.

Problem: If no matching message is available, the instrumented `receive` expression will block for ever. The only way for the process to pause would be to reach a preemption point, but this is not possible without the arrival of a matching message.

Version 2

What we essentially need, is a preemption point in case there is no matching message available. When we were talking about the scheduler, it was assumed that a blocked process is automagically moved from the blocked to the active set, as soon as a matching message arrives. In fact, either synchronous polling of the process or asynchronous process-initiated notification can be used to change the state of the process from blocked to active. Either way, some kind of loop is needed in order to repeatedly check for the arrival of matching messages.

Listing 5.4: Original receive expression

```
receive
  [foo|Tail] = List -> bar;
  Other ->
    baz(),
    gazonk()
end
```

Listing 5.5: Instrumented receive expression (Version 1)

```
receive
  {?UNIQUE, Fresh1, [foo|Tail] = List} ->
    wrapper:rep_receive_notify(Fresh1, [foo|Tail]),
    bar;
  {?UNIQUE, Fresh2, Other} ->
    wrapper:rep_receive_notify(Fresh2, Other),
    baz(),
    gazonk()
end
```

Listing 5.6: Wrapper function (Version 1)

```
rep_receive_notify(From, Msg) ->
  notify_scheduler('receive', {From, Msg}),
  pause().
```

This can be accomplished by a combination of adding an `after` clause with a zero timeout value to the original statement and encapsulating the new statement inside an anonymous function that is passed as an argument to a wrapper function. The result is shown in Listing 5.7.

Function `rep_receive/1` depends on the exact implementation, but, in any case, its structure is similar to that shown in Listing 5.8. If there are no matching messages, the `after` clause is entered, the scheduler is notified of the block and the process pauses. After having been prompted to continue, the process calls `rep_receive` again and repeats the same procedure until a message arrives. When a message is available, the

Listing 5.7: Instrumented receive expression (Version 2)

```
wrapper:rep_receive(
  fun(Fresh1) ->
    receive
      {?UNIQUE, Fresh2, [foo|Tail] = List} ->
        wrapper:rep_receive_notify(Fresh2, [foo|Tail]),
        bar;
      {?UNIQUE, Fresh3, Other} ->
        wrapper:rep_receive_notify(Fresh3, Other),
        baz(),
        gazonk()
    after 0 -> Fresh1()
    end
  end)
```

Listing 5.8: Wrapper function (Version 2)

```
rep_receive(Fun) ->
  Fun(fun() -> notify_scheduler(block, self()),
               pause(),
               rep_receive(Fun)
      end).
```

`after` clause is not entered and the loop is broken.

Problem: This method gives rise to a subtle syntax error, that occurs when there is an assignment to a variable inside the `receive` expression. In the previous example, if the variable `List` gets bound inside the `receive`, it should be visible after that. However, putting the whole statement inside an anonymous function, makes the variable local to that function and hides it from the rest of the program.

Version 3

As it seems, we cannot put the `receive` expression inside a function and pass it as an argument to our wrapper function. Nevertheless, what our wrapper function really needs to know about is the `receive` patterns, so that it can check about matching messages. Therefore, we can instrument the `receive` expression itself as in Version 1 and, before that, add a call to a wrapper function, passing along the patterns in the form of a `case` expression inside an anonymous function. This method is shown in Listing 5.9.
We effectively combine the two previous methods and use a call to `rep_receive` in order to guarantee that a matching message will have arrived by the time the instrumented `receive` expression is executed. The `case` statement is used by the wrapper to manually check the mailbox for matching messages, without actually consuming them, as shown in Listing 5.10. We use `process_info/2` with a `messages` argument to retrieve the process' mailbox in the form of a list and then our function `match/2` is used to check the messages one by one.

If a matching message is found, `rep_receive/1` returns immediately and the program continues to the actual `receive` expression. Otherwise, the process is reported as blocked to the scheduler and enters a busy-loop checking for the arrival of a matching message. As soon as one arrives, the scheduler is notified and the process waits for a prompt to continue its execution and receive the message that has arrived.

In our example, there is a catchall clause (`Other -> ...`) present in the original `receive` expression, which means that the process will block at the wrapper function only if there is no message at all available. If the original expression contains no catchall clause, we have to add one to the instrumented `case` expression, so that the process blocks in the wrapper function in case there are non-matching messages avail-

Listing 5.9: Instrumented receive expression (Version 3)

```
wrapper:rep_receive(
  fun(Fresh1) ->
    case Fresh1 of
      {?UNIQUE, Fresh2, [foo|Tail] = List} -> match;
      {?UNIQUE, Fresh3, Other} -> match
    end
  end),
receive
  {?UNIQUE, Fresh4, [foo|Tail] = List} ->
    wrapper:rep_receive_notify(Fresh4, [foo|Tail]),
    bar;
  {?UNIQUE, Fresh5, Other} ->
    wrapper:rep_receive_notify(Fresh5, Other),
    baz(),
    gazonk()
end
```

Listing 5.10: Wrapper function (Version 3)

```
rep_receive(Fun) ->
  {messages, Mailbox} = process_info(self(), messages),
  case match(Fun, Mailbox) of
    match -> continue;
    nomatch ->
      notify_scheduler(block, self()),
      loop(Fun)
  end.

loop(Fun) ->
  {messages, Mailbox} = process_info(self(), messages),
  case match(Fun, Mailbox) of
    match ->
      notify_scheduler(unblock, self()),
      pause();
    nomatch -> loop(Fun)
  end.

match(Fun, []) ->
  nomatch;
match(Fun, [Msg|Rest]) ->
  case Fun(Msg) of
    match -> match;
    nomatch -> match(Fun, Rest)
  end.
```

able. For example, the `receive` expression of Listing 5.11 is instrumented as shown in Listing 5.12.

Problem: We have successfully dealt with messages between user processes executing instrumented code. However, even if there are no uninstrumented user processes, there is still the Erlang/OTP runtime which may send messages that we are not able to instrument (e.g. `'EXIT'` messages).

Version 4

Actually, part of the mechanism for handling uninstrumented messages has already been implemented by adding a unique atom to the tuple of instrumented messages. This lets us differentiate between instrumented and uninstrumented messages and allows the insertion of additional pattern-action pairs for handling uninstrumented messages, as shown in Listing 5.13. The additional patterns are identical to the patterns

Listing 5.11: Original receive expression without a catchall pattern

```
receive
  [foo|Tail] = List ->
    bar;
  {baz, 42} ->
    gazonk(),
    its_a_talking_dog()
end
```

Listing 5.12: Instrumented receive expression without a catchall pattern (Version 3)

```
wrapper:rep_receive(
  fun(Fresh1) ->
    case Fresh1 of
      {?UNIQUE, _Fresh2, [foo|Tail] = List} -> match;
      {?UNIQUE, _Fresh3, {baz, 42}} -> match;
      {?UNIQUE, _Fresh4, _Fresh5} -> nomatch
    end
  end),
receive
  {?UNIQUE, Fresh6, [foo|Tail] = List} ->
    wrapper:rep_receive_notify(Fresh6, [foo|Tail]),
    bar;
  {?UNIQUE, Fresh7, {baz, 42}} ->
    wrapper:rep_receive_notify(Fresh7, {baz, 42}),
    gazonk(),
    its_a_talking_dog()
end
```

Listing 5.13: Instrumented receive expression (Version 4)

```
wrapper:rep_receive(
  fun(Fresh1) ->
    case Fresh1 of
      {?UNIQUE, Fresh2, [foo|Tail] = List} -> match;
      [foo|Tail] = List -> match;
      {?UNIQUE, Fresh3, Other} -> match;
      Other -> match
    end
  end),
receive
  {?UNIQUE, Fresh4, [foo|Tail] = List} ->
    wrapper:rep_receive_notify(Fresh4, [foo|Tail]),
    bar;
  [foo|Tail] = List ->
    wrapper:rep_receive_notify(unknown, [foo|Tail]),
    bar;
  {?UNIQUE, Fresh5, Other} ->
    wrapper:rep_receive_notify(Fresh5, Other),
    baz(),
    gazonk();
  Other ->
    wrapper:rep_receive_notify(unknown, Other),
    baz(),
    gazonk()
end
```

of the original `receive` expression. Function `rep_receive_notify` is called with an `unknown` argument to indicate that no information is available about the sender of the message.

This is the final version of Concuerror's `receive` expression instrumentation. It works well with both instrumented and uninstrumented messages and does not change the original expressions' semantics.

Timeouts

User-introduced delays or timeouts are outside the scope of tools like Concuerror. We did 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 have to note that in the presence of delays Concuerror may produce interleaving sequences that are not likely to occur in practice. For the program shown in Listing 5.14, the process running `bar` would have to spend more than one second in the send (`!`) operation, in order to produce the event sequence of Figure 5.1 in practice. Concuerror will nevertheless report an erroneous interleaving for this program. Although extremely rare, sequences like this are semantically valid according to the Erlang specification.

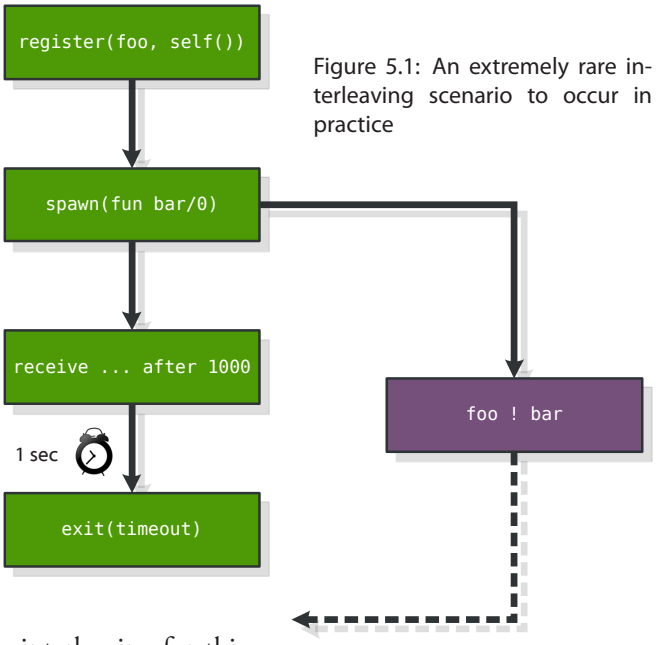


Figure 5.1: An extremely rare interleaving scenario to occur in practice

To instrument `receive` expressions with an `after` clause, we differentiate between a timeout value of `infinity`, which is equivalent to an expression without an `after` clause and is instrumented as before, and any integer timeout value, which is set to zero. The differentiation has to happen at runtime using a `case` clause, because a variable may be used to specify the timeout value. In the second case, there is no need to check for a block, because the `after` clause will be entered any time there is no matching message available. The instrumented expression of Listing 5.15 is shown in

Listing 5.14: Program containing a delay

```
foo() ->
  register(foo, self()),
  spawn(fun bar/0),
  receive
    bar -> ok
  after 1000 -> exit(timeout)
  end.

bar() ->
  foo ! bar.
```

Listing 5.15: Original receive expression with an after clause

```
foo(Timeout) ->
  receive
    [foo|Tail] = List -> bar;
    Other ->
      baz(),
      gazonk()
  after Timeout -> its_a_talking_dog()
  end.
```

Listing 5.16. Similarly to `rep_receive_notify`, function `rep_after_notify` informs the scheduler about the entering of the `after` clause and then pauses the process.

Finally, using the same reasoning as above, the `receive` expression of Listing 5.17 with an `after` clause and no patterns, is instrumented as shown in Listing 5.18.

Listing 5.16: Instrumented receive expression with an after clause

```
foo(Timeout) ->
  case Timeout of
    infinity ->
      wrapper:rep_receive(
        fun(Fresh1) ->
          case Fresh1 of
            {?UNIQUE, Fresh2, [foo|Tail] = List} -> match;
            [foo|Tail] = List -> match;
            {?UNIQUE, Fresh3, Other} -> match;
            Other -> match
          end
        end),
      receive
        {?UNIQUE, Fresh4, [foo|Tail] = List} ->
          wrapper:rep_receive_notify(Fresh4, [foo|Tail]),
          bar;
        [foo|Tail] = List ->
          wrapper:rep_receive_notify(unknown, [foo|Tail]),
          bar;
        {?UNIQUE, Fresh5, Other} ->
          wrapper:rep_receive_notify(Fresh5, Other),
          baz(),
          gazonk();
        Other ->
          wrapper:rep_receive_notify(unknown, Other),
          baz(),
          gazonk()
      end;
      _Fresh7 ->
        receive
          {?UNIQUE, Fresh4, [foo|Tail] = List} ->
            wrapper:rep_receive_notify(Fresh4, [foo|Tail]),
            bar;
          [foo|Tail] = List ->
            wrapper:rep_receive_notify(unknown, [foo|Tail]),
            bar;
          {?UNIQUE, Fresh5, Other} ->
            wrapper:rep_receive_notify(Fresh5, Other),
            baz(),
            gazonk();
          Other ->
            wrapper:rep_receive_notify(unknown, Other),
            baz(),
            gazonk()
        after 0 ->
          wrapper:rep_after_notify(),
          its_a_talking_dog()
        end
  end.
```

Listing 5.17: Original receive expression with an after clause and no patterns

```
receive
after Timeout -> its_a_talking_dog()
end
```

Listing 5.18: Instrumented receive expression with an after clause and no patterns

```
case Timeout of
  infinity -> wrapper:block_for_ever();
  _Fresh1 -> its_a_talking_dog()
end
```

What's next?

The more theoretical part of our presentation is over. We have seen the parse transforms used to instrument the user code and the algorithms used to search the program's state-space. Now it is time to see Concuerror in action. In the next chapter we will develop a simple—though not trivial—concurrent registration server and test it using Concuerror and its GUI.

6 Concuerror by Example

In this chapter we will see how Concuerror can be used in practice as a testing and debugging aid. The presentation of the Concuerror user interface will be done through a fairly simple example program that we will write and test step by step in a loosely test-driven fashion.

We intend to create a generic registration server, that can, for example, be used to manage limited system resources. The desired functionality includes *starting* and *stopping* the server, as well as *attaching* and *detaching* processes to and from the server. However, only a limited number of processes are allowed to be attached to the server at any moment.

Getting started

First, we will create two Erlang modules, one to contain the server code (`reg_server`), the other to contain the tests (`reg_server_tests`). Let us start by writing a `start/0` function for starting the server and a trivial test that checks that the function returns `ok`.

The Erlang/OTP distribution comes with a unit testing framework named EUnit, which provides several macros pertaining to the creation and execution of tests and test suites. Concuerror is not yet fully compatible with EUnit, but some assertion macros can be readily used in Concuerror tests. One of the compatible macros is `?assertEqual`, which tests two expressions for equality. The convention behind its use is that the first argument declares the expected value to which the second argument is to be compared. The `reg_server_tests` module, shown in Listing 6.1, uses an `include_lib` statement to make the above macro available.

➢ Starting with SUnit, an automated unit testing framework for Smalltalk, and JUnit, its Java equivalent, similar frameworks have been created for most languages and are collectively referred to as xUnit.

Our first test uses `?assertEqual` to check that `start/0` returns `ok`. Note that functions with a "`_test`" suffix are auto-exported by EUnit, thus no `export` attribute is needed. The first version of the server module is shown in Listing 6.2.The module exports the `start/0` function, which just returns `ok` for now.

Listing 6.1: The initial testing module

```
-module(reg_server_tests).

-include_lib("eunit/include/eunit.hrl").


start_test() ->
  ?assertEqual(ok, reg_server:start()).
```

➢ Newly added code at each step will be non syntax-highlighted and in bold red.

Listing 6.2: The initial registration server module

```
-module(reg_server).

-export([start/0]).


start() ->
  ok.
```

It is time to run our test using Concuerror. After opening the Concuerror GUI we are faced with several panels. The Modules panel in the upper left side displays a list of imported Erlang modules in Concuerror. These are the modules that will be instrumented when Concuerror begins its analysis. The Functions panel in the lower left side displays a list of exported functions from the selected module. The Log panel in the lower right side displays information about Concuerror actions. Additionally, selecting the Source tab next to Main we can display the source code of the selected module in the Modules panel.

Let us run our test before explaining the rest of the GUI functionality. To import our modules we click the Add button or select File→Add. Through the browse dialog we locate our two modules on the filesystem, select them, and click Open. Our modules have now been imported and added to the Modules panel. By selecting module `reg_server_tests` from this panel, the testing function `start_test/0` appears in the Functions panel. For the time being, we will disable preemption bounding by unchecking the Enable preemption bounding option under Edit→Preferences. We can now select the test

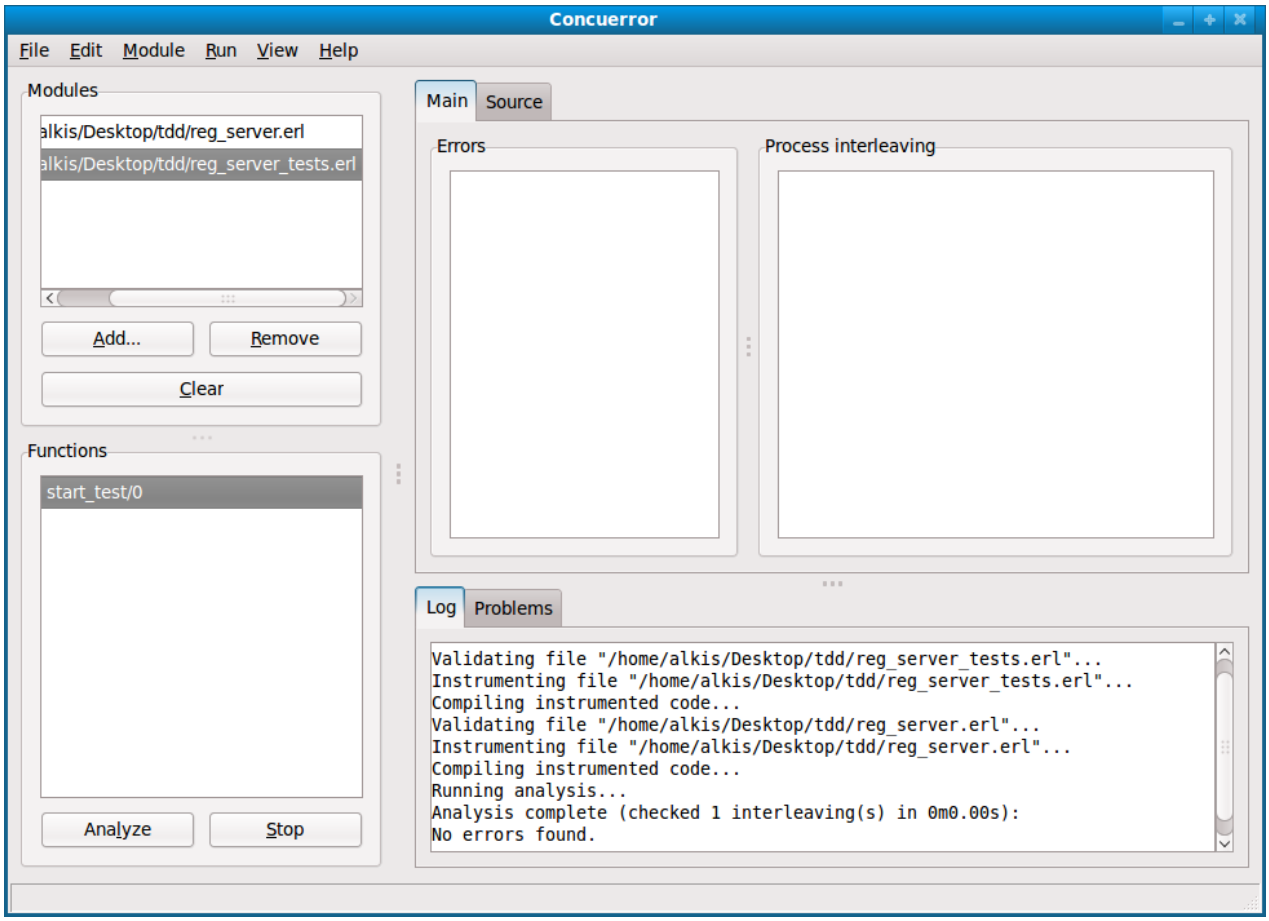


Figure 6.1: Running our first test successfully in the Concuerror GUI

function and click Analyze to execute the test under Concuerror. The Log panel informs us about the successful instrumentation and compilation of our modules as well as about the complete execution of one interleaving sequence without any errors (see Figure 6.1).

Let us now change the return value of `start/0` from `ok` to `error` and run the analysis again. This time an error appears in the Errors panel, namely an assertion violation in line 6 of our testing module. Furthermore, in the Process interleaving panel we can see the corresponding erroneous interleaving sequence. It consists of only one side-effecting action—that of our initial process' abnormal exit (see Figure 6.2). We can also select the Problems tab next to Log and switch to the corresponding panel, which displays additional information about the specific error: The expected value was `ok` but `start/0` returned `error`. We change the return value back to `ok` before proceeding.

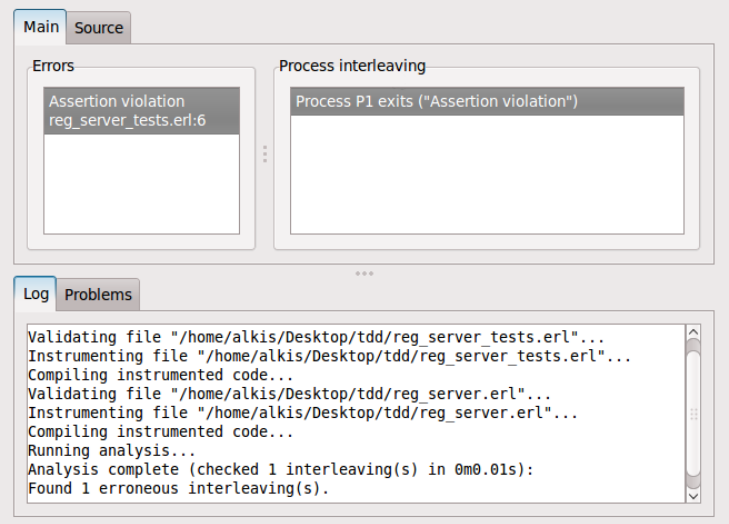


Figure 6.2: Information about an erroneous interleaving

These are the basics about importing modules and running tests using the Concuerror GUI. In the next section we will begin to implement the server's functionality. To keep track of our implementation and testing tasks, we will use a simple TODO list, like the one shown on the right. The current task will be in bold and the finished tasks will be striked out. We will be adding additional tasks or subtasks as they come along.

- **Start server**
- Stop server
- Attach process
- Detach process

Starting and stopping the server: The basics

To start the server, we want to create a server process and register it under the name `reg_server`. To test that the server is spawned and registered as expected, we will create an auxiliary `ping/0` function that returns `pong` if the server responds. In case the server is down, the `ping` function should probably timeout after a while, but we leave this task for later. We also leave for later the case of `start/0` being called more than once—by either one or more processes. For now we want to spawn the server process, register it, and make it respond to a ping call. To this end we create `ping_test/0` shown in Listing 6.3. We call `ping/0` twice to make sure that the server loop is correct.

- **Start server**
  - **Ping**
  - Ping timeout
  - Multiple start calls
- Stop server
- Attach process
- Detach process

To make the test pass, we spawn a new process to start the server, register it under

Listing 6.3: Add a ping test

```
-module(reg_server_tests).

-include_lib("eunit/include/eunit.hrl").


start_test() ->
  ?assertEqual(ok, reg_server:start()).

ping_test() ->
  reg_server:start(),
  ?assertEqual(pong, reg_server:ping()),
  ?assertEqual(pong, reg_server:ping()).
```

Listing 6.4: Add spawn, register and ping to the server

```
-module(reg_server).

-export([ping/0, start/0]).

-define(REG_NAME, reg_server).
-define(REG_REQUEST, reg_request).
-define(REG_REPLY, reg_reply).


start() ->
  Pid = spawn(fun() -> loop() end),
  register(?REG_NAME, Pid),
  ok.

ping() ->
  ?REG_NAME ! {?REG_REQUEST, self(), ping},
  receive
    {?REG_REPLY, Reply} -> Reply
  end.

loop() ->
  receive
    {?REG_REQUEST, Target, ping} ->
      Target ! {?REG_REPLY, pong},
      loop()
  end.
```

the name `reg_server`, and make it run function `loop/0`, as shown in Listing 6.4. Inside its loop, the server receives ping requests and replies with pong messages. Function `ping/0` sends a ping request to the server and waits for a pong response. The macros `REG_REQUEST` and `REG_REPLY` are just used to avoid confusion with messages sent by other processes.

Running our tests in Concuerror results in both of them reporting a deadlock. We should expect this, because the server runs in a loop and is blocked waiting for a request, even after the client process, i.e. the process running the test function, has exited. To avoid this behavior, we have to implement the functionality of stopping the server and use it at the end of our tests. Again, we leave the handling of multiple stop calls for later.

- **Start server**
  - **Ping**
  - Ping timeout
  - Multiple start calls
- **Stop server**
  - Multiple stop calls
- Attach process
- Detach process

➢ When the name of some exported function has changed, a refresh might be necessary so as to update the name in Concuerror. This can be done by selecting Module→Refresh.

To stop the server we create function `stop/0`. Listing 6.5 shows the modified tests and Listing 6.6 shows the implementation of `stop/0`. Similarly to `ping/0`, a message is sent to the server and the latter replies with `ok`. However, in this case, the server terminates instead of returning to the loop.

Now both tests pass when run in Concuerror. Note that even these small tests produce more than one interleaving sequences—in this case three—due to the different exit orders of the two processes participating in the tests. Before we continue, let us refactor the code of Listing 6.6 to remove some of the duplication that was introduced during our last step. In Listing 6.7 we introduce functions `request/1` and `reply/2` to handle the sending and receiving of messages between client and server. The tests still pass after the refactoring, so at this point we are able to perform the basic operations of starting, pinging and stopping our server.

Listing 6.5: Stop the server at the end of our tests

```
start_stop_test() ->
  ?assertEqual(ok, reg_server:start()),
  ?assertEqual(ok, reg_server:stop()).

ping_test() ->
  reg_server:start(),
  ?assertEqual(pong, reg_server:ping()),
  ?assertEqual(pong, reg_server:ping()),
  reg_server:stop().
```

Listing 6.6: Add stop/0 to the server

```
-module(reg_server).

-export([ping/0, start/0, stop/0]).

[...]

stop() ->
  ?REG_NAME ! {?REG_REQUEST, self(), stop},
  receive
    {?REG_REPLY, Reply} -> Reply
  end.

loop() ->
  receive
    {?REG_REQUEST, Target, ping} ->
      Target ! {?REG_REPLY, pong},
      loop();
    {?REG_REQUEST, Target, stop} ->
      Target ! {?REG_REPLY, ok}
  end.
```

Starting and stopping the server: Advanced issues

Let us now handle the case of multiple stop calls. We will first test the case of two consecutive stop calls by one process and then that of two concurrent stop calls by two processes. As shown in Listing 6.8, when a process calls `stop/0` and the server is not running, the return value should be `server_down`. Running this test in Concuerror results in three erroneous interleaving sequences, two due to an exception and one due to a deadlock. Looking at the Process interleaving panel we notice that the first exception happens when the client process attempts to send a message to the already exited server process using its registered name. To fix this error, we can use the `whereis/1` BIF to check whether the server name is registered, i.e. whether the server is running, before sending the stop message. In fact, we can do this inside `request/1`, so that the check is done before any message is sent to the server, as shown in Listing 6.9.

- ~~Start server~~
  - ~~Ping~~
  - Ping timeout
  - Multiple start calls
- ~~Stop server~~
  - **Multiple stop calls (1 proc)**
  - Multiple stop calls (2 proc)
- Attach process
- Detach process

Running the test again, we see that the first error is fixed, but the two others are still there. Now is a good point to try running the tests using EUnit, rather than Concuerror. This can be done by first compiling the two modules and then calling `eunit:test(reg_server_tests)`. EUnit reports that all three tests pass! We can try it

Listing 6.7: The refactored code of Listing 6.6

```
stop() ->
  request(stop).

ping() ->
  request(ping).

loop() ->
  receive
    {?REG_REQUEST, Target, ping} ->
      reply(Target, pong),
      loop();
    {?REG_REQUEST, Target, stop} ->
      reply(Target, ok)
  end.

request(Request) ->
  ?REG_NAME ! {?REG_REQUEST, self(), Request},
  receive
    {?REG_REPLY, Reply} -> Reply
  end.

reply(Target, Reply) ->
  Target ! {?REG_REPLY, Reply}.
```

Listing 6.8: Test for two stop calls by one process

```
multiple_stops_test() ->
  reg_server:start(),
  ?assertEqual(ok, reg_server:stop()),
  ?assertEqual(server_down, reg_server:stop()).
```

Listing 6.9: Use whereis/1 before sending a message to the server

```
request(Request) ->
  case whereis(?REG_NAME) of
    undefined -> server_down;
    _Pid ->
      ?REG_NAME ! {?REG_REQUEST, self(), Request},
      receive
        {?REG_REPLY, Reply} -> Reply
      end
  end.
```

again and again; the result is the same. The reason is that the errors displayed in Concuerror are caused by interleaving sequences that are extremely unlikely to occur in practice. As we mentioned in previous chapters, Heisenbugs are often caused by interleaving scenarios that are almost impossible to reveal with traditional testing tools, like EUnit, but are still probable to unexpectedly occur in practice.

Back to Concuerror and in the Process interleaving panel, we can see that both errors have the same cause: Between the server's reply and its actual exit, the client process manages to squeeze in and call `whereis/1`—it even manages to additionally send its

second stop request in one of the two interleaving sequences. Because the server has not exited yet, the call to `whereis/1` does not return `undefined`. Subsequently, the server exits and, as a consequence, the client either fails with an exception, in case it tries to send a stop request to a process that is not registered anymore, or blocks, in case it has already sent the request and is waiting for an answer from the non-existing server. The detailed sequence for the second case, as viewed in the Concuerror GUI, is shown in Figure 6.3.

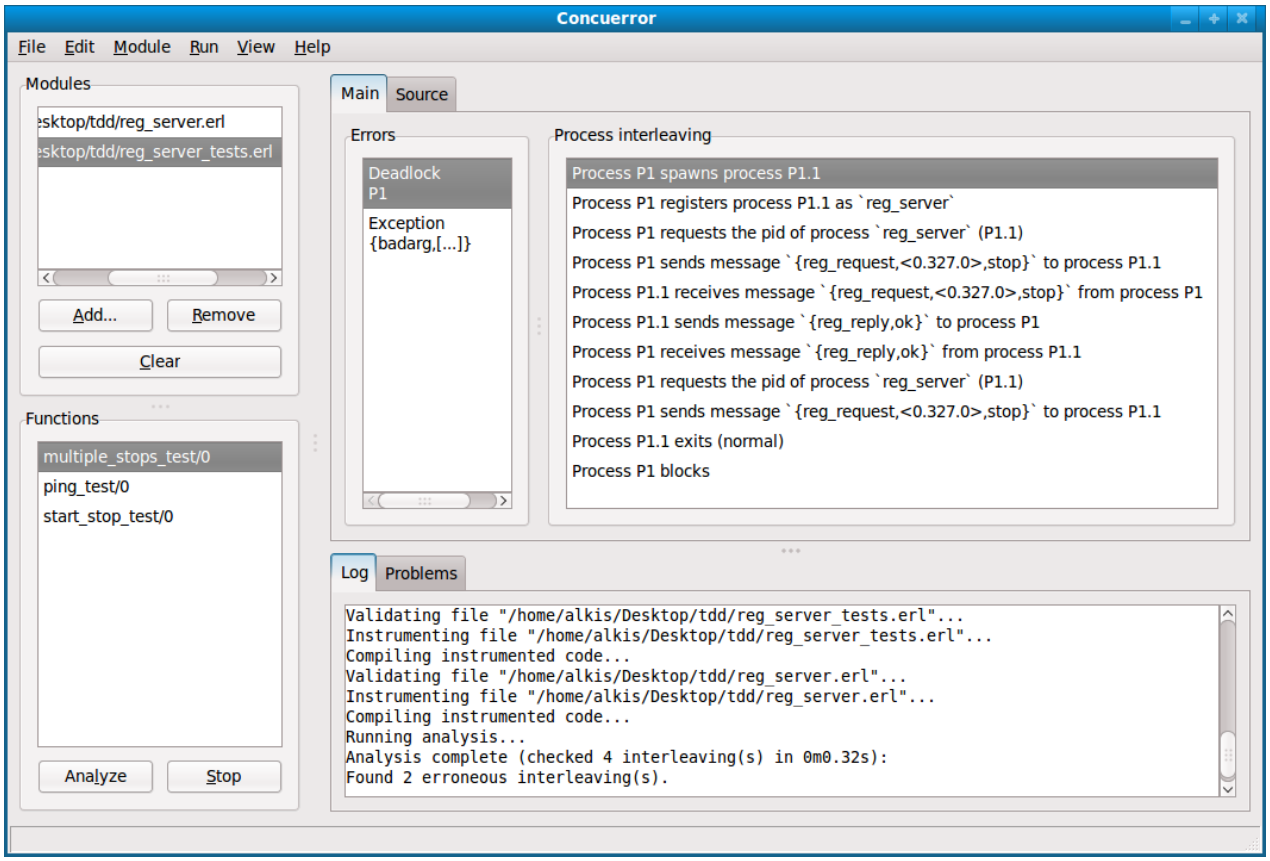


Figure 6.3: Concuerror's detailed interleaving information helps understand and fix the error

The error described above can be solved by making the server unregister itself before replying to the client's stop request. The fixed code is shown in Listing 6.10, which makes all three of our tests pass in Concuerror.

Next, we have to test the case where multiple processes try to stop the server concurrently. We wiil test this scenario by spawning two processes, making them call `stop/0`

- ~~Start server~~
  - ~~Ping~~
  - Ping timeout
  - Multiple start calls
- ~~Stop server~~
  - ~~Multiple stop calls (1 proc)~~
  - **Multiple stop calls (2 proc)**
- Attach process
- Detach process

Listing 6.10: Unregister the server before replying to a stop request

```
loop() ->
  receive
    {?REG_REQUEST, Target, ping} ->
      reply(Target, pong),
      loop();
    {?REG_REQUEST, Target, stop} ->
      unregister(?REG_NAME),
      reply(Target, ok)
  end.
```

on the server, and collecting the return values of their calls. We expect that one call will return `ok` and the other `server_down`, as shown in Listing 6.11.

Running this test in Concuerror results in a large number of erroneous interleaving sequences. Looking at the first one, we notice that both processes call `whereis/1`, before the server stops. The problem is similar to the one we had in our previous step. If the server has exited before the process sends its request, then the requesting process will fail with an exception. Alternatively, if the server exits after the request has been sent, the requesting process will block waiting for a server reply that will never come.

We have to provide a way for a process to be informed whether the server has been stopped by another process after the execution of the first process' `whereis/1` call. To avoid the exceptions caused by the server not being registered after having exited, we can use the PID returned by `whereis/1`, instead of the server's registered name to address the server inside `request/1`. Doing this will result in the transformation of the exception errors into deadlocks due to processes blocking on the `receive` expression of `request/1`. The "trick" to avoid these deadlocks is to use a `monitor/2` call to the server process before sending the request message to the server. If the latter has already exited or exits after the monitor call, the process will receive a `'DOWN'` message from the Erlang runtime and can return `server_down`. Otherwise, the process will receive a normal reply from the server. In any case, a message will be received and, therefore, the process will never block. The changes made in the server code are shown in Listing 6.12. A call to `demonitor/2` with the `flush` option is used to stop monitoring the server and discard the `'DOWN'` message in case the server has already exited.

After making the above changes, all tests pass. Note that our latest test produces a huge number of interleaving sequences (more than 800,000) and takes several minutes to complete. At this point, we can turn on preemption bounding by selecting the Enable preemption bounding option from the Edit→Preferences dialog and use the default value of two for the preemption bound parameter. Running the test again, a considerably lower number of interleaving sequences (about 1,700) are produced in a matter of seconds.

However, we would also like to know if the previous defect would have been detected if a preemption bound of two were used. Temporarily reverting the changes of Listing 6.12 and running the test again, we can see that the defect is indeed detected—in this case the defect is even detected with a preemption bound of zero! This is a validation of the claim we made in Chapter 4 about preemption bounding, i.e. that even a low preemption bound is usually enough to reveal many common concurrency errors.

Listing 6.11: Test for two concurrent stop calls by two processes

```
multiple_concurrent_stops_test() ->
  Self = self(),
  reg_server:start(),
  spawn(fun() -> Self ! reg_server:stop() end),
  spawn(fun() -> Self ! reg_server:stop() end),
  ?assertEqual([ok, server_down],
               lists:sort(receive_two())).

receive_two() ->
  receive
    Result1 ->
      receive
        Result2 -> [Result1, Result2]
      end
  end.
```

Listing 6.12: Monitor the server to deal with multiple concurrent stop calls

```
request(Request) ->
  case whereis(?REG_NAME) of
    undefined -> server_down;
    Pid ->
      Ref = monitor(process, Pid),
      Pid ! {?REG_REQUEST, self(), Request},
      receive
        {?REG_REPLY, Reply} ->
          demonitor(Ref, [flush]),
          Reply;
        {'DOWN', Ref, process, Pid, _Reason} -> server_down
      end
  end.
```

Our next task is dealing with the case of pinging the server when it is not running. The current version of `request/1` should readily handle this case, without the need to use a timeout after all. To check this, we employ the tests shown in Listing 6.13. The first test checks that a call to `ping/0` returns `server_down` before starting the server for the first time as well as after starting and stopping the server. The second test checks that when a process is trying to ping the server at the same time another process is trying to stop it, the `ping/0` call will return either `pong` or `server_down`. Both tests pass successfully and we can proceed to our next task that involves multiple calls to `start/0`.

- ~~Start server~~
  - ~~Ping~~
  - **Ping timeout**
  - Multiple start calls
- ~~Stop server~~
- Attach process
- Detach process

As we did with multiple stop calls, we will test multiple start calls by first using one process and then two. A call to `start/0` should return `already_started` in case the server is already running. The corresponding single process test is shown in Listing 6.14. To make this test work we can call `whereis/1` inside `start/0` before starting the server. If `whereis/1` returns `undefined`, the server is spawned and registered as

- ~~Start server~~
  - ~~Ping~~
  - ~~Ping timeout~~
  - **Multiple start calls (1 proc)**
  - Multiple start calls (2 proc)
- ~~Stop server~~
- Attach process
- Detach process

Listing 6.13: Test calling ping/0 when the server is not running

```
ping_failure_test() ->
  ?assertEqual(server_down, reg_server:ping()),
  reg_server:start(),
  reg_server:stop(),
  ?assertEqual(server_down, reg_server:ping()).

ping_failure_2_test() ->
  reg_server:start(),
  spawn(fun() ->
          Result = reg_server:ping(),
          ?assertEqual(true,
                       lists:member(Result, [pong, server_down]))
        end),
  reg_server:stop().
```

Listing 6.14: Test for two start calls by one process

```
multiple_starts_test() ->
  reg_server:start(),
  ?assertEqual(already_started, reg_server:start()),
  reg_server:stop().
```

before, otherwise the server is already running so `already_started` is returned. The newly added server code is shown in Listing 6.15. The test passes with the new version of `start/0` and we are ready to write the two-process version of the test.

- ~~Start server~~
  - ~~Ping~~
  - ~~Ping timeout~~
  - ~~Multiple start calls (1 proc)~~
  - **Multiple start calls (2 proc)**
- ~~Stop server~~
- Attach process
- Detach process

The test for two concurrent `start/0` calls is shown in Listing 6.16 and is similar to the one we used in Listing 6.11 to test for two concurrent `stop/0` calls. Two processes are spawned and concurrently attempt to start the server. One of them should observe a return value of `ok` and the other a return value of `already_started`. The new test fails when run in Concuerror. As seen from the detailed interleaving information, when the two processes call `whereis/1` before the server is started, they both try to spawn and register the server, which results in one of them failing with an exception. Ideally, we would like the block of code containing `whereis`, `spawn` and `register` to be executed atomically, but this is not possible in Erlang.

Instead, we may use the solution of allowing multiple processes to get spawned, but only one of them to get properly registered as the server process. Subsequently, processes that fail to get registered have to be killed. To accomplish the above, we use a `try...catch` expression around the `register/2` call and a message to the spurious process forcing it to exit, as shown in Listing 6.17. Note that we cannot use a call to `request/1` to kill a spurious process, because that process has not been registered as the server.

Again, our latest test produces a large number of interleaving sequences, thus it is preferable to start with a small preemption bound and gradually increase it, while finding and correcting any errors encountered. At this point all of our tests pass with or without preemption bounding, so we have successfully completed the implementation of the functions for starting and stopping our server.

Attaching processes

- ~~Start server~~
- ~~Stop server~~
- **Attach process**
  - Already attached
  - Max proc reached
- Detach process
  - Detach on exit

Any process should be able to get attached to our server. A unique integer, that for our purposes will be called a registration number, is assigned to each attached process. The server allows for a limited number of attached processes. We will define the maxi-

Listing 6.15: Check if the server is already running before starting it

```
start() ->
  case whereis(?REG_NAME) of
    undefined ->
      Pid = spawn(fun() -> loop() end),
      register(?REG_NAME, Pid),
      ok;
    _Pid -> already_started
  end.
```

Listing 6.16: Test for two concurrent start calls by two processes

```
multiple_concurrent_starts_test() ->
  Self = self(),
  spawn(fun() -> Self ! reg_server:start() end),
  spawn(fun() -> Self ! reg_server:start() end),
  ?assertEqual([already_started, ok],
               lists:sort(receive_two())),
  reg_server:stop().
```

Listing 6.17: Use try…catch to avoid spurious server processes

```
start() ->
  case whereis(?REG_NAME) of
    undefined ->
      Pid = spawn(fun() -> loop() end),
      try register(?REG_NAME, Pid) of
        true -> ok
      catch
        error:badarg ->
          Pid ! {?REG_REQUEST, kill},
          already_started
      end;
    _Pid -> already_started
  end.

[...]

loop() ->
  receive
    {?REG_REQUEST, Target, ping} ->
      reply(Target, pong),
      loop();
    {?REG_REQUEST, Target, stop} ->
      unregister(?REG_NAME),
      reply(Target, ok);
    {?REG_REQUEST, kill} -> killed
  end.
```

mum number of attached processes in the `MAX_ATTACHED_PROC` macro of an external `reg_server.hrl` file, so that it can be included in both the server code and the tests.

To be able to check if a process is attached, we will extend the `ping/0` function to return the calling process' registration number in case it is attached to the server. The attachment of two processes can be checked using the test of Listing 6.18. We use `attach/0` to attach the calling processes to the server. Function `attach/0` returns the registration number that was assigned to the calling process. The test also checks that the registration numbers given to the two processes are not equal. What happens when a process calls `attach/0`, while it is already attached, or when the maximum number of

Listing 6.18: Test for attaching two processes to server

```
attach_test() ->
  Self = self(),
  reg_server:start(),
  RegNum1 = reg_server:attach(),
  spawn(fun() ->
          RegNum2 = reg_server:attach(),
          ?assertEqual(RegNum2, reg_server:ping()),
          ?assertEqual(false, RegNum1 =:= RegNum2),
          Self ! done
        end),
  ?assertEqual(RegNum1, reg_server:ping()),
  receive done -> reg_server:stop() end.
```

attached processes has been reached? Shouldn't a process get detached when it exits? All these matters will be handled later.

For now, let us try to make our latest test pass. The server can use an ordered set to keep track of the free registration numbers. Initially, this set will contain the numbers from 1 to `MAX_ATTACHED_PROC`. Additionally, the server can use a dictionary to store mappings from registered processes' PIDs to their corresponding registration numbers. Both structures can be packed into a record that will represent the *state* of the registration server and will be passed as an argument to its loop. Other than that, the request-reply infrastructure that we created in the previous section can also be used here, so

Listing 6.19: Add attach/0 to the server

```
-module(reg_server).

-export([attach/0, ping/0, start/0, stop/0]).

-include("reg_server.hrl").

[...]

-record(state, {free, reg}).

attach() ->
  request(attach).

start() ->
  case whereis(?REG_NAME) of
    undefined ->
      Pid = spawn(fun() -> loop(initState()) end),
    [...]
  end.

initState() ->
  FreeList = lists:seq(1, ?MAX_ATTACHED_PROC),
  #state{free = ordsets:from_list(FreeList), reg = dict:new()}.

[...]

loop(#state{free = Free, reg = Reg} = State) ->
  receive
    {?REG_REQUEST, Target, attach} ->
      [RegNum|NewFreeList] = ordsets:to_list(Free),
      NewReg = dict:store(Target, RegNum, Reg),
      reply(Target, RegNum),
      NewFree = ordsets:from_list(NewFreeList),
      NewState = State#state{free = NewFree, reg = NewReg},
      loop(NewState);
    {?REG_REQUEST, Target, ping} ->
      case dict:find(Target, Reg) of
        {ok, RegNum} -> reply(Target, RegNum);
        error -> reply(Target, pong)
      end,
      loop(State);
    [...]
  end.
```

that the case of calling `attach/0` when the server is down is handled automatically. The implementation of all the above is shown in Listing 6.19.

Every time a process requests to become attached, the server removes a registration number from its set and adds a 'pid to registration number' mapping to the dictionary. Note that the header file `reg_server.hrl` contains only the attribute `-define(MAX_ATTACHED_PROC, 2).` (A small number is used here to simplify our testing.) We do not need to import this header file in Concuerror; it is automatically recognized since it resides in the same directory as the `.erl` files. With the above additions the test passes successfully.

We will work on our next two tasks in one step because they are fairly simple. The first one is to handle an `attach/0` call from an already attached process.The server can either ignore the call and return the already allocated registration number or, alternatively, return a special value indicating an error. We will choose the first option here and check it with the first test shown in Listing 6.20 to check this. The second task is to handle the case of a process requesting to become attached when the server is already full of attached processes, as determined by `MAX_ATTACHED_PROC`. In this case we would like the `attach/0` call to return `server_full`, as shown in the second test of Listing 6.20.

- ~~Start server~~
- ~~Stop server~~
- ~~Attach process~~
  - **Already attached**
  - **Max proc reached**
- Detach process
  - Detach on exit

For the tests to pass, before attaching the requesting process, we need to check whether it is already attached and, if not, whether there are any free registration numbers. This is easily done inside the server loop as shown in Listing 6.21.

Listing 6.20: Test already attached processes and full server

```
-module(reg_server_tests).

-include_lib("eunit/include/eunit.hrl").
-include("reg_server.hrl").

[...]

already_attached_test() ->
  reg_server:start(),
  RegNum = reg_server:attach(),
  ?assertEqual(RegNum, reg_server:attach()),
  reg_server:stop().

max_attached_proc_test() ->
  reg_server:start(),
  L = lists:seq(1, ?MAX_ATTACHED_PROC),
  Ps = [spawn_attach() || _ <- L],
  ?assertEqual(server_full, reg_server:attach()),
  lists:foreach(fun(Pid) -> Pid ! ok end, Ps),
  reg_server:stop().

attach_and_wait(Target) ->
  reg_server:attach(),
  Target ! done,
  receive ok -> ok end.

spawn_attach() ->
  Self = self(),
  Pid = spawn(fun() -> attach_and_wait(Self) end),
  receive done -> Pid end.
```

Listing 6.21: Handle already registered processes and full server

```
loop(#state{free = Free, reg = Reg} = State) ->
  receive
    {?REG_REQUEST, Target, attach} ->
      case dict:find(Target, Reg) of
        {ok, RegNum} ->
          reply(Target, RegNum),
          loop(State);
        error ->
          case ordsets:to_list(Free) of
            [] ->
              reply(Target, server_full)
              loop(State);
            [RegNum|NewFreeList] ->
              NewReg = dict:store(Target, RegNum, Reg),
              reply(Target, RegNum),
              NewFree = ordsets:from_list(NewFreeList),
              NewState = State#state{free = NewFree, reg = NewReg},
              loop(NewState)
          end
        end;
    [...]
  end.
```

Our new tests passe successfully in Concuerror. The second test produces 822 interleaving sequences for a preemption bound of two. The current execution of our test contains a total of four processes—the initial process, the server process and two additional spawned processes. Let us change the value of `MAX_ATTACHED_PROC` to one, so that our test contains three processes. Leaving the preemption bound at two, there are now only 65 interleaving sequences produced. Trying the same with a `MAX_ATTACHED_PROC` equal to three and four 9,789 and 118,038 interleaving sequences are produced, respectively. This is a clear indication of the exponential dependence of the number of interleaving sequences on the number of interleaved processes.

Detaching processes

- ~~Start server~~
- ~~Stop server~~
- ~~Attach process~~
  - ~~Already attached~~
  - ~~Max proc reached~~
- **Detach process**
  - Detach on exit

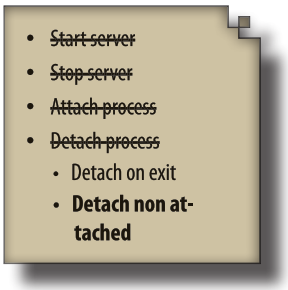
We have finished with our tasks concerning the attachment of processes and proceed to handling the detachment of processes from the server. When a process is detached from the server using a `detach/0` call, it should no longer have an assigned registration number, thus a `ping/0` call after its detachment should return `pong`. This is checked by the test of Listing 6.22. Moreover, the server should make the registration number available to be obtained by other processes in the future. To check this, we can extend

Listing 6.22: Test detaching a process from the server

```
detach_test() ->
  reg_server:start(),
  reg_server:attach(),
  reg_server:detach(),
  ?assertEqual(pong, reg_server:ping()),
  reg_server:stop().
```

the second test of Listing 6.20, so that when one of the processes is detached, another process becomes attached with the same registration number, which is the only one available at the time, as shown in the test of Listing 6.23. To make the tests pass, the server needs to remove the existing mapping from the dictionary and add the freed registration number back to the free set, as shown in Listing 6.24.

After having successfully run our latest tests and before proceeding to our final task, we have to deal with one more issue, namely handling the case of a process calling `detach/0` without being attached. Similarly to the case of the double `attach/0` call that we encountered previously, we can either ignore it and return `ok`, or consider it an error. Again, we choose the silent approach, as shown in the test of Listing 6.25. The fix is pretty obvious and is shown in Listing 6.26.



Listing 6.23: Test reattaching after detaching

```
detach_attach_test() ->
  Self = self(),
  reg_server:start(),
  L = lists:seq(1, ?MAX_ATTACHED_PROC),
  Ps = [spawn_attach() || _ <- L],
  LastProc = spawn(fun() ->
                     RegNum = reg_server:attach(),
                     reg_server:detach(),
                     Self ! RegNum,
                     receive ok -> ok end
                   end),
  receive RegNum -> ok end,
  ?assertEqual(RegNum, reg_server:attach()),
  lists:foreach(fun(Pid) -> Pid ! ok end, [LastProc|Ps]),
  reg_server:stop().
```

Listing 6.24: Add detach/0 to server

```
-module(reg_server).

-export([attach/0, detach/0, ping/0, start/0, stop/0]).

[...]

detach() ->
  request(detach).

[...]

loop(#state{free = Free, reg = Reg} = State) ->
  receive
    {?REG_REQUEST, Target, detach} ->
      RegNum = dict:fetch(Target, Reg),
      NewReg = dict:erase(Target, Reg),
      NewFree = ordsets:add_element(RegNum, Free),
      reply(Target, ok),
      NewState = State#state{free = NewFree, reg = NewReg},
      loop(NewState);
    [...]
  end.
```

Listing 6.25: Test trying to detach an unattached process

```
detach_non_attached_test() ->
  reg_server:start(),
  ?assertEqual(ok, reg:server:detach()),
  reg_server:stop().
```

- ~~Start server~~
- ~~Stop server~~
- ~~Attach process~~
- ~~Detach process~~
  - **Detach on exit**
  - ~~Detach non-attached~~

Currently an attached process is detached only when it calls `detach/0`. If an attached process exits without having been detached on its own, its registration number will remain occupied for ever. Consequently, our last task is to detach a process as soon as it exits. To check this we can modify the test in Listing 6.22 so that the last process being spawned, instead of explicitly detaching itself, simply terminates its execution. The main testing process tries to attach itself after receiving the `'EXIT'` message from this previously attached process. For the `'EXIT'` message to be received, a `process_flag/2` call is used to activate the `trap_exit` flag of the main process. The test is shown in Listing 6.27.

The server needs to know when a process exits, so as to take action and detach it. This means that we have to either use links or monitors to keep track of attached

Listing 6.26: Deal with trying to detach an unattached process

```
-module(reg_server).

[...]

loop(#state{free = Free, reg = Reg} = State) ->
  receive
    {?REG_REQUEST, Target, detach} ->
      case dict:is_key(Target, Reg) of
        false ->
          reply(Target, ok),
          loop(State);
        true ->
          [...]
      end;
    [...]
  end.
```

Listing 6.27: Test for detaching a process as soon as it exits

```
detach_on_exit() ->
  Self = self(),
  reg_server:start(),
  L = lists:seq(1, ?MAX_ATTACHED_PROC - 1),
  Ps = [spawn_attach() || _ <- L],
  process_flag(trap_exit, true),
  LastProc = spawn_link(fun() -> Self ! reg_server:attach() end),
  receive RegNum -> ok end,
  receive {'EXIT', LastProc, normal} -> ok end,
  ?assertEqual(RegNum, reg_server:attach()),
  lists:foreach(fun(Pid) -> Pid ! ok end, [LastProc|Ps]),
  reg_server:stop().
```

processes. There is no reason to use two-sided links here, thus we will have the server create a monitor for every process that gets attached. Also, as soon as the server receives a 'DOWN' message about an attached process, this process will be detached. In Listing 6.28 we implement the above and move the detachment operation into a separate function to avoid code duplication.

At this point, our final test passes successfully in Concuerror, as do all of our previous tests. We do not claim that our code is completely free of bugs. However, the tests that we have written check the basic functionality of our server and make us confident that any scenario that may occur in practice and is covered by our tests will actually work as expected.

Summing up this chapter, we can stress some important points that came up during the presentation of our example. First, for all of our tasks a preemption bound of two was enough to uncover concurrency related errors. It is usually convenient to

Listing 6.28: Detach a process as soon as it exits

```
-module(reg_server).

[...]

loop(#state{free = Free, reg = Reg} = State) ->
  receive
    {?REG_REQUEST, Target, attach} ->
      [...]
            [RegNum|NewFreeList] ->
            NewReg = dict:store(Target, RegNum, Reg),
            monitor(process, Target),
            reply(Target, RegNum),
      [...]
    {?REG_REQUEST, Target, detach} ->
      {Reply, NewFree, NewReg} = detach_proc(Target, Free, Reg),
      reply(Target, Reply),
      NewState = State#state{free = NewFree, reg = NewReg},
      loop(NewState);
    {'DOWN', _Ref, process, Target, _Info} ->
      NewState =
        case dict:is_key(Target, Reg) of
          true ->
            {ok, NewFree, NewReg} = detach_proc(Target, Free, Reg),
            State#state{free = NewFree, reg = NewReg};
          false -> State
        end,
      loop(NewState)
  end.

detach_proc(Target, Free, Reg) ->
  case dict:is_key(Target, Reg) of
    false -> {ok, Free, Reg};
    true ->
      RegNum = dict:fetch(Target, Reg),
      NewReg = dict:erase(Target, Reg),
      NewFree = ordsets:add_element(RegNum, Free),
      {ok, NewFree, NewReg}
  end.
```

start with a low preemption bound and gradually increase it for more thorough testing. Second, we saw the exponential increase in the number of interleaving sequences with respect to the number of processes. This suggests writing our tests to use only a few processes and generalize their results for an arbitrary number of them. Third, we saw that conventional testing using EUnit was not able to expose some of the errors we encountered. Given that our server was destined to be used in a highly concurrent environment, Concuerror not only allowed us to verify that our tests pass under some random interleaving, but gave us the significantly stronger guarantee that under *any* interleaving our program is still robust and correct with respect to our test suite. Last, besides revealing errors, Concuerror also helped us understand their cause by displaying detailed interleaving information. By walking through the erroneous interleaving sequences we were able to quickly understand and correct the source of each problem.

What's next?

The presentation of Concuerror has come to its end. The following chapter concludes this thesis with an overview of related work and a listing of ideas for future additions and enhancements.

7 Epilogue

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 [21], which verifies models expressed in the Promela language [20]. Automatic code translation to the modeling language has been proposed and used in tools like Bandera [12], which translates Java code into one of three modeling languages. Starting with the Verisoft model checker [18], several others, like Java PathFinder [35], CMC [31] and CHESS [30], 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.

Among the techniques used to ameliorate the state explosion problem are partial-order techniques [14, 17, 28], iterative context bounding [27] and even genetic programming [19]. The recent GAMBIT [11] extension to CHESS introduces best-first search based on heuristics that can be customized by the user.

Although Erlang is a concurrency-oriented language, there has not been much effort towards concurrency testing and verification. According to a recent survey [32], Dialyzer [25] and EUnit [5] 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 [7] and message passing errors [8] via static analysis. QuickCheck [2], a property-based testing tool for Erlang, has introduced a user-level scheduler named PULSE [9], which is able to detect some concurrency errors via random process interleaving. Besides the random nature of the testing procedure, whic 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 [22] and the McErlang model checker [15, 16]. 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-grained compared to Concuerror. The introduction of finer-grained preemption points requires the manual placement of commands, which is a strenuous task and, at the same time, alters the original code.

Conclusion and future work

In this thesis we have presented 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. The first indications of using Concuerror in practice look promising. Existing tests can be readily executed in Concuerror and the program under test does not need to be modified at all. Furthermore, Concuerror can be used with a low preemption bound during test-driven development to quickly discover concurrency-related errors, but it can also be used with a higher preemption bound for more thorough program validation, for example as part of overnight testing.

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

- **Further redundancy reduction**

  We have somewhat reduced the number of redundant interleaving sequences, but the actual program state-space is usually still significantly smaller than the number of interleaving sequences that are produced by Concuerror. Partial-order techniques are very common in the model checking literature, and we would like to see the results of using some of them to further reduce redundancy in Concuerror.

- **Full compatibility with EUnit**

  Some of EUnit's assertion macros can already be used with Concuerror. We intend to make Concuerror fully compatible with EUnit, so that users can execute any EUnit test suite in Concuerror with no changes required.

- **Selective instrumentation and layered testing**

  Currently, preemption points are only placed at a number of BIFs. Commonly used Erlang/OTP libraries have to be manually imported so that they get instrumented. To avoid this we can add pre-instrumented library modules to the Concuerror distribution and redirect user code calls to these modules. However, as we noted in p.32, this is not always the best choice. We would like to let users control the "interleaving granularity" by allowing them to choose between placing preemption points at library calls or using a (pre-)instrumented version of the library.

- **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.

- **Extention 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 ouside world (e.g. ports).

- **GUI improvement and visualization**

  The Concuerror GUI can be improved in many ways to become more usable. Some directions are project creation and management, test automation, and vi-

sualization of process interaction to further simplify the grasping and debugging of concurrency errors.

- **Memory overhead optimization**

  The way states are stored and retrieved has a large impact on Concuerror's memory overhead. Erlang ETS tables are currently used for storing states. More efficient state storage and compression schemes could result in a greatly reduced memory consumption.

BIBLIOGRAPHY

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

2. Arts, T., Hughes, J., Johansson, J., and Wiger, U. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 5th ACM SIGPLAN Workshop on Erlang* (New York, NY, USA, 2006), ACM, pp. 2–10.

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

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

5. Carlsson, R., and Remond, M. EUnit: A lightweight unit testing framework for Erlang. In *Proceedings of the 5th ACM SIGPLAN Workshop on Erlang* (New York, NY, USA, 2006), ACM, pp. 1–1.

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

7. Christakis, M., and Sagonas, K. Static detection of race conditions in Erlang. In *Practical Aspects of Declarative Languages: Proceedings of the PADL Symposium* (Berlin, Germany, Jan. 2010), M. Carro and R. Peña, Eds., vol. 5937 of *LNCS*, Springer, pp. 119–133.

8. Christakis, M., and Sagonas, K. Detection of asynchronous message passing errors using static analysis. In *Practical Aspects of Declarative Languages: Proceedings of the PADL Symposium* (Berlin, Germany, Jan. 2011), R. Rocha and J. Launchbury, Eds., vol 6539 of *LNCS*, Springer, pp. 5–18.

9. Claessen, K., Palka, M., Smallbone, N., Hughes, J., Svensson, H., Arts, T., and Wiger, U. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN Internation Conference on Functional Programming* (New York, NY, USA, 2009), ACM, pp. 149–160.

10. Clarke, E. M., Emerson, E. A., and Sifakis, J. Model checking: Algorithmic verification and debugging. *Commun. ACM 52* (November 2009), 74–84.

11. Coons, K. E., Burckhardt, S., and Musuvathi, M. Gambit: Effective unit testing for concurrency libraries. In *Proceedings of the ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2010), ACM, pp. 15–24.

12. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Păsăreanu, C., and Zheng, R. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering* (New York, NY, USA, 2000), ACM, pp. 439–448.

13. Engler, D., and Ashcraft, K. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles* (New York, NY, USA, 2003), ACM, pp. 237–252.

14. Flanagan, C., and Godefroid, P. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (New York, NY, USA, 2005), ACM, pp. 110–121.

15. Fredlund, L.-A., and Earle, C. B. Model checking Erlang programs: The functional approach. In *Proceedings of the 5th ACM SIGPLAN Workshop on Erlang* (New York, NY, USA, 2006), ACM, pp.11–19.

16. Fredlund, L.-A., and Svensson, H. McErlang: A model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2007), ACM, pp. 125–136.

17. Godefroid, P. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, vol. 1032 of *Lecture Notes in Computer Science.* Springer, 1996.

18. Godefroid, P. Model checking for programming languages using Verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (New York, NY, USA, 1997), ACM, pp. 174–186.

19. Godefroid, P., and Khurshid, S. Exploring very large state spaces using genetic algorithms. *International Journal on Software Tools for Technology Transfer (STTT) 6*, (2004), pp. 117–127.

20. Holzmann, G. J. *Design and Validation of Computer Protocols.* Prentice Hall, 1990.

21. Holzmann, G. J. The model checker SPIN. *IEEE Transactions on Software Engineering 23*, 5 (May 1997), pp. 279–295.

22. Huch, F. Verification of erlang programs using abstract interpretation and model checking. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 1999), ACM, pp. 261–272.

23. Johansson, E., Pettersson, M., and Sagonas, K. A high performance Erlang system. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and Practice of Declarative Programming* (New York, NY, USA, 2000), pp. 32–43.

24. Lamport, L. *Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21* (July 1978), pp. 558–565.

25. Lindahl, T., and Sagonas, K. Detecting software defects in telecom applications through lightweight static analysis: A war story. In *Programming Languages and Systems: Proceedings of the 2nd Asian Symposium* (Berlin, Germany, 2004), C. Wei-Ngan, Ed., vol. 3302 of *LNCS*, Springer, pp. 91–106.

26. Lindahl, T., and Sagonas, K. Typer: A type annotator of Erlang code. In *Proceedings of the 4th ACM SIGPLAN Workshop on Erlang* (New York, NY, USA, 2005), ACM, pp.17–25.

27. Musuvathi, M., and Qadeer, S. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), ACM, pp. 446–455.

28. Musuvathi, M., and Qadeer, S. Partial-order reduction for context-bounded state exploration. Tech. Rep. MSR-TR-2007-12, Microsoft Research, 2007.

29. Musuvathi, M., and Qadeer, S. Fair stateless model checking. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), ACM, pp. 362–371.

30. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), USENIX Association, pp. 267–280.

31. Musuvathi, M., Park, D. Y. W., Chou., A., Engler, D. R., and Dill, D. L. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th symposium on Operating Systems Design and Implementation* (2002), pp. 75–88.

32. Nagy, T., and Vig, A. N. Erlang testing and tools survey. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang* (New York, NY, USA, 2008), ACM, pp. 21–28.

33. Shore, J., and Chromatic. *The Art of Agile Development.* O'Reilly Media, 2007.

34. Spolsky, J. *The Best Software Writing I: Selected and Introduced by Joel Spolsky (v. 1).* Apress, 2005.

35. Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. Model checking programs. *Automated Software Engineering 10* (2003), pp. 203–232.