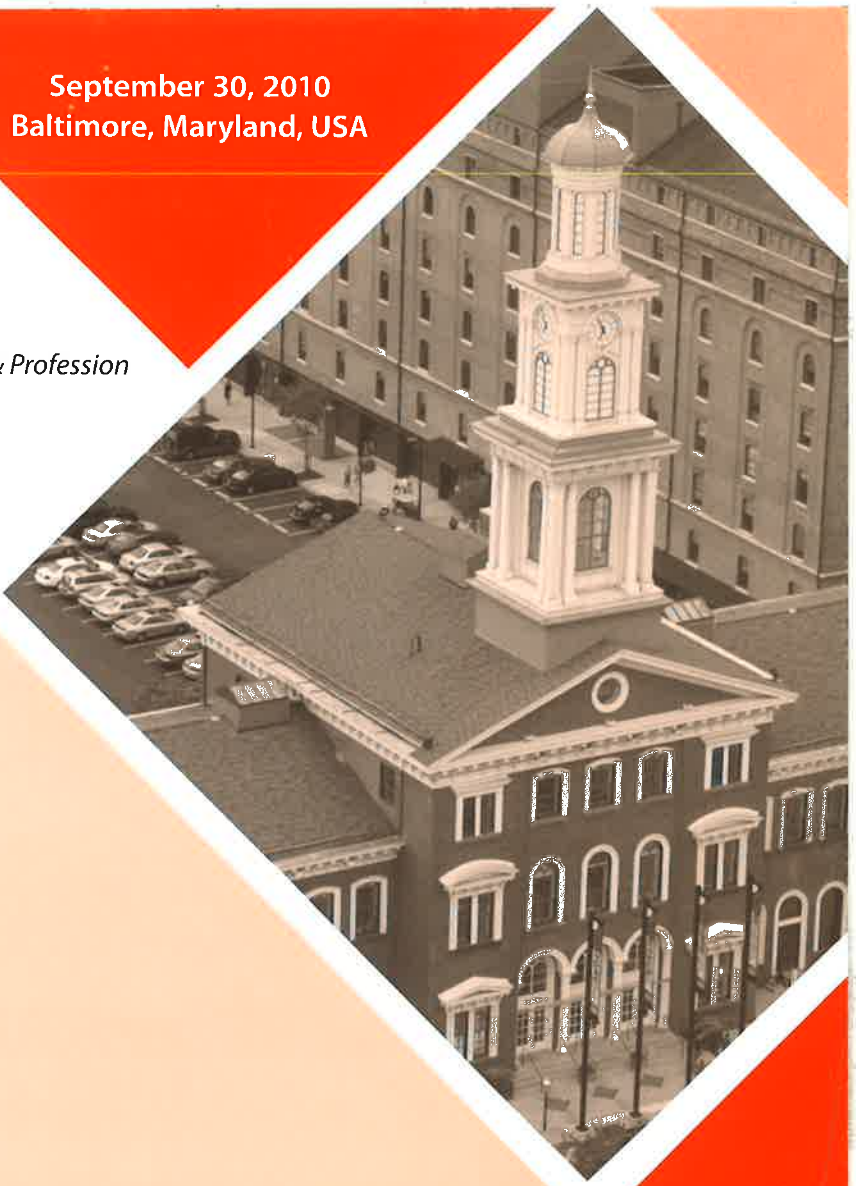
September 30, 2010
Baltimore, Maryland, USA

Association for
Computing Machinery

Advancing Computing as a Science & Profession



Erlang'10

Proceedings of the 2010 ACM SIGPLAN
Erlang Workshop

Sponsored by:

ACM SIGPLAN

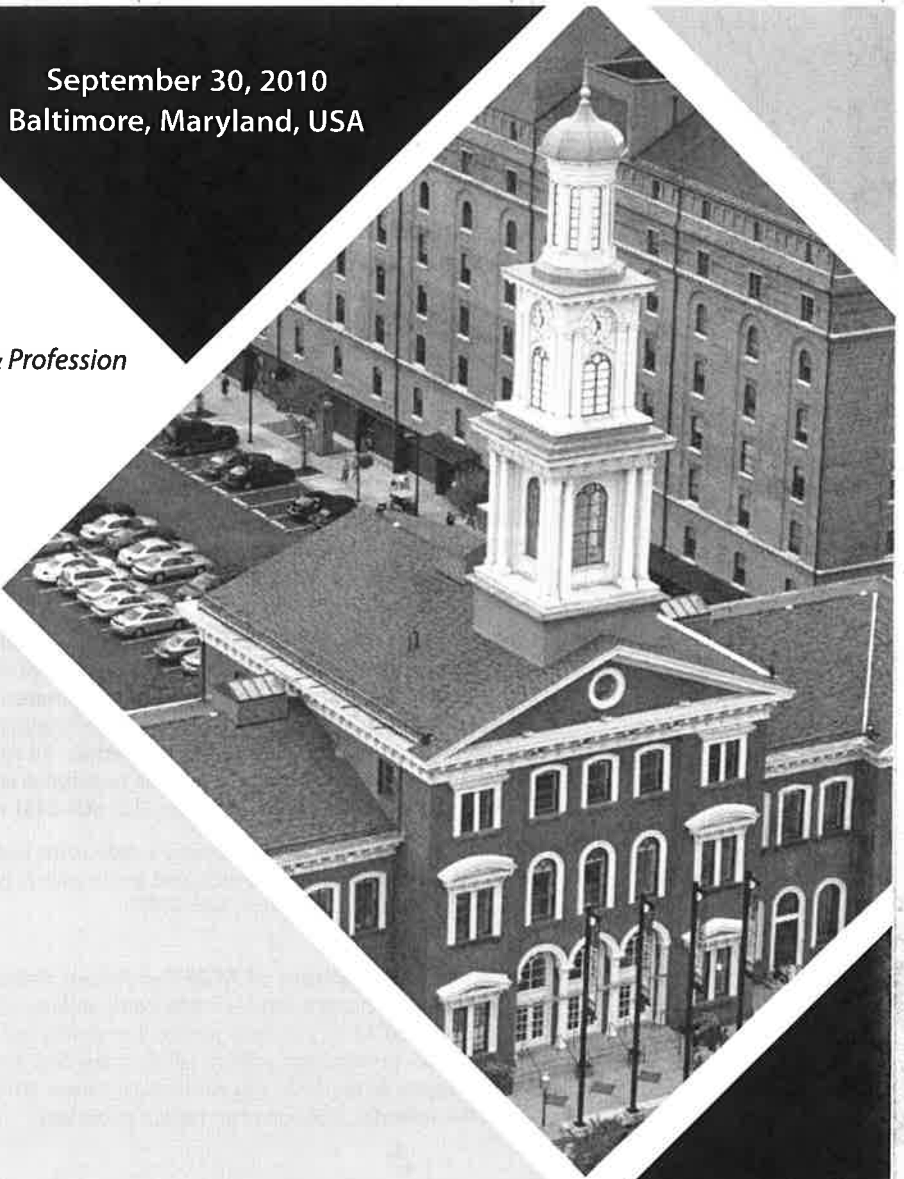
Co-located with:

ICFP'10

September 30, 2010
Baltimore, Maryland, USA

Association for
Computing Machinery

Advancing Computing as a Science & Profession



Erlang'10

Proceedings of the 2010 ACM SIGPLAN

Erlang Workshop

Sponsored by:

ACM SIGPLAN

Co-located with:

ICFP'10

Association for
Computing Machinery

Advancing Computing as a Science & Profession

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York, New York 10121-0701

Copyright © 2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212) 869-0481 or <permissions@acm.org>.

For other copying of articles that carry a code at the bottom of the first or last page, copying is permitted provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Notice to Past Authors of ACM-Published Articles
ACM intends to create a complete electronic archive of all articles and/or other material previously published by ACM. If you have written a work that has been previously published by ACM in any journal or conference proceedings prior to 1978, or any SIG Newsletter at any time, and you do NOT want this work to appear in the ACM Digital Library, please inform permissions@acm.org, stating the title of the work, the author(s), and where and when published.

ISBN: 978-1-4503-0253-1

Additional copies may be ordered prepaid from:

ACM Order Department
PO Box 11405
New York, NY 10286-1405

Phone: 1-800-342-6626 (USA and Canada)
+1-212-626-0500 (all other countries)
Fax: +1-212-944-1318
E-mail: acmhelp@acm.org

ACM Order Number 553102

Printed in the USA

Foreword

It is our great pleasure to welcome you to the 9th *ACM SIGPLAN Erlang Workshop – Erlang 2010,* the annual forum for the presentation of research, theory, implementation, tools and applications of the Erlang programming language. This year's workshop continues the tradition of being co-located with the annual ACM SIGPLAN International Conference on Functional Programming (ICFP).

The call for papers attracted 12 submissions which were reviewed by at least three PC members. The program committee decided to accept seven of these submissions as full papers and two more as short talks followed by a demo during a special demo session during the workshop. These papers cover a variety of topics, including semantics, behavioral and distributed programming in Erlang, testing tools and test driven development, protocol implementations and language issues.

Putting together *Erlang 2010* was a team effort. First of all, we would like to thank the authors of submitted papers and demos for providing the content of the program. We would like to express our gratitude to the program committee and the three additional reviewers, who worked very hard under time pressure in reviewing papers, providing constructive criticism and concrete suggestions for their improvement. We would also like to thank Derek Dreyer and Christopher Stone, this year's ICFP Workshop Chairs, for coordinating all workshops and for putting up with some occasional silences from our part. As usual, special thanks go to members of the workshop's Steering Committee for their guidance and to Simon Thompson in particular. Finally, we appreciate ACM SIGPLAN's continued support and we thank all sponsors of ICFP 2010.

We hope that you will find this program interesting and that the workshop will provide you with a valuable opportunity to share ideas with other researchers, implementors and Erlang practitioners from academia and industry.

Konstantinos Sagonas
Erlang 2010 Program Chair
Nat. Tech. Univ. of Athens, Greece

Scott Lystig Fritchie
Erlang 2010 General Chair
Gemini Mobile Technologies Inc., USA

Table of Contents

Erlang 2010 Workshop Organization

General Chair: Scott Lystig Fritchie *(Gemini Mobile Technologies, USA)*

Program Chair: Konstantinos Sagonas *(National Technical University of Athens, Greece)*

Steering Committee: Bjarne Däcker *(Independent Telecoms Professional, Sweden)*
Clara Benac Earle *(Universitad Polytéchnica de Madrid, Spain)*
Lars-Åke Fredlund *(Universitad Polytéchnica de Madrid, Spain)*
Zoltán Horváth *(Eötvös Loránt University, Hungary)*
Simon Thompson *(University of Kent, UK)*
Tee Teoh *(Canadian Bank Note, Canada)*

Program Committee: Danny Dubé *(Université Laval, Canada)*
Scott Lystig Fritchie *(Gemini Mobile Technologies, USA)*
Garry Hodgson *(AT&T, USA)*
Zoltán Horváth *(Eötvös Loránt University, Hungary)*
Mickaël Rèmond *(Process One, France)*
Konstantinos Sagonas *(National Technical University of Athens, Greece)*
Erik Stenman *(Klarna AB, Sweden)*
Hans Svensson *(Chalmers University of Technology, Sweden)*
Simon Thompson *(University of Kent, UK)*
Ulf Wiger *(Erlang Solutions Ltd, UK)*

Additional reviewers: István Bozó
Máté Tejfel
Melinda Tóth

Sponsor: SIGPLAN

Supporters: CREDIT SUISSE
Erlang SOLUTIONS
galois
JANE STREET CAPITAL
Microsoft Research
Standard Chartered

From Test Cases to FSMs: Augmented Test-driven Development and Property Inference

Thomas Arts
Chalmers / Quviq AB, Gothenburg, Sweden
thomas.arts@ituniv.se

Simon Thompson
University of Kent, Canterbury, UK
S.J.Thompson@kent.ac.uk

Abstract

This paper uses the inference of finite state machines from EUnit test suites for Erlang programs to make two contributions. First, we show that the inferred FSMs provide feedback on the adequacy of the test suite that is developed incrementally during the test-driven development of a system. This is novel because the feedback we give is *independent* of the implementation of the system.

Secondly, we use FSM inference to develop QuickCheck properties for testing state-based systems. This has the effect of transforming a fixed set of tests into a property which can be tested using randomly generated data, substantially widening the coverage and scope of the tests.

Categories and Subject Descriptors D. Software [*D.2 SOFTWARE ENGINEERING*]: D.2.5 Testing and Debugging: Testing tools

General Terms Verification

Keywords TDD, test-driven development, Erlang, EUnit, unit test, QuickCheck, property, inference, finite-state machine

1. Introduction

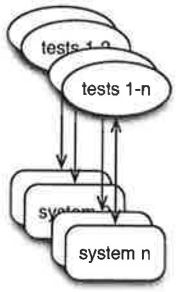
In this paper we show how finite state machines can be automatically extracted from sets of unit tests – here Eunit [6] tests for Erlang programs. We use these FSMs in two ways. First, they can in themselves provide feedback on the adequacy of a set of tests, independently of any implementation. Secondly, they can be transformed and used within Quviq QuickCheck [1, 14] to guide the random generation of test sequences for state-based systems. We discuss these contributions in turn now.

Test-driven Development

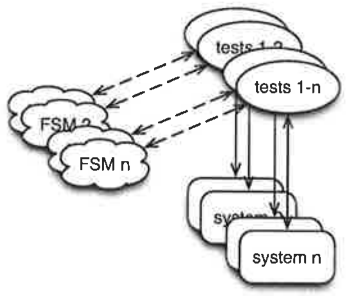
Test-driven development [3, 4] (TDD) advocates that tests should precede implementations. Systems should be developed incrementally, with each increment delivering enough functionality to pass another test, as illustrated here.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09...$10.00



Under this approach, how can we validate the system? It will surely meet the tests, because it has been developed precisely to pass them. The question becomes one of *validating the tests themselves*. In this paper we propose that during TDD of state-based systems we can validate the tests by extracting the finite state machine (FSM) implicit in the current test set.



The FSM is extracted by means of grammar inference [23] over sets of positive and negative traces. This FSM provides feedback on the tests *independently* of any implementation, and thus 'triangulates' the process.

We would argue that that makes the process of test-driven development more robust. In particular, it allows us to give an answer to the question *"When have I written enough tests?"* on the basis of the tests alone, rather than by examining an implementation. We return to this question in Section 5.

We illustrate our approach to TDD by working through a case study in Erlang, developing unit tests in EUnit, and using StateChum [19] to extract a series of FSMs from the test suite as it evolves.

Testing state-based systems

This work was developed within the European Framework 7 ProTest project [18] to develop property-based testing for Erlang.

In particular we seek to develop QuickCheck[1] properties from sets of unit tests, thus providing a migration path from traditional unit testing to property-based testing. To test state-based systems in QuickCheck it is usual to develop a state machine model (using `eqc_fsm` or `eqc_statem`) which encapsulates the permissible sequences of API calls.

We show how the FSM extracted from a set of unit tests can be transformed into a QuickCheck FSM, and thus how a set of unit tests can be combined into a property. This has the benefit of allowing the system to be tested on many more inputs, namely all those permissible in the FSM, and a selection of these can be generated randomly using QuickCheck generators.

Abstraction

In modelling a system using a finite state machine we need to perform abstraction over the state data. In the case study the data consist of a finite collection of resources, and this is modelled by sets with small cardinalities before we seek to identify a general case. In the case of test development, this allows us to identify complete sets of tests for 'small' models before moving to the general case. In QuickCheck this process identifies candidate state data values.

Roadmap

We begin by discussing the background to this work in Section 2. We first introduce test-driven development, and then discuss EUnit and QuickCheck for testing Erlang systems. We also look at grammar inference as a mechanism for inferring finite-state machines from sets of words in the language and its complement. We use StateChum to do FSM inference in our case study.

Section 3 discusses a systematic approach to developing and assessing tests during test-driven development through the case study of a 'frequency server'. We use Eunit to express the tests and StateChum to infer finite state machines from test sets in a fully automated way. While doing this we discuss the question of how to abstract away from particular aspects of the system in forming a model of the system under test.

Section 4 builds on this by developing a QuickCheck state machine for the example. This machine is based on the FSM inferred in the previous section, and we discuss the process of building the QuickCheck machine from this FSM with a view to automating the process as much as possible in the future.

Finally we discuss related work in Section 5 and draw some conclusions in Section 6.

2. Background

In this section we give a brief overview of the main topics which form the background to the work reported this paper, as well as providing references where more information can be found.

2.1 Test-driven development

A manifesto for test-driven development (TDD) is given in Beck's monograph [4]. This gives advice on adopting TDD in practice, as well as answering frequently-asked questions. The thesis of test-driven development is that it is the formulation of tests which should be used to drive the development process.

Specifically, the requirements for the system are given by a series of tests developed incrementally. At each stage the implementor will write enough to satisfy the existing tests and (in theory at least) nothing more. Hence the importance of the tests in specifying the system, and so the importance of finding mechanisms by which the tests can be validated in some independent way. In Section 5 we compare our approach to others in the TDD community.

[1] In this article, QuickCheck refers to Quviq QuickCheck version 1.18

2.2 EUnit

EUnit [5, 6] provides a framework for defining and executing unit tests, which test that a particular program unit – in Erlang, a function or collection of functions – behaves as expected. The framework gives a representation of tests of a variety of different kinds, and a set of macros which simplify the way EUnit tests can be written.

For a function without side-effects, a test will typically look at whether the input/output behaviour is as expected, and that exceptions are raised (only) when required.

Functions that have side-effects require more complex support. The infrastructure needed to test these programs (called a *fixture*) includes a facility to setup a particular program state prior to test, and then to cleanup after the test has been performed.

Examples of EUnit tests are given in the body of the paper, and are explained as they occur. The text [7] gives an introduction to EUnit testing; further details can be found in [5, 6] and the online documentation for the system.

2.3 Grammar and state machine inference

The StateChum tool extracts a finite state machine from sets of positive and negative instances [19]. That is, the user provides sets of words which are in (resp. out) of the language of the state machine, and grammar inference techniques are used to infer the minimal machine conforming to this requirement.

The algorithm uses a *state merging* technique: first the (finite) machine accepting exactly the positive cases is constructed, then states are merged in such a way that no positive and negative states are identified. The particular implementation assumes that the language accepted is prefix-closed, so that in terms of testing a single positive case can be seen as representing a number of positive unit tests. Further details of the algorithm are in [23, 24].

FSM and grammar inference is a well-established field: an early introduction can be found in [22].

2.4 QuickCheck

QuickCheck [1, 14] supports random testing of Erlang programs. Properties of the programs are stated in a subset of first-order logic, embedded in Erlang syntax. QuickCheck verifies these properties for collections of Erlang data values generated randomly, with user guidance in defining the generators where necessary.

When testing state-based systems it makes sense to build an abstract model of the system, and to use this model to drive testing of the real system. Devising this model is crucial to the effective testing, and the technique outlined in this paper facilitates model definition from existing test data rather than from an informal description of the system under test.

QuickCheck comes with a library (`eqc_fsm`) for specifying test cases as finite state machines. For each state in the FSM it is necessary to describe a number of things.

- The possible transitions to other states.
- A set of preconditions for each transition.
- How to actually perform the transition (that is, a function that performs whatever operations are necessary).
- Postconditions to make a check after the state transition.
- A description of the changes on the state as a result of the transition.

This information is supplied by defining a set of callback functions; we will see an example of this in practice in Section 4.

3. Test-driven development

In this section we introduce a procedure for systematically developing the unit tests that are used in the test-driven development process of systems. This is illustrated through the running example of a simple server.

3.1 Example: a frequency server

As a running example (taken from [7]) we write tests for a simple server that manages a number of resources – frequencies for example – which can each be allocated and deallocated. The server process is registered to simplify the interface functions, so that it is not necessary to use the process identifier of the server to communicate with it. The Erlang type specification for the interface functions is as follows:

```
-spec start([integer()]) -> pid().
-spec stop() -> ok.
-spec allocate() -> {ok,integer()} |
                    {error, no_frequency}.
-spec deallocate(integer()) -> ok.
```

The `start` function takes a list of frequencies as argument and spawns and registers a new server that manages those frequencies. The `stop` function communicates with the server to terminate it in a normal way.

The `allocate` function returns a frequency if one is available, or an error if all frequencies have already been allocated. The `deallocate` function takes a previously allocated frequency as argument and has the server release that frequency.

3.2 Testing start/stop behaviour

As straightforward as this server seems to be, it is still a good idea to define some tests before we write the code. We use EUnit [6] as a framework for writing our unit tests, but the principles in this paper apply however we write unit tests.

We start by defining tests for starting and stopping the server, not worrying about allocation and deallocation. Of course we want a test in which we start and stop the server, but we also want to test that we can start it again after stopping. Since the second test subsumes the first, we only define the second.

```
startstop_test() ->
    ?assertMatch(Pid1 when is_pid(Pid1),start([])),
    ?assertMatch(ok,stop()),
    ?assertMatch(Pid2 when is_pid(Pid2),start([1])),
    ?assertMatch(ok,stop()).
```

We start the server twice, each with a different list of resources, more or less an arbitrary choice. The second call to `stop` is performed to clean up and return to the state in which no server is registered. Note that we match the returned values of the calls, viz. `ok` for stop and a pid for `start`, precisely as required by the specification.

Note that although we have defined this as a single EUnit test, it can also be seen as representing four separate tests, one performed by each `?assertMatch` expression. The four tests check that that the system can be started, that it can be started and then stopped, and so forth: one test case for each prefix of the sequence of `?assertMatch` statements.

Now we would be able to write our first prototype, but it is obvious that if we write `start` and `stop` to just return the correct return types, then the test would pass. This indicates that we have too few tests for a proper test-driven development of a non-trivial server. How do we find out which additional tests to add?

One answer is to appeal to our programmers' intuition, but a more satisfactory – and principled – approach is to look at the set of tests and see what state space is implicit in these. More specifically, we can extract the minimal finite state machine (FSM) from the traces, and then judge the adequacy of the resulting FSM in modelling the proposed system, thereby assessing the tests themselves.

```
passive
config debugMode true
+ start stop start stop
```

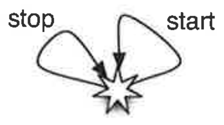


Figure 1. Start/stop behaviour: first model

3.3 Visualizing the state machine

In this section we demonstrate how we can use the StateChum library [19], to improve our set of unit tests by generating a Finite State Machine which represents the minimal FSM implicit in the tests. Inspecting that FSM allows us to decide which tests should be added (or indeed removed) in order to make the state space correspond to the intended model, and thus to establish the completeness of the test data set.

Translating EUnit tests to sequences

In order to use StateChum on a set of given EUnit tests, we need an algorithm to translate EUnit tests to sequences that are given as input to StateChum. The translation we start of with is to replace each `?assertMatch(`$Result$`,`$Fun(A_1, ..., A_n)$`)` in a test by the function name Fun to obtain a sequence of function calls. In particular, the `startstop_test()` above is translated into the sequence:

```
+ start stop start stop
```

where the leading '+' indicates that this is a *positive trace*, that is a trace that is to be accepted by the inferred FSM.

Note also that the algorithm used by StateChum assumes that the positive traces are closed under initial segments, so that the single trace is in fact equivalent to

```
+ start
+ start stop
+ start stop start
+ start stop start stop
```

Finally it should be noted that the transformation from the EUnit tests to the StateChum input can be fully automated.

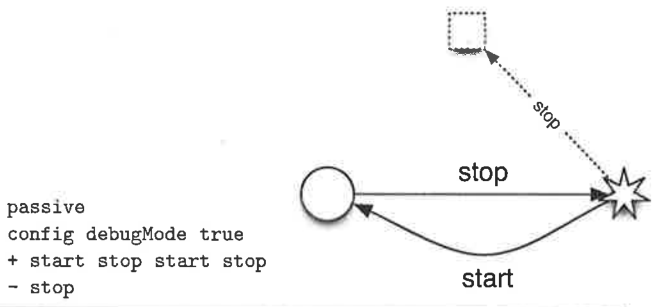
3.4 Using the derived FSM to assess tests

In order to use StateChum on our example, we need to abstract from the data part in our test case and concentrate on the sequence of function calls performed. This sequence is input to StateChum and this input together with the derived FSM is shown in Fig. 1.

This figure indicates that there is a single state in which it is possible both to start and to stop the server. Starting and stopping the server don't result in a state change; at least, not on the basis of this single test case. In particular, the picture suggests that one can successfully perform a stop in the initial state, and also start the system twice.

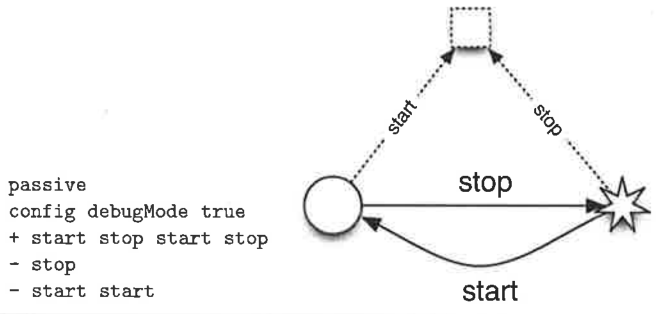
In order to make two distinguishable states we need to supply StateChum with two *negative sequences*, which correspond to two negative test cases, that is, test cases that result in erroneous behaviour of some kind. The first test case verifies that one cannot stop in the initial state. This is added to the input for StateChum by adjoining the line `- stop`. After doing so, we observe an FSM with three states, depicted in Fig. 2.

In the initial state – indicated by a *starred* icon – a `stop` leads to the error state and a `start` leads to a second state. From that



```
passive
config debugMode true
+ start stop start stop
- stop
```

Figure 2. Start/stop behaviour: second model



```
passive
config debugMode true
+ start stop start stop
- stop
- start start
```

Figure 3. Start/stop behaviour: final model

second state there is a `stop` transition back to the initial state, but no further transitions. The third state is a 'dead state' – denoted by a *square* – and this is the result of a `stop` move from the initial state. The two traces: one negative and one positive, are insufficient to predict what happens when a `start` call is made in the second state.

So, we need to add another negative test case, stating that starting a system that is already running will result in an error. The new FSM derived is shown in Fig. 3, where we now have an extra transition to the error state. This picture describes the complete behaviour of the `start` and `stop` functions and all sequences thereof. Therefore, we are justified in concluding that the set containing one positive and the two negative tests is sufficient for testing the start/stop aspects of the system behaviour.

3.5 Writing negative tests in EUnit

When writing negative tests, we can either choose to specify which exception should occur, or just match on any possible exception. Many testers like the first alternative best, since one also tests whether the code fails for the right reason. However, in our case the reason is not specified, and, by adding it to the test we limit the possibilities in the design.

For example, if we were to decide that an initial `stop` raises an exception with reason `not_running` and were then to decide to implement the server using the standard OTP behaviour `gen_server`, then the error generated by the implementation would be a `noproc` exception rather than `not_running`, and so the negative test would fail. We could change the exception sought, but rather than overspecify the error exception, we choose the second alternative above and match on any possible exception.

```
stop_without_start_test() ->
     ?assertException(_,_,stop()).

start_twice_test_() ->
     {setup,
      fun() -> start([]) end,
      fun(_) -> stop() end,
      fun() -> ?assertException(_,_,start([])) end}.
```

If we stop a non running server, an exception is raised and starting an already running server similarly raises an exception. The reason for writing the last test case as a test generator[2] with set-up code, clean-up code and actual test code is that EUnit raises an exception as soon as the `?assertException` would fail, e.g., when the second start succeeds. In such cases one still wants to clean up and stop the already running server.

Translating EUnit tests to sequences

Although we used StateChum to derive a full set of tests by first supplying the negative sequences and then writing the additional test cases, we still strive after a translation from EUnit tests to these sequences. We extend therefore our translation in such a way that any command sequence in EUnit that ends with an `?assertException` is a negative tests and the translation of these assertions is given by:

$[\texttt{?assertException(}E1\texttt{,}E2\texttt{,}Fun\texttt{(}A_1,...,A_n\texttt{))}] \rightarrow Fun$
$[\{\texttt{setup,}$
    $\texttt{fun() -> } InitSeq \texttt{ end,}$
    $\texttt{fun() -> } StopSeq \texttt{ end,}$
    $\texttt{fun() -> } Seq \texttt{ end}\}] \rightarrow [InitSeq\ Seq]$

Thus ignoring the cleanup code and assuming at most the last assertion is an exception assertion, which determines the test to be a negative sequence.

Taking the examples given at the start of this subsection we generate the sequences:

```
- stop
- start start
```

as shown in Figure 3.

3.6 Initial implementation: start/stop behaviour

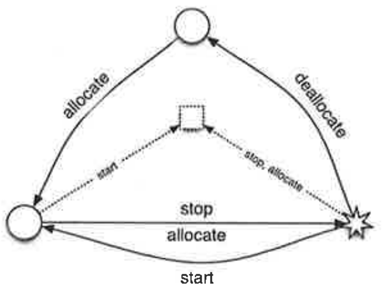
We can now run the tests and all three fail with notifications pointing to the fact that `start` and `stop` are as yet undefined! We now write the code for starting and stopping the server.[3] thus:

```
start(Freqs) ->
   {ok,Pid} =
     gen_server:start({local,?SERVER},
                      ?MODULE,Freqs,[]),
   Pid.

stop() ->
   gen_server:call(?SERVER,stop).

%% callbacks
init(Freqs) ->
    {ok, Freqs}.

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

All tests pass and by having seen the correspondence between the test cases and the FSM in Fig. 3, we have strong confidence that adding more tests is superfluous and that we can proceed with

[2] Note the subtle addition of '_' after the function name, which transforms a direct test into a test generator. See [6] for details.

[3] An alternative implementation of the system is provided in Ch. 5 of [7].



```
passive
config debugMode true
+ start stop start stop
- stop
- start start
+ start allocate deallocate allocate stop
- start allocate allocate
```

Figure 4. Single frequency: second model

specifying the tests for the additional functionality of allocating and deallocating frequencies.

3.7 Defining tests for a data-dependent state space

After having defined test cases for starting and stopping the server, we would now like to allocate and deallocate frequencies. Whether or not allocation succeeds depends on the number of frequencies that are available. In other words, depending on how many frequencies we start with and how many allocations we perform, we get different successful and failing test cases.

Starting by defining a set of test cases for four frequencies would immediately result in a large number of possible allocation and deallocation scenarios, let alone taking a realistic example of several hundreds of resources. We therefore start by defining the tests for systems with one and two frequencies available and make sure that we get a complete set of tests for each of these, trusting that we can generalise from these to the general case.

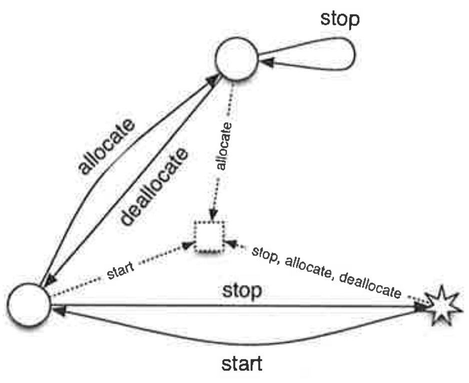
3.8 A single frequency

A typical test case would be to allocate a frequency and then deallocate it. Another typical test case would be to allocate it once more after deallocation. Since the first test case is subsumed in the second one, we only write the second.

```
alloc_dealloc_alloc_test_() ->
  {setup,
      fun() -> start([1]) end,
      fun(_) -> stop() end,
      fun () ->
         ?assertMatch({ok,1}, allocate()),
         ?assertMatch(ok,deallocate(1)),
         ?assertMatch({ok,1}, allocate())
     end
      }.
```

Note that the frequency value 1 used here is arbitrary. (We assume it to be more likely to find an error in the implementation by adding more different scenarios than by trying more different values for the specific frequencies.) This test must allocate the same value twice since there is only one value to be allocated.

We use StateChum again to visualize the FSM, which is equivalent to Fig. 3 with the addition of an arbitrary allocations and deallocations after starting the server. So, we do not capture the fact that it is possible to allocate all available frequencies and that an error is returned in that case. In order to add a general test case for the exhaustion of frequencies, we need to know how many frequencies there are. We propose to get the tests right for one frequency first, then take the two frequency case and see if we can generalise from there.



```
passive
config debugMode true
+ start stop start stop
- stop
- start start
+ start allocate deallocate allocate stop
- start allocate allocate
- deallocate
- allocate
```

Figure 5. Single frequency: third model

Using StateChum we can quickly observe what happens if we add a negative test for allocating two frequencies in case we only have one. The result is shown in Fig. 4 and it is immediately clear that we have to add a few more test cases to make a sensible picture out of this FSM.

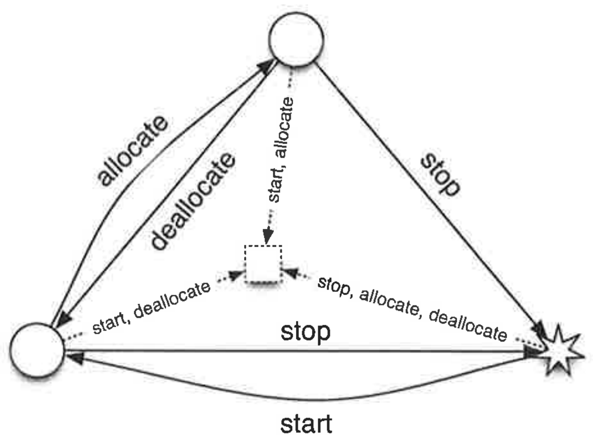
According to Fig. 4, from the initial state we can perform a `deallocate` and then an `allocate`. We need to exclude that possibility by stating that deallocation (and indeed also allocation) can only be done after a `start`; this results in Fig. 5.

In the FSM of Fig. 5 a `start` can only be followed by an `allocate`, which after deallocation allows a new allocation. The only strange part is that one can `stop` indefinitely often after allocation; one would like instead to have a `stop` transition back to the initial state. In fact, it is good to observe this in a visualisation of a state space, since it is domain-dependent whether or not one would allow a server that allocates frequencies to just stop or that one would need to deallocate the frequencies first. In Erlang it is most natural to perform the deallocation as side-effect of stopping. We add a test to ensure that we can start again after stopping with one allocated resource.

The tests[4] added for the server with one frequency are:

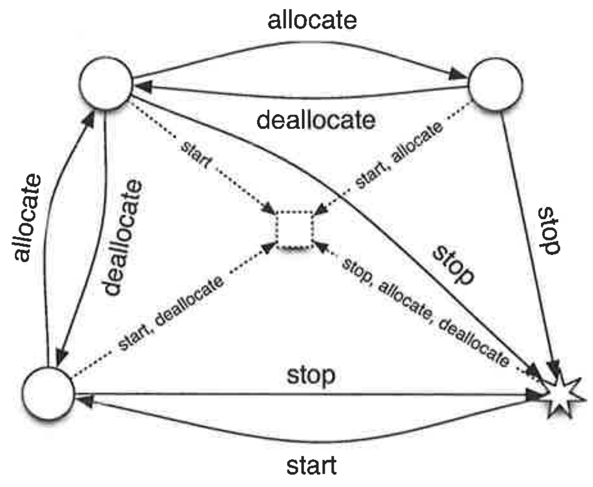
```
allocate_without_start_test() ->
    ?assertException(_,_,allocate()).

deallocate_without_start_test() ->
    ?assertException(_,_,deallocate(1)).

running_server_test_() ->
    {foreach,
     fun() -> start([1]) end,
     fun(_) -> stop() end,
     [fun() ->
       ?assertMatch({ok,1} ,allocate()),
       ?assertMatch(ok,deallocate(1)),
```

[4] EUnit allows to combine a few of these tests with the `foreach` primitive instead of `setup`



```
+ start stop start stop
- stop
- start start
+ start allocate deallocate allocate stop
- start allocate allocate
- allocate
- deallocate
+ start allocate stop start
- start deallocate
- start allocate start
```

Figure 6. One frequency: final model and StateChum test set



```
+ start stop start stop
- stop
- start start
+ start allocate allocate deallocate allocate
  deallocate deallocate stop
- start allocate allocate allocate
- allocate
- deallocate
+ start allocate stop start
+ start allocate allocate stop start
- start allocate deallocate deallocate
- start allocate start
- start allocate allocate start
```

Figure 7. Two frequencies: final model and StateChum test set

```
      ?assertMatch({ok,1},allocate())
    end,
   fun() ->
      ?assertMatch({ok,1} ,allocate()),
      ?assertMatch({error,no_frequency} ,allocate())
   end,
   fun() ->
      ?assertMatch({ok,1} ,allocate()),
      ?assertMatch(ok,stop()),
      ?assertMatch(Pid when is_pid(Pid),start([1]))
   end]}.
```

Note that in the above test cases we use domain knowledge to interpret the error value returned from allocation as a negative test case, expressing the condition that starting the server and performing two allocations is impossible. Were we to be given an API for our frequency server that raised an exception for a failing allocation, then the test case would be identified as a negative test case much more easily.

At this point we could conclude, if we were confident that all the transitions shown are as expected. However, the StateChum tool diagnostics for this input are:

```
#Prescribed: 5
#Proscribed: 5
#Unknown: 2
```

This output states that, of the twelve possible transitions in the machine, five make a transition to an accepting state and another five to the dead state: two transitions are as yet undetermined. The two transitions in question are: whether it is possible to deallocate before any allocation, and, whether it is possible to start the machine again after the one frequency is allocated.

We can rule these out with two negative test sequences that come at the end of the complete set of cases listed in Fig. 6 and these generate the machine in that figure. The data might appear to be skewed in favour of the negative tests: there are 7 negative and 3 positive tests. However, noting the prefix-closure property of the positive tests, we can see these three tests as embodying 10 distinct positive test cases, and under this interpretation we have of the same order of positive and negative tests.

Translating EUnit tests to sequences

We need to extend the translation of EUnit tests to the `foreach` construct, which is equivalent to the translation of several `setup` commands. In addition we have to add that an assertion that matches an error produces a negative sequence. As explained, this is somewhat controversial and probably one would like to enforce the design to raise an exception instead.

3.9 Two frequencies

Now we look at the case where there are two frequencies to be allocated, and develop a set of tests along the lines of the one frequency machine in Section 3.8. The set of tests – described in StateChum input format – are shown in Fig. 7.

The greyed-out tests are identical to the previous case, while the other tests are developed by a similar process to that in Section 3.8. Counting distinct prefixes as separate tests, we have 15 positive tests and 8 negative ones. A number of the later tests are included to avoid loops, such as looping on stop behaviour rather than returning the system to the start state when it is stopped; others are to prevent starting a system that is already running, whatever state it is in.

Note that in the EUnit tests the specific frequency that we allocate and deallocate was not significant when there was just one frequency available. However, now that we have two frequencies to choose from, a choice has to be made about which frequency is to be allocated. Now we have either to specify in our test case how the algorithm implements the choice, or to abstract away from the allocation algorithm. In EUnit tests this difference manifests itself as the difference between the following two test cases. In the first case, the test requires an implementation that takes frequencies from the head of the list:

```
twofreq_server_test_() ->
    {setup,
     fun() -> start([1,2]) end,
     fun(_) -> stop() end,
     fun() ->
       ?assertMatch({ok,1} ,allocate()),
       ?assertMatch({ok,2},allocate()),
       ?assertMatch(ok,deallocate(2)),
       ?assertMatch({ok,2},allocate()),
       ?assertMatch(ok,deallocate(1)),
       ?assertMatch(ok,deallocate(2))
      end}.
```

The alternative is a test that does not enforce any order on the allocation of frequencies:

```
twofreq_server_test_() ->
    {setup,
     fun() -> start([1,2]) end,
     fun(_) -> stop() end,
     fun() ->
       ?assertMatch({ok,F1} ,allocate()),
       ?assertMatch({ok,F2},allocate()),
       ?assertMatch(ok,deallocate(F2)),
       ?assertMatch({ok,F3},allocate()),
       ?assertMatch(ok,deallocate(F1)),
       ?assertMatch(ok,deallocate(F3))
      end}.
```

The latter test seems preferable in a test-driven development process, since it does not over-specify implementation details. Moreover, if the set of frequencies is extended to contain more than two frequencies, the test makes still sense without having to re-evaluate how the choice of frequencies is actually implemented. In this case, it is likely that re-use of frequencies is preferred to assigning as-yet-unused frequencies.

3.10 Data abstraction

With the translation of EUnit tests to sequences for StateChum we abstract from the data in the EUnit test cases. According to the API of the frequency server, the `start` and `deallocate` operations are parameterised by a list of frequencies and the frequency to be deallocated, respectively. These parameters play different roles.

- The list parameter is the `start` value for the particular run of the server, and it can be any legitimate integer list; of course its size will constrain the behaviour of the system, but the call to `start` is bound to succeed if and only if the system is not already running. This (pre-)condition is encapsulated in the structure of the FSMs seen in Figs. 6 and 7.
- On the other hand, the parameter to `deallocate` is assumed to be a frequency that is already allocated. This condition is not something that can be modelled in the FSM without 'hard wiring' the set of frequencies into the FSM itself. Supposing that there are n frequencies available, this would give rise to some 2^n states, each one representing a different subset of the n states having been allocated.

So, we can safely abstract in our EUnit tests from the specific frequency that is returned by `allocate`, i.e. we do not need to know the exact allocation algorithm. But, we cannot easily abstract from the specific frequency that is passed to `deallocate`; that frequency has to be remembered in our test case. Therefore, the abstraction

```
- start allocate deallocate deallocate
```

is only a valid abstraction if both deallocations refer to the same frequency. This means that the translation from EUnit test cases to sequences that we have developed fails in some cases, such as this:

```
twofreq_server_test_() ->
    {setup,
     fun() -> start([1,2]) end,
     fun(_) -> stop() end,
     fun() ->
       ?assertMatch({ok,F1} ,allocate()),
       ?assertException(_,_,deallocate(3-F1))
      end}.
```

One solution would be the 'hard-wiring' of the frequencies discussed earlier, which would involve two allocation operations, `allocate1` and `allocate2` and two deallocation operations, `deallocate1` and `deallocate2`. However the state machine resulting from that approach suffers from an exponential state explosion (as described earlier).

Instead we use another abstraction. We can 'loosen' our model, so that `deallocate(N)` can be applied whether or not `N` has been allocated or not. A problem with this is that this makes the FSM non-deterministic, since in the case that `N` is not already allocated the result of the transition will be that the set of available states is unchanged.

We can then interpret an exception for a deallocation as a possibility in a positive sequence, which is similar to changing the API for the `deallocate` function so that

```
-spec deallocate(integer()) -> ok | error.
```

with the `error` result indicating that no actual reallocation has taken place, because the argument frequency was not allocated. We can then distinguish the normal termination and error termination by translating the EUnit

```
?assertMatch(error,deallocate(...))
```

into a `failDA` operation. This would restore determinism in the model. Taking this approach, we add the `failDA` transition and the following test cases to those in Fig. 7

```
+ start failDA stop
+ start failDA failDA stop
- start allocate failDA allocate allocate
- start allocate allocate failDA
- failDA
```

and obtain the state machine of Fig. 8. However, the translation of EUnit tests to sequences needs to be adapted to treat certain error cases as part of a positive sequence and others as making the test case negative. This requires the user to specify the differences and therefore this method is not entirely satisfactory if full automation is the goal.

As shown in Fig. 8, this resolves all the `failDA` operations, which are only permissible when zero or one frequencies have been allocated. The labels on the transitions to the 'dead' state have been elided for readability in the figure.

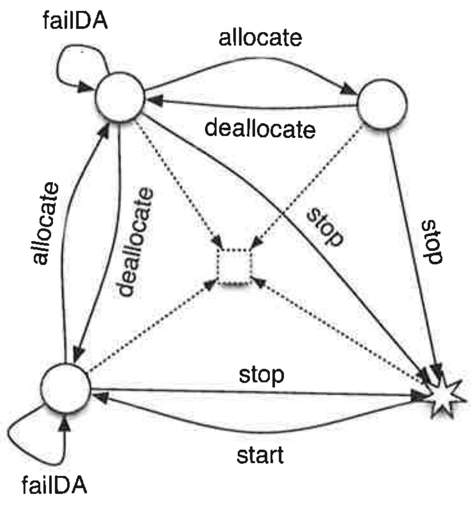


Figure 8. Two frequencies with failed deallocation, `failDA`

3.11 One, two, many ...

A pattern is emerging in Figs. 6 and 7: an FSM to model the server with n frequencies will have $n+3$ states: an initial state, a dead state and $n+1$ states representing the different numbers of frequencies that have been allocated.

We contend that the case of allocation from a set of two frequencies should be sufficient to test the general case, since it allows us to examine the case of allocation and deallocation when some frequencies have been allocated and some not.

Of course, it is possible for an implementation to have special case 'Easter egg'[5] behaviour for particular collections of frequencies, but any finite set of tests will be vulnerable to this. So, making the assumption that our implementation is generic in the frequency set we repeat our contention. Probably a careful tester would extend the model to contain three resources in order to be able to test re-use of a frequency in the middle, but it seems a large investment to go any further than that. We have already to define 17 EUnit tests to capture the behaviour of two frequencies and 20 to capture the behaviour of three frequencies. This corresponds to about 100 lines of Erlang test code for an implementation that is itself smaller than that.

If one is interested in testing even more possible combinations of allocating and deallocating resources, one would rather generate a large number of random combinations for a random collection of frequencies. We can do precisely this by using the QuickCheck finite state machine library to generate the test cases.

4. QuickCheck finite state machine

QuickCheck comes with a library (`eqc_fsm`) for specifying test cases as finite state machines. Given a few callback functions for this state machine, the QuickCheck machinery is able to generate and run test cases that are sequences generated from these callback functions.

Here we present an approach to generate QuickCheck state machine specifications from EUnit tests in contrast to the more common manual generation from informal specifications of the software under test. The advantage of using QuickCheck, as we will see in this section, is that with little extra effort, we get many new tests cases that actually test meaningful sequences, not covered by the EUnit tests.

As demonstrated in the previous section, we use StateChum to generate a state machine from the EUnit tests in order to obtain states names and their transitions. We may potentially improve the state machine by adding tests cases, but, as explained before, the data part restricts us to test cases with only little data involved. Now we translate the obtained state machine in a QuickCheck specification.

4.1 Sequence of calls

Each state obtained by StateChum is translated into a unary Erlang callback function[6] that returns a list with possible next states and the transitions thereto. For example, the state machine described in Fig. 3 has 3 states; state names are randomly chosen by the tool, but manually translated into something meaningful, say *init*, *started* and *error*. From *init* there are two possible transitions and in the QuickCheck library `eqc_fsm`, this is specified thus:

```
init(_) ->
  [ {started,{call,?MODULE,start,[nat()]}},
    {error,{call,?MODULE,stop,[]}}].

started(_) ->
  [ {error,{call,?MODULE,start,[nat()]}},
    {init,{call,?MODULE,stop,[]}}].

error(_) ->
   [].
```

Note that Fig. 3 has no arguments for the functions; this information is present in the EUnit tests, but not in the abstracted state machine. Therefore, we have to retrieve it from the EUnit tests. At this moment we may realize that starting the server with an empty list and a list with only one element has been a completely arbitrary choice in our EUnit tests. In fact, we would like to start the server with an arbitrary, positive number of frequencies `nat()`.

Each transition is encoded as a tuple with first argument the name of the next state and as second argument a symbolic call to an Erlang function, in this case the `start` and `stop` functions in the module (?MODULE) we define our specification in, which differs from the implementation module `frequency.erl`. The reason for a local version of the start and stop function is that we expect these to potentially raise an exception and similar to the `assertException` in EUnit we have to notify QuickCheck that exceptions may be valid. Moreover, we use a maximum number of frequencies to compute the list with consecutive sequences in the start function.

```
start(Freqs) ->
    catch frequency:start(lists:seq(1,Freqs)).

stop() ->
    catch frequency:stop().
```

In the EUnit tests, the return values of the calls to start and stop are checked in the assertions. These assertions translate into postconditions in the QuickCheck specification. Postconditions are callbacks with five arguments: a `From` state, a `To` state, the data in `From` state, the symbolic call and the result of that call. Thus, we check that indeed the positive calls return the right value and that whenever we enter the error state, it was because of a call that raised an exception.

```
postcondition(init,started,_,{call,_,start,_},R) ->
    is_pid(R);
```

[5] Some hidden message or feature, coded as a surprise in software and other artefacts.

[6] The argument of the state is the state data.

```
postcondition(started,init,_,{call,_,stop,_},R) ->
    R == ok;
postcondition(_From,error,_,{call,_,_,_},R) ->
    case R of
       {'EXIT',_} -> true;
       _ -> false
    end.
```

Finally, we need to write a QuickCheck property to run the test cases. First an arbitrary sequence of start and stop commands is created using the state machine description and then that sequence is evaluated. In order to make sure that we start in a known state (even if a previous test has failed), we both stop the frequency server at the beginning and end of each test, relying on the catch when the server is not running.

```
prop_frequency() ->
   ?FORALL(Cmds,commands(?MODULE),
      begin
         stop(),
         {H,S,Res} = run_commands(?MODULE,Cmds),
         stop()
         Res == ok
     end).
```

4.2 Adding state data

The advantage of running many different sequences of starting and stopping the server may not be so obvious for this example. The real benefit of using a QuickCheck state machine specification shows when the state data is used to represent the allocated frequencies.

We choose to use the state machine from Fig. 7 as our starting point. In the state *started* we should add a transition to a state in which one frequency is allocated. From that new state, we create a transition to yet another one where two frequencies are allocated, etc. Of course, the state names have to be generalised and we use QuickCheck's support for parametrized states, i.e. each state is represented by a tuple of which the first argument is the state name and the second argument is a parameter, the number of allocated frequencies in our case.

Note that the state machine in Fig. 7 was obtained from tests with two frequencies and is in fact an abstraction of tests with two allocations. We would like to generalise this to an arbitrary number of frequencies, but start with setting a maximum of 2 for the moment.

```
-define(MAX,2).
```

We introduce a record to represent an abstraction of the state of the frequency server: the free frequencies and the used frequencies.

```
-record(freq,{used=[], free=[]}).
```

We rename the state *started* into `allocated` and add appropriate transitions. We fix the maximum number of allocations to 2 and deallocation of frequencies that have not been allocated is smoothly added as a transition.

```
init(_) ->
    [ {{allocated,0},{call,?MODULE,start,[?MAX]}},
      {error,{call,?MODULE,stop,[]}}
     ].

allocated(N,S) ->
    [ {error,{call,?MODULE,start,[nat()]}} ] ++
    [ {{allocated,N+1},{call,?MODULE,allocate,[]}}
                            || N < ?MAX] ++
    [ {error,{call,?MODULE,allocate,[]}}
                            || N == ?MAX] ++
    [ {{allocated,N-1},{call,?MODULE,deallocate,
                             [elements(S#freq.used)]}}
                           || N > 0] ++
    [ {init,{call,?MODULE,stop,[]}}].

error(_) ->
    [].
```

The list comprehensions are used to lazily compute the state parameter and only include the alternatives that are valid for that particular state. Starting an already started server may take any argument, hence no `?MAX` there but an arbitrarty positive number.

The deallocation functions depends on the state data. As an argument to `deallocate` we supply an arbitrary element of the list `S#freq.used`.

In order to successfully test these cases, QuickCheck need to know more about the state data. This is achieved by defining call-back functions that operate on the data.

The state data gets modified by the `next_state_data` callback function, which takes five arguments. The first argument is the state from which the transition originates and the second argument the state that the transition leads to. The third argument is the state data, i.e., the record that we defined above. The fourth argument is the (symbolic) result of the evaluation of the symbolic call in the last argument.

```
next_state_data(_,_,S,V,{call,_,start,[Max]}) ->
    S#freq{used=[], free=lists:seq(1,Max)};
next_state_data(_,_,S,V,{call,_,allocate,[]}) ->
    case S#freq.free == [] of
        true -> S;
        false ->
           S#freq{used=S#freq.used++[V],
                  free=S#freq.free--[V]}
    end;
next_state_data(_,_,S,V,
                {call,_,deallocate,[Freq]}) ->
    S#freq{used=S#freq.used--[Freq],
           free=S#freq.free++[Freq]};
next_state_data(_,_,S,V,{call,_,stop,[]}) ->
    S#freq{used=[], free=[]}.
```

In this way, we know which frequencies are allocated and which are free. Note that if all frequencies are allocated, then an allocation will result in an error and the state stays unchanged.

Similar to the start and stop command before, we add local commands for allocation and deallocation. This time we use the local function to modify the return value, since our model is cleaner when we get a frequency returned from `allocate`:

```
allocate() ->
    case frequency:allocate() of
        {ok,Freq} -> Freq;
        Error -> Error
    end.


deallocate(Freq) ->
    frequency:deallocate(Freq).
```

Finally, we add postconditions for allocation and deallocation to complete our QuickCheck specification.

```
postcondition(_,_,S,{call,_,allocate,[]},R) ->
    case R of
        {error,no_frequency} ->
          S#freq.free == [];
        F when is_integer(F) ->
          lists:member(F,S#freq.free)
    end;
```

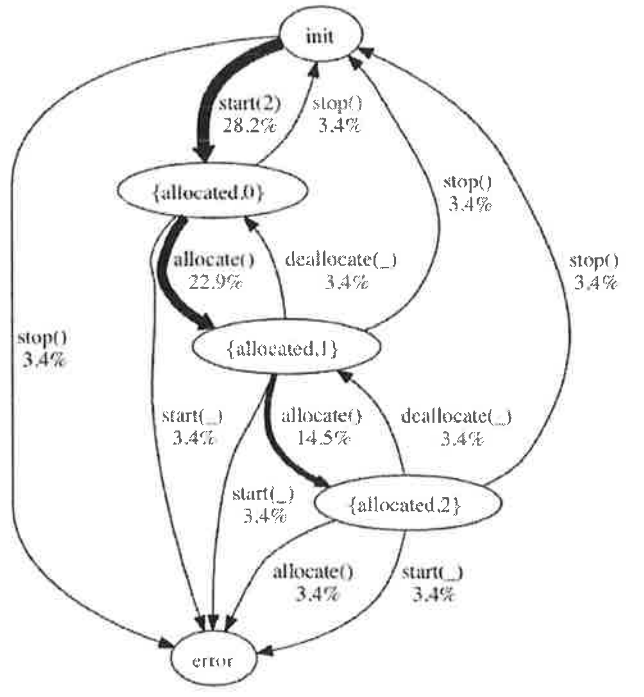


Figure 9. Visualization of QuickCheck specification

```
postcondition(_,_,S,{call,_,deallocate,[Freq]},R) ->
    R == ok;
```

This specification can be used to generate many different sequences of calls to start, allocate, deallocate and stop. QuickCheck can compute a fair distribution for the occurrences of the commands, such that we increase the likelihood to obtain sequences that indeed allocate all available resources instead of just starting and stopping the server all the time. In a visualization of the QuickCheck state machine the weights for each transition are provided as a percentage (see Fig. 9).

A typical example generated with this state machine could be a test case like:

```
 {set,{var,1},{call,frequency_eqc,start,[2]}},
 {set,{var,2},{call,frequency_eqc,stop,[]}},
 {set,{var,3},{call,frequency_eqc,start,[2]}},
 {set,{var,4},{call,frequency_eqc,stop,[]}},
 {set,{var,5},{call,frequency_eqc,start,[2]}},
 {set,{var,6},{call,frequency_eqc,allocate,[]}},
 {set,{var,7},{call,frequency_eqc,allocate,[]}},
 {set,{var,8},{call,frequency_eqc,deallocate,
                                       [{var,6}]}},
 {set,{var,9},{call,frequency_eqc,allocate,[]}}
```

4.3 Additional error transitions

There is still a subtle difference between the QuickCheck state machine in Fig. 9 and the state machine obtained from EUnit tests in Fig. 7, viz. a number of transitions to the error state are missing.

For example, the deallocation in the state with zero allocated frequencies leads to the error state in Fig. 7. We have neglected this case in our specification, but we can add it by adding one more transition to the state defining callback function `allocated`:

```
    [ {error,{call,?MODULE,deallocate,
             [elements(S#freq.free)]}} || N == 0] ++
```

Note that we must pick a frequency from the free frequencies, since none is in use yet. Alternatively, we could take any arbitrary frequency using `nat()`.

The `next_state_data` function can stay as is since we jump to the error state and no more transitions are allowed from there, hence the specific state of the server is not important. The postcondition has, of course, to be adapted, since a transition to the error state should be caused by an exception:

```
postcondition(_,To,S,{call,_,deallocate,[Freq]},R)
                                 when To =/= error ->
    R == ok;
```

Tests generated from this specification expect an exception raised when we deallocate after starting the server. We need to add a catch in the local function of `deallocate` as well. However, when we run the tests against our frequency implementation, we obtain immediate feedback from QuickCheck that the postconditions of this deallocation is falsified. In other words, our implementation follows the specification and indeed always had `deallocate` return `ok`.

Inspecting the EUnit test cases shows that indeed we never test starting the server and then deallocating. The transition in Fig. 7 was added because of insufficient information. In fact, one can argue that the transition should not be there at all, but that incorrect deallocations are either not allowed, which should be guaranteed by the clients of the server, or that the specification of the API should be enriched with a possible error result for `deallocate`.

Rather would we now add a transition that deallocation of free resources should have no effect. This can be done by adding another transition to the state machine:

```
[ {{allocated,N},{call,?MODULE,failDA,
                    [elements(S#freq.free)]}}] ++
```

We use `failDA` instead of `deallocate` to avoid getting ambiguous transitions in the state machine. QuickCheck cannot compute good test case distribution when the model is ambiguous.

The `failDA` function is simply calling the deallocation in the implementation module. The next state function for `failDA` leaves the state untouched and the postcondition checks that an `ok` is returned. When running QuickCheck with this property we found an error in our implementation, since we expected the clients to obey the rule that they would not release the same frequency twice and always added a released frequency to the list of available frequencies. This gave a list with duplicates in the newly constructed test cases and the postcondition for `allocate` found the mismatch by checking that the given frequency is indeed free.

4.4 Increasing number of frequencies

With the definition of the macro `MAX` we can now easily create a state machine that tests sequences that have 4 frequencies and all possible combinations of allocations and deallocations. The only thing to do is to recompile the code with a larger constant. But, testing with a small number of frequencies thoroughly may reveal more faults than when testing a larger number of frequencies in a less exhaustive manner.

The QuickCheck specification is about 100 lines of code, which is similar to an exhaustive EUnit test suite, but it covers a wider range of tests. For larger, more realistic example, the size of the QuickCheck specification tends to grow less fast than an EUnit test suite does.

5. Related work

In this section we examine related work in test-driven development, grammar inference and testing methodologies.

Test-driven development

As we mentioned in Section 2, Beck's [4] answers a number of frequently-asked questions. In replying to *"How many tests should you write?"* he provides a simple example of a function to classify triangles: this elicits an answer inspired by equivalence partitioning. No state-based systems are discussed. The question *"How do you know if you have good tests?"* relates to the quality of individual tests, rather than the effect of the collection as a whole.

Fowler advocates mutation testing as a mechanism for assessing the adequacy of a set of tests [21]. Astels [3] in discussing TDD for Java also advocates mutation testing with Jester [13], as well as code coverage analysis with Clover [9] and NoUnit [16].

Of course, these methods can only be used when there is an implementation to hand. In the context of TDD there is a circularity to this, since the implementation has been developed specifically to meet the set of tests. By contrast, our method gives feedback on the test set independently of any implementation.

Random testing

Random testing for functional programs was first introduced by QuickCheck for Haskell [8] and more recently developed for Erlang [1]. It has also inspired related systems for Scheme, Standard ML, Perl, Python, Ruby, Java, Scala, F# and JavaScript.

QuickCheck testing is based on the statement of logical *properties* which are then tested for random inputs generated in a type-based manner. Simple logical statements of properties suffice for functional behaviour; state based systems are tested by driving them from an FSM which gives an abstract model of the system.

Fuzz testing or fuzzing [20] is a related technique used particularly with protocol testing, an area where QuickCheck FSMs can also be used. Fuzzing is a "brute force" approach, typically generating inputs at random, rather than having their generation being guided by a model such as an FSM. Fuzzing is perceived, however, as a mechanism providing a high benefit:cost ratio.

A comprehensive overview of other approaches to random testing is given in Pacheto's thesis [17]. Pacheto's thesis also examines ways that random testing can be 'directed' with extra tests being generated as a consequence of examining the results of already executed test cases.

Inference and testing

There is a substantial literature on inferring higher-level structures from trace or event-based data. Among the earliest is Cook and Wolf's [10] which infers an FSM from event-based trace data. More recent work by Artzi *et. al.* [2] uses those techniques to general legal test inputs – that is legal sequences of calls to APIs – to OO programs, again based on execution traces; this paper also provides a useful overview of other work in this area. Walkinshaw and others [24] use the Daikon tool [12] as part of an interactive process of model elicitation.

Daikon implements invariant inference, and has been extended to the DySy tool [11] which augments the Daikon approach based on test set execution with dynamic symbolic execution. Xie and Notkin [26] infer specifications from test case executions, and based on this develop further test cases.

Our approach differs from these in being based on the test cases themselves rather than on their execution: it can therefore be used independently of any implementation.

The Wrangler refactoring tool for Erlang [25] provides clone detection and elimination facilities [15], and in the latest release (0.8.8) implements the facility to transform a cloned test into a QuickCheck property, thus generalising the range of possible tests of the system.

6. Conclusions and Future Work

We have shown the value of extracting the finite state machine implicit in a set of EUnit tests not only for understanding the adequacy of the tests developed as a part of the process of test-driven development but also in defining a QuickCheck FSM which can be used for property-based testing of the system under test. In doing this we noted a number of points.

- The negative tests – that is those that lead to an `error` value of some sort, raise an exception or cause another form of error – are as important as the positive tests in delimiting the correct behaviour of the system implicit in the tests. This is due in part to the nature of the extraction algorithm [23] but is also due to the fact that without these tests there would be no explicit bounds on the permissible behaviour.
- We assume that we can extract the call sequences within tests by static examination of the test code. This is not unreasonable since many test cases consist of straight line code, particularly for the state-based systems that we examine here.
- Some aspects of the process can be automated with ease, including the extraction of the function call sequences and the naive conversion of an FSM into QuickCheck notation. Others require manual intervention, including the choice of data values for the 'small' states and the choice of state data for the QuickCheck FSM.
- Given that the model we develop is an abstraction of the actual system, it is natural for non-determinism to creep into the model. This can be resolved by renaming some of the transitions to avoid non-determinism. The old and new transitions can then be seen as having pre-conditions which will be explicit in the QuickCheck model.

The next step for us to take is to refine the process described here into a procedure which automates as much as possible of the FSM development. This will allow QuickCheck properties for state-based systems to be extracted from tests in a semi-automated but user-guided way.

We would like to acknowledge the support of the European Commission for this through the FP7 Collaborative project *ProTest* [18], grant number 215868.

References

[1] T. Arts *et. al.* Testing Telecoms Software with Quviq QuickCheck In Proceedings of the Fifth ACM SIGPLAN Erlang Workshop, ACM Press, 2006.

[2] S. Artzi *et. al.* Finding the Needles in the Haystack: Generating Legal Test Inputs for Object-Oriented Programs. In *M-TOOS 2006: 1st Workshop on Model-Based Testing and Object-Oriented Systems*, 2006.

[3] D. Astels. *Test-driven Development: A Practical Guide.* Prentice Hall, 2003.

[4] K. Beck. *Test-driven Development: By Example.* Addison-Wesley, 2002.

[5] R. Carlsson EUnit - a Lightweight Unit Testing Framework for Erlang. In *Proceedings of the fifth ACM SIGPLAN Erlang Workshop*, ACM Press, 2006.

[6] R. Carlsson and M. Rémond. *EUnit - a Lightweight Unit Testing Framework for Erlang.* `http://svn.process-one.net/contribs/trunk/eunit/doc/overview-summary.html`, last accessed 07-06-2010.

[7] F. Cesarini and S. Thompson. *Erlang Programming.* O'Reilly Inc., 2009.

[8] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, ACM Press, 2000.

[9] Clover. *Clover: Java code coverage & test optimization.* `http://www.atlassian.com/software/clover/`, last accessed 07-06-2010.

[10] J. E. Cook and A. L. Wolf. Discovering Models of Software Processes from Event-Based Data. In *ACM Transactions on Software Engineering and Methodology*, 7, 1998.

[11] C. Csallner *et. al.* DySy: Dynamic Symbolic Execution for Invariant Inference. In *ICSE08*, ACM Press, 2008.

[12] M. D. Ernst *et. al.* The Daikon system for dynamic detection of likely invariants. In *ACM Transactions on Software Engineering and Methodology*, 69, 2007.

[13] E. R. Harold. *Test your tests with Jester.* `http://www.ibm.com/developerworks/library/j-jester/`, last accessed 07-06-2010.

[14] Hughes, J. QuickCheck Testing for Fun and Profit. In: 9th Int. Symp. on Practical Aspects of Declarative Languages, Springer (2007)

[15] H. Li and S. Thompson. Similar Code Detection and Elimination for Erlang Programs. In *12th Int. Symp. on Practical Aspects of Declarative Languages*, Springer LNCS 5937, 2010.

[16] NoUnit. `http://nounit.sourceforge.net/`, last accessed 07-06-2010.

[17] C. Pacheco *Directed Random Testing*. Ph.D. thesis. MIT Department of Electrical Engineering and Computer Science, 2009.

[18] ProTest. `http://www.protest project.eu/`, last accessed 07-06-2010.

[19] StateChum. `http://statechum.sourceforge.net/`, last accessed 07-06-2010.

[20] M. Sutton, A. Greene, P. Amini *Fuzzing: Brute Force Vulnerability Discovery*, Addison Wesley, 2007.

[21] B. Venners. *Test-Driven Development: A Conversation with Martin Fowler, Part V.* `http://www.artima.com/intv/testdrivenP.html`, last accessed 07-06-2010.

[22] E. Vidal. Grammatical inference: An introductory survey. In *Grammatical Inference and Applications*, LNCS 862, Springer, 1994.

[23] N. Walkinshaw *et. al.* Reverse-Engineering State Machines by Interactive Grammar Inference. In *14th IEEE Working Conference on Reverse Engineering (WCRE'07)*, IEEE Press, 2007.

[24] N. Walkinshaw *et. al.* Iterative Refinement of Reverse-Engineered Models by Model-Based Testing. In *FM'09*, volume 5850 of Lecture Notes in Computer Science, Springer, 2009.

[25] Wrangler. `http://www.cs.kent.ac.uk/projects/wrangler/`, last accessed 07-06-2010.

[26] T. Xie and D. Notkin. Mutually Enhancing Test Generation and Specification Inference. In *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, LNCS Vol. 2931, Springer, 2003.

Coordinating and Visualizing Independent Behaviors in Erlang

Guy Wiener   Gera Weiss
Ben-Gurion University, Beer-Sheva, Israel
{gwiener,geraw}@cs.bgu.ac.il

Assaf Marron
Weizmann Institute of Science, Rehovot, Israel
assaf.marron@weizmann.ac.il

Abstract

Behavioral programming, introduced by the LSC language and extended by the BPJ Java library, enables development of behaviors as independent modules that are relatively oblivious of each other, yet are integrated at run-time yielding cohesive system behavior. In this paper we present a proof-of-concept for infrastructure and a design pattern that enable development of such behavioral programs in Erlang. Each behavior scenario, called a behavior thread, or b-thread, runs in its own Erlang process. Runs of programs are sequences of events that result from three kinds of b-thread actions: *requesting* that events be considered for triggering, *waiting* for triggered events, and *blocking* events that may be requested by other b-threads. A central mechanism handles these requests, and coordinates b-thread execution, yielding composite, integrated system behavior. We also introduce a visualization tool for Erlang programs written in the proposed design pattern. We believe that enabling the modular incremental development of behavioral programming in Erlang could further simplify the development and maintenance of applications consisting of concurrent independent behaviors.

Categories and Subject Descriptors D.2.11 [*Software*]: Software Architecture—Patterns; D.3.2 [*Programming Languages*]: Language Classifications—Erlang

General Terms Design, Languages

Keywords Design Patterns, Behavioral Programming, Live Sequence Charts

1. Introduction

Scenario-based programming, or behavioral programming, is a programming paradigm introduced by the language of Live Sequence Charts (LSC) [1] and its Play-Engine implementation [3]. This work was extended in [6] through the BPJ library that implements behavioral principles in a traditional Java programming context. In behavioral programming, behaviors are programmed relatively independently of each other, and are interlaced at run-time to create a cohesive, integrated, system behavior. This approach turns out to be very natural, and enables incremental development of highly modular system, where the decomposition of the system is according to behaviors - software components that may cross subsystem boundaries and are not necessarily tied to a specific class or object. LSC and BPJ represent two different approaches for behavioral programming. LSC is based on centralized execution of a collection of sequence charts [7, 9] enhanced with modalities that control what must, may or must not be done. The Play Engine examines all charts in the specification, and triggers events such that the execution satisfies the modal specification. The LSC language also includes rich programming constructs such as objects with properties, control flow, conditions, variables, symbolic objects and implemented functions that expand the capabilities of the developed applications. Behavioral programming in BPJ is based on running behaviors in Java threads (behavior threads, or b-threads, for short) that call API functions to announce events that they request, wait for, or block, and to invoke a coordination mechanism that weaves these requests yielding integrated system behavior.

In both approaches, candidate next events from each behavior are considered for triggering. One of these is selected, subject to the condition that it is not forbidden, or blocked, by other behaviors. Behaviors affected by the triggered event advance and perform arbitrary processing. All behaviors then are synchronized and coordinated, resulting in selection of next event in the system.

In Section 8 we outline in some more detail the different approaches of both LSC and BPJ.

In this paper we adopt the approach used by BPJ and propose (through a proof-of-concept) implementing b-threads as Erlang processes. We provide a central coordination mechanism, and an interface that b-threads can use to report the events that they request, wait for, or block. We propose a design pattern for coding behavioral programs, and provide a visualization tool that depicts b-threads coded in the proposed pattern as transition systems.

We believe that enabling the modular incremental development of behavioral programming in Erlang could further simplify the development and maintenance of applications consisting of concurrent independent behaviors.

The sections of this paper follow largely the section structure used by "The Gang of Four" [2] for documenting design patterns, including sections such as intent, motivation, applicability, structure, sample code, and related patterns. The visual tools is described in the Structure section.

2. Intent

We propose a design pattern called BP and an associated module (called bp), for iteratively creating a sequence of events, where the next event is chosen with the help of the bidding protocol described below. The bidders are Erlang processes registered as behavior threads (b-threads). In each iteration:

1. Each b-thread places a bid:
    - Watched events: events that the b-thread waits for and asks to be notified of.
    - Requested events: events that the b-thread proposes that they be considered for triggering.
    - Blocked events: event that the b-thread forbids.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09...$10.00

2. When all b-threads place their bids, an auction takes place:
   - An evaluation mechanism chooses an event requested by some b-thread and not blocked by any b-thread.
3. B-threads are notified of the auction outcome:
   - The b-threads that asked to be notified (the selected event is in their watched set) are resumed.
4. B-threads can execute arbitrary computations before placing their next bid in the next iteration.

3. Motivation

The motivation for proposing the design pattern is to provide a simple mechanism through which systems can be constructed from software components each of which controls and coordinates a particular behavior. As behaviors may cross object boundaries, such construction complements the traditional approach to software development where programs modularity revolves around objects and data-structures. Using events as markers of system behavior, and applying the proposed bidding mechanism for choosing events, the resulting integrated system behavior is an event sequence that reflects, at every step, each b-thread's view of how the system should proceed.

The BP design pattern helps programmers maintain b-thread independence, by unifying the occurrence of events requested by multiple b-threads and not notifying b-threads of events that they do not watch. Particularly, requesting scenarios need not care how events that they trigger affect other scenarios.

4. Applicability

The BP design pattern should be used when the system's behavior can be naturally decomposed into, or described as an interleaved execution of relatively independent scenarios as described in [1, 3]. BP allows for programming each of these scenarios in an explicit and natural way in its own module, as an alternative to allowing the scenario to emerge implicitly from code that is scattered in multiple participating objects.

In this context, it is worth noting that interesting behaviors often emerge already in early stages of software design and specification, even with small sets of b-threads. Thus, BP allows for programming initial designs and software specifications to be presented to users for feedback (e.g., for finding errors in the way requirements are understood).

More generally, the BP design pattern is suitable for incremental development, as it allows adding and removing behaviors with little or no change to existing code.

Another context where BP can be particularly applicable is end-user customization. For example, imagine a (remote) control for a video player. If that player's behavior is programmed in BP, users can customize it without going into existing code. They can add or remove behaviors, say for simplifying certain activities, or for avoiding common mistakes, by adding to the system b-threads that handle specific sequences of user actions.

5. Structure

5.1 Participants

In this design, the participants are:

The BP controller: A central server process.

- Receives the synchronization requests from b-threads.
- Decides on the next event.
- Sends the next event to b-threads that are waiting for it.

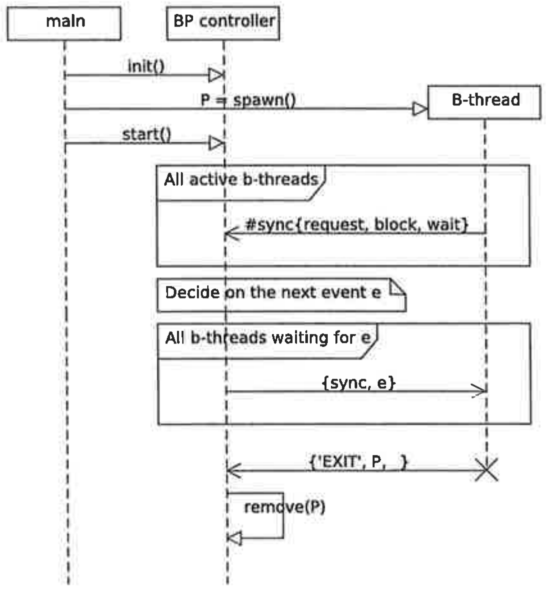


Figure 1. The collaboration between processes using BP. Filled arrows mark function calls, vee arrows mark messages.

b-threads: The work-doing processes.

- Bid for the next event by sending a synchronization request to the BP controller with the following parameters: (1) Requested events. (2) Blocked events. (3) Watched events.
- Wait until the BP controller sends them an event that they are waiting for.

5.2 Collaboration

The collaboration between the BP controller and the b-threads follow these steps (see Figure 1):

1. The BP controller is initialized.
2. B-thread processes are spawned and added to the controller.
3. The BP controller is started.
4. B-threads call **bp:sync** to send their requests to the BP controller, and are suspended until they receive a response from the controller.
5. The BP controller waits until all the active registered b-threads send their requests.
6. The BP controller decides on the next event (see the next section) and sends responses to the b-threads that wait for it.
7. The b-threads that receive the event continue their computation until they call **bp:sync** again.
8. When a b-thread exits, it is removed from the BP controller.

5.3 Model

The BP design pattern is based on the following mathematical model [6]:

First, the code of each b-thread is abstracted as a transition system whose states are valuations of program variables at the times where the b-thread places its bids (synchronizes with the other b-threads). To model the bidding, we attach to each state s a set $R(s)$ of requested events and a set $B(s)$ of blocked events, as formalized in Definition 1. Recall that a (labeled) transition system is defined (see e.g [8]) as a quadruple $\langle S, E, \rightarrow, init \rangle$, where S is a set of states, E is a set of events, $\rightarrow$ is a transition relation contained in $(S \times E) \times S$, and $init \in S$ is the initial state.

The runs of such a transition system are sequences of the form $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_i} s_i \cdots$, where $s_0 = init$, and for all $i = 1, 2, \cdots, s_i \in S, e_i \in E$, and $s_{i-1} \xrightarrow{e_i} s_i$.

Definition 1 (behavior thread). A b-thread is a tuple $\langle S, E, \rightarrow, init, R, B\rangle$, where $\langle S, E, \rightarrow, init\rangle$ forms a labeled transition system, $R \colon S \rightarrow 2^E$ is a function that associates each state with the set of events requested by the b-thread when in that state, and $B \colon S \rightarrow 2^E$ is a function that associates each state with the set of events blocked by the b-thread when in that state.

Using this abstraction, we can formalize the auction mechanism as an operator for composing transition systems, as given in Definition 2.

Definition 2 (runs of a set of b-threads). We define the runs of a set of b-threads $\{\langle S_i, E_i, \rightarrow_i, init_i, R_i, B_i\rangle\}_{i=1}^n$ as the runs of the labeled transition system $\langle S, E, \rightarrow, init\rangle$, where $S = S_1 \times \cdots \times S_n$, $E = \bigcup_{i=1}^n E_i$, $init = \langle init_1, \ldots, init_n\rangle$, and $\rightarrow$ includes a transition $\langle s_1, \ldots, s_n\rangle \xrightarrow{e} \langle s'_1, \ldots, s'_n\rangle$ if and only if

$$\underbrace{e \in \bigcup_{i=1}^{n} R_i(s_i)}_{\text{e is requested}} \qquad \bigwedge \qquad \underbrace{e \notin \bigcup_{i=1}^{n} B_i(s_i)}_{\text{e is not blocked}}. \qquad (1)$$

and

$$\bigwedge_{i=1}^{n} \Big(\underbrace{(e \in E_i \implies s_i \xrightarrow{e}_i s'_i)}_{\text{affected b-threads move}} \wedge \underbrace{(e \notin E_i \implies s_i = s'_i)}_{\text{unaffected b-threads don't move}}\Big) \quad (2)$$

This definition specifies a transition system whose runs are the interleaved executions of the composed system. Specifically, we say that a sequence of events is a run of the system if, at each step, the selected event is requested by some b-thread and not blocked by any. The second part of the definition says that the selected event may change the state of the b-threads that have it in their alphabet.

Note that this mathematical model allows for nondeterminism. In particular, if there are multiple requested events that are not blocked the model allows choosing any of them. In implementations, however, it is easier to program deterministic systems. In particular, we prioritize the b-threads and event requests such that event selection is deterministic.

5.4 Code Structure

Our implementation of the BP design pattern is based on a supporting module, bp. The bp module exports the following functions:

`init/0` Initialize the BP controller.

`add/2` Add a process with a given priority.

`start/0` Start the controller.

`sync/1` Send a synchronization request, that includes requested, blocked and watched events to the controller and wait until one of the watched events is selected. The `sync/1` function takes a record as an argument. The record definition is `-record(sync, {request, wait, block})`. The record fields match the arguments described in Section 5.3, and have a default value of an empty list. For example:

```
bp:sync(#sync{
        request=[E1],wait=[E1,E2],block=[E3]})
```

`remove/1` Remove a process from the controller.

For its basic operation, the bp module imposes a few constraints on the structure of the code. B-threads must be spawned as processes, and the program must include initialization code for the bp process. Section 5.2 describe the initialization sequence in more detail. There are no other constraints on the calculations that b-threads perform before or after the calls to `bp:sync`.

```
morning() ->
  [sync(#sync{wait=[morning], request=[morning]}) ||
   _ <- lists:seq(1,3)].

evening() ->
  [sync(#sync{wait=[evening], request=[evening]}) ||
   _ <- lists:seq(1,3)].

interleave() ->
  bp:sync(#sync{wait=[morning], block=[evening]}),
  bp:sync(#sync{wait=[evening], block=[morning]}),
  interleave().

display() ->
  Event = bp:sync(#sync{wait=[morning, evening]}),
  io:format("Good ~w~n", [Event]),
  display().

test() ->
  bp:init(),
  bp:add(spawn(fun morning/0), 1),
  bp:add(spawn(fun display/0), 2),
  bp:add(spawn(fun evening/0), 3),
  bp:add(spawn(fun interleave/0), 4),
  bp:start().
```

Figure 2. "Hello, World!" - An example of using the bp module

```
Good morning
Good evening
Good morning
Good evening
Good morning
Good evening
```

Figure 3. Output of code from Figure 2

Figure 2 shows a basic example of code that follows the classical ""Hello, World!" program and uses the bp module to issue "Good morning" and "Good evening" greetings. The example includes four b-threads. The events in the system are `morning` and `evening`. The `morning` and `evening` b-threads request one of these events three times and terminate. The `interleave` b-thread, in an alternating manner, blocks one of these events while waiting for the other, causing the interleaving of the two independent event sequences. The `display` b-thread prints the selected event. The expected result — alternating `Good morning` and `Good evening` greetings — is shown on Figure 3.

It should be noted that at the current step of the development, bp is not a generic module: It does not take another module as an argument and does not define a behavior for another module. Instead, it allows for starting b-threads by spawning a process. We find this approach more flexible and free-form. However, if the code follows specific conventions, listed in the next section, an auxiliary visualization module can produce a diagram of it.

5.4.1 Code Visualization

The bp module supports any unstructured code, as long as it uses `init`, `add`, `start` and `sync`. However, if the code has a specific structure, it is possible to produce a visual representation of its behavior automatically: A module named bp_vis produces a diagram of a transition system depicting the b-thread. The diagram

```
case bp:sync(#sync{...}) of
  x -> state1();
  y -> state2();
  ...
end.
```

Figure 4. An example of a state case

is generated as a Graphviz[1] file. There are two ways to generate the diagram: Use `bp_vis` as a `parse_transform` module, or run `bp_vis:visualize(BeamFile, GvFile)` directly. In the first case, the generated file has the same base name as the module.

The code visualizer makes the following assumptions about the structure of the code:

1. The code for each b-thread is contained in a separate module.
2. Each function that is called in a clause of a state case is a state function, that represents the next state where:
   - A *state function* is a function that its last term is a *state case*.
   - A *state case* is a case statement where the expression is a call to `bp:sync` and each clause maps from an event to a function call. Figure 4 shows an example of a state case.
3. If a function called `start` is found in the module, it is assumed to be the first state.

The format of the generated diagram follows.

- State functions appears as ellipses with multiline labels. The first line is the function signature. The following lines show the content of the request, wait and block arguments of the call to `bp:sync` in the state case.
- An edge from ellipse A to ellipse B appears if one of the clauses in the state case of A is a function call to B.
- Each edge has a label. The label format for an edge from A to B is "`Event (when Guard) / Call`". The `Event` is the head of the case clause whose body is the function call to B. The `Guard` is the guard part of the head of the clause, which is optional. The `call` is the exact function call to B, including arguments.
- If one of the state functions is called "`start`", it is emphasized by a small arrow, starting from a black dot.

For example, suppose that some device can print, scan, send a fax and stop. The b-thread module in Figure 5 has the following behavior: It tries to print 3 times. If the printing starts, it waits for it to end. If it fails more then 3 times, it gives up. When waiting for printing, it prevents the machine from stopping. Figure 6 shows a visualization of that code. Since this b-thread does not have a designated start function, it does not include the additional arrow. The transition diagram figures in Section 7.2 were generated using this code visualization tool, while designating a start function.

In the future we expect that the structural requirements that simplify the visualization process will be expanded to include other design patterns, or accommodate less constrained code. It is our view that the code for behavioral program should be as free as possible, and need not be aligned with the state transitions used for the formal definitions or visualization. In Section 8 we discuss the relation of our work to state transition coding with `gen_fsm`, using callback-functions as well as explicit state.

[1] `http://graphviz.org`

```
-module(printjob).
-compile([{parse_transform, bp_vis}]).
-include("bp.hrl").
-define(LIMIT, 3).

pending(N) ->
  case bp:sync(#sync{request=[print],
             wait=[print, scan, fax],
             block=[stop]}) of
    print -> working();
    _ when N < ?LIMIT -> pending(N+1);
    _ when N >= ?LIMIT -> idle()
  end.

working() ->
  case bp:sync(#sync{wait=[finish],
                     block=[stop]}) of
    finish -> idle()
  end.

idle() ->
  case bp:sync(#sync{wait=[stop]}) of
    stop -> ok
  end.
```

Figure 5. A b-thread module with a structured code

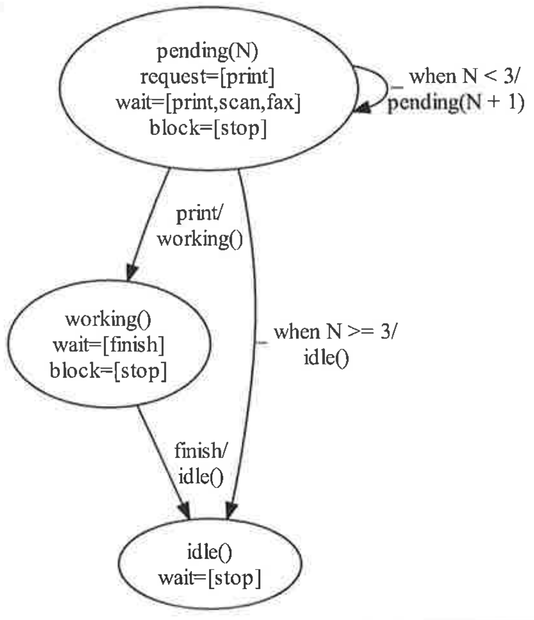


Figure 6. A visualization of the module in Figure 5

6. Consequences

Behavioral programming is based on consensus: It requires all participating processes to agree on the next step. Therefore, it requires a synchronization point for all b-threads that are not waiting. To exit the blocking call to `bp:sync`, each active b-thread must wait until all other active b-thread call `bp:sync`. Although this is a reasonable demand for an agreement protocol, a specialized protocol for a specific problem can be more efficient.

Reaching an agreement also requires sharing information between processes. In our implementation, we use a central process (the BP controller) to collect and handle the requests from all participating b-threads. Therefore, this process is critical to the operation of the system. In a production system, special attention should be given to this potential single-point-of-failure, and common actions such as monitoring, automatic recovery and redundancy should be considered.

In the proof-of-concept implementation, the execution rate can reach thousands of events per second, with thousands of participating b-threads. Detailed performance analysis remains as future work. Such analysis should take into consideration, among others, the amount of event-based synchronization as compared with other processing and computation performed by the b-threads. For applications where synchronization may introduce execessive delays we are considering for example clustering of b-threads in synchronization groups, or using design patterns in which long-running processes do not run as b-threads, but instead use dynamically-created b-threads to communicate with the rest of the behavioral program. Lastly it should be noted, that certain benefits of programming multiple independendent b-threads may be manifested also when the number of processors is small, or even one, as complex behaviors are decomposed into simpler ones in a natural way.

7. Examples and Sample Code

7.1 Coordinated Sequential Processing

To illustrate how the BP design pattern can be used, we discuss, as an example, the structure of applications that require bulk processing of a large volumes of records to perform business operations. Examples of business operations include time based events (e.g. generation of periodic correspondence), periodic application of complex business rules processed repetitively across very large data sets (e.g. interest accrual or rate adjustments), or the integration of information that is received from internal and external systems that typically requires formatting, validation and processing in a transactional manner into the system of record. Such batch processing systems are used to process billions of transactions every day for enterprises around the world. Easy interweaving of such processes can be of great value. For example - consider the printing of different notices as well as paper advertisements and coupons for insertion in each customer's envelope. Sequential processes may independently customize individual messages to customers in a large database, but they need to be coordinated, such that all messages to a given customer are printed consecutively. The BP design pattern enables such coordination with minimal dependency across the different sequential processes.

In BP terms, sequential batch processing can be formulated as an iterative bidding/consensus process where, for each record of data, a set of independent b-threads collaborate by expressing their views of how the record should be processed (using the request/wait/block idioms). More specifically, the system can be programmed using a sequencer b-thread that controls the sequencing of the records and a set of b-threads that model different considerations of how the records should be processed.

To demonstrate the technique, we present an implementation of the Sieve of Eratosthenes algorithm in Figure 7. The `sequencer` is a b-thread that leads the sequential processing of the natural numbers, and attempts to declare each one as a prime. The `pFactors` b-thread blocks the multiples of a prime number from being declared as prime. The `factory` b-thread is responsible for spawning and registering a `pFactors` b-thread whenever a prime number is discovered. Note that such dynamic addition of b-threads is an extension of the basic collaboration described above, and requires further attention in definition and development. The `start` method starts an instance of the `sequencer` and the `factory` b-threads. This code does not conform with the assumptions outlined in Section 5.4.1 above and therefore cannot be automatically visualized by `bs_wis`.

```
sequencer(I) when I < 100 ->
  sync(#sync{wait=[I], request=[I]}),
  T = sync(#sync{
    wait=[prime,not_prime],
      request=[prime,not_prime]}),
  io:format("~w is ~w ~n", [I,T]),
  sequencer(I+1);

sequencer(I) -> io:format("---~n").

pFactors(I) -> pFactors(2*I,I).

pFactors(N,I) ->
  sync(#sync{wait = [N]}),
  sync(#sync{block=[prime], wait = [N+1]}),
  pFactors(N+I,I).


factory(I) ->
  I = sync(#sync{wait = [I]}),
  T = sync(#sync{wait = [prime,not_prime]}),
  if
    T == prime ->
       add(spawn(fun() -> pFactors(I) end), 1);
    true -> ok
  end,
  factory(I+1).

run() ->
  init(),
  add(spawn(fun() -> sequencer(2) end), 3),
  add(spawn(fun() -> factory(2) end), 4),
  start().
```

Figure 7. Illustration of coordinated sequential batch processing: BP version of the Sieve of Eratosthenes.

7.2 Tic-Tac-Toe

As another (larger) example of a code that uses the proposed BP design pattern, we describe an implementation of a computer program that plays the well known game of Tic-Tac-Toe. Game playing behavior by humans is naturally decomposed into independent behaviors of complying with the rules, random or default moves, and an attempt to apply various tactics accumulated with experience. Thus, the purpose of this example is to show how the BP pattern can be used to construct a composite behavior from a set of simpler, intuitive b-threads.

The game involves two players: one marked x is played by the human, and a second marked o is played by the computer. The events in the program are pairs of the form $\langle \mathsf{x}, Sq\rangle$ or $\langle \mathsf{o}, Sq\rangle$, where Sq references a square in the 3×3 board, and is an integer between 1 and 9 (see figure below).

A game is played as a sequence of events; e.g the sequence $\langle x,1\rangle$, $\langle o,5\rangle$, $\langle x,9\rangle$, $\langle o,3\rangle$,$\langle x,7\rangle$, $\langle o,8\rangle$, $\langle x,4\rangle$, describes a game round in which x wins, and its final configuration is:

X 1	2	O 3
X 4	O 5	6
X 7	O 8	X 9

Below we list the b-threads of the program and use the visualization (automatically generated from the Erlang code) to explain their behavior.

detect_win: This b-thread detects the occurrence of winning scenarios. Independently of how the game is played, the basic rule of the game that says "the first player to get three in a line wins" can be directly translated to a simple b-thread as shown in Figure 8.

enforce_turns: Another rule for this game is that "players alternate placing xs and os on the board". This, again, translates directly to a simple b-thread as depicted in Figure 9.

disallow_square_reuse: This b-thread prevents a given square from being marked twice. See Figure 10.

default_moves: This b-thread simply requests the marking of all squares. The order of the requested events determines their priorities in our strategy: try to mark the center square first, then the corners, and only then the remaining squares. Requested moves will be triggered only when not blocked and when there are no higher priority unblocked requests. With the addition of this b-thread, the program can now play legally and can complete any game - though its strategy is quite simplistic.

prevent_line_with_two: One of the first rules of thumb, taught to someone learning how to play the game, is that when your opponent (in our case the x player) is about to complete a line of three, you should put your mark (in our case o) on that line to preempt the attack. This rule can be directly translated to a b-thread, as shown in Figure 12.

complete_line_with_two: This b-thread implements another basic rule of thumb of tic-tac-toe: whenever you can complete a (vertical, horizontal, or diagonal) line and, by that, win the game – do it. Note that, since the winning is immediate, the attack can have a higher priority than defense. In particular, when `prevent_lines_with_two` requests to preempt an attack and `complete_line_with_two` requests the winning move – the latter request should be chosen. This is implemented in the BP design pattern using the priority mechanism – events requested by a b-thread with higher priority are chosen over requests of b-threads with lower priorities. The `complete_line_with_two` b-thread is depicted in Figure 13.

intercept_ single_ fork: This multi-instance b-thread defends against situations where a future marking by player x will present him/her with the choice of winning in one of two different lines. For example, following a mark of square 6 and square 8 by x, player o will try to mark square 9. See Figure 14.

intercept_ double_ fork: This b-thread defends against situations where a future marking by player x will present him/her with the choice of creating two forks as described above by marking two opposite corners when o marks the center. The defense used by this b-thread is to attack, by marking square 2 and forcing the opponent to defend and abandon his own attack. See figure 15.

This set of b-threads constitutes a complete computer program that plays tic-tac-toe against a user. The strategy for the o player, played by the computer, emerging from the composition of the b-threads is optimal in the sense that the o player will never lose the game (the x player, played by the user, can force a tie but can never win). In addition to the b-threads, the code includes one additional module, not listed here, containing initialization code that spawns and registers copies of the b-threads.

8. Known Uses and Related Work

The event-based and state-like nature of BP makes b-threads somewhat similar to the gereric finite state machine (`gen_fsm`) module[2] from the Erlang standard library. Both the `gen_fsm` and `bp` modules deal with a state-based reaction to events. However, there are several differences between the two modules:

- `gen_fsm` does not provide the option to block events.
- `gen_fsm` does not deal with coordinating between several instances.
- `gen_fsm` does not distiguish between events that are waited for and other events. It will handle any call to `gen_fsm:send_event`.

One can view the `bp` module as an extension of `gen_fsm`, designed for coordinating between several processes.

In addition to the general capabilities and broad usage of Erlang in concurrent processing, particular attention to programming independent behaviors in Erlang can be seen in systems such the ERES rule-production system [11] or the eXat agent programming system [10]. What distinguishes b-thread synchronization in the proposed pattern from the classical programming of concurrent behaviors in Erlang is the ability of one process to prevent the occurrence of an event requested by another process, without each party's explicit awareness of the existence of the other party. It will be interesting to explore the addition of the proposed synchronization approach with its compact blocking idiom into the above systems.

The proposed pattern and module for the Erlang language follows in the footsteps of scenario based programming and behavioral programming of LSCs [1, 3]and BPJ[6].

The formal visual language of live sequence charts (LSCs) was defined in [1]. The LSC language extended message sequence charts (MSC)[7], and the then-current UML sequence diagrams mainly by adding modalities to events (UML sequence diagrams were later enhanced to express some of these notions [9]). The LSC language adds to the sequence diagram a notation that distinguishes between events that must happen ("hot"), events that may happen ("cold"), and events that must not happen (marked explicitly or implicitly as forbidden). These modalities enable the direct execution of LSC specification where the Play Engine tool processes the LSC specification and generates a sequence of events that satisfies the specification. The Play Engine does this by keeping track of the next candidate events on each of the charts and selecting a next event to be triggered based on the specified modalities. If no event can be selected without violating the specification - the system stops. Together with other constructs (objects and properties, flow control, variables, access to functions in other languages, symbolic messages, symbolic objects, and a notion of time), the LSC language and the Play Engine show that independent units of behavior description can be used not only in requirements and specifications, but in building the final executable.

The same concepts were implemented in Java through BPJ library [5, 6] (and can be similarly implemented in other textual, procedural languages). Each behavior is coded in its own Java thread - called a b-thread. The b-thread calls the behavioral synchronization function of the BPJ library (called bSync), passing to it three parameters - a set of requested events, a set of watched events and

[2] `http://www.erlang.org/doc/man/gen_fsm.html`

a set of blocked events. The calls to BPJ also synchronize all b-threads, by suspending each caller until all registered b-threads post their wishes. Then, a central coordination mechanism selects an event that is requested by some b-thread and is not blocked by any b-thread, and resumes all b-threads that either requested the same event, or announced it as a watched event. In BPJ the b-thread relies on the underlying language for flow control, variables, objects and other programming necessities. The library and source code examples are available at [5].

9. Conclusion and Future Directions

A proof-of-concept is described for a design pattern and a supporting module for composing an application from a set of behavior threads that independently request, block, and wait for events. We believe that this design pattern can create valuable synergy between behavioral (or scenario-based) programming and functional programming in Erlang.

Among the useful features of behavioral programming are:

- Behavioral Modularity: B-threads can be coded relatively independent of each other – interacting mainly through events that are part of overall system behavior. The behavior of multiple b-threads is successfully interwoven, even though each b-thread has little or no awareness of the identity of the other b-threads or of their internal structure.
- Incremental Development: New modules that add or restrict behavior can be added to an existing system with little or no change to existing modules. The new module relates to the behavior and the events of the existing system, and not to its structure and code. For example, applications written in this behavioral approach can be more readily patched to correct errors or handle small changes in requirements. The patch, or the new module, can watch out for the event sequences whose handling should be changed, and override the existing behavior with new behavior.

  The incrementality of behavioral programming allows for observing meaningful behavior from early stages of development. As each b-thread generates observable system behavior, incomplete versions of the systems can be used to start validating or refining requirements and specifications.

  An additional aspect of incremental development can be post-deployment system customization, where an end-user can modify the system behavior by adding b-threads for simplifying certain user tasks.
- Naturalness: In natural language conversations and in requirements documents, people often describe behaviors of systems in terms of scenarios. Therefore, behavioral programming seems like a natural approach to development of software systems.

  Additionally, due in part to the behavioral modularity feature, behavioral programs can more readily "explain" their decisions (and behavior). Events that caused transitions in a recently executed chain of events can provide important insight into the rationale for the program's progress, something that may be harder to infer from a usual trace. This may be useful in developing and debugging behavioral applications, in using b-threads for monitoring, and in developing intelligent agents, expert systems or systems capable of learning.
- Suitability to multi-core and distributed systems: In behavioral programming each behavior thread is associated with an executable system process or thread. This "automatically" structures the developed system as a set of concurrent processes. In the context of Erlang, this enables leveraging the ease and efficiency of handling concurrent processes in this language together with natural decomposition of system behavior, towards a system that utilizes resources effectively while maintaining a natural and robust structure.

The proposed pattern and module are in early development stages and can be considered a proof-of-concept for demonstrating the principles of behavioral programming in Erlang.

Future directions for research include devising higher level idioms to control behavior, and developing domain specific languages based on behavioral programming principles.

For the tool, future work includes robustness improvements and adding functionality that exists already in the BPJ library. For example, BPJ supports event filters - calling of a function to determine membership of an event in the sets of blocked or watched events. This allows more flexible definition of the event sets, and handling of very large, or possibly infinite, sets.

The visualization tool we propose here is focused on comprehension of individual b-threads. It will be interesting to explore visualization techniques that assist in comprehension of sets of b-threads, in particular the interaction between different b-threads, possibly along the line of visualizing LSC dependencies as done in [4].

As a general programming paradigm, behavioral programming is still in very early stages. It is presently manifested in the visual, multi-modal language of LSC, and in the Java library BPJ. We hope that the design pattern proposed here for functional programming in Erlang will help expand the reach of this promising concept, and drive additional research and development required for its success.

Acknowledgments

The research of the first two authors was supported by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University. The research of the third author was supported by an Advanced Research Grant to David Harel from the European Research Council (ERC) under the European Community's FP7 Programme.

References

[1] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995. ISBN 0201633612.

[3] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[4] D. Harel and I. Segall. Visualizing inter-dependencies between scenarios. In R. Koschke, C. D. Hundhausen, and A. Telea, editors, *SOFTVIS*, pages 145–153. ACM, 2008. ISBN 978-1-60558-112-5.

[5] D. Harel, A. Marron, and G. Weiss. The BPJ Library. `www.cs.bgu.ac.il/~geraw`.

[6] D. Harel, A. Marron, and G. Weiss. Programming coordinated scenarios in java. In *Proc. 24th European Conference on Object-Oriented Programming (ECOOP 2010)*, 2010. to appear.

[7] ITU. *International Telecommunication Union Recommendation Z.120: Message Sequence Charts*. 1996.

[8] R. Keller. Formal verification of parallel programs. *CACM*, 19(7): 371–384, 1976. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360248.360251.

[9] OMG. *Unified Modeling Language Superstructure Specification, v2.0*. Aug. 2005. URL `http://www.omg.org`.

[10] A. D. Stefano and C. Santoro. Using the erlang language for multi-agent systems implementation. In *IAT*, pages 679–685. IEEE Computer Society, 2005. ISBN 0-7695-2416-8.

[11] A. D. Stefano, F. Gangemi, and C. Santoro. Eresye: artificial intelligence in erlang programs. In Erlang Workshop pages 62–71. ACM, 2005. ISBN 1-59593-066-3.

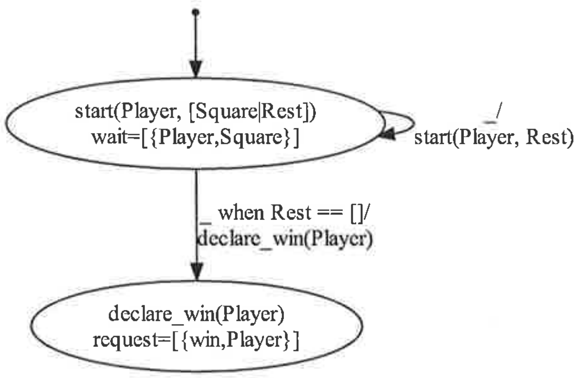


Figure 8. The `detect_win` b-thread. An instance is spawned for each of the 6 permutation of the 3 events that comprise one of the 8 winning lines (3 vertical, 3 horizontal, and 2 diagonal) for each of the two players (total of 96 instances). For example, an instance that waits for $\langle x, 9\rangle$, $\langle x, 5\rangle$, $\langle x, 1\rangle$, and announces a win by x is started by the line **spawn**`(detect_win, start, [x, [9,5,1]])`. For this example, in the first state, `Square` is matched with 9 and `Rest` with `[5,1]`. The b-thread waits for $\langle x, 5\rangle$ and, if that event occurs, `start(x, [5,1])` is called where `Square` is matched with 5 and `Rest` with `[1]` and so on. Eventually, if $\langle x, 5\rangle$ and afterwards $\langle x, 1\rangle$ occur, the b-thread requests the event $\langle win, x\rangle$ to announce that x won the game.

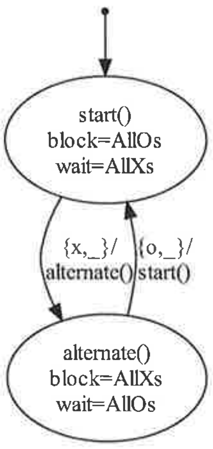


Figure 9. The `enforce_turns` b-thread. The variables `AllO` and `AllX` are lists of all the o events and all the x events, respectively. Turn enforcement is achieved by alternately blocking all o or all x events.

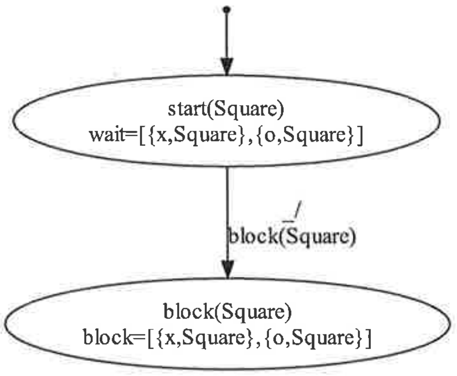


Figure 10. The `disallow_square_reuse` b-thread. Each of the nine instances of this b-threads waits for either $\langle x, Sq\rangle$ or $\langle o, Sq\rangle$, for a particular value of $Sq \in \{1, \ldots, 9\}$ and, when one of these two events is observed, the b-thread blocks them both forever.

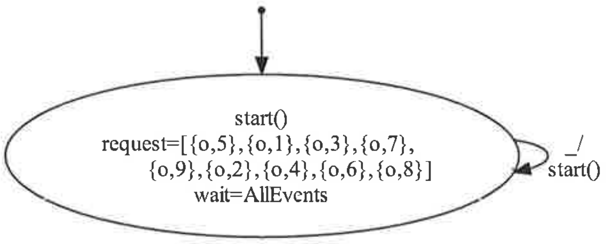

start()
request=[{o,5},{o,1},{o,3},{o,7},
{o,9},{o,2},{o,4},{o,6},{o,8}]
wait=AllEvents
_/
start()

Figure 11. The `default_moves` b-thread. Requested moves will be triggered only when not blocked and when there are no higher priority (unblocked) requests. The order of the requested events determines their priorities in our strategy: try to mark the center square first, then the corners, and only then the remaining squares.

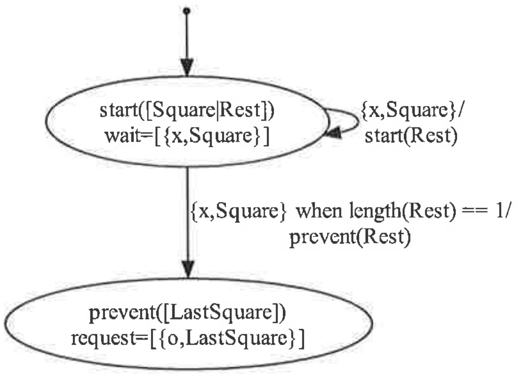


Figure 12. The `prevent_lines_with_two` b-thread. The parameter to the first state of the b-thread (called start) is a list of length three containing the three squares of a (vertical, horizontal, or diagonal) line that the b-thread is protecting (a copy of this b-thread is instantiated for every permutation of the squares of each line). Initially, the b-thread is waiting for the opponent to play the first square on the line. When this happens, the b-thread takes a self transition to the state `start` and waits for the opponent to mark the second square . Then, if the opponent marks the second square, the b-thread moves to the state `prevent` where a request to put an o on the last square is issued.

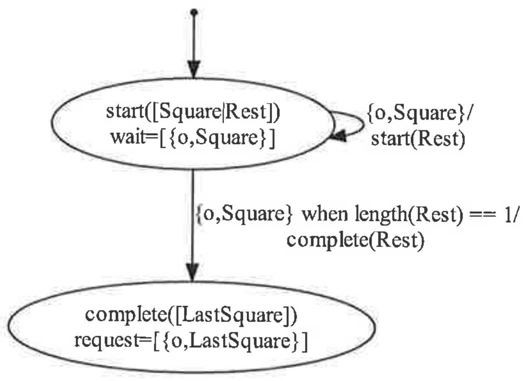


Figure 13. The `complete_lines_with_two` b-thread. Similar in structure to the `prevent_lines_with_two` b-thread, shown in Figure 12. The difference is that this b-thread is waiting for o moves. Note that the b-thread is independent of the scenarios that lead to requests of the first two o moves (issued by other b-threads).

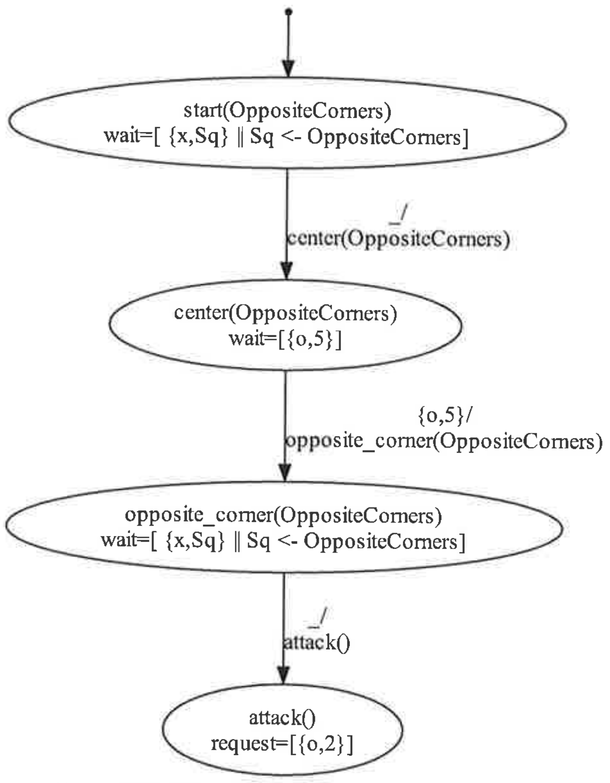

start(OppositeCorners)
wait=[{x,Sq} || Sq <- OppositeCorners]
_/
center(OppositeCorners)
center(OppositeCorners)
wait=[{o,5}]
{o,5}/
opposite_corner(OppositeCorners)
opposite_corner(OppositeCorners)
wait=[{x,Sq} || Sq <- OppositeCorners]
_/
attack()
attack()
request=[{o,2}]

Figure 15. The `intercept_double_fork` b-thread. Instances correspond to two opposite corners. The b-thread waits for x to mark one of the corners, then for o to mark the center and, lastly, if x marks the opposite corner, the b-thread requests to mark square number 2 (the middle of the upper row) with an o.

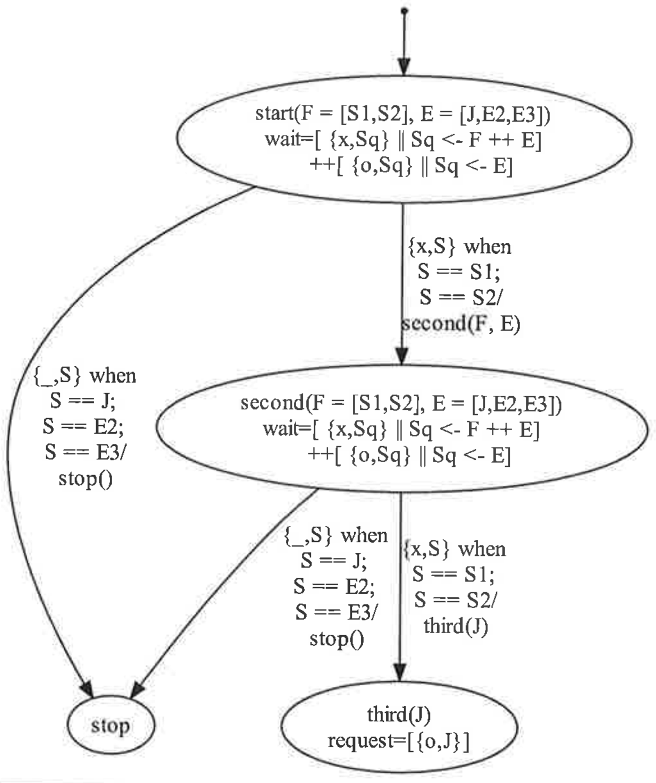


Figure 14. The `intercept_single_fork` b-thread. Instances of this b-thread correspond to five squares: `S1,S2,J,E1,E2` that form two intersecting lines where J is the junction, i.e., the square where both lines intersect. The thread takes action if `E1,E2` remain empty and `S1,S2` are marked by x. When this happens, the thread requests to mark J with o.

A Unified Semantics for Future Erlang

Hans Svensson
Department of Computer Science and Engineering,
Chalmers University of Technology,
Göteborg, Sweden
hanssv@chalmers.se

Lars-Åke Fredlund Clara Benac Earle
Facultad de Informática,
Universidad Politécnica de Madrid,
Spain
{fred,cbenac}@babel.ls.fi.upm.es

Abstract

The formal semantics of Erlang is a bit too complicated to be easily understandable. Much of this complication stems from the desire to accurately model the current implementations (Erlang/OTP R11-R14), which include features (and optimizations) developed during more than two decades. The result is a two-tier semantics where systems, and in particular messages, behave differently in a local and a distributed setting. With the introduction of multi-core hardware, multiple run-queues and efficient SMP support, the boundary between local and distributed is diffuse and should ultimately be removed. In this paper we develop a new, much cleaner semantics, for such future implementations of Erlang. We hope that this paper can stimulate some much needed debate regarding a number of poorly understood features of current and future implementations of Erlang.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics

General Terms Languages, Verification

Keywords Erlang, Semantics

1. Introduction

In the past years quite a lot of effort has been spent on describing and defining the semantics of Erlang; from the early work of Petterson [5], via the formal definition of single node semantics by Fredlund [3] and the more high-level language philosophy description by Armstrong [1], to the definition of the distributed Erlang semantics by Claessen and Svensson [2] later refined by Svensson and Fredlund [6]. The end result is an accurate formal semantics of Erlang, that in detail describe how the current (Erlang/OTP R11-R14) implementation behave. The successful implementation of McErlang (by Fredlund and Svensson [4]) is further evidence that the semantics is actually useful and can be used for practical purposes.

Unfortunately, the formal semantics is a bit too complicated to be easily understandable. This is a bit of a nuisance, since the language design philosophy [1] is very clean and easy to grasp. When analyzing the current Erlang semantics we could see that much of the complexity stems from the strive to very closely follow the current implementation of the Erlang run-time system (ERTS). The result is a two-tier semantics where systems, and in particular messages, behave differently in a local and a distributed setting. Also, the Erlang language has evolved over the years, leaving some legacy constructions that could have been avoided if the language was re-constructed today. The most striking example is the overlap between *monitors* and *links*, where there is very little practical difference between a *trapped* link-message and a monitor-message.

With the current trend being the introduction of multi-core hardware, and the latest advances in the ERTS with multiple run queues and efficient SMP support; the boundary between local and distributed is diffuse and should ultimately be removed. In this paper we propose a new, much cleaner semantics for a future implementation of Erlang, where there is no boundary between local and remote processes and where the semantics does not in itself hinder parallelization.

It is not our intent to define the future semantics of Erlang, but we hope to initiate, and stimulate, the discussion and debate of this topic. We believe that it is vital to lift some of the current restrictions in order for Erlang to continue to scale to upcoming multi-core architectures. We are also well aware of all the complications involved in (drastically) changing the semantics of a mature programming language. The vast amount of legacy code, and backwards compatibility are large hurdles to overcome. Nevertheless, the earlier we start thinking about these improvements, the more time we have to make them before reality catches up!

Paper Organization The paper is organized as follows. In Sect. 2 we begin by giving a high-level intuitive overview of the new semantics, before we formally state definitions and rules of the new semantics in Sect. 3. In Sect. 4 we describe the inner workings of the node controller. Thereafter, in Sect. 5, we restrict the possible execution sequences by stating a rule for fairness. Thereafter, in Sect. 6 we make a few illustrative comparisons between the new and the current semantics. Finally, we conclude and add some future directions in Sect. 7.

2. Intuitive Semantics

Before we start listing definitions and semantic rules, we provide an informal but hopefully intuitive high-level overview of the new semantics.

In our setting, a complete distributed system consists of a number of *nodes*. The nodes are the top-level containers in such a system. A node contains a *node controller* (an addition in this new semantics), and a number of *processes*. One should note that a node is not equivalent to a physical machine, rather the opposite, a single machine can host many nodes. This high-level description of a distributed system is illustrated in Figure 1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09...$10.00

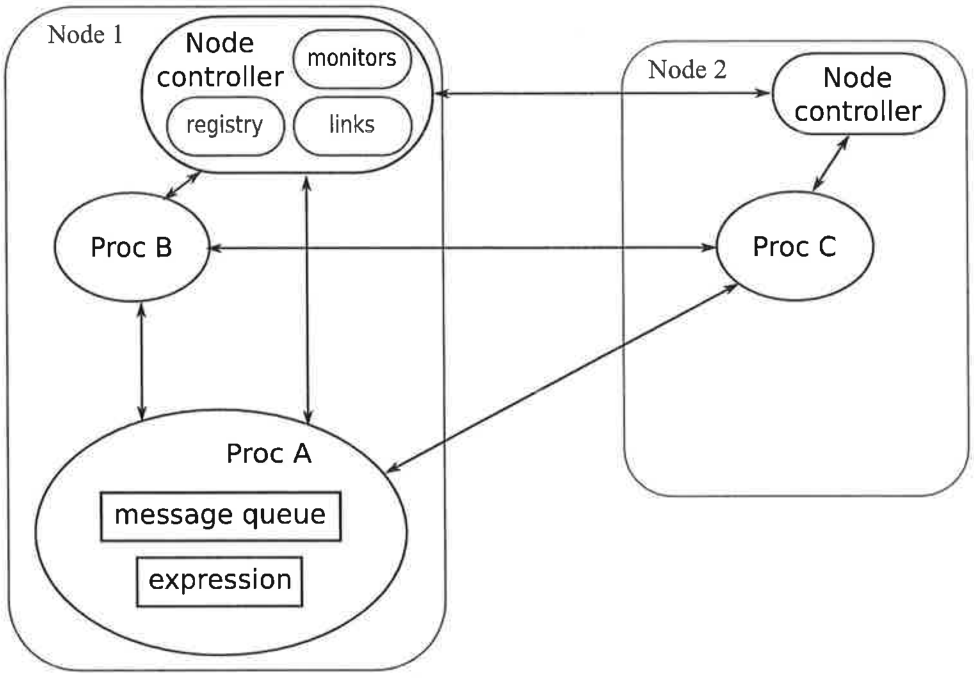


Figure 1. Distributed system - Nodes, node controllers, and processes

2.1 Everything is Distributed

As mentioned in the introduction, with the introduction of multi-core architectures, the boundary between local and remote is more diffuse. Therefore, the new semantics treats all messages (including messages sent to one-self!) equally, and all messages are sent through a (virtual) system message queue (the *ether*). In practice this means that every message-passing consists of two steps, sending and delivering. Thus, messages sent between different pairs of processes can be freely re-ordered.

We have also chosen to make (almost) all *side-effecting* actions (such as spawning a new process, linking to a process, etc.) asynchronous. To stress that many side effects have a similar impact on the system, they are treated in a uniform way by the introduction of a *node controller*. The node controller is responsible for all node-local administration.

As an example of a side effect, consider registration of a name for a process. This is done by sending a signal to the node controller. Some time later the node controller receives the signal, decides whether the name can be registered and sends a reply to the process doing the registration. This might seem a bit impractical from a user perspective, but nothing stops us from defining a higher-level function that sends the register-signal **and** waits for the reply.

2.2 Uni-directional Links Only

In the new semantics we provide both the concept of *links* and *monitors*. However, we do not have any functionality like *trapping exits*[1] that exists in the current semantics. This means that if a linked process terminates, the linking process is also terminated, no 'but's and 'if's. (The trap exit functionality is in practice the same as a monitor.) We have also opted for making links and monitors uni-directional. That is if process A is linking to process B, a failure of A does not affect B.

As a practical example let us consider how to re-implement the supervisor behavior using uni-directional links and monitors. Assuming that we want to supervise a child process, specified as the tuple `{M,F,A}`, so that when the child terminates the supervisor is informed, and if the supervisor terminates abnormally, the child terminates too, then the following supervisor code fragment suffices:

```
SupervisorPid = self(),
ChildPid =
  spawn(fun () ->
          link(SupervisorPid), apply(M,F,A)
        end),
MonitorRef = monitor(process, ChildPid),
```

2.3 A Built-in Registry

We have decided to make an addition to the semantics by including a process registry in the semantics. This inclusion is questionable, but we think that the ability to communicate with named processes is such a central concept in Erlang that it deserves a place in the semantics.

As in current Erlang, the basic operations supported are sending a message to a named process (`atom!msg`), sending a message to a named process on a remote node (`{atom,node}!msg`), and registering, unregistering and name lookup. However, for uniformity, in this semantics names can be registered for remote processes (i.e., `register(name,pid)` does not fail if `pid` is a remote process), and registering a local process at a remote node is supported too (using the operation `register(node,name,pid)`). As a consequence, when a message is sent to a remote node using the syntax `{atom,node}!msg` there is no guarantee that the process that should receive the message is located at `node`; thus it may be necessary to relay the message to a process on yet another node.

[1] In the current semantics, processes are able to trap exit-signals, and treat them as ordinary (information-)messages, thus avoiding termination upon receiving an exit-signal.

2.4 Message-Passing Guarantees

There are few message-passing guarantees in general in the new semantics, but for each pair of processes the order of messages is guaranteed. That is, if process A sends a stream of messages $M_1, M_2, M_3, \cdots$ to process B, then process B will receive the messages in exactly that order. (With the possibility of dropping messages at the end of the sequence because of a node disconnect.)

It should be noted that this guarantee matches exactly what is outlined by Armstrong [1] in his thesis. However note that as observed in Svensson and Fredlund [7], current Erlang implementations does not provide this guarantee, as when distributed processes communicate, messages may be lost.

Further note that in the semantics there are no guarantees regarding the ordering of messages delivered to a process if that process is addressed both directly (using its pid) and indirectly through a registered name. To exemplify we assume that a process P executes the code fragment `Q!msg1, q_name!msg` where `Q` is bound to the pid of a process Q also registered under the name `q_name`. In such a scenario there are no guarantees provided by the semantics regarding whether `msg1` or `msg2` is delivered first to the mailbox of Q.

3. Formal Semantics

In this section we present the semantics in a style similar to earlier Erlang formal semantics. We use this style since it is straightforward and easy to follow, while still being detailed enough to allow precise arguments about correctness. We make the necessary definitions before presenting the semantic rules.

Definition 1 A *process*, ranged over by $p \in$ Process, is a triplet: Expression $\times$ ProcessIdentifier $\times$ MessageQueue, written $\langle e, pid, q\rangle$ such that

- e is an expression currently run by the process,
- pid is the process identifier of the process,
- q is a message queue.

The expression (e) in a process should be interpreted as a normal (Erlang) expression similar to the expressions defined in [3].

Definition 2 A *process group*, ranged over by $pg \in$ ProcessGroup, is either an empty process group $\emptyset$, a single process, or a combination of process groups pg_1 and pg_2, written as $pg_1 \parallel_P pg_2$.

Definition 3 A *node controller*, ranged over by $nc \in$ NodeController, is a triplet: $\mathcal{P}$(ProcessIdentifier $\times$ ProcessIdentifier) $\times$ $\mathcal{P}$(MonitorReference $\times$ ProcessIdentifier $\times$ ProcessIdentifier) $\times$ $\mathcal{P}$(ProcessIdentifier $\times$ ProcessName), written $\langle lnks, mns, reg\rangle$ such that

- $lnks$ is a set of links, i.e. tuples ($link\_from$, $link\_to$),
- mns is a list of monitors, i.e. tuples (mon_name, mon_from, mon_to),
- reg is a set of registered names, i.e. tuples ($name$, pid).

Definition 4 A *System message queue*, also named an *ether*, ranged over by $eth \in$ SystemMessageQueue, consists of a finite sequence of triplets Identifier $\times$ Identifier $\times$ Signal. Let ϵ denote the empty sequence, $(\cdot)$ is concatenation and $(\ \setminus\ )$ is deletion of the first matching triplet, e.g.:

$$\begin{aligned} eth &= (a_2, b_1, c_1)\cdot(a_1, b_2, c_1)\cdot(a_1, b_2, c_2)\cdot(a_1, b_2, c_1) \setminus (a_1, b_2, c_1) \\ &= (a_2, b_1, c_1)\cdot(a_1, b_2, c_2)\cdot(a_1, b_2, c_1) \end{aligned}$$

Definition 5 A *node*, ranged over by $n \in$ Node, is a triple: ProcessGroup $\times$ NodeIdentifier $\times$ NodeController, it is written $[pg, nid, nc]$ such that

- pg is a group processes running at the node,
- nid is a unique identifier for the node,
- nc is a node controller.

Let Identifier be the union of process identifiers (ProcessIdentifier) and node identifiers (NodeIdentifier), and let identifiers be ranged over by $id \in$ Identifier.

Definition 6 A *node system*, ranged over by $ns \in \mathcal{P}$ Node, is either an empty node system $\emptyset$, a single node, or a combination of node systems ns_1 and ns_2, written as $ns_1 \parallel_N ns_2$.

Definition 7 A *system*, ranged over by $s \in$ System, is a tuple: $\mathcal{P}$ Node $\times$ SystemMessageQueue, written $[\![ns, eth]\!]$ such that

- ns is a node system, and
- eth is an ether (system message queue).

Intuitively, the composition of processes into process groups and nodes into node systems should be thought of as a set of processes (nodes). We will take care to define the semantics in such a way as to ensure that the operators $\parallel_P$ and $\parallel_N$ are commutative and associative.

To keep the semantic rules reasonably short and readable we use some supportive functions to abbreviate some lengthy (and often repeated) constructions.

Definition 8 Let the function isNid(i) (where $i \in$ Identifier) return true if the identifier i represents a node identifier, and false if it represents a process identifier.

Definition 9 Let the function node(p) (where $p \in$ ProcessIdentifier) return the node identifier for a given process identifier.

Definition 10 Let the function destNid(sig) return the node identifier of the remote node involved in the signal sig (note that the return value is `undefined` for some signals), e.g.

$$\begin{array}{lll} \mathsf{destNid}(\mathbf{link}(pid)) & \longrightarrow & \mathsf{node}(pid), \text{ and} \\ \mathsf{destNid}(\mathbf{spawn}(e, ref)) & \longrightarrow & \mathtt{undefined} \end{array}$$

Definition 11 ethMatch(eth,to,$from$), is a function that given a system message queue, a sender identity ($from$), and a receiver identity (to) returns the first message in the queue sent by $from$ to to, e.g.

$$\begin{array}{lll} eth & = & (a_2, b_1, c_1)\cdot(a_1, b_2, c_1)\cdot(a_1, b_2, c_2)\cdot(a_1, b_2, c_1) \\ & \Longrightarrow & \mathsf{ethMatch}(eth, a_1, b_2) = c_1 \end{array}$$

Definition 12 Let the functions pids(pg) and nids(ns) return the set of process identifiers belonging to processes in the process group pg and the set of node identifiers in the node system ns respectively. Further let a process group and a system be *well-formed* if its process identifiers and node identifiers are unique, i.e., a process identifier belongs to at most one process and a node identifier to at most one node, and an identifier is either a node identifier or a process identifier.

In the following we assume that all (Erlang) node systems are well-formed and that they only contain well-formed process groups.

In the semantics, *signals* are items of information transmitted between a sending and a receiving process (or node controller). A *process action*, committed by a process, group of processes, or node, is either a silent action, an input action, an output action, a node termination, or a node disconnect.

Definition 13 (Process signals) The process signals, ranged over by $sig \in \mathsf{Signal}$ are:

$$
\begin{array}{rll}
sig ::= & \mathbf{message}(v) & \text{message} \\
\mid & \mathbf{link}(pid) & \text{linking with process} \\
\mid & \mathbf{unlink}(pid) & \text{unlinking process} \\
\mid & \mathbf{monitor}(pid, ref) & \text{monitor process} \\
\mid & \mathbf{unmonitor}(ref) & \text{unmonitor process} \\
\mid & \mathbf{monitor\_node}(nid) & \text{monitor node} \\
\mid & \mathbf{unmonitor\_node}(ref) & \text{unmonitor node} \\
\mid & \mathbf{whereis}(name) & \text{lookup pid for name} \\
\mid & \mathbf{register}(name, pid) & \text{register name for pid} \\
\mid & \mathbf{spawn}(e, ref) & \text{spawn a process} \\
\mid & \mathbf{spawn\_node}() & \text{spawn a node} \\
\mid & \mathbf{nsend}(name, v) & \text{named send} \\
\mid & \mathbf{exit}(v) & \text{external termination signal} \\
\mid & \mathbf{died}(id, v) & \text{termination signal}
\end{array}
$$

Definition 14 (Process actions) The process actions, ranged over by $\alpha \in \mathsf{Action}$ are:

$$
\begin{array}{rll}
\alpha ::= & \tau & \text{silent action} \\
\mid & pid\ !_{from}\ sig & \text{output action} \\
\mid & pid\ ?_{from}\ sig & \text{input action} \\
\mid & \mathbf{die}(nid) & \text{node termination} \\
\mid & \mathbf{disconnect}(nid_1, nid_2) & \text{node disconnect}
\end{array}
$$

In the following we define formally the possible computation steps of an Erlang system. In this definition we assume the existence of a set of transition rules for expressions, a subset of $\mathcal{P}(\mathsf{Expression} \times \mathsf{exprAction} \times \mathsf{Expression})$, which can be found in [3]. For completeness we repeat below the definition of the expression actions:

Definition 15 (Expression actions) The expression actions, ranged over by $\alpha \in \mathsf{exprAction}$, are:

$$
\begin{array}{rll}
\alpha ::= & \tau & \text{computation step} \\
\mid & pid\ !\ v & \text{output} \\
\mid & \mathbf{exiting}(v) & \text{exception} \\
\mid & \mathbf{read}(q, v) & \text{read from queue} \\
\mid & \mathbf{test}(q) & \text{checking queue contents} \\
\mid & f(v_1, \ldots, v_n) \rightsquigarrow v & \text{built-in function call}
\end{array}
$$

Intuitively, $f(v_1, \ldots, v_n) \rightsquigarrow v$ corresponds to the call of a built-in function f which returns the value v.

Definition 16 Let the function $\mathsf{mkAction}(msgs)$ be defined as follows:

$$
\begin{array}{ll}
\mathsf{mkAction}(\epsilon) & \longrightarrow \tau \\
\mathsf{mkAction}((to, from, sig) \cdot msgs) & \longrightarrow to\ !_{from}\ sig\ ; \mathsf{mkAction}(msgs)
\end{array}
$$

Definition 17 The system transition relation is the least relation satisfying the transition rules in Tables 1 - 9

It should be pointed out that some rules should be combined together to achieve the desired effect. The most obvious examples are perhaps the output-rules, where the message is sent in the *side_eff*-rule in Table 2 and delivered in the *output*-rule in Table 5.

Most of the semantic rules are self explaining, but we go through each table below and point out some subtleties and explain the more complicated rules.

Expression evaluation in process context Table 1 contains expression evaluation that is local to a process. Note that all these actions take place in a larger context (a system), but since the rules do

$$
\mathit{silent}\ \frac{e \xrightarrow{\tau} e'}{\langle e, pid, q\rangle \xrightarrow{\tau} \langle e', pid, q\rangle}
$$

$$
\mathit{read}\ \frac{e \xrightarrow{\mathbf{read}(q_1, v)} e'}{\langle e, pid, q_1 \cdot v \cdot q_2\rangle \xrightarrow{\tau} \langle e', pid, q_1 \cdot q_2\rangle}
$$

$$
\mathit{test}\ \frac{e \xrightarrow{\mathbf{test}(q)} e'}{\langle e, pid, q\rangle \xrightarrow{\tau} \langle e', pid, q\rangle}
$$

$$
\mathit{self}\ \frac{e \xrightarrow{\mathtt{self}() \rightsquigarrow pid} e'}{\langle e, pid, q\rangle \xrightarrow{\tau} \langle e', pid, q\rangle}
$$

Table 1. Rules for process-local expression evaluation

$$
\mathit{side\_eff}\ \frac{e \xrightarrow{side\_eff(args) \rightsquigarrow res} e' \qquad (res, id, sig) = \mathsf{mkSig}(\mathsf{node}(pid), side\_eff, args)}{\langle e, pid, q\rangle \xrightarrow{id\ !_{pid}\ sig} \langle e', pid, q\rangle}
$$

Table 2. Rules for side effecting expression evaluation

$$
\begin{array}{ll}
\mathsf{mkSig}(nid, !, [pid, v]) & \longrightarrow (v, pid, \mathbf{message}(v)) \\
\mathsf{mkSig}(nid, !, [\{name, nid'\}, v]) & \longrightarrow (v, nid', \mathbf{nsend}(name, v)) \\
\mathsf{mkSig}(nid, !, [name, v]) & \longrightarrow (v, nid, \mathbf{nsend}(name, v)) \\
\mathsf{mkSig}(nid, \mathtt{exit}, [pid, v]) & \longrightarrow (\mathtt{ok}, pid, \mathbf{exit}(v)) \\
\mathsf{mkSig}(nid, \mathtt{link}, [pid]) & \longrightarrow (\mathtt{ok}, nid, \mathbf{link}(pid)) \\
\mathsf{mkSig}(nid, \mathtt{unlink}, [pid]) & \longrightarrow (\mathtt{ok}, nid, \mathbf{unlink}(pid)) \\
\mathsf{mkSig}(nid, \mathtt{spawn}, [e, ref]) & \longrightarrow (ref, nid, \mathbf{spawn}(e, ref)) \\
\mathsf{mkSig}(nid, \mathtt{spawn}, [nid', e, ref]) & \longrightarrow (ref, nid', \mathbf{spawn}(e, ref))
\end{array}
$$

Table 3. Definition of mkSig()

not depend on anything outside the process, the context is not visible in the rules. (The rules are lifted to the system level by the *internal*-rule defined in Table 5.) In the rule *silent*, if the expression e has a transition $e \xrightarrow{\tau} e'$ (a normal computation step) then the process (and in the greater scheme the whole system) $\langle e, pid, q\rangle$ has a transition labelled by the silent action τ to the process $\langle e', pid, q\rangle$.

In the *read*-rule, a transition from $\langle e, pid, q_1 \cdot v \cdot q_2\rangle$ to the target process $\langle e', pid, q_1 \cdot q_2\rangle$ is enabled whenever the process mailbox (queue) can be split into three parts $q_1 \cdot v \cdot q_2$, and the expression transition *receive*: $e \xrightarrow{\mathbf{read}(q_1, v)} e'$ is derivable at the expression level. (See [3] for the exact definition of the *receive*-rule.) Thus, the rules *read* and *receive* together ensure the intuitive semantics of the `receive` construct. The side effecting actions are handled by the rule in Table 2.

Side-effecting (node controller) expression evaluation Table 2 contains the rule for side effecting actions. Many of these actions require involvement of the node controller. The *side_eff* rule is generic, and include the actions: *send, exit, link, unlink, monitor, unmonitor, spawn, register, whereis, monitor_node, and unmonitor_node*. All rules in the table are *asynchronous*, e.g. `whereis()` returns immediately with `ok`, whereas the "real" result (for example for *whereis* and *unlink*) arrive later (as a normal receivable signal). The asynchronous nature of these operations results in deceptively simple semantic rules. In Table 7 the node controller side of these rules is presented, and in Sect. 4 the more complex internals of the node controller is explained. The *side_eff*-rule uses the translation

$$\mathit{node}\ \frac{e \xrightarrow{\mathtt{node}() \rightsquigarrow nid} e'}{[\![[\langle e, pid, q\rangle \parallel_P pg, nid, nc] \parallel_N ns, eth]\!] \xrightarrow{\tau} [\![[\langle e', pid, q\rangle \parallel_P pg, nid, nc] \parallel_N ns, eth]\!]}$$

$$\mathit{termination}\ \frac{sig = \mathbf{died}(pid, \mathtt{normal})}{[\![[\langle v, pid, q\rangle \parallel_P pg, nid, nc] \parallel_N ns, eth]\!] \xrightarrow{nid\ !_{pid}\ sig} [\![[pg, nid, nc] \parallel_N ns, eth \cdot (nid, pid, sig)]\!]}$$

$$\mathit{exiting}\ \frac{e \xrightarrow{\mathbf{exiting}(v)} e' \qquad sig = \mathbf{died}(pid, v)}{[\![[\langle e, pid, q\rangle \parallel_P pg, nid, nc] \parallel_N ns, eth]\!] \xrightarrow{nid\ !_{pid}\ sig} [\![[pg, nid, nc] \parallel_N ns, eth \cdot (nid, pid, sig)]\!]}$$

$$\mathit{spawn}_{node}\ \frac{e \xrightarrow{\mathtt{spawn\_node}() \rightsquigarrow nid'} e' \qquad nid' = \mathsf{fresh}()}{[\![[\langle e, pid, q\rangle \parallel_P pg, nid, nc] ns, eth]\!] \xrightarrow{\tau} [\![[\langle e', pid, q\rangle \parallel_P pg, nid, nc] \parallel_N [\emptyset, nid', \langle \emptyset, \emptyset, \emptyset\rangle] \parallel_N ns, eth]\!]}$$

Table 4. Process rules for expression evaluation at system level

function mkSig to construct an appropriate *signal*, this function is defined in Table 3.

Expression evaluation in node context Table 4 contains the simple rule for evaluation of node() in the system context. The function returns immediately (i.e. it is not asynchronous). Because the result is depending on the node context, the rule is separated from the rules in Table 1.

The *termination*- and the *exiting*-rule are also evaluated at the system level. Terminated processes are removed from the system as seen in Table 4.

The last rule in Table 4 handle the slightly odd action of spawning a new *node*. This action is an odd bird in the semantics, and the obvious place for such a rule would be in the general *side_eff*-rule. However, since the result is the *creation* of a whole node it does not fit into the general pattern. Moreover, making the pattern even more general was not too appealing, instead the rule is listed on its own here. The rule makes sure that a new node is created, with a fresh node identifier.

Node level input- and output-rules Table 5 and Table 6 contains input and output rules. The *output*-rule deliver messages to the system message queue (the *ether*), while the *internal*-rule simply lifts non-output expression evaluations to the system level (as discussed above).

For the *input*-rules we should note that the rules can be applied in an arbitrary order for pairs of a sender and a receiver. This means that messages (from different senders and receivers) can possibly be reordered. However, at the same time this introduces a problem, namely that a certain (sender,receiver)-pair is never considered. That means that the delivery of some messages could potentially be delayed forever. Many properties can not be proved for such a non-fair situation, to deal with this problem we have to state a fairness rule (in Sect. 5). Also, since nothing stops processes from sending messages to dead processes, we need rules (the *missing*-rules) to eventually remove such messages. The $\mathit{missing}_{node}$-rule is also responsible for sending some node controller reply-messages, such as link-replies (**noproc**) and spawn-replies (a useless *pid*), by using the ncEffect()-function. The ncEffect()-function is defined in Sect. 4.

The *exit*-rule handles external abnormal termination of processes, the reason for termination could be a triggered link or an explicit call to `exit()`. The receiving process is terminated and the node controller is (eventually) informed.

The presented message-passing mechanism is totally asynchronous, even messages sent to oneself are delivered through the system message queue. Finally, note that messages to node controllers are dealt with in Table 7.

Node controller (meta-)rules Table 7 contains meta-rules for the *node controller*. The rules describe how node controller signals are treated. There are two cases, in the normal case (the *nc*-rule) the signal is at its final destination and needs only to be handled at this node controller. In the second case (the $nc_{forward}$-rule) we handle a signal involving a remote pid at the sender side; i.e. the signal should be passed on after doing local actions. Specifying these actions and replies is the meat of these rules. The actions (i.e. the definitions of the function ncEffect) are presented in Sect. 4. To distinguish local signals from remote signals the function destNid() is used, it is defined in Table 8.

It should be noted that the node controller does not have a separate message queue. Signals sent to the node controller are consumed immediately upon delivery. Thus, there is no selective receive for node controllers. The *nc*-rule and the $nc_{forward}$-rule are also special in the sense that they (potentially) have more than one action attached to their transitions, both an input action and output action(s). The interpretation of multiple actions for one transition is straightforward; the actions are ordered, just as if there had been several consecutive transitions with a single action. The function mkAction() is used to create multiple actions from a sequence of messages.

$\mathsf{destNid}(\mathbf{link}(pid))$	$\longrightarrow$	$\mathsf{node}(pid)$
$\mathsf{destNid}(\mathbf{unlink}(pid))$	$\longrightarrow$	$\mathsf{node}(pid)$
$\mathsf{destNid}(\mathbf{spawn}(e, ref))$	$\longrightarrow$	$\mathtt{undefined}$
$\mathsf{destNid}(\mathbf{nsend}(name, v))$	$\longrightarrow$	$\mathtt{undefined}$

Table 8. Definition of destNid()

Node failure rules Table 9 contains rules for failing nodes, either when a single node crash or when two nodes are disconnected. In both cases the node controllers are informed. Also here we see how deferring most of the work to the node controller saves us from complex semantic rules. The corresponding rules in the distributed Erlang semantics fills the majority of a page. Nevertheless, the complexity does not fully disappear, we still have to deal with some bookkeeping inside the node controller.

$$\textit{internal}\ \frac{p \xrightarrow{\tau} p'}{\llbracket [p \parallel_P pg, nid, nc] \parallel_N ns, eth \rrbracket \xrightarrow{\tau} \llbracket [p' \parallel_P pg, nid, nc] \parallel_N ns, eth \rrbracket}$$

$$\textit{output}\ \frac{p \xrightarrow{to\,!_{from}\,sig} p'}{\llbracket [p \parallel_P pg, nid, nc] \parallel_N ns, eth \rrbracket \xrightarrow{to\,!_{from}\,sig} \llbracket [p' \parallel_P pg, nid, nc] \parallel_N ns, eth \cdot (to, from, sig) \rrbracket}$$

Table 5. System output rules

$$\textit{input}\ \frac{\mathsf{ethMatch}(eth, to, from) = sig = \mathbf{message}(v)}{\llbracket [\langle e, to, q\rangle \parallel_P pg, nid, nc] \parallel_N ns, eth \rrbracket \xrightarrow{to\,?_{from}\,sig} \llbracket [\langle e, to, q \cdot sig\rangle \parallel_P pg, nid, nc] \parallel_N ns, eth \setminus (to, from, sig) \rrbracket}$$

$$\textit{exit}\ \frac{\mathsf{ethMatch}(eth, pid, from) = \mathbf{exit}(v)}{\begin{array}{l}\llbracket [\langle e, pid, q\rangle \parallel_P pg, nid, nc] \parallel_N ns, eth \rrbracket \xrightarrow{pid\,?_{from}\,\mathtt{exit}(v)\ ;\ nid\,!_{pid}\,\mathbf{died}(pid,v)} \\ \qquad \llbracket [pg, nid, nc] \parallel_N ns, eth \setminus (pid, from, \mathbf{exit}(v)) \cdot (nid, pid, \mathbf{died}(pid, v)) \rrbracket\end{array}}$$

$$\textit{missing}_{process}\ \frac{\begin{array}{c}\mathsf{ethMatch}(eth, to, from) = sig \qquad \neg\mathsf{isNid}(to) \\ (\mathsf{node}(to) = nid\ \wedge\ to \notin \mathsf{pids}(pg))\ \vee\ \mathsf{node}(to) \notin \mathsf{nids}(ns)\end{array}}{\llbracket [pg, nid, nc] \parallel_N ns, eth \rrbracket \xrightarrow{to\,?_{from}\,sig} \llbracket [pg, nid, nc] \parallel_N ns, eth \setminus (to, from, sig) \rrbracket}$$

$$\textit{missing}_{node}\ \frac{\begin{array}{c}\mathsf{ethMatch}(eth, to, from) = sig \qquad \mathsf{isNid}(to) \qquad to \notin \mathsf{nids}(ns) \\ (\_, msgs) \longleftarrow \mathsf{ncEffect}([\emptyset, to, \langle\emptyset, \emptyset, \emptyset\rangle], from, sig)\end{array}}{\llbracket ns, eth \rrbracket \xrightarrow{to\,?_{from}\,sig} \llbracket ns, eth \setminus (to, from, sig) \cdot msgs \rrbracket}$$

Table 6. System input rules

$$\textit{nc}\ \frac{\begin{array}{c}\mathsf{ethMatch}(eth, nid, from) = sig \qquad \mathsf{destNid}(sig) = nid\ \vee\ \mathsf{destNid}(sig) = \mathtt{undefined} \\ ([pg', nid, nc'], msgs) \longleftarrow \mathsf{ncEffect}([pg, nid, nc], from, sig)\end{array}}{\llbracket [pg, nid, nc] \parallel_N ns, eth \rrbracket \xrightarrow{nid\,?_{from}\,sig\ ;\ \mathsf{mkAction}(msgs)} \llbracket [pg', nid, nc'] \parallel_N ns, eth \setminus (nid, from, sig) \cdot msgs \rrbracket}$$

$$\textit{nc}_{forward}\ \frac{\begin{array}{c}\mathsf{ethMatch}(eth, nid, from) = sig \qquad \mathsf{destNid}(sig) = nid' \qquad nid \neq nid' \\ ([pg', nid, nc'], msgs) \longleftarrow \mathsf{ncEffect}([pg, nid, nc], from, sig) \qquad msgs' = (nid', from, sig) \cdot msgs\end{array}}{\llbracket [pg, nid, nc] \parallel_N ns, eth \rrbracket \xrightarrow{nid\,?_{from}\,sig\ ;\ \mathsf{mkAction}(msgs')} \llbracket [pg', nid, nc'] \parallel_N ns, eth \setminus (nid, from, sig) \cdot msgs' \rrbracket}$$

Table 7. Meta-rules for node controller

$$\textit{node}_{failure}\ \frac{msgs = \{(nid', nid, \mathbf{died}(nid, \mathtt{nodedown})) \mid nid' \in \mathsf{nids}(n)\}}{\llbracket [pg, nid, nc] \parallel_N ns, eth \rrbracket \xrightarrow{\mathsf{die}(nid)\ ;\ \mathsf{mkAction}(msgs)} \llbracket ns, eth \cdot msgs \rrbracket}$$

$$\textit{node}_{disconnect}\ \frac{}{\begin{array}{c}\llbracket [pg_1, nid_1, nc_1] \parallel_N [pg_2, nid_2, nc_2] \parallel_N ns, eth \rrbracket \xrightarrow{\mathtt{disconnect}(nid_1, nid_2)} \\ \llbracket [pg_1, nid_1, nc_1] \parallel_N [pg_2, nid_2, nc_2] \parallel_N ns, eth \cdot (nid_1, nid_2, \mathbf{died}(nid_2, \mathtt{disconnect})) \cdot (nid_2, nid_1, \mathbf{died}(nid_1, \mathtt{disconnect})) \rrbracket\end{array}}$$

Table 9. Node failure rules

4. Node Controller

In this section we thoroughly describe the node controller, and define how the different signals are handled. Looking at the variety of signals the node controller handle, it is a fairly complex construction. However, as we see below, much of the complexity is imaginary, the handling of each signal alone is quite straightforward; it is mostly a matter of bookkeeping.

For brevity we refrain from introducing all node controller definitions and rules. Many rules are very similar to rules presented here; for example monitor and monitor_node behave very similar to link (with the addition of *references*), register is quite similar to named send, etc. First we need to define yet another couple of functions, then we look at in turn handling of: **link**, **unlink**, **dead**, **spawn**, and **nsend** signals.

Definition 18 Let the function deleteDead(*pid*,*nc*) be defined in the obvious way; deleting all occurrences of *pid* from the node controller structure nc. In the case when the *pid* represents a node (i.e. it is in fact a *nid*), the nc should be cleared of all *processes* at that node as well.

***Node controller* – link/unlink** The rules for **unlink** in Table 11 are more simple than the rules for **link** in Table 10 since they do not depend on the unlinked process being alive or not. The first unlink-rule is the local instance, where the unlink is treated at the local node controller. In this case the node controller acknowledges the removal of the link. In the remote case (second unlink-rule) the link is silently removed. Combined with the rules for link one can see that as soon as the link is removed locally, there is no risk of getting an **exit**-signal later.

There are three rules for **link**, the first rule handle the case when the link is successful (either local or remote), and the last two handle the case when the to-be-linked process does not exist. In the second case, if it was a remote link (the to-be-linked process was on a different node than the linking process) the linking process is informed via a message to its node controller (the last rule), if it was a local link an **exit**-signal is constructed directly (the second last rule). In the first rule, we simply record the link in the $lnks$-set in the node controller, if it is a remote link, there might be a quick reply from the remote node controller with a **died**-signal, but that is handled by the rules for **died** (described in the next section).

At first it might seem a bit strange that there is no acknowledgement (i.e. no **message**({**reply**,...})-signal) for links, but since it is not possible to trap exits in this semantic there is no real difference between an acknowledgement followed by an **exit**-signal, or just an **exit**-signal.[2]

***Node controller* – died** The two rules in Table 12 capture all possible combinations of links and monitors to both processes and nodes. The rule is in fact too general for some cases, but the result of being too general is simply empty message-sets.

To illustrate how the rules works in practice, we consider two different situations. In the first situation a local process has terminated; this triggers the first rule. The node controller needs to send exit and monitor signals to the local processes linked to and monitoring the terminated process. The node controller also needs to communicate the termination of the process to all other node controllers where a process is linked to or monitoring the terminated process. Lastly, the node controller should remove all links, monitors and registered names for the terminated process, since these are not active anymore, this is done by the function deleteDead. It should be noted that $local\_mons$ is a list, and that $remote\_nodes$ and $local\_links$ are true sets. I.e. duplicates are possible in the $local\_mons$, but not in the others.

In the second situation a remote node has crashed (or disconnected), resulting in a **died**-signal with a node identifier being sent to the node controller. This triggers the second rule. This situation involves a bit more work for the node controller, not only should it notify local process monitoring the crashed node and then remove these monitors from the mns-set. The node controller also needs to find all links and monitors for *processes* located at the crashed node, and construct appropriate messages for them. However, in the second rule there is no need to inform remote nodes, they are already informed. Finally, in this situation as well, we clean up the node controller structure by applying the function deleteDead.

***Node controller* – spawn** In Table 13 the rule for **spawn** is presented. The handling of spawn is straightforward, a new process is created with a *fresh* pid, and the new pid is communicated back to the spawning process.

***Node controller* – nsend** In Table 14 the rules for **nsend** are presented. The two rules handle the different lookup cases. In the first rule there is a process registered for $name$ and we proceed by sending the signal to that process. In the second rule, the lookup-call returns `undefined` and the node controller simply drops the message.

5. Fairness

As we noted above, the input-rules, i.e. the rules in Table 6, can be applied in such a way that some messages are never delivered. I.e. the rules themselves does not ensure that messages are delivered in a fair manner. This is generally a bad thing, since many properties can not be proved in a non-fair system. Therefore we need to define a fairness rule that excludes certain unwanted behavior of the system. Fairness is defined in terms of permissible *execution sequences*.

Definition 19 An *execution sequence* is a sequence of node systems ns_i, together with corresponding system actions α_i written:

$$ns_0 \xrightarrow{\alpha_0} ns_1 \xrightarrow{\alpha_1} ns_2 \xrightarrow{\alpha_2} \ldots.$$

Definition 20 [Fairness for execution sequences] It should hold for all execution sequences, $(\vec{ns}, \vec{\alpha})$:

$$\forall i. \left\{ ns_i \xrightarrow{pid\ !_{from}\ sig} ns_{i+1} \Rightarrow \exists j > i. \left(ns_j \xrightarrow{pid\ ?_{from}\ sig} ns_{j+1} \right) \right\}$$

I.e. every message sent is eventually delivered.

6. Discussion

In this section we make a few illustrative comparisons between the current and the new semantics. We also identify some practical consequences of changing Erlang to follow the new semantics.

6.1 Everything is Distributed

The biggest, and in our eyes the most important, difference between the distributed semantics of Erlang [6] and the semantics we define in this paper is the changes made to message passing. In the current semantics (describing the current Erlang/OTP implementation) messages behave differently in a local and a distributed setting, i.e. there are different guarantees for message ordering, and

[2] Here is a distinct difference between links and monitors, since we are allowed to have more than one monitor for a pair of processes, each monitor needs to be identified by a unique reference. Thus there is an acknowledgement signal for monitor, containing this reference. Apart from this, monitors are handled similarly to links.

$$\frac{\mathit{pid} \in \mathsf{pids}(\mathit{pg}) \;\vee\; \mathit{nid} \neq \mathsf{node}(\mathit{pid})}{\mathsf{ncEffect}([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle], \mathit{from}, \mathbf{link}(\mathit{pid})) \longrightarrow ([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks} \cdot (\mathit{from}, \mathit{pid}), \mathit{mns}, \mathit{reg}\rangle], \epsilon)}$$

$$\frac{\mathit{pid} \notin \mathsf{pids}(\mathit{pg}) \;\wedge\; \mathit{nid} = \mathsf{node}(\mathit{pid}) \qquad \mathit{nid} = \mathsf{node}(\mathit{from})}{\mathsf{ncEffect}([\mathit{pg}, \mathit{nid}, \mathit{nc}], \mathit{from}, \mathbf{link}(\mathit{pid})) \longrightarrow ([\mathit{pg}, \mathit{nid}, \mathit{nc}], (\mathit{from}, \mathit{pid}, \mathbf{exit}(\mathtt{noproc})))}$$

$$\frac{\mathit{pid} \notin \mathsf{pids}(\mathit{pg}) \;\wedge\; \mathit{nid} = \mathsf{node}(\mathit{pid}) \qquad \mathit{nid} \neq \mathsf{node}(\mathit{from})}{\mathsf{ncEffect}([\mathit{pg}, \mathit{nid}, \mathit{nc}], \mathit{from}, \mathbf{link}(\mathit{pid})) \longrightarrow ([\mathit{pg}, \mathit{nid}, \mathit{nc}], (\mathsf{node}(\mathit{from}), \mathit{nid}, \mathbf{died}(\mathit{pid}, \mathtt{noproc})))}$$

Table 10. Node controller effect and reply for **link**

$$\frac{\mathit{nid} = \mathsf{node}(\mathit{from})}{\begin{array}{l}\mathsf{ncEffect}([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle], \mathit{from}, \mathbf{unlink}(\mathit{pid})) \longrightarrow \\ \qquad ([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks} \setminus (\mathit{from}, \mathit{pid}), \mathit{mns}, \mathit{reg}\rangle], (\mathit{from}, \mathit{nid}, \mathbf{message}(\{\mathtt{reply}, \mathtt{unlink}, \mathit{pid}\})))\end{array}}$$

$$\frac{\mathit{nid} \neq \mathsf{node}(\mathit{from})}{\mathsf{ncEffect}([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle], \mathit{from}, \mathbf{unlink}(\mathit{pid})) \longrightarrow ([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks} \setminus (\mathit{from}, \mathit{pid}), \mathit{mns}, \mathit{reg}\rangle], \epsilon)}$$

Table 11. Node controller effect and reply for **unlink**

$$\frac{\begin{array}{c}\neg\mathsf{isNid}(\mathit{pid}) \\ \begin{array}{rcl} \mathit{local\_links} & = & \{(\mathit{pid}_1, \mathit{pid}_2, \mathbf{exit}(v)) \mid (\mathit{pid}_1, \mathit{pid}_2) \leftarrow \mathit{lnks} \;\wedge\; \mathsf{node}(\mathit{pid}_1) = \mathit{nid} \;\wedge\; \mathit{pid}_2 = \mathit{pid}\} \\ \mathit{remote\_nodes} & = & \{(\mathsf{node}(\mathit{pid}_1), \mathit{nid}, \mathbf{died}(\mathit{pid}, v)) \mid (\mathit{pid}_1, \mathit{pid}_2) \in \mathit{lnks} \cup \mathit{mns} \;\wedge\; \mathsf{node}(\mathit{pid}_1) \neq \mathit{nid} \;\wedge\; \mathit{pid}_2 = \mathit{pid}\} \\ \mathit{local\_mons} & = & [(\mathit{pid}_1, \mathit{pid}_2, \mathbf{message}(\{\mathtt{reply}, \mathtt{monitor}, \{\mathit{ref}, \mathit{pid}, v\}\})) \\ & & \;\|\; (\mathit{ref}, \mathit{pid}_1, \mathit{pid}_2) \leftarrow \mathit{mns} \;\wedge\; \mathsf{node}(\mathit{pid}_1) = \mathit{nid} \;\wedge\; \mathit{pid}_2 = \mathit{pid}]\end{array}\end{array}}{\begin{array}{l}\mathsf{ncEffect}([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle], \mathit{from}, \mathbf{died}(\mathit{pid}, v)) \longrightarrow \\ \qquad ([\mathit{pg}, \mathit{nid}, \mathsf{deleteDead}(\mathit{pid}, \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle)], \mathit{local\_links} \cdot \mathit{local\_mons} \cdot \mathit{remote\_nodes})\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{isNid}(\mathit{nid}') \\ \begin{array}{rcl} \mathit{local\_links} & = & \{(\mathit{pid}_1, \mathit{pid}_2, \mathbf{exit}(v)) \mid (\mathit{pid}_1, \mathit{pid}_2) \leftarrow \mathit{lnks} \;\wedge\; \mathsf{node}(\mathit{pid}_1) = \mathit{nid} \;\wedge\; \mathsf{node}(\mathit{pid}_2) = \mathit{nid}'\} \\ \mathit{local\_mons} & = & [(\mathit{pid}_1, \mathit{pid}_2, \mathbf{message}(\{\mathtt{reply}, \mathtt{monitor}, \{\mathit{ref}, \mathit{pid}, v\}\})) \\ & & \;\|\; (\mathit{ref}, \mathit{pid}_1, \mathit{pid}_2) \leftarrow \mathit{mns} \;\wedge\; \mathsf{node}(\mathit{pid}_1) = \mathit{nid} \;\wedge\; (\mathit{pid}_2 = \mathit{pid} \;\vee\; \mathsf{node}(\mathit{pid}_2) = \mathit{nid}')]\end{array}\end{array}}{\begin{array}{l}\mathsf{ncEffect}([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle], \mathit{from}, \mathbf{died}(\mathit{nid}', v)) \longrightarrow \\ \qquad ([\mathit{pg}, \mathit{nid}, \mathsf{deleteDead}(\mathit{nid}', \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle)], \mathit{local\_links} \cdot \mathit{local\_mons})\end{array}}$$

Table 12. Node controller effect and reply for **died**

$$\frac{\mathit{pid}' = \mathsf{fresh}()}{\mathsf{ncEffect}([\mathit{pg}, \mathit{nid}, \mathit{nc}], \mathit{from}, \mathbf{spawn}(e, \mathit{ref})) \longrightarrow ([\langle e, \mathit{pid}', \epsilon\rangle \parallel_p \mathit{pg}, \mathit{nid}, \mathit{nc}], (\mathit{from}, \mathit{nid}, \mathbf{message}(\{\mathtt{reply}, \mathit{ref}, \mathit{pid}'\})))}$$

Table 13. Node controller effect and reply for **spawn**

$$\frac{\mathsf{lookup}(\mathit{name}, \mathit{reg}) = \mathit{pid}}{\mathsf{ncEffect}([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle], \mathit{from}, \mathbf{nsend}(\mathit{name}, v)) \longrightarrow ([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle], (\mathit{pid}, \mathit{from}, \mathbf{message}(v)))}$$

$$\frac{\mathsf{lookup}(\mathit{name}, \mathit{reg}) = \mathtt{undefined}}{\mathsf{ncEffect}([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle], \mathit{from}, \mathbf{nsend}(\mathit{name}, v)) \longrightarrow ([\mathit{pg}, \mathit{nid}, \langle \mathit{lnks}, \mathit{mns}, \mathit{reg}\rangle], \epsilon)}$$

Table 14. Node controller effect and reply for **nsend**

message delivery depending on whether the receiving process is local or remote. The change to a fully asynchronous message passing should enable a more efficient implementation of the run-time system. However, there is also a downside with the more permissive semantics. Less restrictions results in more possible interleavings, and in effect a larger state space to explore during verification.

Further, the choice of making all side effects (with the exception of `spawn_node`) purely asynchronous was motivated two-fold. (1) it made the resulting semantics a lot shorter, and therefore easier to understand, and (2) it should make the operations easier to parallelize in a future implementation of the semantics. These bare asynchronous behaviors would in many situations be rather inconvenient for the programmer to use, for example when spawning a process or registering a named process. But nothing stops us from defining (as a BIF or equivalent) the well-known synchronous variants that are used in Erlang today, by simply encapsulating the side-effecting action and the waiting for confirmation.

6.2 Uni-directional Links Only

Another part of the current semantics that we have found non-intuitive is the bi-directional links. We find it a bit strange that a process can affect the behavior of another process behind its back by linking to it. (E.g. in the situation where process A first links to process B and then exits abnormally, resulting in that B, unless it is trapping exits, suddenly crashes.) Also the functional overlap of monitors and trapped exits (links) makes for extra complications in the semantics. Instead we propose to only have uni-directional links (and monitors, but they are already uni-directional), and to remove the whole exit-trapping functionality since a monitor serves the same purpose.

The practical consequences of this change are not too dramatic, but it becomes a bit more complicated to build the important *supervision trees*. However, since supervision structures are most of the time built using the OTP-behavior (`supervisor`), it should be enough to change the supervisor implementation to compensate. The `spawn_link` construction is also easily mimicked by a slight change to its definition.

6.3 A Built-in Registry

The built-in registry is another difference between the current and the new semantics. However, its workings should be easily made equivalent to the standard process registry implementation in Erlang. Also, because of the introduction of a general treatment of side-effecting actions, the inclusion of a registry affects the size of the semantics marginally. The size argument is important, since the main priority of the new semantics is to keep it reasonably small.

6.4 Message-Passing Guarantees

The message-passing guarantees has changed dramatically in the new semantics. There are no longer any difference between local and distributed message-passing, and only the order of messages between a pair of processes is guaranteed.

7. Conclusions and Future Directions

We have presented a proposal for a future semantics of Erlang. However, the observant reader might have noticed that there are very few references to Erlang in the description of the semantics. That is, perhaps the language for this semantics is not Erlang, maybe there is another language waiting around the corner?

Nevertheless, we hope that the presented semantics is easy enough to follow that it can serve as a discussion starter when it comes to improving and changing Erlang in the future. It is our strong belief that the current message-passing guarantees must be removed in order to fully take advantage of an extreme (32+) multi-core architecture.

On the other hand, perhaps we can continue to work with bi-directional links, although they are slightly unintuitive they have after all worked fairly well for many years. It might not be worth trying to make the change.

Another side to the discussion is the implications for model checking. From a model checking perspective it makes perfect sense to impose stronger guarantees for message-passing, as we have in the current semantics. The result is a smaller state space, a crucial detail when model checking. Therefore, from a model checking perspective, the new, more permissive, semantics is potentially going to cause problems. Everything is not lost, it should be possible to find reduction techniques to compensate for the enlarged state space, but it will require a bit of thought.

Future Directions Since we have access to an experiment platform, McErlang [4], the obvious next step is to implement this semantics there. Having such an implementation is going to make it easier to analyze semantic design decisions in more detail.

Since, as mentioned above, the new semantics results in larger state spaces during model checking we should also put some effort into partial order reduction techniques for the new semantics. Otherwise, it will become hard to perform model checking for systems under the more asynchronous semantics.

Acknowledgments

This research was sponsored by EU FP7 Collaborative project *ProTest*, grant number 215868. Many thanks to the anonymous reviewers, whose comments significantly improved the presentation of the semantics. Thanks also to Koen Claessen for stimulating discussions and helpful comments.

References

[1] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, December 2003.

[2] K. Claessen and H. Svensson. A semantics for distributed Erlang. In *Proc. of the ACM SIGPLAN workshop on Erlang*, pages 78–87, New York, NY, USA, 2005. ACM Press.

[3] L-Å. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

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

[5] M. Petterson. A definition of Erlang (draft). Manuscript, Department of Computer and Information Science, Linköping University, 1996.

[6] H. Svensson and L-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Proc. of the SIGPLAN workshop on Erlang*, pages 43–54, New York, USA, 2007. ACM.

[7] H. Svensson and L-Å. Fredlund. Programming distributed Erlang applications: pitfalls and recipes. In *Proc. of the SIGPLAN workshop on Erlang*, pages 37–42, New York, USA, 2007. ACM.

Chain Replication in Theory and in Practice

Scott Lystig Fritchie
Gemini Mobile Technologies, Inc.
slfritchie@snookles.com

Abstract

When implementing a distributed storage system, using an algorithm with a formal definition and proof is a wise idea. However, translating any algorithm into effective code can be difficult because the implementation must be both correct and fast.

This paper is a case study of the implementation of the chain replication protocol in a distributed key-value store called Hibari. In theory, the chain replication algorithm is quite simple and should be straightforward to implement correctly. In practice, however, there were many implementation details that had effects both profound and subtle. The Erlang community, as well as distributed systems implementors in general, can use the lessons learned with Hibari (specifically in areas of performance enhancements and failure detection) to avoid many dangers that lurk at the interface between theory and real-world computing.

Categories and Subject Descriptors H.2.4 [*Database Management*]: Systems—Distributed Databases; C.4 [*Performance of Systems*]: Reliability, availability, and serviceability

General Terms Algorithms, Design, Reliability, Theory

Keywords Chain replication, Erlang, Hibari, key-value store

1. Introduction

A data store, whether a key-value store or file system or other kind of database, may be distributed across two or more machines for any combination of the following reasons:

- Availability — It is unacceptable for data to be inaccessible or lost when a single machine fails.
- Performance — A single machine cannot service its intended workload within acceptable limits (e.g., minimum throughput or maximum latency limits).
- Capacity — A single machine cannot physically store the required amount of data (e.g., RAM capacity or disk capacity).
- Cost — A single machine may be meet all of the above goals but is too expensive to purchase and/or maintain. Both hardware and software (e.g., software license fees) are considered.

The challenge of building any distributed, mutable-state service is managing changes to replicated copies of state data in a predictable manner. A distributed systems architect might turn to a collection of formal specifications and proofs such as Lynch's *Distributed Algorithms* [13]. Unfortunately, translating a distributed algorithm into reliable running code is a subtle, poorly understood art.

Lamport's Paxos algorithm [12, 16] is now a well-known distributed algorithm for maintaining shared consensus, but as staff at Google [5] and Microsoft [11] have written, it is very difficult to preserve the algorithm's correctness and simultaneously reach performance goals. In the Erlang community, Arts et al. [2, 3] presented leader election algorithms, their implementations, the testing methods they used, and the flaws that they discovered long after the code was considered finished.

This paper is a case study of the implementation of the chain replication protocol in a distributed key-value store called Hibari[1]. In theory, the chain replication algorithm is simpler than the Paxos algorithm. In practice, however, there are plenty of implementation details that have hampered creating a product that is both correct and sufficiently fast. The experience presented here can help others in the Erlang community and, more broadly, all distributed systems developers to create robust distributed systems that actually work correctly.

An outline of this paper's topics is as follows:

- Summaries of the chain replication technique and of an Erlang application, Hibari, that uses chain replication for replica management.
- Practical problems caused by disk latency and the need for rate control.
- Status monitoring, including how Erlang messaging infrastructure can hide network failures.
- Hibari's implementation of consistent hashing and replica placement strategies.
- Observations about Hibari that don't merit their own sections.
- Related work and concluding remarks.

2. Chain Replication

The chain replication algorithm is described by van Renesse and Schneider [24]. The paper specifies a variation of master/slave replication where all servers that are responsible for storing a replica of an object are arranged in a strictly-ordered chain. The head of the chain makes all decisions about updates to an object. The head's decision is propagated down the chain in strict order. See Figure 1 for a diagram of message flows.

The number of replicas for an object is determined by the length of the replica chain that is responsible for that object. To tolerate f replica server failures, a chain must be at least $f + 1$ servers long.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09...$10.00

[1] In Japanese, "hibari" means "meadowlark." The two Kanji characters used for "hibari", 雲雀, literally mean "cloud sparrow."

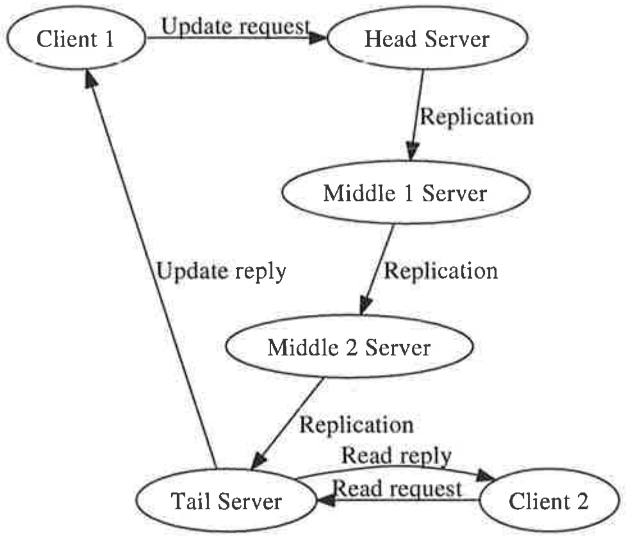


Figure 1. Write and read operations upon a chain of length four.

Operations on objects within the chain are linearizable (chapter 13 of [13]) when all updates are processed by the head of the chain and all read-only queries are processed by the tail of the chain. Strong consistency[2] is maintained because read requests are processed only by the tail of the chain.

When a chain member fails, the following steps are taken to repair the chain.

1. The chain is shortened to remove the failed brick (call it B) from service, and un-acked updates are re-sent down the chain. All clients are notified about the new chain configuration.
2. When brick B restarts, it is added to the end of the chain, and all out-of-date keys are copied to B. Meanwhile, brick B is ignored by all clients. If B receives a client request by mistake, the request is ignored. (See also section 12.3.)
3. When the key copying phase is complete, the chain is reconfigured to make B a full member of the chain in the tail role, and all clients are notified about the new chain configuration.

See [24] for a full description of procedures necessary to recover from chain member failure.

Client workloads with extremely large read/write ratios can potentially imbalance individual server workloads: 100% of read operations are sent to the same server, the tail of the chain. The chain replication implementation in CRAQ [22] allows read operations to be handled by other servers in the chain without violating strong consistency. Hibari does not use CRAQ's optimizations but instead uses data placement policies to balance server workloads; see section 11 for more detail.

3. Hibari Overview

Hibari is a distributed, fault tolerant, high availability key-value store written in Erlang. Through use of chain replication (section 2), all operations by Hibari clients read strongly consistent updates. Hibari is one of the few distributed key-value stores that can atomically update multiple keys in a single client operation (section 9). By default, all updates are persistent: each server flushes all updates to local stable storage before replying to a client.

[2] Read operations can only return an object's last update.

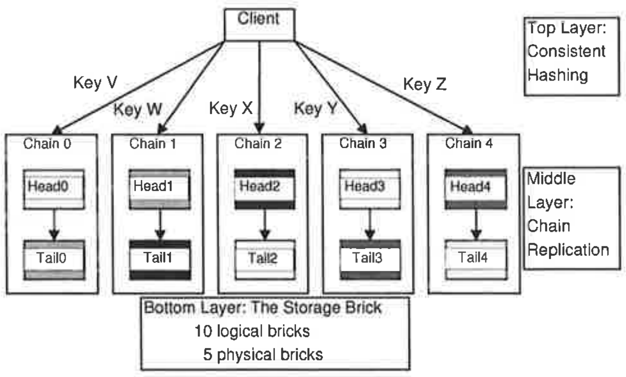


Figure 2. Hibari logical architecture: consistent hashing, chain replication, and basic storage.

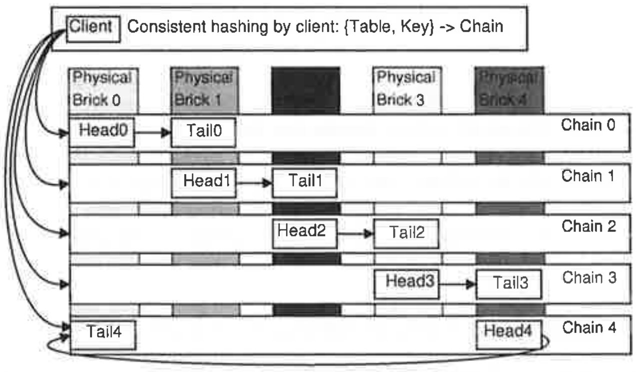


Figure 3. Hibari architecture: an alternate view of Figure 2 with each physical brick represented in a vertical column.

Hibari faithfully implements the chain replication algorithm as described in [24]. Erlang's messaging model makes it trivial to support the original algorithm's asymmetric message passing for updates. Local data logging, inter-server messaging, and chain replica repair (after the failure of a server) are implemented as described.

Hibari's performance is typically quite good, individually and as a cluster. On typical commodity 1-2U rack-mountable server hardware, Hibari can sustain a throughput of several thousand updates of 1 kilobyte values per second per server. Clients see throughput increase linearly as servers are added to the cluster.[3]

High availability is achieved by using replication chains longer than one server and by reacting quickly (e.g., within a few seconds) when a server fails. Availability is further assisted by distributing keys across the cluster using consistent hashing: when a chain has failed, all keys managed by other chains are unaffected by the failure. Also, machines may be added to or removed from Hibari clusters without interrupting service.

3.1 Physical and Logical Bricks

A brick is a server that stores persistent data. Figures 2 and 3 portray a cluster of five physical bricks (mapping one-to-one onto physical machines, typically Linux-based) with five replication chains. Each chain is of length two; the chain's data is available for reads and writes as long as the total number of failures within the chain is less than two.

[3] Assuming that chains are evenly balanced across physical machines; see also section 11.

```
CREATE TABLE foo (
  BLOB key;
  BLOB value;
  INTEGER timestamp;
  INTEGER expiration_time;          -- 0 = no expiry
  ERLANG_PROPERTY_LIST proplist;  -- Usually empty
) UNIQUE PRIMARY KEY key;
```

Figure 4. SQL-like representation of a Hibari table.

As far as Hibari Admin Server (see section 7) is concerned, data is stored by a "logical brick." The five physical machines, called "physical bricks," in Figure 2 are each configured with two logical bricks. The logical bricks are striped across the physical bricks so that each physical brick contains one logical brick in a "head" role and one logical brick in a "tail" role. Using consistent hashing (section 3.3) and techniques described in section 11, each physical brick's CPU, RAM, and disk workloads will be in balance under most conditions.

The Admin Server monitors the status of each logical brick. If a physical brick crashes, then clearly all logical bricks on that machine will fail. The reason for a physical brick's crash is not usually important: any hardware failure, such as power supply or disk volume, that disrupts a logical brick is sufficient to trigger reaction by the Admin Server.

If deployed on virtualized hardware, "physical brick" could mean either the physical hardware or the virtualized hardware. Virtualized hardware complicates management of the actual physical locations of logical bricks. Hibari's current code base does not make any attempt to enforce replication on distinct physical bricks. During development, it is useful to test clusters of hundreds or thousands of logical bricks on a single physical machine (with or without hardware virtualization). But right now, it is ultimately a human administrator's responsibility to ensure the physical diversity of each logical brick within a chain.

3.2 A Client's View of a Hibari Cluster

Each key-value pair is stored in a Hibari table. Tables were first implemented to provide separate key namespaces, that is, to permit storing the key "foo" multiple times, each in a different table. Later, tables became a convenient administration tool for configuring behaviors such as consistent hashing (section 3.3) and key migration (section 10).

Data in a Hibari table is stored in one or more chains. Each chain stores data for only one Hibari table. Each logical brick stores data for only one Hibari chain. Figure 3 depicts a typical layout for chains for a single Hibari table; see section 11 for discussion of replica placement strategies.

Each key in a Hibari cluster has the attributes depicted in pseudo-SQL in Figure 4; technically, each key stored by Hibari is actually part of a 5-tuple. Going forward, the more familiar term key-value pair will be used instead, and the other attributes will be mentioned only when the context requires it.

Each Hibari client receives status updates from the Hibari Admin Server that contain server status updates. Using the mapping data within each status update, each client knows the head and tail bricks for all chains in all tables within the cluster. Clients usually send their requests directly to the correct brick and do not incur intra-cluster query forwarding penalty, except during cases of key migration (also called key repartitioning, see section 10).

All attributes in Figure 4 except `value` are always stored in RAM by the logical brick. The `value` attribute may be stored on disk or in RAM as a per-table configuration option. As a result, Hibari logical bricks can consume a lot of RAM, proportional to the number of keys stored in the brick and (for RAM-based value blob storage) the sum of all value blob sizes.

3.3 Consistent Hashing

Hibari uses a consistent hashing technique [10] to map a $\{T, K\}$ tuple to the name of the chain that is responsible for storing that key K in table T. The key K, or configurable prefix of K, for example, the first 4 bytes, or all bytes between the first two ASCII '/' (slash) characters, is hashed using the MD5 algorithm[4] and mapped onto the unit interval $[0.0, 1.0)$. The unit interval is divided into an arbitrary number of ranges, where each range represents a chain name. Each chain can appear one or more times in the range map. The top third of Figure 5 depicts a range mapping of two chains onto the unit interval; each of the two chains has an identical chain weighting factor. (The bottom two-thirds of Figure 5 is discussed in section 10.)

The relative size of each range is determined by the chain weighting factor. Assume a hypothetical chain mapping where the chain weighting factor for chain C_1 is twice as large as the weighting factor for chain C_9. The sum of the size of range interval(s) found in the range map will be twice as large for chain C_1 as the sum for chain C_9. Smaller weighting factors can be used to bias distribution of keys away from some chains (and therefore away from some physical bricks/machines) that have lower capacity (e.g., slower CPU, less RAM, or smaller disk capacity).

Two hash mappings are used to implement key migration (or key repartitioning). In normal operation, the maps are the same. During key migration, the maps are used to calculate the current and new/desired location of a key. Key migration is a dynamic, online process that can expand or shrink a cluster as well as to change the relative chain weighting factors.

3.4 Single Data Center

Hibari was designed to provide strong consistency within a single data center. All deployments to date are in a single data center. A Hibari chain can have members in multiple data centers, but there are several practical complications in such a scheme:

- By definition, each update operation must traverse the entire replication chain. If chain members are in different data centers, each update operation will pay a penalty of the sum of the wide-area network latencies of all network links between the data centers.
- A client application in data center D_1 that attempts to read a key stored by a tail brick in data center D_2 must pay the penalty of the wide-area network latency between D_1 and D_2.
- The Hibari Admin Server is not currently designed to run simultaneously in multiple data centers.

4. Problems with Disk Write I/O Latency

All distributed systems architects have to face the tough facts of economic reality: if a system costs too much, then it won't be built. Deployment on cheap-enough hardware can also mean deployment on not-fast-enough hardware. The pressure of meeting performance goals can make cutting algorithmic corners very tempting. Architects must never forget that any change to a distributed algorithm, no matter how small or innocent the change seems, may in fact invalidate the algorithm.

For the purposes of this paper, "big data" means that the total amount of data and metadata stored (including all replicas) is larger

[4] MD5 was chosen for convenience and relatively low computation cost. Neither the larger output range or collision resistance of more recent cryptographic checksum algorithms is necessary.

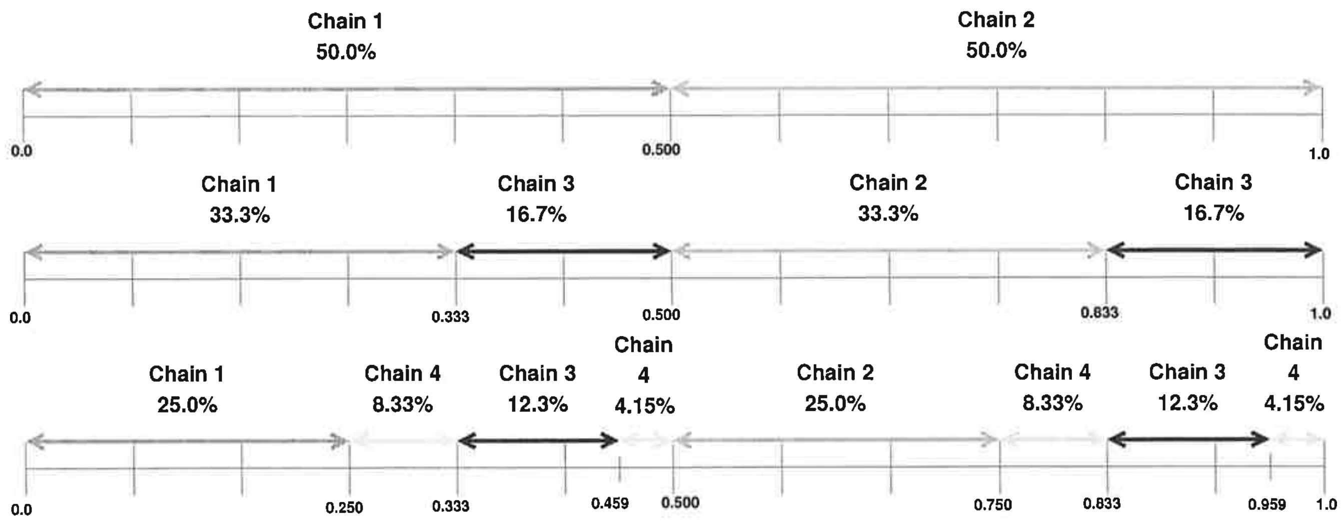


Figure 5. Consistent hashing map after two migrations: 2 chains, then 3 chains, then 4 chains. All chains have equal weighting factors.

than the sum of RAM in the cluster. The price of RAM-based and flash RAM-based storage is too high for many big data applications. Only traditional hard disks are cost-effective enough to support free or advertising-supported applications such as email services. Free email services now routinely provide email users with storage quotas in the range of 1-25 gigabytes per user, with "unlimited" storage available at extremely low monthly fees.

Rotating disk media can provide total storage capacity at a low-enough cost for most big data applications, but their average random I/O operation latencies are quite high: at least 4 milliseconds for top-of-the-line, 15K RPM SCSI disks and at least double that amount for slower, cheaper disks. As a result, it is in any application's best interest to minimize the number of random I/O operations that it generates.

The write-ahead log technique has been used by database systems for decades to aggregate disk write operations, minimizing random disk I/O operations by appending log entries to a log file. Another common technique is group commit, which flushes many transactions' worth of log entries to stable storage with a single `fsync(2)` system call (or OS equivalent).

Hibari uses both techniques, a write-ahead log and group commit, to minimize random disk I/O required to store reliably all updates received by a brick. The latency penalty of the `fsync(2)` system call is required to avoid data loss in the event of a cluster-wide catastrophe, such as a data center power failure.

Systems calls, such as `write(2)` and `fsync(2)`, that operate on local file systems can block for tens of milliseconds (or more) per call. Such blocking delay is unacceptable in almost any latency-sensitive application. To solve this problem, each `gen_server`-based Hibari logical brick sends all write and sync requests to a central write-ahead log (WAL) process (shared by all logical bricks on that Erlang node) to request `write(2)` and `fsync(2)` system call execution. The `gen_server` process is then free from blocking to work on other tasks while the I/O operations are pending. The WAL process sends messages back to the brick when the I/O calls have finished.

The chain replication protocol already requires that each write operation have a serial number associated with it and that each update propagates down the chain in serial number order. The WAL process uses these serial numbers to signal to each brick the largest serial number that has been safely flushed to disk. Each brick can then send those updates to downstream bricks at its leisure.

Unfortunately, Hibari's early implementations of the communication between brick and WAL server processes were fraught with subtle, difficult-to-find race conditions: writes were written to the local WAL out of order, `fsync(2)` operations were acknowledged with wrong log serial numbers, and bricks sent log replay messages downstream in incorrect order. The QuickCheck software testing tool [17] has been invaluable for helping create the conditions necessary to exploit the very small windows of vulnerability of many of these bugs.

Many of the bugs above have caused data loss. Most would never have been created if the code could ignore the reality of dealing with slow disk devices. Few data buffering hacks go unpunished.[5]

5. Problems with Disk Read I/O Latency

Each Hibari data server maintains key and key metadata in RAM but stores value blobs either in RAM or on disk. For big data purposes, Hibari must store value blobs on disk. As a consequence, any client `'get'` or `'get_many'` operation may trigger disk I/O. If the available cache is too small, and/or if the client application's access pattern doesn't provide sufficient temporal locality of reference, then disk read I/O operations are inevitable.

Hibari tries to minimize the number of disk operations required to read a key's value blob by always storing in RAM the value blob's exact storage location: write-ahead log file number, starting byte offset, and blob size. The brick can read any value blob with a single pair of `lseek(2)` and `read(2)` system calls.

Hibari's `'get'` and `'get_many'` operations are not the only source of disk read I/O workload. Two more significant sources are the following.

1. Chain repair — Chain repair can generate a huge amount of disk read I/O. For each value blob that a brick under repair does not have, the upstream brick must read the blob from disk before sending it downstream to the repairee brick.
2. Key migration — Sometimes called key repartitioning, resharding, or rebalancing, the Hibari key migration process moves keys (and associated values and metadata) from one chain to another (see section 10). Any key that must be moved during a

[5] Borrowed very loosely from Patrik Nyblom.

key migration must have its value read from disk by a brick in the source chain before it is copied to the destination chain.

5.1 Read Priming

As described in section 4, many system calls involving the local file system can block the caller. Any file `open(2)` or `read(2)` system call has the potential to block a Hibari logical brick process for many milliseconds each. On extremely overloaded systems, each call can easily take over 100 milliseconds, which in turn can have enormous negative effects on latency-sensitive applications.

To avoid blocking brick processes with read-only disk I/O, Hibari borrows techniques used by the Squid HTTP proxy [19] and Flash HTTP servers [15]. Before a brick attempts to open or read a file, it first spawns a "primer" process that asynchronously opens the file and reads the desired data. This process acts like adding water to a pump to "prime" the pump: all necessary file metadata and data is read into the OS page cache. The primer process uses the standard Erlang `file` API to do its work. When finished, the primer process sends a message to the logical brick process that the priming action is complete. Then the brick process can read the blob (using the same API) with very little probability of blocking.

This priming technique has the disadvantage of reading the same data twice: once by the short-lived primer process and once by the long-lived brick process. However, even with value blobs up to 16 megabytes in size, the overhead isn't big enough to worry about. The major advantages are that the Erlang `file` module already supports all operations that the primer process requires, and the probability of blocking the brick process is reduced to practically zero. The reduction of average read latency significantly outweighs the disadvantages.

5.2 Access by Lexicographic vs. Temporal Orders

For both chain repair and key migration workloads, the primer technique only hides a portion of the latency required to read large numbers of value blobs from disk. Both workloads generate I/O based on the lexicographic sort order of the keys. However, the value blobs are stored on disk in temporal order, that is, relative to the time when they were received.

The mismatch between lexicographic and temporal orderings can create a significant amount of random I/O workload, as far as the underlying disks are concerned. For key migration workloads, the I/O cost is largely unavoidable. Brick repair times can take a few seconds or several days, depending on several factors.

- If the brick under repair was down for only a short time, the total number of keys that require repair is likely to be small, and their value blobs are likely to be in the OS page cache.
- For a brick that is completely empty (e.g., a new machine with a new, empty file system), a manual function is provided that transmits keys and value blobs in an order that is sorted by each value blob's location within the write-ahead log. The sorting can help reduce the amount of random pattern read disk I/O required to read a large number of value blobs. The savings can be very significant when the total size of value blobs is in the range of hundreds or thousands of gigabytes.
- For repair tasks that fall in the middle, the number of keys to repair is high, but the cost of starting repair completely from scratch is even higher. In this middle case, there is no choice other than wait for the standard repair technique to finish and to accept the amount of random read disk I/O required to do it. For a chain that contains a terabyte of data or more, the time required to finish chain repair can be minutes (best case), hours, or even days (worst case). System planners and operations staff must keep this in mind as they plan their data redundancy strategy, that is, how big should each brick be and how long should each chain should be.

6. Rate Control

Modern hard disks are orders of magnitude slower than other components in the system: CPU, RAM, system buses, and even commercial gigabit Ethernet interfaces and switches are all less likely to be the slowest system component. To avoid overloading disk subsystems even further, rate control mechanisms are necessary to control anything that can generate disk I/O.

Hibari has explicit controls for both batch sizes (e.g., number of keys per iteration of an algorithm loop) and bandwidth (e.g., total number of bytes/second) for the following.

1. Chain repair operations, key migration operations (see section 5), and number of primer processes for prefetching value blobs from disk.
2. Log "scavenging" operations, which reclaim space from Hibari's otherwise infinite-sized write-ahead log. The scavenger's activity can create a large amount of extra disk I/O, more than enough affect clients by increasing latency. See the Hibari Systems Administrator's Guide at [9] for a detailed description.

During key migration periods, it is possible for a client's request to be forwarded back and forth between a key's old chain location and its new chain location. This forwarding loop is usually quickly broken once the key has been stably written to the new chain. If a forwarding loop is detected (using a simple hop counter), an exponential delay is added at each forwarding hop to try to avoid overloading bricks in either chain. Also, the loop will be broken if the total number of hops exceeds a configurable number.

Hibari also has an implicit limit on the number of simultaneous client operations that a single brick can support. The simple technique is borrowed from SEDA [26]: if the client request is too old, then drop the request silently. Sending a reply to the client will create even more work for an overloaded server to do, so the cheapest thing to do is to do nothing. Each Hibari client request contains a wall clock timestamp. If that timestamp is too far in the past, the Hibari brick will ignore the request, assuming that the request waited in the brick's Erlang mailbox for so long that the brick must be overloaded.

To help synchronize system clocks, it is strongly recommended that all Hibari machines, servers and clients, run the NTP (Network Time Protocol) service. Synchronization down to the femtosecond is not necessary; all clocks within even 100 milliseconds is good enough. However, deployment on virtual machines, such as Xen or VMware, should be avoided unless the guest OS's clock can reliably match the host OS's clock (which is assumed to be stable).

Unsynchronized guest OS clocks can cause bricks to drop client requests silently, via the mechanism described two paragraphs earlier. The silent drops cause client-side timeouts that can be very confusing, unless you happen to be looking at "operation too old" counter statistics. That counter should only ever increment during periods of server overload; any other time is a near-certain symptom of unsynchronized OS clocks.

7. Cluster Management and Monitoring

The original chain replication paper [24] describes a single master server that is responsible for managing chain state and monitoring the status of each server within each chain. To be accepted commercially, however, single points of failure must be avoided or minimized to the greatest extent possible.

Hibari implements the single master entity as a single Erlang/OTP application that is managed by the Erlang kernel's "application controller." The application controller coordinates multiple

Erlang nodes to run the management/monitoring application, the Admin Server, in an active/standby manner. This indeed creates a single point of failure: if the machine running the active Admin Server instance crashes, the Admin Server's services are lost.

Failure of the Admin Server is not usually a significant problem. The Admin Server is required only when bricks crash or restart within the cluster, or if an administrator wishes to query cluster status or to reconfigure the cluster. Without the Admin Server, Hibari client nodes may continue operation without error, as long as other bricks also fail while the Admin Server is down.

The Admin Server requires approximately 10 seconds to restart. The Admin Server's private state (including histories of the down/-up status of each logical brick and chain) is also distributed across bricks within the cluster. If the Admin Server's private state were managed with chain replication, then there would be a "chicken and the egg" problem when the Admin Server bootstraps itself. To avoid a circular dependency, the private state is replicated using quorum voting-style replication.

7.1 Detecting Brick Failure and Network Partition

The original chain replication paper [24] makes two assertions that are extremely problematic in the real world. The first is "Servers are assumed to be fail-stop." The second is "A server's halted state can be detected by the environment." If either assumption is violated, the system can quickly make bad decisions that can cause data loss.

The biggest problem with detecting halted nodes is the problem of network partition. Partitions are usually caused by failure of network equipment, such as an Ethernet switch or failure of network links such as a telecom data circuit. However, any failure of hardware and/or software that creates arbitrary message loss can be considered a network partition.

With message passing alone, it's impossible to tell the difference between a network partition, a failed node, or merely a very slow node. The built-in Erlang/OTP message passing and network distribution mechanisms cannot adequately handle network partition events by themselves.

To fix the worst problems caused by network partition, Hibari includes an OTP application called "partition_detector." Running on all Hibari servers, the sole task of this application is to monitor two physical networks, the 'A' and 'B' networks, for possible partition. All Erlang network distribution traffic is assumed to use network 'A' only. UDP broadcast packets are sent periodically on both networks. When broadcasts by an Erlang node are detected on network 'B' but have stopped on network 'A', then a partition of network 'A' may be in progress.

The Erlang/OTP application controller can still make faulty decisions when a network partition happens; the application controller does not interact with partition_detector application. However, after the application controller restarts an Admin Server instance, the partition_detector application can abort the initialization of that instance when it believes there is a partition in effect, raise an alarm, and leave the Admin Server processes in an idle state. This situation must then be resolved by a human administrator.

7.2 Fail Stop Means ... Stop?

Violation of the "fail stop" assumption have also caused problems for Hibari. Hibari's sponsor, Gemini Mobile Technologies, is not responsible for day-to-day operations and monitoring of its customer's systems, so all we know and theorize comes from after-the-fact analysis of failures in customer lab or production systems. In these post mortem analyses, we identified two significant problems:

1. A bug within the Erlang/OTP `'net_kernel'` process that can cause deadlock and thus cause communication failures between Erlang nodes. One instance of this bug hit a customer's system on at least 10 different machines within a 30 minute interval, including both nodes that managed the Admin Server's active/standby fail-over.
2. System `'busy_dist_port'` events can trigger interference in process scheduling and extremely high inter-node messaging latencies. All Erlang messaging traffic to a remote Erlang node is sent through a single Erlang port which represents a TCP connection. If the sending node detects congestion (e.g., a slow receiver, intermittent network failure), then any Erlang process on the local node that attempts to send a message to the remote node will be blocked: the Erlang process is removed from the scheduler and will remain unschedulable until the distribution port is no longer congested.

The combination of `'net_kernel'` deadlock, wild variations in message passing latency, and process de-scheduling can create a situation that is difficult to diagnose. If a brick is merely slow to respond to status queries by the Admin Server, Arpaci-Dusseau et al. suggest calling it "fail stutter" [1]. But if the brick responds too slowly, the Admin Server may interpret a performance problem as a failure instead.

One such problem, affecting both the Admin Server node and many others within a cluster, caused Hibari's largest deployment to suffer from multi-hour transient availability failures. If a brick does not respond to a status query by the Admin Server, it is considered failed and removed from the chain. A few seconds later, the brick would catch up and answer new queries. The Admin Server would force the brick to crash, triggering automatic repair and eventual rejoining the chain. If the situation is bad enough, the chain can be (and has indeed been) whittled down to zero bricks.

One solution to this problem has been a small patch to the Erlang virtual machine to make the buffer size for inter-node network distribution ports configurable. The default size of the `erts_dist_busy` constant is 128 kilobytes. However, even a value of 4 megabytes appears to be too small for the amount of messaging data that Hibari bricks send during bursty traffic patterns.

Another solution uses information from Hibari's partition_-detector application to supplement the monitoring info that the Admin Server uses. If a system monitor `'nodedown'` message is received, the partition detector's state is queried to check if a network partition was a possible cause of the message. The same is done if a query of a remote brick's general health status fails due to timeout or `'nodedown'` reasons.

In hindsight, the single Admin Server process has had more problems in production than we had anticipated. The solutions outlined above have not been in use long enough to judge their effectiveness. However, given the problems that we know have happened in production networks, it is likely that a distributed manager application would likely have been fooled by the same conditions and made similarly bad decisions.

7.3 The Admin Server as a Single Entity

The single running Admin Server instance has a convenient consequence: behavior during network partition events is easy to reason about. An administrator knows where the Admin Server might run: all eligible nodes are configured statically, so there are (typically) only two or three machines where the Admin Server may be running. Furthermore, using Figure 6 as an example:

- If an entire chain is on the same side of a partition as the Admin Server, then Chain 1 is healthy and usable by Client 1. Client 2 is the far side of the partition and therefore cannot access Chain 1.

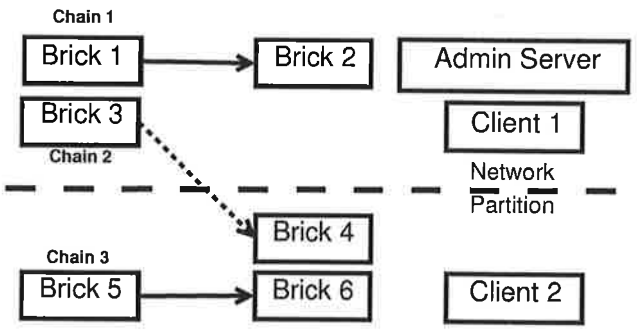


Figure 6. A network partition scenario.

- If the entire chain is on the far side of the partition relative to Admin Server, then Chain 3 is healthy and usable by Client 2. Client 1 is the far side of the partition and therefore cannot access Chain 3.
- For chains that are split by the partition, the bricks on the same side of the partition as the Admin Server will be reconfigured into a new chain. In Figure 6, Chain 2 will be reconfigured to a chain of length one that contains only Brick 3. The new Chain 2 is accessible by Client 1 but not by Client 2.

The CRAQ paper [22] proposed a distributed chain monitoring and management scheme. Hibari's Admin Server pre-dates the CRAQ paper and therefore couldn't take advantage of its suggestions.

8. Erlang Messaging Is Not Reliable

The original chain replication paper [24] says, "Assume reliable FIFO links." There is no such thing in the real world. Erlang's messaging model is frequently described anecdotally as "send and pray."

The original chain replication paper says that the tail of the chain, after processing an update successfully, acknowledges the update back "upstream" all the way to the head. These acknowledgments are vital for chain repair purposes. The Hibari implementation avoids the cost of an acknowledgement per update (which can be as high as several thousand updates per second per chain) by using a once per second acknowledgment that contains the largest update serial number that has been processed by the tail.

The once per second optimization is fine if all communication links are indeed reliable and FIFO. However, Erlang's communications links are not reliable.

Svensson and Fredlund describe in [21] under what conditions the usually-reliable messaging between two Erlang nodes can turn unreliable. It is possible for a brick to send three updates downstream, with serial numbers S_1, S_2, and S_3, respectively. It is possible for the Erlang network distribution mechanism to deliver the messages containing S_1 and S_3 and drop the message containing S_2. In this situation, a Hibari tail brick might acknowledge serial S_3 as the last processed serial number, and the head brick will mistakenly assume that the S_2 update has been processed by all members of the chain.

Fortunately, the Erlang process monitoring BIF `monitor()` will deliver a `{'DOWN',...}` message to the receiver when the connection between nodes has been broken and may have dropped messages. Hibari now uses this mechanism to detect dropped messages. If a `{'DOWN',...}` message is received, then subsequent replication log replay messages must pass a series of strict sanity checks. If the checks fail, then the receiving brick will crash itself.

Sample key	Data stored in value blob
`/42/1`	Text of post #1
`/42/1/1`	Text of comment #1 on post #1
`/42/1/2`	Text of comment #2 on post #1
`/42/2`	Text of post #2
`/42/summary`	Next post number, number of active posts, number of deleted posts, ...

Figure 7. A sample Web blogging application's `posts` table.

It is imperative that all bricks replay all replication log messages in exactly the same serial order. By crashing immediately, the brick that first notices a dropped message will avoid propagating out-of-order messages downstream. Hibari's automatic chain repair will compensate for the lost message.

9. Micro-Transactions

Though not mentioned in the original chain replication paper [24], [22] mentions the possibility of implementing a "micro-transaction": an atomic update of multiple keys by a single transaction in certain limited situations. All update operations are sent to the head of a chain, and the head can make any decision it wishes, including a non-deterministic decision.[6] Therefore, the head can decide on the fate of multiple operations that are sent in a single client request: "commit" by applying changes for all operations in the request, or "abort" by applying none of them.

Hibari has implemented a similar transaction feature. A client can send multiple primitive query and update operations in a single protocol request to a Hibari data server. The limiting factor is that all keys for the primitive operations in the request must be keys that the brick is responsible for. This limit is the reason for using the word "micro-transaction" instead of "transaction."

To implement request forwarding, for example, when a client sends a request to the wrong brick, each Hibari brick is already aware of what range of keys it is responsible for. Micro-transactions introduce a second reason why Hibari bricks must maintain this awareness: if a brick detects that a micro-transaction attempts to operate on keys stored in multiple chains, the micro-transaction must be aborted.

To use micro-transactions effectively, the client application must be aware of the key prefix scheme used by each table. It is the client's responsibility to create micro-transactions where all keys are managed by the same chain. This implicit knowledge could be made explicit by changing the client API: add a parameter to specify the consistent hash string, similar to a "bucket" in the Riak client API [18]. By using either implicit key prefixing or an explicit bucket-like grouping, the client controls whether any two keys must be stored in the same chain.

For example, assume a need to build a simple Web blogging application. On a per-user basis, the application requires storage for user authentication data, biographical data and preferences settings, blog postings, and comments on blog postings. The blog postings and comments would be stored in a single table called `posts`. The hashing key prefix, configured when the `posts` table was created, would be a variable prefix delimited by two slash characters.

See Figure 7 for example keys that would be stored in the `posts` table for user #42. The value of the `/42/summary` key would contain metadata for the user's collection of postings: the number to assign to the user's next post, the number of active/undeleted posts, the number of deleted posts, etc. All comments for post #1 would be retrieved by a `'get_many'` operation with the `{binary_prefix,`

[6] Non-deterministic choices are mentioned in [24].

```
add_new_post(UserID, PostText) ->
  Prefix = "/" ++ integer_to_list(UserID) ++ "/",
  MetaKey = Prefix ++ "summary",
  {ok, OldTS, OldVal} =
      brick_simple:get(posts, MetaKey),
  #post{next_id = NextID, active = Active} =
      OldMeta = binary_to_term(OldVal),
  NewMeta = OldMeta#post{next_id = NextID + 1,
                         active = Active + 1},
  PostKey = Prefix ++ integer_to_list(NextID),
  %% replace op: Abort if the key does not exist
  %%             or if current timestamp /= OldTS.
  %% add op: Abort if the key already exists.
  Txn = [brick_server:make_txn(),
         brick_server:make_replace(MetaKey,
                                  term_to_binary(NewMeta),
                                  0, [{testset, OldTS}]),
         brick_server:make_add(PostKey, PostText)],
  [ok, ok] = brick_simple:do(posts, Txn).
```

Figure 8. Example code to add a new Web blog posting using a micro-transaction.

"/42/1/"} option to limit results to only those keys that have a prefix that matches post #1's comments.

To create a new posting, the micro-transaction feature would be used to keep the metadata in the `summary` key consistent despite races with other metadata updates. A simple function, without error handling code for sake of simplicity, is shown in Figure 8.

10. Automatic Key Partitioning and Migration

Some key-value stores in the open source world [14, 23] do not include automatic support for key partitioning (also called "key sharding"): they assume the client will implement it. Unfortunately, coordinating the actions of many distributed clients in a 100% bug-free manner is a very difficult task.

Other distributed storage systems place significant restrictions on key migration/repartitioning. For example, the MySQL Cluster RDBMS did not support repartitioning until April 2009, and then only to expand the size of the cluster [20] — reducing cluster size was not supported.

Reducing cluster size is a valuable feature. Also, support for heterogeneous hardware is very desirable. It is nearly impossible to buy the same hardware more than three months after a system has been deployed in the field, much less three years or more in the future.

The original chain replication paper [24] is silent on the subject of key migration. Hibari provides support for key migration as well as support for heterogeneous hardware. Both are accomplished by its consistent hashing implementation.

Each Hibari server and client node maintains two complete consistent hashing maps for each Hibari data table: one old/current map and one new map. During normal operations, the two maps are identical. However, during a key migration period, the two maps will be different: the current map describes where keys are stored in the current scheme, and the new map describes where keys are stored in the desired scheme.

The bottom two-thirds of Figure 5 shows an example of the chain mappings used to migrate a table from two chains to three chains and later four chains. A key K_1 with an MD5 hash that maps to 0.1 on the unit interval would be stored in Chain 1 and would not move in either key migration. A key K_2 with an MD5 hash that maps to 0.49 on the unit interval would initially be stored on Chain 1. The first key migration would move K_2 from Chain 1 to Chain 3. The second key migration would move K_2 from Chain 3 to Chain 4.

The method demonstrated in Figure 5 attempts to minimize key movement and to evenly distribute migration workload. However, the Admin Server API permits the flexibility to choose arbitrary map definitions for a key migration. As a planning tool, an API function is provided to calculate how many keys would be moved between all pairs of chains, given a specific hash map.

Hibari's key migration is performed dynamically, while all bricks and clients are in full operation. The chain head brick selects a "sweep window," a range of keys (in lexicographic sort order), and copies the keys to their respective destination chains. When all destination chains have acknowledged successful writes, the sweep window is advanced, and the process repeats. All chain heads perform key migration sweeps in parallel. Operations by clients on keys inside the sweep window are deferred until the sweep window advances.

Due to the realities of message passing asynchrony, it is possible for clients to send queries to the wrong brick in the wrong chain. Each brick will determine if a query has been sent incorrectly and, if so, forward the query to the appropriate brick. Most forwarding involves only one extra hop or are loops that exist for very small periods of time (typically much less than one second). The forwarding delay and maximum hop mechanism described in section 6 take care of rare, long-lived forwarding loops.

Hibari's key migration implementation is currently missing a feature requested by at least one customer: the ability to halt a migration. If the I/O workload caused by migration causes severe latency problems for normal Hibari client applications, the customer wishes to suspend migration until peak client workload subsides.

Aborting a migration entirely would be much more difficult. The sweep key mechanism would have to "run backward": a sweep of the key space in reverse order would send each key-value pair from its destination chain backward to its source chain. Rather than implement this complex feature, it is much easier to permit the current migration to map M_n to finish, and then trigger a new migration to map M_{n+1} where $M_{n-1} = M_{n+1}$ to move all keys back to their original location.

11. Replica Placement

Terrace and Freedman [22] have discussed replica placement strategy: where should various chain members be located physically and logically? For example, all replicas within a chain should not be within the same physical data center rack: rack-wide power failures and network outages are too common, even in well-managed data centers.

One of chain replication's nice features is that it doesn't make many demands on replica placement policy, giving an administrator great flexibility. For example, a Gemini customer decided that chain lengths of three would be sufficient to meet its availability goals. The Hibari default replica placement arranges bricks within chains as if the underlying physical machines were in a ring: chain 1 uses machines $A \rightarrow B \rightarrow C$, chain 2 uses machines $B \rightarrow C \rightarrow D$, and so on. In the case of 26 physical machines, the final chain would use machines $Z \rightarrow A \rightarrow B$. (See also Figure 2 for an example of a ring of five machines.)

Using the above ring strategy, the resources of each machine are likely to be used equally: each physical machine would host an equal number of head, middle, and tail bricks. This balance was pleasing to the customer. However, the operations impact of key migration did not appear so pleasing when considering expanding the size of the cluster. If the new machines are inserted into the ring between A and Z, then the machines nearby, that is, machines A, B, Y, and Z, will endure greater load caused by key migration than the other 22 original nodes.

Original Machines			Original Machines			New Machines		
Machine A	Machine B	Machine C	Machine D	Machine E	Machine F	Machine G	Machine H	Machine I
Head0 →	Middle0 →	Tail0.	**Head3** →	Middle3 →	Tail3.	**Head6** →	Middle6 →	Tail6.
Tail1.	**Head1** →	Middle1 →	Tail4.	**Head4** →	Middle4 →	Tail7.	**Head7** →	Middle7 →
Middle2 →	Tail2.	**Head2** →	Middle5 →	Tail5.	**Head5** →	Middle8 →	Tail8.	**Head8** →

Figure 9. Replica placement using groups-of-3-machines strategy: start with six machines, then add three more. Machines that maintain head bricks are bold-faced to highlight the striping pattern.

This customer decided to use a different placement strategy, called groups-of-three-machines strategy. For each Hibari table, a group of three chains is striped across a small group of three machines. This process would repeat until all machines were accounted for. See Figure 9 for an example. The result provides equal workload sharing: each machine still has an equal number of head, middle, and tail bricks. Also, adding new machines (in groups-of-three) will create a balanced workload during key migration: assuming that all chain weightings are equal, then roughly 50% of all keys in chains on machines A through F will migrate to chains on machines G through I.

On top of the groups-of-three placement strategy, the customer is free to use rack-aware placement also. For example, each physical machine in a group-of-three can be placed in a different rack.

12. Other Observations

This section contains a number of observations about Hibari's implementation and production deployments that don't merit entire sections to themselves.

12.1 Using `gen_server`

Hibari's implementation makes heavy use of the Erlang/OTP `gen_server` behavior, largely because its serial method of handling messages maps very well onto the serialization that a well-behaved chain replication server must do. However, a single Erlang process cannot consume more than a single CPU core's worth of computation resources. Due to Hibari's one-to-one mapping of logical bricks to Erlang processes, an administrator who wishes to take full advantage of multi-core and multi-CPU systems must provision more chains than strictly necessary so that many logical bricks will be assigned to a single physical brick.

The extra logical bricks come at a cost of management complexity. The Admin Server now must keep track of more bricks and chains than is otherwise strictly necessary. The overhead of monitoring each brick is small, but when monitoring a few thousand bricks, the total cumulative workload can cause problems. The biggest single bottleneck is updating the Admin Server's private state storage bricks. For the sake of simplicity, updates to the private state bricks are serialized. When a cluster with over 3,000 logical bricks are booted simultaneously, the number of state transitions that are generated each second can exceed the state storage bricks' maximum update rate. A future release of Hibari will fix this problem.

12.2 Chain Reordering

Chain reordering doesn't appear in either the original chain replication paper [24] or the CRAQ paper [22], but it's valuable from an operations perspective. As originally described, a chain can become reordered by the failure and repair of member servers. In the long term, such reordering can destroy an administrator's intended balance of workload across hardware resources.

For example, if a chain is configured as $B_1 \rightarrow B_2 \rightarrow B_3$ and brick B_2 fails, then after repair is finished, the chain's order will be $B_1 \rightarrow B_3 \rightarrow B_2$. If brick B_1 fails later, the chain's order will be $B_3 \rightarrow B_2 \rightarrow B_1$ (again, after repair). Without reordering, the chain will remain in this order until yet another brick fails. Hibari, however, will reorder the chain back to $B_1 \rightarrow B_2 \rightarrow B_3$ once the repair of B_1 is complete.

12.3 Key Timestamps

Each key in a Hibari server has a timestamp associated with it. Each server enforces a rule that each update must strictly increase the key's timestamp. This feature prevents multi-client races that attempt to update the same key. The timestamp can also be used for "test and set" behavior, which will abort a micro-transaction if the key's current timestamp does not exactly match a timestamp observed in an earlier operation.

Key timestamps have subsequently become extremely important for optimizing brick repair. Other projects such as Dynomite [7] and Riak [18] use Merkle trees to quickly calculate which keys two servers share.

Hibari uses a simple "I have"/"Please send" iterative protocol to identify keys that need repair. Each key and its timestamp are sent in the "I have" phase. Because all keys and timestamps are stored in RAM, no disk I/O is required (by either the upstream/online brick or the downstream/repairing brick) to complete the "I have" phase. Disk I/O (to retrieve value blobs) is required only for keys that are missing or out-of-date on the downstream brick.

12.4 Read-Ahead

Read-ahead optimizations by the operating system's disk subsystem can often degrade performance. Most of Hibari's read-only disk operations are random in nature across mostly small pieces of data, usually only a few kilobytes each. Read-ahead mechanisms that try to read hundreds or thousands of kilobytes merely create higher latency for all disk operations.

There is one case where Hibari could use very aggressive read-ahead buffering by the OS: during brick initialization's sequential scan of the brick's write-ahead log. The Erlang virtual machine does not support system calls like `fadvise(2)` and `mincore(2)`, so it has no custom control over read-ahead behavior. We have not yet been desperate enough to try using the newer R13B Erlang NIF feature or an older-style driver to implement these system calls, but we probably will, someday.

12.5 File Checksums

Hibari stores all log data on disk with MD5 checksums. Any data corruption detected by an MD5 checksum will cause a brick to take itself out of service. Automatic chain repair will identify the keys lost due to corruption and re-replicate those keys. The bad file is moved to a separate directory to prevent future access. By not deleting the bad file, Hibari hopes to avoid reusing the bad disk block(s) that caused the original problem.

Unlike GFS [8], Hibari's network messages are not yet protected by checksums. It is possible for a bit error to escape the network protocols stack's checksum regime and affect Hibari clients and/or downstream bricks.

12.6 Murphy's Law

Anything can and does happen in a production environment. The "impossible" is possible, and if something can go wrong, it will.

In one memorable case, the Erlang/OTP "kernel" application's error_logger process was overwhelmed by over 85,000 error messages that were triggered by Erlang's system_monitor() BIF in under one minute. We know that such system messages can be generated extremely quickly by the Erlang virtual machine. Hibari's process that receives the system events will only forward 40 events per second to the error_logger. [7]

So, how can a process that throttles itself to generate only 40 error messages/second send over 85,000 messages to error_logger in under one minute? After all, (40 messages/sec)(1 minute = 60 seconds) = 2,400 messages. Intensive code review of the rate limiting mechanism has found no fault. And we know for certain that all 85,000 messages were generated in under 60 seconds. This mystery will probably never be solved.

12.7 Other observations

- It is too easy, especially when subject to schedule pressures, to shoot yourself in the foot with Erlang. Code with side-effects is difficult to understand, to test, and to support ... yet management of side-effects (i.e., mutable state) is Hibari's reason for being. If we were to rewrite Hibari from scratch, we would be extra careful to segregate code with and without side-effects to simplify testing by QuickCheck and other tools.
- An early implementation decision for Hibari, left ambiguous by the original chain replication paper and discarded by the CRAQ paper, was that any single logical brick can be a member of only one chain. In hindsight, it was a good decision. The complexity of implementing the key migration logic for a logical brick that stores keys for multiple chains would have been painful.
- Hibari relies on Erlang's network distribution service for all significant cluster communication. The short term impact is positive: Erlang message passing "just works."[8] The long term impact is negative: nobody knows the largest practical size of a single Erlang cluster. To build a Hibari cluster with thousands of nodes, we may have to move away from Erlang's built-in messaging or, perhaps, re-write Ericsson's network distribution code.

13. Related Work

Citations of related work have appeared throughout the paper; however, a few other prior works should be mentioned.

As described in the introduction, others have written about the experience of implementing distributed algorithms; citations of [3, 5, 11] only scratch the surface of recent publications. Any Erlang developer who attempts to implement a distributed algorithm from scratch or modify one should read [2] and [21] before starting.

Consistent hashing was introduced by Karger et al. in [10]. The technique has become popular for replacing central directory services for many key $\longrightarrow$ location mapping needs. Central directory servers, in most cases, run on a single host and are therefore a single point of failure for availability and also a likely performance bottleneck.

Amazon's Dynamo distributed hash table [6] uses a layer of indirection in its consistent hashing implementation. Hibari's implementation is quite similar. The main differences are in naming and that Hibari's number and size of hash interval partitions can be changed by using key migration. Also, Hibari's hash partition sizes may be heterogeneous, as demonstrated in Figure 5.

Replica placement is the main topic of [25] and is also discussed in [8] and [22]. The Cassandra distributed database has plug-in API [4] that helps encourage experiments with different placement policies.

14. Conclusion

Using the Hibari distributed key-value store as a case study, we have shown that the path from a pure, proven algorithm to real-world implementation is not smooth. Most of the problems we've encountered with Hibari, both with implementation correctness and with performance, apply not only to Erlang but all distributed computing environments.

We all share the same limitations, such as hard disk drives that grow ever slower relative to the computers they are paired with, failure-prone networks, the maximum speed of light, the fundamental properties of asynchronous messaging, and the problem of making theoretical ideas such as "fail stop" into an equivalent reality. We must never forget that any change to a distributed algorithm or its environment or implementation, no matter how small or innocent the change may appear, may in fact invalidate the algorithm ... and it may be weeks, months, or even years before we notice the error.

15. Availability

In July 2010, Gemini released the Hibari source code under the Apache Public License version 2.0. Its source code and documentation, including Systems Administrator's Guide and Developer's Guide, is available at `http://hibari.sourceforge.net/`.

Acknowledgments

Many thanks to Gemini Mobile Technologies, Inc.: to the Shibuya development team for code and documentation reviews and to the Shibuya field engineer team for their invaluable customer support. I also owe thanks (and probably quite a few drinks) to the ACM's reviewers and to Olaf Hall-Holt, Satoshi Kinoshita, James Larson, Romain Lenglet, Jay Nelson, Joseph Wayne Norton, Gary Ogasawara, Mark Raugas, Justin Sheehy, Ville Tuulos, and Jordan Wilberding. Finally, thank you to Louise Lystig Fritchie for her unstinting patience and editing wizardry. All remaining errors are mine.

References

[1] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 33–38, 2001.

[2] T. Arts, K. Claessen, J. Hughes, and H. Svensson. Testing implementations of formally verified algorithms. In *Software Engineering Research and Practice*, 2005.

[3] T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in erlang. In *Lecture Notes in Computer Science*, pages 140–154. Springer, 2005.

[4] Cassandra Wiki. URL `http://wiki.apache.org/cassandra/-Operations`. Accessed on 31 July 2010.

[5] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, 2007.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of SOSP*, pages 205–220, 2007.

[7] The message flow is: Erlang VM $\longrightarrow$ system event receiver process $\longrightarrow$ error_logger process.

[8] An important exception is described in section 8.

[7] Dynomite key-value store. URL `http://github.com/-cliffmoon/dynomite`. Accessed on 31 July 2010.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on operating systems principles*, pages 29–43, New York, NY, 2003.

[9] Hibari. URL `http://hibari.sourceforge.net/`. Accessed on 31 July 2010.

[10] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. P. Abstract. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In Proc. 29th ACM Symposium on Theory of Computing (STOC)*, pages 654–663, 1997.

[11] I. Keidar and L. Zhou. Building reliable large-scale distributed systems: When theory meets practice. *ACM SIGACT News*, 40(3), September 2009.

[12] L. Lamport and K. Marzullo. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[13] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[14] Memcached. URL `http://memcached.org/`. Accessed on 31 July 2010.

[15] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Annual Technical Conference*. USENIX, 1999.

[16] R. D. Prisco and B. Lampson. Revisiting the paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG 97), volume 1320 of Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1997.

[17] Quviq AB. QuickCheck property-based software testing tool. URL `http://www.quviq.com/`. Accessed on 01 August 2010.

[18] Riak key-value store. URL `http://wiki.basho.com/display/-RIAK/Riak`. Accessed on 31 July 2010.

[19] Squid. Squid http proxy. URL `http://www.squid-cache.org/`. Accessed on 31 July 2010.

[20] Sun Microsystems. Sun announces mysql cluster 7.0 for real-time, mission-critical database applications. URL `http://www.mysql.com/news-and-events/generate-article.php?id=2009_06`. Accessed on 01 August 2010.

[21] H. Svensson and L.-A. Fredlund. Programming distributed erlang applications: Pitfalls and recipes. In *ACM Erlang Workshop*. ACM Press, 2007.

[22] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*, San Diego, CA, 2009.

[23] Tokyo Tyrant. URL `http://1978th.net/tokyotyrant/`. Accessed on 31 July 2010.

[24] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI*, 2004.

[25] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. ACM, 2006.

[26] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on operating systems principles*, pages 230–243, New York, NY, 2001.

Analysis of Preprocessor Constructs in Erlang *

Kitlei Róbert
Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University
kitlei@inf.elte.hu

Bozó István
Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University
bozo_i@inf.elte.hu

Kozsik Tamás
Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University
kto@aszt.inf.elte.hu

Tejfel Máté
Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University
matej@inf.elte.hu

Tóth Melinda
Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University
toth_m@inf.elte.hu

Abstract

Program analysis and transformation tools work on source code, which – as in the case of Erlang – may contain macros and other preprocessor directives. Such preprocessor constructs have to be treated in an utterly different way than lexical and syntactical constructs. This paper presents an approach to treat preprocessor constructs in a non-invasive way that is reasonably efficient and supports code transformations and analyses in an Erlang specific framework.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

General Terms Algorithms, Design, Languages, Reliability

Keywords Erlang, macro, preprocessor, refactoring

1. Introduction

Proper management of preprocessor constructs in a program analysis and transformation tool poses a challenge. This paper proposes a graph representation of such constructs in RefactorErl [2]. This representation stores all syntactical and lexical elements of a given Erlang source code exactly once, while clearly indicating the used preprocessor constructs. The paper also introduces the necessary infrastructure to automatically address the changes that have to be made to the representation if the source code is transformed so that it affects preprocessor constructs.

The hosting tool, RefactorErl, is an Erlang-specific tool that comprises a back-end for representing the AST built upon Erlang source code. The primary concern of RefactorErl, as its name suggests, is refactoring[1], however, it is capable of running different analyses on the source code as well.

The rest of the paper is structured as follows. In Section 2, the RefactorErl tool is introduced shortly. Section 3 describes the preprocessor constructs that can appear in Erlang source code. Section 4 discusses how preprocessor constructs are represented. Transformations may involve changes in the AST that also affect these preprocessor constructs; the way this is handled is presented in Section 5. Section 6 describes related and future work, and Section 7 concludes the paper.

2. RefactorErl

RefactorErl is a source code analyser and transformation tool originally developed for refactoring Erlang programs. The latest release of the tool contains 22 refactoring transformations. RefactorErl provides additional facilities for program manipulation such as clustering functions and other Erlang forms based on how tightly they are coupled [5]. It introduces a powerful query language [3] to collect information about an Erlang program for analysis or refactoring as well. This query language operates on the level of semantic entities of Erlang and is directly accessible through the user interface of RefactorErl, making it possible for programmers to obtain knowledge about the structure and the semantic relationships of, and also to compute different software complexity metrics [4] about, an Erlang program.

RefactorErl represents an Erlang program as a three layered *semantic program graph (SPG)*. The SPG is a rooted, directed graph which contains lexical, syntactic and semantic nodes and edges. The syntactic nodes and edges constitute the abstract syntax tree of the represented Erlang program and provide the basis for constructing the SPG. After parsing, the different asynchronous semantic analyser modules of RefactorErl add semantic nodes and edges to the SPG such as binding information about variables or the call graph. The semantic analyzer framework is incremental, and it automatically restores the consistency of the semantic program graph after a source code transformation and also after manual editing of the source code.

* Supported by KMOP-1.1.2-08/1-2008-0002 and Ericsson Hungary

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09...$10.00

[1] Refactoring is the process of changing the structure of a program without changing its externally observable behaviour.

The query language is implemented with traversals in the SPG based on *path expressions*. A path expression can be defined by giving a start node and a list of edge labels to follow. The graph edges can be traversed forwards and backwards, and it is possible to filter the result with syntactic or semantic information.

This paper describes how the various preprocessor constructs of Erlang are represented in the semantic program graph, as well as how refactorings interact with these constructs.

3. Preprocessor constructs in Erlang

In order to make decisions about the choice of representation, let us determine the kinds of preprocessor usage first. There are five preprocessor constructs to be considered: macro definitions and undefinitions, macro substitutions, conditional compilation, and file inclusion. Other constructs such as records may be affected by the preprocessor (`snmp_types.hrl` in the `snmp` application does conditionally define the fields of a record, as seen in Listing 1). The approach that we present in the rest of the paper is applicable with minimal modifications to represent these constructs, therefore we shall not elaborate on them further.

```
-ifdef(SNMP_USE_V3).
-record(message, {version, vsn_hdr, data}).
-else.
-record(message, {version, community, data}).
-endif.
```

Listing 1. Conditional record definition in snmp_types.hrl

The most important question we are seeking the answer to is the following: what is the list of tokens a macro can be expanded to when a macro substitution is encountered?

Erlang/OTP R13B04 contains 1.45 million lines of effective code, 23024 macro definitions, 1204 conditional compilation constructs (out of which 305 of them contain an `-else`), and 2530 instances of file inclusion. Out of the 23024 macro definitions, 11499 (almost 50 %) are simple numerical constants. The other half of the macro definitions is quite diverse: variables, strings, binaries, tuples and lists of constants, record accesses, references to other macros and function calls are all among them in various proportions.

Simple macro use The simplest case is where a macro is defined, and then it is used, as in Listing 2. In most of the cases, the body of the macro is an atom or a number. The substitution is trivially made.

```
-define(m, 123).

f() -> ?m.
```

Listing 2. A simple macro definition and substitution

Macros may have parameters, even of arity zero, which adds a little technical hurdle, as seen in Listing 3. In this example, `?m1` is defined with no parameters, while `?m2` is defined with an empty list of arguments.

```
1 -define(m1, g).
2 -define(m2(), g).
3
4 f1() -> ?m1().          % returns 'h'
5 f2() -> ?m2().          % returns 'g'
6
7 g() -> h.
```

Listing 3. The role of parameters in macro substitutions

The difference is subtle: the macro substitution on line 5 involves only `?m1`, while the substitution on line 4 involves the parentheses as well: `?m2()`. The preprocessed code is shown in Listing 4.

```
f1() -> g().         % returns 'h'
f2() -> g.           % returns 'g'

g() -> h.
```

Listing 4. Listing 3 after preprocessing

The rules for macros were made more lenient in the recent R14A release of Erlang/OTP: the source code in Listing 5 compiles, with `f` returning `{f, g, h}`. Previous releases of the compiler reject this source code, complaining that macros cannot be redefined.

```
-define(m, f).
-define(m(), g).
-define(m(X), X).

f() ->
        {?m, ?m(), ?m(h)}.
```

Listing 5. Defining macros with the same name but different arities

All of the above macros are well-behaved: they are expanded into a series of tokens that form a syntax subtree. However, this is not always the case; later we will describe cross-cutting macros that do not have this property.

Another interesting case is the vanishing macro. The vanishing macro is a macro, usually having no arguments or an empty argument list, that has an empty body. Listing 6 demonstrates this along with the peculiarity present in the R13 release series of Erlang/OTP that the ending parenthesis of macro definitions can be left out as a "nicety" (as of Erlang/OTP R14A, ending parentheses are required). Counterintuitively, neither function `no_params` nor function `no_params2` has any parameters. Since the substitution of a vanishing macro is expanded to no tokens, how can it be connected to the syntax tree? A simple solution is to represent vanishing macros as special comments.

```
-define(vanish,.
-define(vanish2(),.

no_params(?vanish) -> 1.
no_params2(?vanish2()) -> 2.
```

Listing 6. The vanishing macro

An example of the vanishing macro can be found in the `ttb.erl` module of the `observer` application of Erlang/OTP. The code is shown in Listing 7; line 8 contains the vanishing definition of the macro.

```
1  -ifdef(debug).
2  -define(get_status,
3            ;get_status ->
4                erlang:display(
5                    dict:to_list(NodeInfo)),
6                    loop(NodeInfo)).
7  -else.
8  -define(get_status,).
9  -endif.
```

Listing 7. A conditional from ttb.erl, one case is always non-syntactic, the other is always vanishing

A very unfortunate and badly readable case of macro application is when the tokens resulting from the substitution do not form the front of a full syntax subtree; this is called a syntactically cross-cutting macro substitution. In Listing 8, we see a macro that is used in two similar expressions. There is one notable difference: the substitution cross-cuts the syntax tree on line 4 while it does not on line 3 because of the left-associativity of the addition operator. Thus, we can conclude that syntactical well-behaviour is the property of the macro substitution, not the macro definition. While this example can be amended by additional parentheses, there are even worse cases of non-syntactical macros; Listing 16 on page 4 shows two particular cases.

```
1  -define(two, 1+1).
2
3  f() -> ?two + 1.
4  g() -> 1 + ?two.
```

Listing 8. A macro with both a syntactical and a non-syntactical expansion

The first macro definition on line 2 in the already mentioned Listing 7 shows an example of a non-syntactic macro in actual code. In contrast to the macro in Listing 16 (where it could be determined only at the point of substitution whether the macro is syntactically cross-cutting), this definition of the `?get_status` macro always cross-cuts the syntax if used, because it begins with a semicolon that is syntactically always below a clause, and never its first token.

Let us examine the only use of the macro in the code of `ttb.erl`, the abridged version of which is shown in Listing 9. Supposing the crosscutting definition of the macro is in effect, the code is transformed into the one shown in Listing 10. The semicolon is part of the substitution, but the whole receive construct is not the outcome of the substitution, therefore the substitution is cross-cutting.

```
1  loop(NodeInfo) ->
2      receive
3          % several cases...
4          {stop,FetchOrFormat,Sender} ->
5              a_long_body
6                  % note the missing ; here
7          ?get_status
8      end.
```

Listing 9. Usage of ?get_status in ttb.erl

```
1  loop(NodeInfo) ->
2      receive
3          % several cases...
4          {stop,FetchOrFormat,Sender} ->
5              a_long_body
6                  % note the missing ; here
7              ;get_status ->
8                  erlang:display(
9                      dict:to_list(NodeInfo)),
10                     loop(NodeInfo)
11     end.
```

Listing 10. ?get_status is substituted

Undefinition Macros can be undefined in the code, ensuring that they have no definition thereafter (Listing 11). Of course, undefined macros cannot be dereferenced.

```
1  -undef(m).
```

Listing 11. A macro is undefined

External macro definition It is not always possible to determine the configuration of a macro by only using static analysis of the source code itself: macros can originate from the command line as well, as seen in Listing 12. Macros defined on the command line cannot have parameters, and they are restricted to constructing a single atomic value, making all of their uses syntactical.

```
1  erlc -Dx=ok test.erl      # x is defined
2  erlc test.erl             # x is not defined
```

Listing 12. A command line macro definition

If we encounter a substitution for a macro that was neither defined nor undefined previously, we have to conjecture that the definition of the macro is supplied externally.

Conditional compilation A conditional compilation construct is replaced during preprocessing by one of two series of forms, depending on the definedness (`-ifdef`) or undefinedness (`-ifndef`) of a certain macro. If conditional compilation is used, a macro substitution has the potential to be expanded in more ways than one. In Listing 13, `f()` will return either `'a'` or `'b'`, depending on the configuration of `'x'` at line 1. We will indent conditionally compiled forms for enhanced readability, although this is not common practice in real code.

```
1  -ifdef(x).
2      -define(y, a).
3  -else.
4      -define(y, b).
5  -endif.
6
7  f() -> ?y.
```

Listing 13. A macro with ramified definitions

Since it is the intention of the programmer that both cases of the conditional compilation be present in the source code, a refactoring tool cannot ignore one of the cases: if the condition is reversed, the forms from the other branch will be used, therefore all refactorings have to deal with both branches. In Listing 14, for example, when renaming function `f`, both instances of `f` have to be renamed.

```
1 -ifdef(x).
2     f() -> 1.
3 -else.
4     f() -> 2.
5 -endif.
6
7 g() -> f().
```

Listing 14. A function with ramified definitions

It is possible to write conditional compilation constructs that leave the configuration of a macro garbled, as seen in Listing 15. While this code can be compiled, if the definition of `x` is removed from line 1, the code becomes invalid. For this reason, we shall require (and check) that the configurations of all macros be compatible at the end of each conditional compilation branch. The undefined configuration of the macro is only compatible with the undefined configuration (see Listing 31); all other configurations can be combined.

```
1 -define(x, 1).
2 -ifdef(x).
3     -undef(y).
4     -define(y, 1).
5 -else.
6     -undef(y).
7 -endif.
8 f() -> ?y.
```

Listing 15. Conditional macro configurations

Furthermore, it is possible to write code in which different definitions of the same macro cross-cut the syntax tree in different ways, as seen in Listing 16. Depending on whether `?x` is present, the substitution on line 6 is expanded either to `1*2+g()` or `1+m:g()`.

Such macros make code maintenance extremely hard. An approach for handling such macros was described in [10], however, we have decided to restrict our representation so that whenever a macro has more than one body, they are required to be expanded to a syntactically correct subtree, the top node of which is an expression.

```
1 -ifdef(x).
2     -define(y, * 2 +).
3 -else.
4     -define(y, + m :).
5 -endif.
6 f() -> 1 ?y g().
```

Listing 16. Macro with two different non-syntactical bodies

Fortunately, the majority of conditional compilation cases are used for debugging purposes, similar to the one shown in Listing 17. Such cases can be represented more conveniently than the general case, as described in Section 6.

```
1 -ifdef(my_debug).
2     -define(dbg, io:format("debug␣msg")).
3 -else.
4     -define(dbg, ok).
5 -endif.
```

Listing 17. A very common case of conditional compilation

Include files Finally, include files can also affect the configuration of macros. A common way is that a macro is defined in a `.hrl` file (less often, in an `.inc` file), and it is included in an `.erl` file (possibly through a chain of includes). Out of the mentioned 23024 macro definitions in Erlang/OTP, 15996 (69 %) are found in `.hrl` files. As a prime example, `wx.hrl` contains some record definitions and 3333 macro definitions, all of which are constants or calls to a function that returns constants.

A simple case of file inclusion is seen in Listings 18 and 19.

```
1 % header.hrl
2 -define(m, 1).
```

Listing 18. A macro is defined in a header file, header

```
1 % includer.erl
2 -include("header.hrl").
3
4 f() -> ?m.
```

Listing 19. A macro is defined in a header file

In some cases, a macro is used in a `.hrl` file, either as a condition in conditional compilation, or in the body of a function. This is particularly worrisome, as the code that appears on line 2 in Listing 20 contributes to two distinct syntactical parts, but in a different way: once it appears in the syntax tree of `includer1.erl` from Listing 21 with the body `1`, and once it appears in the syntax tree of `includer2.erl` from Listing 22 with the body `2`. In Listing 23, we have to go outside the language to determine the body, as it is externally supplied. Worse still, these macro substitutions may be part of rather dissimilar ASTs: imagine the macro bodies of Listing 16 coming from two include files in the manner of Listing 20. This is clearly not good: at some point, we will have to output a textual source file, and by allowing such syntactically bad cross-cutting macro bodies, we would have a hard time reassembling the text. Instead, we have chosen to disallow syntactically cross-cutting macro bodies if the macro has more than one body; with syntactically well-behaved bodies, we can join their top nodes and insert this node in the syntax tree. This way, `?m` on line 2 can be represented with one node that has two trees below it, one coming from the body `1` and another from the other body `2`. This representation is described in detail in Section 4.

```
1 % header.hrl
2 f() -> ?m.
```

Listing 20. A macro is passed to a header file

```
1 % includer1.erl
2 -define(m, 1).
3 -include("header.hrl").
```

Listing 21. A macro is passed to a header file

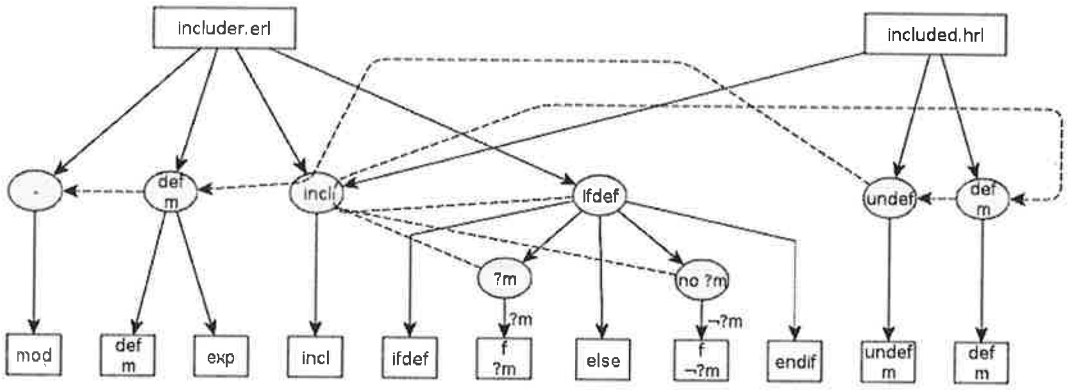


Figure 1. Macro configurations

```
% includer2.erl
-define(m, 2).
-include("header.hrl").
```

Listing 22. A macro is passed to a header file

```
% includer3.erl
% ?m is externally defined
% when compiling includer3.erl
-include("header.hrl").
```

Listing 23. A macro is passed to a header file

As RefactorErl contains powerful querying facilities, it is not too difficult to find macros in the loaded files that are ambiguous or otherwise inconvenient. While changing these code parts might be very difficult (see Listing 1 and Listing 9), at least it is possible to identify the worrisome code parts using the representation that is described in Section 4.

Disallowing circular includes In our representation, we disallow circular includes. They present lots of subtle complexities, and probably because of them, they are not common in practice. In the following, we present a nontrivial case study.

Listings 24 and 25 show a subtle combination of include directives, conditional compilation, and macro undefinition. When compiling `messy.erl`, some macros are set, and then `header.hrl` is included for the first time. Among others, `?x` is defined, therefore the first branch of the conditional construct is inserted on line 2. This branch undefines `?x`, then includes `header.hrl` for the second time. Since `?x` is now undefined, the second branch is taken this time around, which undefines `?x` again – this has no effect, `?x` is still undefined. Now we expand `?callOther`, with the parameters `?f1` and `?f2`, which currently expand to `f1` and `f2`. Thus, we get the first function, line 5 in Listing 26. Now, we continue with the remainder of the first branch from the first inclusion. Through two redefinitions, we have now exchanged the roles of `?f1` and `?f2`. Finally, expanding `?callOther` again, we now get the mirroring countdown function `f2/0`.

```
1  % header.hrl
2  -ifdef(x).
3      -undef(x).
4      -include("header.hrl").
5      -undef(f1).
6      -define(f1, f2).
7      -undef(f2).
8      -define(f2, f1).
9  -else.
10     -undef(x).
11 -endif.
12
13 ?callOther(?f1, ?f2).
```

Listing 24. Messy circular include

```
% messy.erl
-module(messy).

-export([f1/1]).

-define(callOther(ThisFun, OtherFun),
        ThisFun(1) -> true;
        ThisFun(N) -> OtherFun(N-1)).

-define(x, x).
-define(f1, f1).
-define(f2, f2).
-include("header.hrl").
```

Listing 25. Messy circular include

```
1  -module(messy).
2
3  -export([f1/1]).
4
5  f1(1) -> true;
6  f1(N) -> f2(N-1).
7
8  f2(1) -> true;
9  f2(N) -> f1(N-1).
```

Listing 26. Messy circular include, preprocessed

Note that postulating the non-circularity of includes does not disqualify the multiple inclusion of the same include file. Some interesting macro constructs are still present, e.g. header guards, shown in Listings 27 and 28.

```
% includer.erl
-include("header.hrl").
-include("header.hrl").
```

Listing 27. Header guard

```
% header.hrl
-ifndef(guard).
    -define(guard).
    % ...
-endif.
```

Listing 28. Header guard

Summary of possible configurations In this section, we have investigated the possible configurations that a macro can take, which has to be considered when encountering a macro substitution. The following five distinct configurations are possible.

- A single body, either allowing or disallowing it to be syntactically unfitting in substitutions.
- Multiple bodies, disallowing them to be syntactically unfitting.
- Externally defined.[2]
- Undefined.
- A combination of an external definition and one or more bodies is also possible, if an include file is included both from a file that does not reference the macro, and an include file that defines one or more bodies for the macro.

Limitations and applicability We can handle preprocessor constructs with certain – not too limiting – restrictions.

- We do not handle programs with compilation errors.
- We require that the configurations of all macros be compatible at the end of each conditional compilation branch. This is checked when the tool loads the files.
- We restrict our representation so that whenever a macro has more than one body, they are required to be expanded to a syntactically correct subtree. This is also checked when the tool loads the files.
- We do not handle circular includes. This restriction could be relaxed if necessary. This is also checked when the tool loads the files.

[2] If the configuration of the macro is previously unknown, this is the only available configuration choice.

4. Representation

We now define the SPG representation of Erlang programs that stores all syntactical and lexical elements exactly once, while clearly indicating the relations of preprocessor constructs present in the source code.

Macro environments An Erlang source file consists of a list of forms. We split these forms into macro environments. The boundaries of all macro environments (except for two notable differences, see below) are determined so that they contain the longest range of forms where all macros have the same configuration. In general, such a macro environment will begin with a `-define` or `-undef` directive, and contain function forms or non-preprocessor directive forms. Macro environments following a conditional or include environment may start with the latter, changing the configuration of no macros from the previous environment.

We assign conditional compilation constructs a specific kind of environment. This environment can simultaneously change the configuration of several macros, because it has two lists of environments under it: one for each form list of the conditional construct. The macro configuration changes of the two branches have to conform to each other.

Include files also have their designated macro environments. The macro environments of the included file act as if they were inserted in place.

Listings 29 and 30 with the accompanying Figure 1 illustrates most possible combinations of macro environments. The graph shows the file nodes on top, the macro environments in grey, and the forms of the file in the bottom, with slightly abbreviated notation. The contents of the macro environments indicate the changes of macro configurations that take place in the environment. The first environment of `includer.erl` begins with a `-module` form, which does not change any macros. The second form in the file does change the configuration of `?m`, therefore, it has to be put into a new macro environment. The third form, a function form, also becomes a member of this environment. Next up are two special forms: an include form and a conditional `-ifdef` form. The conditional form has two sub-environment lists, both of which have only one element. All of the edges and nodes of these two branches are labelled with `?m` and ¬`?m` respectively, indicating that these parts of the graph should only be visible depending on whether `?m` is defined.

```
% includer.erl
-module(includer).
-define(m, ?MODULE).
-export([f/0]).
-include("header.hrl").

-ifdef(m).
    f() -> ?m.
-else.
    f() -> 1.
-endif.
```

Listing 29. Macro configurations

```
% header.hrl
-undef(m).
-define(m, ok).
```

Listing 30. Macro configurations

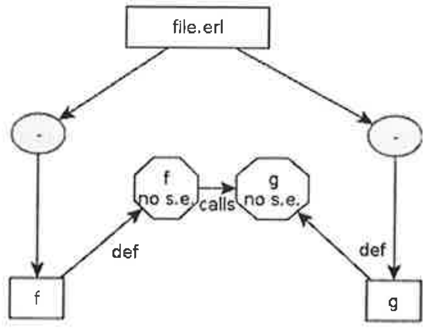


Figure 2. An abstract view of Figure 6 with condition $\neg?m$

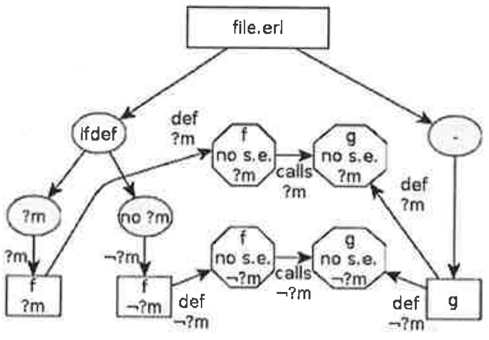


Figure 3. Figure 5 before merging is done

Tags of conditions The representation also tags all nodes and edges by the set of conditions that they are under.[3] These tags are used to temporarily create an abstract view of the graph that shows only the parts relevant to a specific set of conditions. Figure 2 shows such a view on Figure 6 with the condition $\neg?m$; the original source code is shown in Listing 48. As the condition of the `-ifdef` is already determined, only the second branch is present. Also, the tags $?m$ and $\neg?m$ are no longer visible.

After the abstract syntax tree is constructed, automatic semantic analysers are run on all forms for all configurations that the form contains. The analysers add semantic nodes and edges to the graph, tagged with the configuration that they are run under. After an analyser is run in all possible configurations, nodes and edges with different tags but the same semantic contents are merged. This helps us keep the graph small, and decreases exponential explosion. Figure 3 shows the semantically analysed graph of Listing 47, which is merged into Figure 5.

Preceding environments All macro environments have a set of preceding environments. For the very first environment of a file that is not included from any other file, this set is empty. For top-level environments that have no parent environment, the set of predecessors contains the last sub-environments of the previous environment in the file if the previous environment is a conditional construct or an include file, and the previous environment as the only element in all other cases. The first sub-environments have the same predecessor as the parent environment. The predecessors of the first environment of an include file are the environments (last sub-environments) preceding forms that include the file.

Figures 1 and 4 show examples of predecessor environments in dashed lines. Figure 1 contains no environments that have more than one predecessor. As the environment before the `-ifdef` construct is an `-include` construct, making the last environment of the include file (where `?m` is defined) the predecessor of the `-ifdef` environment. Both sub-environments of the `-ifdef` environment inherit this predecessor from the parent environment. As `header.hrl` is included by only one form, the first environment of the include file (an undefiniton of `?m`) has only one preceding environment: the second macro environment of `includer.erl`. However, the first macro environment of the included file in Figure 4 has four preceding environments, as the file is included in three places. Both sub-environments of the `-ifdef` environment before the `-include` construct in `file1.erl` contribute to the set of preceding environments.

The configuration of a macro at a substitution can be determined by inspecting the changes that the preceding macro environments introduce. The following configurations are possible.

- If no previous environment changes the configuration of the macro, we conclude that the macro is externally defined[4]. Externally defined macros cannot cross-cut the syntax.
- If a single `-define` is found, the macro has one body, and it is allowed to be syntactically cross-cutting.
- If several bodies are defined in the paths of a conditional construct or in different including files (see Listing 20), the macro has multiple bodies, and they are not allowed to cross-cut the syntax.

Loading the files Files are loaded into the system one after the other. When an environment has more than one preceding environment, they are checked for conflicts: non-conforming macro configurations such as the one in Listing 31, where one path has `?x` defined and the other does not, are not allowed.

```
-ifdef(m).
    -define(x, a).
-else.
    -undef(x).
-endif.
```

Listing 31. Non-conforming macro configurations

When a macro substitution is encountered, the system checks the current configuration of the macro. If it has exactly one body, it is expanded to tokens (possibly cross-cutting the syntax), with additional edges marking the tokens as results of a macro substitution. If the macro is not allowed to be syntactically cross-cutting, a syntactical macro node is inserted, similar to the workings of the `epp_dodger` module of the standard library of Erlang/OTP [1]. These nodes are similar to the environment nodes of conditional compilation: they split the syntax tree into different paths, all nodes and edges of which are tagged with their originating configuration, as seen in Figure 4, which is based on Listings 32-35. In this example, three files set up different configurations for the same macro, then include `header.hrl`, which contains a substitution of the

[3] The edges have another kind of tag as well, which is used for syntactical construction, and makes convenient queries possible. For further elaboration, see [8, 9].

[4] Analysers may take into account the build scripts used, and can precisely determine whether all externally defined macros are present. As build scripts come in such a wide variety, we did not take this approach. It would not take too much effort, however, to introduce an annotation that explicitly identifies externally defined macros.

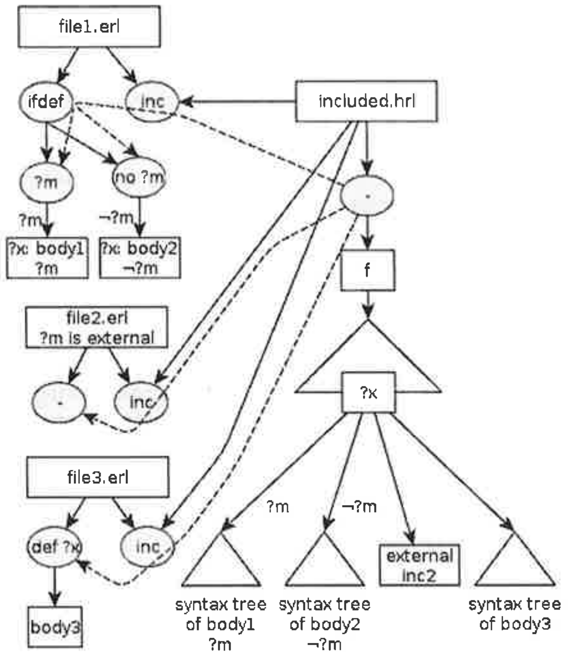


Figure 4. Multiple substitutions for `?m`

macro. Since three bodies and an external definition is associated with the macro at the environment where the macro substitution occurs, all bodies of the macro are required to comprise a syntax subtree. A disambiguation node is placed in the syntax tree, and the different bodies are all constructed below, with the appropriate condition tags ($?m$ and $\neg?m$ for the two configurations from `file1.erl`, and no restrictions for the other two cases).

```
% file1.erl
-ifdef(m).
    -define(x, body1).
-else.
    -define(x, body2).
-endif.
-include("header.hrl").
```

Listing 32. Multiple substitutions for `?m`

```
% file2.erl
-include("header.hrl").
```

Listing 33. Multiple substitutions for `?m`

```
% file3.erl
-define(x, body3).
-include("header.hrl").
```

Listing 34. Multiple substitutions for `?m`

```
% header.hrl
f() -> ?x.
```

Listing 35. Multiple substitutions for `?m`

An include file is reloaded every time it is included by another file. Forms with macro substitutions are updated to reflect the changes in macro configurations.

5. Analysis and transformation

Transformations in RefactorErl are formulated as transformations of the syntax tree: insertions and deletions of nodes and subtrees, and updates of nodes. After the transformations are done, semantic analysers are run, which bring the SPG up to date so that the new semantic nodes and edges reflect the new connections (variable bindings, function calls etc.) in the code. This section describes how to transform the AST core of the SPG with respect to preprocessor constructs, and how the analyser infrastructure is prepared to make correct nodes and edges for all macro configurations.

One possibility would be to prepare the transformations themselves to deal with preprocessor constructs. However, since macros can occur almost anywhere in the source code, this approach would likely result in compromises: operations adressing the correct use of macros would be scattered in the code of the tool thus making it error-prone and hard to maintain. Also, this would make writing new transformations more difficult.

Our choice is to let the transformations describe only the syntactical changes, and create an infrastructure that automatically handles the preprocessor constructs on the way. [5]

Let us examine how the syntactical changes made by the transformations can interact with the various preprocessor constructs.

The `-undef` directive affects only transformations that insert new forms with macro substitutions in them. The transformation has to ensure that the macros used inside the function are bound appropriately. For example, if `?x` is extracted from line 3 in Listing 36, the transformation has to place the extracted function directly before or directly after line 3, otherwise the substitution `?x` would not be expanded as before.

```
1 -undef(x).
2 -define(x, 1).
3 f() -> ?x.
4 -undef(x).
```

Listing 36. Extracting a macro expression to a function

As only syntactical changes are made, the syntax nodes of macro substitutions can be temporarily changed. In Listing 37, there are two calls to `f/0`, both coming from macro substitutions. If the function is renamed to `g/0`, the transformations, unaware of the macro expansions, will change the underlying *function application name* syntax nodes.

```
-define(m1, f()).
-define(m2(X), X()).
f() -> ?m1, ?m2(f).
```

Listing 37. Macro substitutions before 'f' is renamed to 'g'

One possibility is to remove the affected macro substitutions from the code, and leave only the expanded nodes, as seen in Listing 38. This can always be done, and it does not need to change the original definitions of the macro.

[5] Of course, there are several transformations that deal specifically with preprocessor constructs: renaming, moving, inlining or extracting a macro, or moving functions, macros or records to an include file. Obviously, in these cases, the transformations themselves have to handle the preprocessor constructs.

```
-define(m1, f()).
-define(m2(X), X()).
g() -> g(), g().
```

Listing 38. Macros inlined after renaming

However, the intention of the programmer is likely different: if he did not want the macro calls in the code, he would not have put them there in the first place. Therefore, if possible, it is better to change the macro substitution arguments and the macro definition to fit the changes in the syntactical structure, as seen in Listing 39.

```
-define(m1, g()).
-define(m2(X), X()).
f() -> ?m1, ?m2(g).
```

Listing 39. Macro substitutions after 'f' is renamed to 'g'

Tracing the changes back to the definition and the arguments is not always possible. Listing 40 shows a case where the body of `?m` is used once as the name of a function, and once as an atom. If the function is renamed, changing the macro body would also change the value of the atom.

```
-define(m, f).
f() -> fun ?m/0, ?m.
```

Listing 40. Semantically inconsistent macro use

Side effects (message passing, file operations etc.) and type conversions make it impossible to do refactorings right in every case. Let us consider the code in Listing 41, which shows a scheme that is similar to some functions that can be found in production code. This function calls another one from the same module, except that the function name is saved to a file, transformed arbitrarily and then read back. The halting problem assures us that it is impossible to tell even whether the function call on line 6 is reached (`change_file_arbitrarily` might run forever), and the function name might be changed both by the algorithm and other actors transforming the file while `change_file_arbitrarily` is running. Renaming any function in the module raises the question whether the atom `original_fun_name` should be changed as well; this question cannot be answered in general, consequently all function renamings should be disabled by a conservative approach.

Since such problems are quite common, refactoring tools have to make a compromise between correctness and usability by extending range of decidable cases (e.g. RefactorErl uses data flow to detect expressions that bind the same value, one application of which is to trace where a function name might have come from), and helping the user in the undecidable cases as much as possible.

```
1  f() ->
2      A1 = original_fun_name,
3      write_fun_name_to_file(A1, "file.txt"),
4      change_file_arbitrarily("file.txt"),
5      A2 = read_fun_name_from_file("file.txt"),
6      erlang:apply(?MODULE, A2, []).
```

Listing 41. Name transformation

Include files introduce other realistic cases of the same problem. In Listings 42-44, the macro substitutions in the two modules both expand to function applications, but the called function is not the same: `includer1:f/0` in one module, `includer2:f/0` in the other. Thus, if one is renamed, the substitution of `?m` cannot be amended by changing the macro definition in the include file, lest the other function call is changed as well.

```
% header.hrl
-define(m, f).
```

Listing 42. Semantically inconsistent macro uses in including files

```
% includer1.erl
-module(includer1).
-export([f/0]).
-include("header.hrl").
f() -> ?m().
```

Listing 43. Semantically inconsistent macro uses in including files

```
% includer2.erl
-module(includer2).
-export([f/0]).
-include("header.hrl").
f() -> ?m().
```

Listing 44. Semantically inconsistent macro uses in including files

In Listing 45, the macro `?m` is expanded to a function application with a module qualifier. If a transformation removes the module qualifier, the macro definition is not applicable anymore.

```
-define(m(X, Y), X:Y()).
f() -> ?m(?MODULE, f).
```

Listing 45. Macro with a module qualifier

Besides inlining the substitution, another valid approach is to offer a newly devised macro that is as close to the original one as possible, as seen in Listing 46.

```
-define(m(X, Y), X:Y()).
-define(m2(Y), Y()).
f() -> ?m2(f).
```

Listing 46. A new macro is introduced

Conditional compilation Probably the hardest question is how conditionally compiled code parts should be transformed. A primary concern is the soundness of the transformation, but efficiency cannot be disregarded: introducing a new conditional compilation construct splits the code into two paths, foreshadowing exponential explosion.

In the following, we describe our approach to handle the transformation of conditional compilation constructs. First, we present a basic approach that is generally applicable but not too efficient in certain cases, then we will shortly discuss the possibilities for improvement.

Refactoring is done in two phases. The first phase collects information by walking the edges of the SPG and querying its nodes, while the second phase makes the actual transformations: insertions and deletions of edges, and insertions, deletions and updates of nodes.

Conditional compilation constructs branch the code into two paths[6] depending on the presence or absence of a macro. Let us call a subset of all such macros a conditional configuration. Each conditional configuration gives us an abstract view of the SPG, as the path chosen at each conditional construct is known.

[6] If no `-else` directive is present, the second path contains no forms.

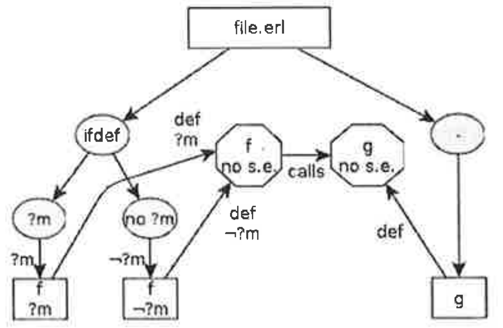


Figure 5. Two code paths, neither has side effects

We perform the refactoring on all possible conditional configurations, then we merge similar nodes, as we want all syntactic nodes to be represented only once in the graph. This is illustrated in Listing 47 and Figure 5: the code on line 6 is present in both code paths, which leaves two distinct g function nodes after renaming f to g in both paths, shown in Figure 3.

However, storing the whole graph for all configurations is too costly. Our current measurements show that for one macro configuration, each kilobyte of source code generates approximately 150 kilobytes of internal representation. While this requirement is imposing, it is within the limitations of current personal computers to load a typical stand-alone application of 400 kB-3 MB, which will require around 60-450 MB of storage. Dedicated machines should handle larger systems as well, such as all applications of Erlang/OTP (around 50 MB of code, which requires somewhat less than 8 GB of storage when loaded). However, adding a macro configuration naïvely doubles the space requirement. Therefore, our choice of representation uses tags to distinguish between the different code paths, as described in Section 4.

Since conditional compilation cannot cross-cut forms, semantic nodes and edges that do not cross form boundaries, such as variables, are not directly affected. However, they can still pass on information, e.g. through data flow.

```
-ifdef(m).
    f() -> 1.
-else.
    f() -> 2.
-endif.
g() -> f().
```

Listing 47. Two code paths

In Listing 48, after the two runs of the function semantic analyser has produced the two distinct semantical nodes in Figure 6, they cannot be merged: one has a side effect, the other does not.

```
-ifdef(m).
    f(X, Y) -> X + Y.
-else.
    f(X, Y) -> X ! Y, Y.
-endif.
g(X) -> f(X).
```

Listing 48. Two code paths, one with side effects

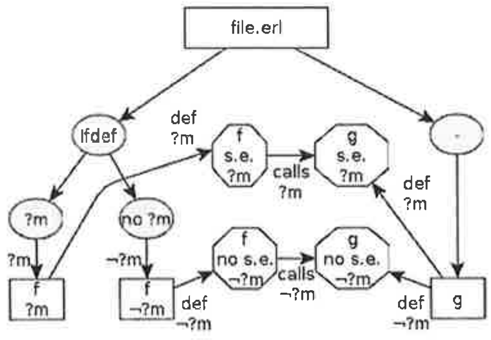


Figure 6. Two code paths, one with side effects

6. Related and future work

There are several possible approaches to representing preprocessor constructs for analysis and transformation purposes. It is possible to choose to work solely with preprocessed code, but it does not capture the intent of the programmer, who used the preprocessor constructs on purpose. When handling preprocessor constructs, two approaches are common.

It is possible to restrain the use of preprocessor constructs, requiring them to be "meaningful" in some way, and rejecting code that contain "ugly" constructs. An obvious restriction is that the code should only contain syntactically well-behaved macros. With this restriction, the source code with macros can be represented as a tree-based extension of the syntax tree.

This is the approach taken by the `epp_dodger` module in the standard library of Erlang/OTP [1], which is used in the refactoring tools Tidier [6] and Wrangler [7]. This approach can handle most code if the programmers are not overtly adventurous in their use of macros. However, some rare cases may go beyond these limits. Also, as the syntax tree is not regarded as a whole (the subtrees of macro bodies are not connected to the substitution node), extra effort is required to correctly handle all subtle cases.

The other common approach is to represent as much of the preprocessor language as possible including cases that are very hard to handle. In spite of these hardships, there are several such results for the C programming language and its preprocessor language. C has a very similar preprocessor language to Erlang, and its facilities are heavily used in most large projects.

Garrido describes in her dissertation [10] how preprocessor constructs in C are represented in CRefactory. As C and Erlang have many similarities regarding their preprocessors, it is no surprise that the two approaches are very similar; this dissertation was a major influence on our chosen representation. The main difference is that our internal representation contains semantic information as part of the SPG, while CRefactory collects semantic information while walking the graph.

Ernst, Badros and Notkin [11] present an empirical study of the preprocessor usage in C. While this paper does not focus on such statistics, it is possible to compile one in the future. A free suite of software of comparable size is available for Erlang: Erlang/OTP R13B04 contains approximately the same number of lines, 1.4 million, as is investigated in the aforementioned study. Also, the site `trapexit` features 1445 open source projects in Erlang, which could be analysed.

Improvements on the basic algorithm The approach described in this paper can be improved in various ways. An obvious way to improve the performance of preprocessing is to branch the transformations on conditional configurations only as needed. If no path expression that the transformation runs passes any edge or node that is tagged with a condition, then the transformation ends normally, without interfering with preprocessor constructs. Currently, refactorings in RefactorErl generally take a little but noticeable time to complete; if this interval is increased exponentially by the number of conditional configurations, the user may easily get upset.

Another possibility for improvement is to treat conditional compilations such as Listing 17 specially, so that they do not require a separate pass for the simpler branch. The main branch with the debug message still has to be represented, as it may contain variables or expressions.

One further possibility is to make use of our knowledge about the behaviour of certain semantic analysers, and instead of separately running them for both conditional branches and then compacting the resulting nodes, the appropriate nodes can be made in one pass.

7. Conclusion

We have presented some challenges that handling preprocessor constructs poses. In this paper, we have presented a way to overcome these challenges, which allows us to do more comprehensive analysis and transformation of Erlang source code. The challenges are partly representational, but they stem from the algorithmical question: how can transformations on the source code be made so that they conform to the will of the programmer, when preprocessor constructs are present in the source code?

After assessing the possible cases of preprocessor usage, we have proposed an algorithm and a fitting representation. The purpose of the algorithm was that neither the transformations nor the semantic analysers should be aware of the presence of preprocessor constructs in the code. Rather, the problem is solved on a higher level of abstraction by the transformation framework, transparently handling the conditional configurations.

References

[1] Syntax_Tools Reference Manual. `http://www.erlang.org/doc/apps/syntax_tools/`

[2] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Bozó, I., Tóth, M., Király, R.: Modeling semantic knowledge in Erlang for refactoring, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Cluj-Napoca, Romania, Studia Universitatis Babeş-Bolyai, Series Informatica, vol. 54(2009) Sp. Issue

[3] Lövei, L., Hajós, L., Tóth, M.: Erlang Semantic Query Language, On the 8th International Conference on Applied Informatics, ICAI 2010

[4] Király, R., Kitlei, R.: Implementing structural complexity metrics in Erlang, On the 8th International Conference on Applied Informatics, ICAI 2010

[5] Lövei, L., Hoch, Cs., Köllő, H., Nagy, T., Nagyné-Víg, A., Horpácsi, D., Kitlei, R., and Király, R.: Refactoring Module Structure, In Proceedings of the 7th ACM SIGPLAN workshop on Erlang, pages 83–89, Victoria, British Columbia, Canada, Sep 2008.

[6] Sagonas, K., Avgerinos, T.: Automatic Refactoring of Erlang Programs Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming, 2009, pp. 13-26. ACM, Coimbra, Portugal, 2009. ISBN 978-1-60558-568-0.

[7] Li, H., Thompson, S.: Tool support for refactoring functional programs, Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pages 199-203. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-977-7.

[8] Kitlei, R.: Reconstructing syntax in RefactorErl PhD workshop, Central Europen Functional Programming Summer School, May 2009

[9] Kitlei, R., Lövei, L., Tóth, M., Horváth, Z., Kozsik, T., Király, R., Bozó, I., Hoch, C., and Horpácsi, D.: Automated syntax manipulation in RefactorErl In 14th International Erlang/OTP User Conference, Stockholm, Sweden, Nov 2008

[10] Garrido, A.: Program refactoring in the presence of preprocessor directives Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005

[11] Ernst, M.D., Badros, G. J., Notkin, D.: An empirical analysis of C preprocessor use, IEEE Transactions on Software Engineering, vol. 28, no. 12, Dec. 2002, pp. 1146-1170.

Generic Load Regulation Framework for Erlang

Ulf Wiger
Erlang Solutions Ltd
ulf.wiger@erlang-solutions.com

Abstract

Although Telecoms, the domain for which Erlang was conceived, has strong and ubiquitous requirements on overload protection, the Erlang/OTP platform offers no unified approach to addressing the problem. The Erlang community mailing list frequently sports discussions on how to avoid overload situations in individual components and processes, indicating that such an approach would be welcome. As Telecoms migrated from carefully regulated single-service networks towards multimedia services on top of best-effort multi-service packet data backbones, much was learned about providing end-to-end quality of service with a network of loosely coupled components, with only basic means of prioritization and flow control. This paper explores the similarity of such networks with typical Erlang-based message-passing architectures, and argues that a robust way of managing high-load conditions is to regulate at the input edges of the system, and sampling known internal choke points in order to dynamically maintain optimum throughput. A selection of typical overload conditions are discussed, and a new load regulation framework – JOBS – is presented, together with examples of how such overload conditions can be mitigated.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Reliability, availability and serviceability

General Terms Regulation, Performance

Keywords Erlang, Performance, Throughput, Regulation

1. Introduction

The Erlang programming language (Armstrong 2007) is predominately used in server-side applications and various forms of messaging gateways. These systems are often exposed to bursty traffic, and need a strategy for coping with overload conditions. Still, overload remains one of the most prominent causes of service outage (See Figure 1). As Erlang is a highly concurrent, message-passing language, overload conditions have much in common with congestion problems in communication networks and other traffic engineering systems. Indeed, Erlang-based applications are often also part of such communication networks, and must take responsibility for delivering predictable throughput, in order not to become a congestion point and cause even greater problems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09...$10.00

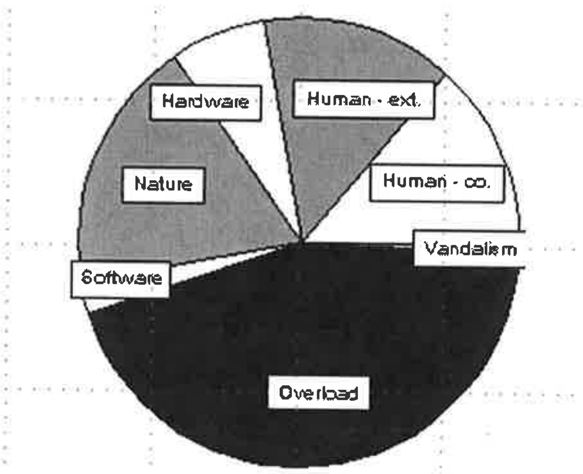


Figure 1. Failures in the US Public Switched Telephony Network, outage minutes by cause, see (Kuhn 1997).

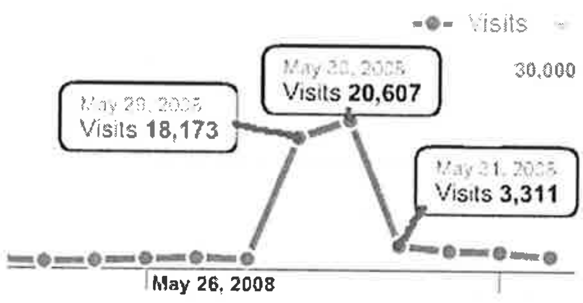


Figure 2. Result of high ranking on Reddit.com, see (Wilson 2008).

While the telecoms domain has changed since Erlang was invented – nowadays, telecommunication is mainly regarded as a specific form of Internet multimedia – Erlang-based applications are still often used in systems serving human communication patterns – instant messaging, Voice over IP, search and document rating networks. As Web users on a global scale can "flock" towards information, online services can be exposed to extremely bursty input loads (See Fig 2).

Some notable Erlang-based products have boasted impressive resistance to overload (see Fig 3 and Fig 4), but not much has been written about how it is done.

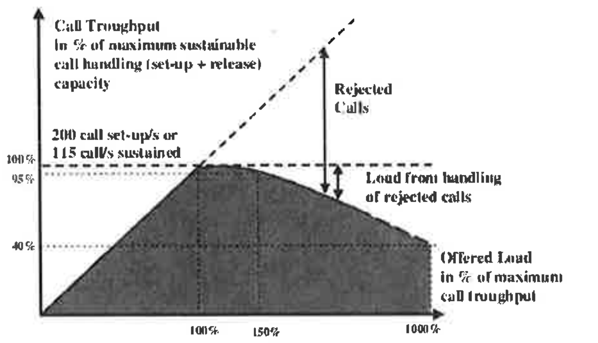


Figure 3. AXD 301, throughput under load, see (Wiger 2001).

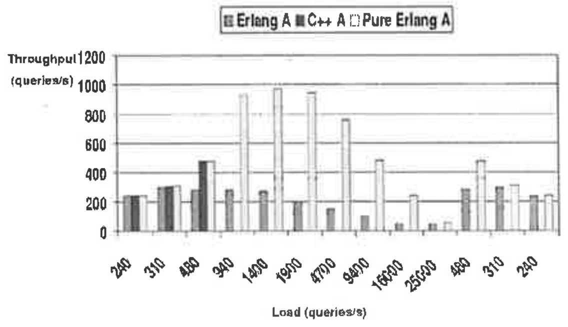


Figure 4. Dispatch Manager, increasing and then decreasing load, C++ vs Erlang; in this test, the C++ application dies after 480 queries/s, see (Nyström et al. 2008).

In the following sections, we will list some typical problems that can arise in Erlang-based applications due to overload, and describe some common mitigation strategies. We will then outline a general approach to load regulation of Erlang programs, and describe a framework designed to offer generic support for load regulation, addressing these problems.

A perhaps unusual approach in this paper is to compare regulation of Erlang-based systems with the problem of achieving Quality of Service (QOS) for multimedia traffic over IP networks. An intuition for this might be that Erlang, being a concurrency-oriented language, supports the building of message-passing systems, somewhat similar to that of a communication network. Erlang was designed for soft real-time, where response times are usually quite good, but no hard guarantees are given. This could also be said for IP networks with DiffServ QOS (see (Peuhkuri 2010)).

Our hypothesis is that lessons can be learned by studying the migration of voice traffic from dedicated circuit-oriented networks to packet-oriented best-effort IP networks, with an intemediate step based on ATM – a packet-oriented technology capable of supporting guaranteed bit rates. ATM lost to IP much due to its higher complexity, cost of interfaces and failure to provide high-speed interfaces for the core network in a timely fashion (see (Gray 2000)).

Our proposed approach is to consign load regulation to the edges of the system, and to be very careful with adding flow-control measures in core components. The reasoning behind this is further expanded in subsection 3.3, Stateless Core.

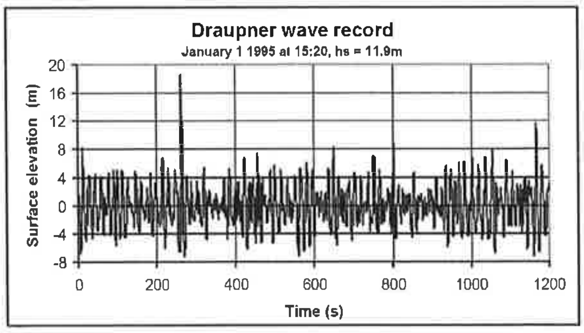


Figure 5. Time history including a possible freak wave event, Draupner platform, North Sea Jan 1 1995, see (Haver 2003).

2. Common Problems

2.1 Active sockets

All communication between Erlang processes and external entities is done through `ports`. Ports behave more or less like normal processes, in that they send a message to the "port owner" process when there is incoming data, and they are instructed to send data by sending them a message in return.

Ports are by default interrupt-driven, but in the case of `inet` ports (sockets), they can operate in different modes:

- `{active,true}` – incoming data on the socket is immediately passed to the port owner.
- `{active,false}` – nothing is sent until the port is instructed to do so.
- `{active,once}` – the port sends one message as soon as there is available data, then reverts to `{active,false}` mode.

In the case of TCP sockets, keeping the port in `{active,true}` mode means that senders will immediately be free to send more data. This can be fatal if the Erlang system is not able to process incoming data fast enough. It is generally considered prudent to keep the sockets in `{active,false}` or `{active,once}` mode. It should also be noted that the low-level POSIX socket API is "passive", in that data must be explicitly read from the buffer.

2.2 Memory Spikes

Garbage-collected languages are known to sometimes exhibit bursty memory allocation behaviour, and Erlang is no exception. Certain job types may be more demanding for the garbage collector, e.g. by building large terms on the process heap, causing repeated garbage collection sweeps that fail to free up data.

Traditionally, this has been a well-known, but reasonably manageable problem. Erlang offers the ability to trace on garbage collection events in real-time, making it possible to quickly identify processes that cause memory allocation bursts.

However, multi-core architectures introduce the possibility of multiple schedulers injecting this sort of behaviour simultaneously, potentially causing lethal memory allocation peaks. We have seen this happen during load testing of otherwise robust message-passing systems. The phenomenon is non-deterministic and can require hours of load test to discover. For lack of a better analysis, we called them "monster waves" (see Figure 5) although we do not yet know exactly what causes it. If the analogy would turn out to be appropriate upon further study, it would be highly interesting to explore ways to predict such patterns before they occur (see e.g. Figure 6).

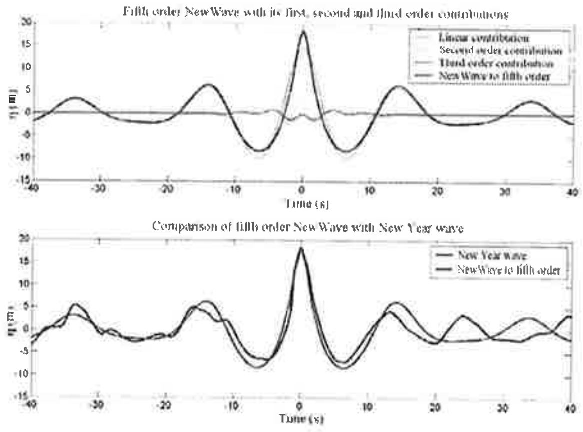


Figure 6. Approximating the Draupner monster wave using a 5^{th} order non-linear equation, see (Taylor et al. 2006).

2.3 Asynchronous Producers

Message passing in Erlang is asynchronous. If a synchronous dialogue is needed, this must be accomplished with a request/reply pair of messages. Reply messages are necessary if the client needs confirmation that the request has been handled, but it also has a flow-control effect.

Some behaviours in Erlang are primarily asynchronous. The OTP behaviour `gen_event` is one example. An event notification is done with an asynchronous message, whereas the processing of the event involves a sequence of (synchronous) calls to user-defined handler functions. The `error_logger` process typically pretty-prints each event and logs it to disk, spending far more effort on each event than the sender does. Therefore, it is quite easy to overwhelm the error logger and potentially killing the whole system. It should be noted that this can happen without any particular wrongdoing on the part of the user, even though it has happened that unsuspecting developers have triggered this problem by relying on the `error_logger` for debugging output

2.4 Unnecessary Serialization

Getting the right granularity of concurrency as well as the right balance between synchronous and asynchronous communication is quite difficult, especially for the novice programmer. One example of a beginner's error is to think that each component must have its own process – typically a `gen_server`. In large organizations, this model can be favoured for organizational reasons – it is easier to perform unit tests if you have your own process. As noted above, it also introduces natural flow control, as the caller necessarily must wait until the work is done.

This style of programming can easily lead to excessive serialization, and the introduction of bottlenecks. We are reminded of Amdahl's argument (Amdahl 1967), stating that "the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude." Simply put, while excessive serialization may give benefits in terms of flow control, it is likely to come at the expense of throughput.

2.5 Excessive Contention

There are very few shared data structures in Erlang. ETS tables are a form of off-heap storage, either hash tables or b-trees, which can optionally be shared among all processes in the system. This means that the ETS tables must be protected internally by mutexes.

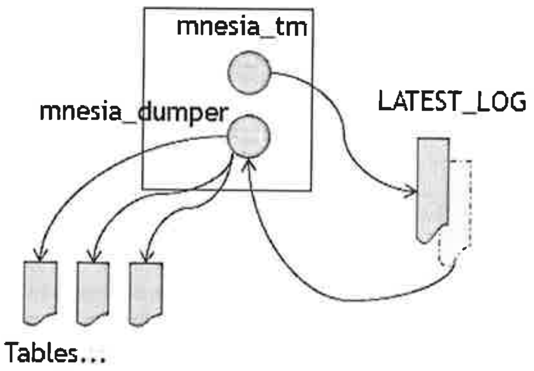


Figure 7. Dumping of Mnesia's transaction log.

ETS tables became ubiquitous early on, when contention was not a problem (there was only one cpu, and one scheduling thread), and putting the data in ETS was often a way to speed up processing compared to using functional data structures on the process heap. But with the introduction of multi-core and multiple schedulers, lock contention has significantly altered this relationship.

Another area where contention can become a problem is `mnesia` transactions. `Mnesia` employs deadlock prevention, which is a reasonably scalable method of avoiding deadlocks in a distributed environment. However, deadlock prevention introduces the risk of false positives (it allows lock dependencies to flow only one way, thus making deadlocks impossible, but also punishing some transactions that were never in danger of deadlocking). The greater the number of concurrent transactions operating on the same data set, the greater the risk of such false positives.

2.6 Large Message Queues

Erlang's support for selective message reception is a great strength, but the implementation – pattern-matching over a single message queue - has rather poor complexity. Behaviours such as `gen_server` normally pick the first message in the queue (a constant-time operation), but in the handling of a message, it is quite possible that the server communicates with other processes and resorts to selective message reception. If the server cannot keep up with incoming requests, each time it performs a selective receive, it must scan all messages in the queue – an operation that becomes more costly the more it falls behind.

In OTP R14B, an optimization enables functions like `gen_server:call()` to run in constant time, regardless of the length of the message queue, but processes that use selective message reception in places where they also receive 'normal' messages, will not be able to benefit from the optimization (see e.g. subsection 2.7).

2.7 Mnesia Overload

Mnesia supports checkpointing to disk of data that is supposed to be persistent. Specifically for data that resides both in RAM and on disk, at transaction commit, Mnesia logs the persistent operations to disk by appending them to a commit log. At periodic intervals, Mnesia reads the commit log (starting a new log for future commits) and distributes the changes into table-specific logs, periodically merging those logs into the actual table image (see Figure 7).

This procedure is quite fast, as it relies entirely on streaming data to and from the disk, but on occasion, the next log dump may be triggered before the previous log dump has finished. When this happens, Mnesia will report that it is overloaded.

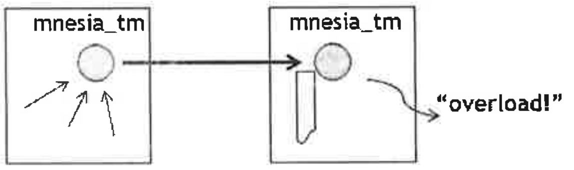


Figure 8. Overload of the mnesia transaction manager.

Another form of overload in Mnesia is when a transaction manager is not able to keep up with updates originating on remote nodes. This can cause the message queue to grow in the transaction manager process, slowing it down and causing it to fall even further behind (see subsection 2.6). Mnesia detects this and reports that it is overloaded, but the 'overloaded' event is only issued on the node where the overload was detected – not on the nodes where the load originated (see Figure 8).

3. Regulation Strategies

Before looking at concrete techniques for mitigating the problems mentioned, we should pause to consider what kind of system an Erlang-based program comprises from a regulation standpoint – if indeed such a distinction can be made. There are several possible strategies for regulating the work of a system.

3.1 Feedback Control

In the textbook "Applied Optimal Control and Estimation" (see (Lewis 1992)), Frank L. Lewis gives the following description: "Feedback control is the basic mechanism by which systems, whether mechanical, electrical, or biological, maintain their equilibrium or homeostasis. [It] may be defined as the use of difference signals, determined by comparing the actual values of system variables to their desired values, as a means of controlling a system. An everyday example of a feedback control system is an automobile speed control, which uses the difference between the actual and the desired speed to vary the fuel flow rate. Since the system output is used to regulate its input, such a device is said to be a *closed-loop* control system."

Feedback Control saw its first applications in ancient Greece, where in 270 B.C. Ktesibios invented a float regulator for a water clock. The regulator kept the water level in a tank at a constant depth, which resulted in a constant flow of water through a tube at the bottom of the tank, filling a second tank at a constant rate. The water level in the second tank could then be used to measure elapsed time.

The development of Feedback Control stalled when Bagdad fell to the Mongols in 1258, but was revitalized during the Industrial Revolution, and has now evolved into a discipline based on mathematics (modern control theory) and engineering. Given that a system can be described in terms of a mathematical model of the input signal and the desired output, a feedback circuit can be constructed. Regulation can then be tuned by combining suitable measures of proportional, derivative (change-oriented) and integral (stabilizing) feedback (see also (Theorem.net 2001)). Special care must be taken to ensure that the feedback loop doesn't in fact make things worse, e.g. by introducing oscillations in the system. A large part of literature on how to do this deals exclusively with linear and deterministic systems.

To establish a secure footing in control theory, we should require deterministic mathematical models of the processes we wish to regulate, but this seems problematic given our context. Not only do such models generally not exist for the problems we wish to attack, but requiring such models would introduce a very high threshold for most Erlang programmers. This view also finds support in (Welsh et al. 2001).

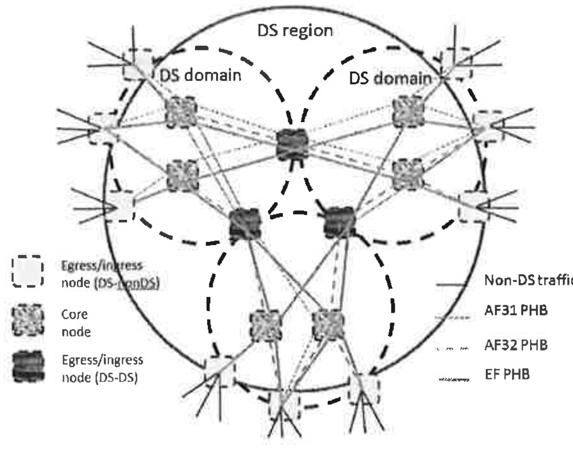


Figure 9. Principle of a DiffServ network, see (Peuhkuri 2010).

However, by constructing a regulation framework which provides the means to measure and control throughput rate and load characteristics, we might well be able to lay a foundation on which we can apply simple "engineering-style"[1] feedback control, and perhaps add more sophisticated regulation techniques later on.

3.2 Frequency Regulation

Depending on the characteristics of the system, it may be desirable to regulate based on frequency. One might want to control that the system outputs work at a given rate, or estimate the frequency of incoming requests.

Output rate regulation is relatively straightforward: the very simplest form would be to start an interval timer, which periodically sends a message to a producer or performs a given task.

Frequency estimation is a bit more difficult. It is common to assume that the incoming traffic follows a random distribution, and can be modeled as a Poisson process (see e.g. (Welsh et al. 2001) and (Wikipedia 2010)). In the case of estimating arrival frequency for the purposes of load regulation, we ought in particular to look at non-homogeneous Poisson processes – where the rate is not expected to be constant over time. There is an OTP library, `overload`, which implicitly estimates request frequency (see 4.2).

3.3 Stateless Core

Along with increasing demand for performance-critical services on top of IP networks, much effort has gone into engineering Quality Of Service guarantees on top of TCP/IP. These efforts have suffered from much the same problems that plagued ATM, which led to the currently dominant trend of using IP routers in the core network with only minimal support for packet classification and prioritization. This model is known as the DiffServ model (see (Peuhkuri 2010) and figure 9), or the stateless-core model, in contrast to the Integrated Services (IntServ) model, which dictates per-flow handling throughout the network. The word 'stateless' in this context refers to the router's knowledge – or lack thereof – of individual packet flows.

One of the arguments favouring a stateless core network is the amplification problem, where errors in the core network have much greater impact than errors in the edge network. This has particularly

[1] as in: relying more on intuition and pragmatic experimentation than a mathematical model

influenced network administrators to favour simple solutions (see (Bell 2003)).

We believe that this dynamic is vital even in software component design. Adding logic to handle congestion issues in core software components leads to increased complexity. Also, as these components tend to have little information about the particulars of each application (due to their generic nature), it may be necessary to support different regulation scenarios (see e.g. 6.1).

To the greatest extent possible, we advocate that generic components should stay neutral to load regulation strategies, and focus on being as simple and generic as possible. We do propose adding diagnostic functions allowing users to sample performance characteristics. This is important not least for debugging, but can also be used e.g. in connection with the JOBS framework.

4. Specific Techniques

4.1 Worker Pool

Worker pools are commonplace not least in programming languages that depend on POSIX threads for concurrency. As creating such a thread is quite a heavy operation, it is usually better to create a thread pool at startup, and then pick the first available thread for a unit of work.

In Erlang, this technique is reasonably common in connection with socket servers (e.g. HTTP servers). A predefined number of acceptors can be created, and indeed, all can call accept() on the same socket simultaneously. An incoming request is routed to one of them, and this worker can either acknowledge immediately to an acceptor pool manager, allowing a new worker to be created, or it can do so once it has finished serving the request. The latter would serve as a form of overload control, whereas the former would primarily increase throughput.

4.2 Request Frequency Estimation

The Erlang/OTP framework has a library module called `overload`, which was developed for the very first commercial Erlang-based product, Mobility Server. It is a server which simply keeps track of the frequency of requests to perform work. It grants requests, as long as the frequency stays below a predefined threshold, and then denies requests above that limit (using a hysteresis function to keep denying requests until the frequency has come well below the threshold again). The server uses a simple exponential formula to calculate the frequency (see (Ericsson 2010b)):

$$i(n) = K + i(n-1)e^{-K(T(n)-T(n-1))}$$

where i is the intensity (frequency), K, or 'kappa', is a constant which regulates how quickly the calculation responds to changes in intensity[2], and $T(n)$ is a timestamp signifying the time of event n.

While simple, the module lacks the ability to discriminate between different types of work. Also, in many cases, it is not sufficient to simply 'deny' a request, so some form of queueing system is typically needed. If we have a queue, the rate of jobs entering the queue is not nearly as interesting as the number of jobs leaving it, and we don't need an exponential distribution to regulate that. Setting a timer corresponding to the period $\frac{1}{f}$, and dispatching the next job(s) when it fires is sufficient[3].

4.3 Credit System

Credit systems are often combined with queueing semantics. The basic principle is that each client is assigned a quota of sorts, and is allowed to do work until the quota is exhausted; at this point, it must wait until it is assigned a new quota. Erlang's reduction-counting scheduler works according to this principle, and the TCP request window is also a form of credit system.

At the Erlang application level, one may implement a form of quota system with a `gen_server` and simple counters, e.g. using `ets:update_counter/3`. A process wanting to start a task, makes a synchronous call to the quota server, which grants permission if there are credits left; otherwise, it saves the request and responds to it only when already running tasks have finished and returned credits. One complication is that one must handle the case where tasks do not finish in an orderly fashion. If each task runs in a dedicated process, credits can be returned when the process terminates.

A clear advantage of the Erlang programming model with lightweight, pre-emptively scheduled processes, is that the process asking for permission can simply wait in a (blocking) synchronous call, assuming that the process is dedicated to one task, or sequence of tasks.

4.4 Smoothing

Subsection 2.2 dealt with memory spikes, especially in multi-core systems.

A queue-based regulation system may have the effect of smoothing out bursts of jobs, reducing the risk of "monster waves". In our own experience, identifying jobs that resulted in large message heaps (using `erlang:system_monitor/2`) and then regulating them with a combination of a queue and a counter-based credit system (see subsections 4.6 and 4.3) can mitigate this problem.

4.5 Back-pressure

Subsection 2.1 dealt with the possibility to keep network sockets in passive mode. From a load regulation standpoint, it is essential to make use of this technique in order to benefit from the transmit window in TCP.

If the clients are cooperative, or able to use some custom protocol, back-pressure can be achieved through a credit system involving the client (see subsection 4.3). However, in many cases, using standard protocols is part of the product requirements, and few of these protocols support any form of credit or backpressure mechanism. See (Wang 2010) for an interesting discussion on the subject.

Even internally to the system, it may be critical to monitor the status of potential bottlenecks and feed back information to the load regulator, allowing it to adjust the throughput rate.

4.6 Job queue

Erlang-based systems are replete with queues: scheduling queues, message queues, I/O buffers, etc. As a rule, the Erlang runtime system handles all queueing of processes and messages, with the exception of process message queues, which are a fundamental concept in the language.

We can see that queues are a common way to achieve fairness and balance among multiple tasks, and it makes sense to use them at the application level as well as for low-level units of work. We can thus model a type of queue that deals with more coarse-grained tasks. We call this type of queue a "job queue". At the moment, we do not care about how this queue is implemented, only how it is used.

In short, we classify incoming requests as the beginning of a specific type of work (a "job"), and enter it into the appropriate job queue. At some point, we extract the job from the queue and either accept it or reject it. Depending on job type, the queue may be configured with a timeout, at which point we must reject the job, as it may not make sense to continue. An example of this is where protocol timers define a maximum wait time.

[2] The time it takes for a change in intensity is approximately $\frac{1}{K}$.

[3] We would have to adjust for timer drift due to scheduling delays and the time it takes to perform the actual dispatch.

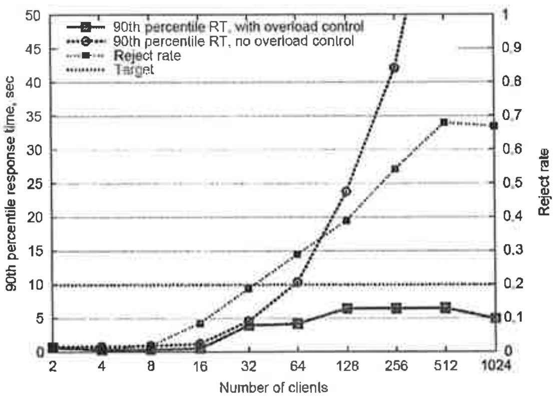


Figure 10. SEDA: Response times with and without overload control, see (Welsh et al. 2001).

For time-limited requests, it may make sense to handle jobs in LIFO order rather than FIFO. During normal operation, it makes no difference in practice, as the queues are expected to be short, and all job requests handled within specified time limits. However, during bursts of incoming traffic, the last entries may be those most likely to complete, as they have used up less of the alloted time waiting in line. Given that the position in line is stochastic anyway, there is not much use in being "fair" to those jobs that happened to arrive at the head of the burst; they are no more "deserving" of being served than the last entries.

Why bother at all about the order, then? Depending on queue implementation, there may be a significant cost difference between the two alternatives. If the queue is implemented as a list, LIFO-style enqueue/dequeue are both O(1), whereas with e.g. the (FIFO) `queue` module in OTP, enqueue is still O(1), but dequeue is amortized O(N). Presumably, lower queueing overhead contributes to increased capacity in the number of jobs that can be handled, which would seem to be the most fair overall.

A LIFO list-based queue also has the advantage that if there is a point where remaining jobs can be discarded (e.g. because the client has already given up waiting), this can be done by simply discarding the tail of the list. On the other hand, the queue will be subject to garbage collection sweeps, which incur a cost proportional to the amount of live data for each sweep.

A queue can also be implemented as an `ordered_set` ETS table, sorted on timestamp, in which case FIFO and LIFO have the same complexity – O(logN). This sort of queue has worse constant factors, but predictable behaviour overall.

In both cases, one should queue only a reference to the requested work, not the full set of input parameters, in order to avoid unnecessary copying.

4.7 The SEDA framework

The Staged Event-Driven Architecture (SEDA), described in e.g. (Welsh et al. 2001), (Welsh and Culler 2001) and (Welsh et al. 2000) is a comprehensive, and highly interesting example of the use of job queues to achieve high throughput and load tolerance in Internet services (see Figure 10).

We believe that the findings from the SEDA framework corroborate our own, and it is perhaps worth noting that we have reached very similar conclusions, while we have approached the problem from different angles.

An important difference between SEDA and the JOBS framework is that SEDA was designed with 'conventional' languages (such as Java and C++) and concurrency models in mind. The stated premise was that there are two generally accepted models for implementing concurrency in high-availability Internet service frameworks: POSIX threads and event-driven programming (or a combination of the two).

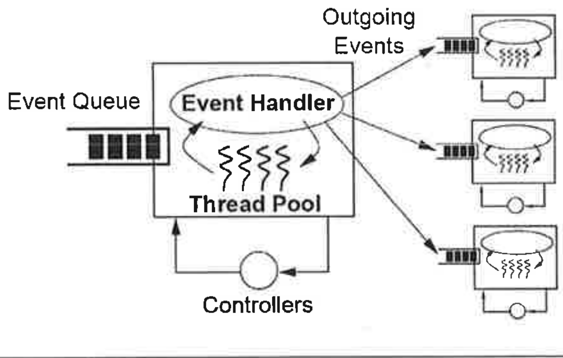


Figure 11. A SEDA Stage, see (Welsh et al. 2001).

"The key idea behind our framework is to use event-driven programming for high throughput, but leverage threads (in limited quantities) for parallelism and ease of programming. In addition, our framework addresses the other requirements for these applications: high availability and maintenance of high throughput under load. The former is achieved by introducing fault boundaries between application components; the latter by conditioning the load placed on system resources." (see (Welsh et al. 2000), pg 2).

While this is a perfectly acceptable premise. Erlang offers significantly different characteristics. The main criticism that POSIX threads are resource hungry and have scalability problems, and that event-based programming can get complex, are not as relevant in an Erlang context, as Erlang's processes handling is quite lightweight and scalable.

In a sense, a significant part of SEDA attempts to provide benefits that are already available to the Erlang programmer. Our assumption is that the load regulation framework should not try to enforce a concurrency model on the programmer, as it would not likely be an improvement over the existing concurrency model in Erlang. For this reason, the JOBS framework is quite similar to a subset of SEDA – the actual request queueing and dispatching component, called a SEDA Stage (see Figure 11). Of course, with a thread pool of size 1, the SEDA Stage also looks quite similar to an Erlang process.

We suggest that the SEDA framework can be a great source of inspiration on how to evolve the JOBS framework, but we imagine a much more coarse-grained division into load-regulated blocks – primarily doing admission control at the input edges of the system, and otherwise relying on Erlang's concurrency model and design patterns to enable high throughput.

5. Load Sampling

The following subsections will give examples of common sampling points, for determining overload conditions. Which methods to choose will vary depending on the characteristics of the system as well as the Erlang/OTP version used.

5.1 Run queue

In a single-core Erlang VM, the run queue is a fairly reliable indicator of high load. The run queue is normally zero, or close to zero in a system that is not overloaded. Sampling the run queue is easy, and a very cheap operation (`erlang:statistics(run_queue)`).

On a multi-core system, this indicator is likely less practical, as sampling the total run queue length involves acquiring a global lock on all scheduler run queues. It is possible that future implementations will loosen the atomicity requirement, making the function more scalable.

5.2 Memory

There are a number of memory-related indicators available in the Erlang runtime system. It is important to understand that the sampled values are rough estimates, and do not reflect things like memory fragmentation, etc.

Depending on the nature of the application, it might be best to use either total memory, the size of the shared binary heap, or the total size of the process heaps as a main indicator. Profiling of the system is needed to determine the most suitable choice.

5.3 ETS Tables

The total amount of data in ETS tables can sometimes be a reliable indicator of the "load" of a system. The total amount of data can either be sampled using `erlang:memory(ets)` or by summing the results of `ets:info(Table, memory)`. The former is more accurate, according to the OTP team.

It might sometimes be the case that a single table, or a subset of all tables, best reflect the system load. Again, profiling is needed to determine this.

5.4 Number of Processes

In principle, Erlang can handle a huge number of concurrent processes – up to 268 million (see (Ericsson 2010a)). However, due to memory constraints (pre-sizing the process table), the configurable hard limit is usually set much lower than that, and the default process count limit is currently 32768. This means that it is quite possible for a system to run out of processes. In some systems, the number of concurrently executing processes might also be a good measure of the load.

In the old days, one could get the current number of processes by calling `length(processes())`, which has O(N) complexity. The function `erlang:system_info(process_count)` yields the same result, but in a much more efficient way. The function `erlang:system_info(process_limit)` might also be useful.

5.5 Message Queue Size

Subsection 2.6 dealt with the problems of large message queues. It is possible to sample the message queue length through `process_info(Pid, message_queue_len)`. Rather than sampling all message queues in the system, one might want to single out a few strategic processes. In most cases, the BIF `erlang:system_monitor(Pid, [{large_heap, Sz}])` should be able to detect large message queues indirectly, as the message queue is part of the process heap. However, if a significant portion of the data in the queue are binaries (which are stored on a separate binary heap), such queues may go unnoticed. The problem with large message queues depends on the number of messages, not the amount of data.

5.6 Response Times

Increasing response times is often a fairly good indicator of overload problems. One of the untold stories from the AXD 301 days is that it was made to sample the response times of neighbouring nodes, since some of them (of a competing brand) had a tendency to die when exposed to more than 150% load, e.g. while re-establishing connections after a link failure. The AXD 301 was capable of slowing down, queueing up traffic, so that other nodes in the network wouldn't topple over from overload.

In this case, sampling response times had a smoothing effect (see subsection 4.4), but it would also be possible to detect local overload by measuring one's own response times. This could be done by measuring the difference between input and output rates, or by simply timing some strategic `gen_server:call()` operations in the system.

6. Rules of Thumb

6.1 Regulate at the Input Edges

In section 3.3, we described the current trend in IP networking to rely on minimal classification in the core network, and relegate regulation to the edges of the network.

Our experience from regulating Erlang applications is similar. It is costly and awkward to implement overload controls in each core component. It is generally better to make them as fast as possible, and insert minimal logic to allow them to indicate if they are getting overloaded. One could imagine complex components like mnesia using a load regulation framework for tasks like transaction log dumps.

In fact, this could also serve to illustrate our point: there is an option, `dump_log_load_regulation true | false`, which makes it possible to have mnesia run the transaction log dumps at lower priority. This can make sense in e.g. a telecoms system, if mnesia is used primarily for configuration data, and performance-critical processes do not write to persistent tables at all. In a system where writes of persistent data are frequent, it makes no sense to perform this type of regulation, as the log dumps must rather be as fast as possible in order to keep up with the traffic load.

6.2 Regulate Only Once

An additional problem with regulating in generic components is that it becomes difficult to know if a job has already been regulated. If classification is done at the input edge, where a unit of work enters the system, it is possible to classify and regulate only once, thereby reducing overhead, and allowing for more predictable behaviour.

However, if a job crosses node boundaries inside the system, it is wise to regulate also on the receiving node. One should bear in mind that the job has now been accepted by the system, and work has already been invested in it. Therefore, it should receive higher priority (e.g. higher throughput) than new job requests entering the system on that node.

With a queue-based regulation framework, classifying the request as an already accepted request would involve putting it in a separate queue.

There may be other cases where it makes sense to regulate multiple times, and we can consider that the SEDA approach is to view multiple regulation steps as building blocks to be combined (see (Welsh et al. 2001)).

6.3 Regulate All Types of Work

It is easy to make the mistake of regulating only the most important jobs. However, ensuring that only the important jobs submit to load regulation and queueing, while other jobs are allowed to run unrestricted, essentially introduces priority inversion.

Therefore, all significant jobs should submit to load regulation. By significant, we mean that the cost of performing load regulation on the job is insignificant compared to the cost of performing the actual work.

6.4 Rejectable vs Non-Rejectable Jobs

A very common type of activity for Erlang-based applications is session establishment (phone calls, logging in to chat groups, etc.) We may note immediately that a request to establish a session may be rejected, whereas it makes little sense to reject a request to end a session. Even in the case where the dialogue times out and the user gives up waiting, we should remember the request to terminate the session, thereby freeing up resources in the system.

In other cases, regulatory requirements may forbid us to reject even session establishment requests. For example, it is not acceptable for a telephone switch to reject emergency calls. They should be serviced even under extreme overload.

In other words, we must be able to distinguish between rejectable and non-rejectable jobs and treat them accordingly.

6.5 Reject Window

There are three cases to consider once we have decided not to service a request:

1. The client does not expect a reply
2. We have time to send a reject message to the client
3. The client has already given up waiting for a reply

It is considered good form not to send reject messages after the expiration of the protocol window during which the client expects to receive such a message. If we hold requests in a queue until they can be served, we should be able to detect when jobs are too old to even be rejected, and simply discard them all.

6.6 Priorities/Weights

Erlang has some support for process priorities, but it is generally believed in the Erlang community that priorities should be used sparingly or not at all.

In our approach, we choose to prioritize work units rather than processes. It is possible in this model to assign different weights to different jobs, and this may be advantageous for a few reasons:

- Some jobs may be defined as more important than others (e.g. emergency calls vs normal calls)
- We may already have invested processing time in a job. This could happen if part of the job is handled on one node and the rest on another. The job should be given greater priority on the other node, compared to jobs we have not yet committed to.

6.7 Do not optimize for fair weather

It is quite easy to conclude that load regulation must be as cheap as possible, and start designing schemes that perform very well under normal conditions. An example of such a scheme is a credit system, where a pool of credits is periodically refilled, and jobs are allowed to execute immediately as long as there are credits in the pool.

Such a scheme may have excellent properties for other reasons, but it is important to keep in mind that it usually serves no purpose to save CPU cycles when there is plenty of CPU capacity to spare. As justification for this, consider that the nature of Internet service systems, or telecoms systems, is to serve the offered traffic, and nothing else. Any surplus capacity remains unused, and should merely be viewed as spare capacity in preparation for traffic peaks. An application running on a multi-purpose server, on the other hand, cannot easily make this assumption.

Load regulation needs to be the most efficient when there is overload, and resources are scarce. Queueing models tend to display this behaviour; they seem unnecessarily expensive when there is no overload (and therefore no need to regulate), but become increasingly effective when throughput is high, because of caching and batching effects.

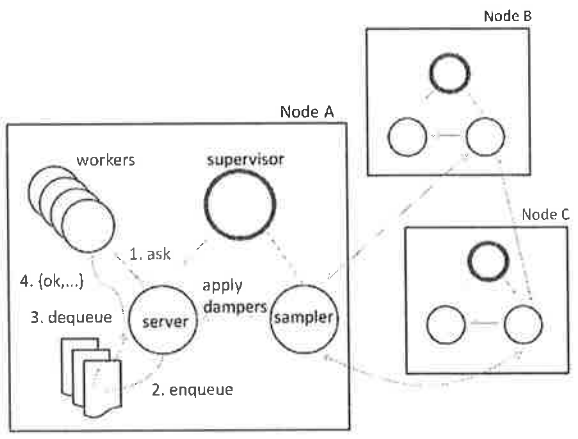


Figure 12. Architecture of the JOBS framework.

7. The JOBS Framework

7.1 Introduction

The `JOBS` framework brings together experiences from several different load regulation techniques, and tries to facilitate most of the techniques above.

7.2 Architecture

Figure 12 illustrates the process model of the JOBS framework.

The job server selects a queue based on job type, and fetches the configured set of regulators for that queue. There are a number of different regulator types:

- `rate` – given a frequency *f*, ensures that the rate of accepted jobs does not exceed *f*.
- `counter` – ensures that the number of simultaneously executing jobs of the given type does not exceed the defined value.
- `group_rate` – the total rate of jobs accepted from all queues with the same group rate can not exceed the given frequency.

7.2.1 Counter-based regulation

Counter-based regulation provides a form of credit system. It can be used to reduce contention e.g. when running mnesia transactions, and has been found to actually improve throughput that way.

The counter system is based on gproc's counters and aggregated counters (see (Wiger 2007)), mainly to avoid reinventing the wheel. Each worker is assigned a counter with some given value. We make use of gproc's ability to handle complex aliases, and name the counter after the associated regulator, with a JOBS-specific prefix. Counters in gproc are "shared properties", so each process can have its own instance, which is exactly what we need. When the regulator is instantiated, an aggregated counter is also created. As new jobs are dispatched, the aggregated counter is automatically updated with the corresponding increment. This way, the aggregated counter provides the total value for regulation. If the worker process dies, gproc detects this and removes the counter, automatically adjusting the aggregated counter as well.

It would be possible for the dispatched process to access the counters, e.g. to reduce them incrementally for long-running jobs. No API for this has been implemented, but it could be done through the normal gproc counter API, and any changes would immediately be reflected in the aggregated counter.

7.2.2 Rate-based regulation

Rate-based regulation is done by remembering the time of the latest dispatch for each regulator. The number of jobs to dispatch at the time of a check is $\lfloor \frac{T_c - T_l}{I} \rfloor$, where T_c is the current time, T_l is the time of the last dispatch, and I is the pre-calculated dispatch interval corresponding to the configured maximum rate.

After each dispatch, if the queue is non-empty, a timer is set to the time remaining until the next dispatch, or 0, if the next dispatch is already overdue. If the queue is empty, it will not be checked again, until a new request arrives.

In other words, the rate regulator will always attempt to dispatch jobs at the maximum allowed rate. The dispatch rate will of course never be higher than the arrival rate.

7.2.3 Group rates

It is possible to group regulators, by specifying the `group_rate` option. When a regulator belongs to a group, the rate parameters of the group are updated each time a regulator is used. The least value from comparing the group regulator and the specific regulator is picked. This can be used to specify a maximum total rate of a group of requests, while allowing for greater peaks in the individual request types.

7.2.4 Feedback modifiers

Distributed feedback-based regulation is accomplished by letting the sampler processes exchange status information. The job server process receives an instruction to apply "modifiers", each indicating a specific sampler type and giving a type-specific "degree" of overload (a degree of 0 means no overload).

Each regulator in each queue then determines individually how to respond to each damper, e.g. by reducing the nominal rate by some multiple of the degree. The simplest configuration is to provide a `{Modifier,Factor}` tuple, in which case the `Modifier` value is multiplied by `Factor` to produce a reduction value in percent of the defined maximum rate.

A more advanced case would be to name a function which, when called with a modifier value returns a reduction value. Different sampler callbacks (see section 7.8) could export such modifier functions for convenience and as examples for the user.

7.3 Regulation API

The basic API for submitting to load regulation is:

```
jobs:ask(JobType) ->
    {ok, Opaque} | {error, Reason}
  Reason = rejected | timeout
```

If there is no registered job type matching `JobType`, an exception is raised.

The recommended way to end the job is to let the process terminate, i.e. create a temporary process for each job. If the process needs to be kept in order to perform more work, it must explicitly tell the job regulator when it has completed the job:

```
jobs:done(Opaque)
```

It is also possible to do this as a one-liner:

```
jobs:run(JobType, Fun) -> Result
```

Result is the result of calling Fun(), when the job request has been accepted. If the job request is denied with `{error,Reason}`, `erlang:error(Reason)` is raised.

7.4 Management API

It is possible to dynamically add, remove and reconfigure queues. This is partly intended for continuous maintenance and evolution of a system, but could also be used for dynamic regulation, e.g. changing the regulation parameters based on policy control, sampling feedback or operator intervention.

One could also imagine implementing a "training" function, where the feedback from the sampler is used to find an optimal throughput level. No experiments have been done with this yet.

7.5 Inspection API

A vital part of the run-time management function is to be able to read the current configuration. Thus, all configuration values are available at run-time. Furthermore the queue and regulator plugins must support an `info/2` function for real-time inspection of key performance indicators (e.g. current queue length). It is possible to provide information on custom attributes; the default return value for unsupported attributes is `'undefined''`.

7.6 Regulator Plugins

It will be possible to define a custom regulator plugin, although at the time of writing, the API has not yet been defined. Most likely, it will have a very narrow regulation interface: one function which is called at each check interval, and a facility for outside control.

The suitable time to define this API is when there is a good case for a new regulator type. Suggestions are most welcome.

7.7 Queue Plugins

The default queue implementation is an ordered-set ETS table, maintaining a FIFO queue of `{Timestamp, JobRef}` tuples. It is possible to define a different type of queue through a callback interface. The API looks roughly as follows:

```
-type timestamp() :: integer().
-type job()       :: {pid(), reference()}.
-type entry()     :: {timestamp(), job()}.

%%
%% Create a new instance of a queue; return a
%% #q{} record.
%%
-spec new(options(), #q{}) -> #q{}.

%%
%% The queue is being deleted. Delete any files,
%% ets tables, etc. that belong to the queue.
-spec delete(#q{}) -> true.

%%
%% Enqueue a job reference; return the updated queue
%%
-spec in(timestamp(), job(), #q{}) -> #q{}.

%%
%% Dequeue a batch of N jobs; return the modified queue.
%%
-spec out(N :: integer(), #q{}) -> {[entry()], #q{}}.

%%
%% Return all the job entries in the queue
%%
-spec all(#q{}) -> [entry()].

%%
%% Return information about the queue.
%%
-spec info(atom(), #q{}) -> any().
```

```
%%
%% Return all entries that have been in the queue
%% longer than Q#q.max_time.
%%
-spec timedout(#q{}) -> [entry()].

%%
%% Check whether the queue is empty.
%%
-spec is_empty(#q{}) -> boolean().
```

7.8 Sampler Plugins

As the indicators for determining overload can vary significantly between systems, the sampler behaviour provides a plugin API. The callback API is designed to be as simple as possible:

```
%% Initialize the plugin;
%% called at startup or when plugin is added
init(Argument) ->
   {ok, InternalState}.

%% A message (e.g. a mnesia event) has arrived
%% to the sampler
handle_msg(Msg, Time, State) ->
   {log, Value, NewState} | {ignore, NewState}.

%% A sample interval has triggered.
%% Sample and return the result.
sample(Time, State) ->
   {Value, NewState}.

%% Calculate an overload factor, based on the history
%% of samples and possible indicators based on incoming
%% messages.
calc(History, State) ->
   {Factor, NewState}.
```

The history is simply a list of `{Time, Value}` tuples. A default function is provided for assessing the contents of the list:

```
calc(Type, Template, History) -> Factor :: term().
  Type     = time | value
  Template = [{Threshold, Factor}]
```

For example, a cpu load sampler plugin might provide a template like `[{80, 1}, {90, 2}, {100, 3}]`, meaning that the `Factor` is set to 1 at 80% overload, 2 at 90%, etc.

When e.g. sampling mnesia, the only information we can extract is whether it is overloaded or not – not the degree of load. It is then better to check the duration of the overload condition, e.g. with a template like `[{0,1}, {30,2}, {45,3}, {60,4}]`. If there is currently no overload, we do not even check the template. If there is overload (`Value == true`), we check how long it has been true, and find the lowest corresponding threshold in the template. In this case, if overload has persisted for 35 seconds, `Factor = 2`.

These values need to be tuned for the system in question.

(Some administrative functions, such as terminate() and code_change() will probably be added as well, in line with standard OTP behaviours.)

7.9 Evaluation

It is important to note that, at the time of writing, JOBS has not yet been used in commercial operation. Promising results from prototypes have earned it a place in products currently under development.

In the prototype tests, we have used JOBS to regulate a system that uses a distributed, persistent mnesia database. The system tends to be disk-bound, and without load regulation, it risks triggering mnesia overload by outrunning the transaction log dumper (see section 2.7). If the load persists, eventually the mnesia transaction manager (`mnesia_tm`) will also fall behind, building up a very large message queue. As each new application-level request is handled in a separate process, eventually enough processes will be backed up waiting for mnesia that the node crashes, either running out of memory, or running out of ETS tables (each mnesia transaction creates an ETS table for the temporary transaction store).

We regulated the system by adding a counter-based queue and empirically adjusting the number of allowed concurrent requests. Just by doing this, we increased the request rate within which we were able to meet the response time requirements. Our assumption is that reduced low-level contention is the main reason for this (see section 2.5).

At some point, however, we still observed mnesia overload and subsequent eventual node crashes. We addressed this by adding a mnesia load sampler, and configuring the request queue to reduce the number of concurrent requests while mnesia overload was observed. This improved the situation, but as we were testing on different hardware (and virtual instances) with radically different disk throughput, we observed that the optimal throughput level and necessary reduction seemed to differ between the systems. We then extended JOBS to allow the samplers to detect *persisting* overload, and correspondingly increase the factor by which the regulators should reduce throughput in order to cope (see section 7.8).

A unit test exists that exercises a number of different scenarios. An interesting finding was that when running a sequential loop, calling an empty function via the JOBS framework (in essence measuring the overhead of the framework), the maximum sustained rate on a budget dual-core laptop was 500 requests/s. Using parallel evaluation instead, starting all requests asynchronously, a batch of 500 parallel jobs finished in 110 ms if the maximum rate was set to 5000 requests/s. This seems to be the highest reachable rate given the type of hardware, giving an amortized per-request overhead of the JOBS framework of less than 200 μs, before optimizations. We believe this to be acceptable for many types of request traffic.

The JOBS framework also incorporates the experience from load-testing a highly scalable instant-messaging system, where we encountered the "monster waves" alluded to in sections 2.2 and 4.4. The cure in that case was to identify jobs that caused large process heaps (using the `system_info()` BIF), and regulating them with - using JOBS terminology - a counter-based queue, limiting the number of such jobs that could run concurrently.

8. Conclusion

We have designed a generic load regulation framework based on experience from several Erlang-based products designed to withstand significant levels of overload. The basic principle is one of classifying and queueing jobs at the input edges of the system, in a manner similar to that used by the DiffServ mechanism for achieving quality of service in IP networks.

The idea of performing admission control at the input edges of the system is mainly inspired by the AXD 301 system and its derivatives, but it could just as well have been borrowed from the SEDA framework. JOBS is a more general implementation than the AXD 301-based load regulation, with a runtime management API and a plugin approach to queue management and feedback control.

We have found this to be a strategy that suits Erlang very well. We have tested the framework in a prototype for a commercial system, and found that the distributed feedback mechanism works. Thus, it is possible to configure the framework to detect choke points and reduce the accepted traffic, as well as reduce it even more if the overload condition persists.

Possible future developments are development of more sampler plugins, a "training facility" which automatically finds an appropriate throughput level, and possibly also a parameterized queue type capable of performing load-balancing. Another thought is that with a queue configured to invoke a certain function, we could have "producer queues", and essentially an adaptive and scriptable load generation tool.

The code for JOBS has been released as Open Source (ESL 2010).

Acknowledgments

I wish to thank the anonymous reviewer who brought the work on SEDA (Welsh et al. 2001) to my attention.

References

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM. doi: http://doi.acm.org/10.1145/1465482.1465560.

Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, July 2007.

Gregory Bell. Failure to thrive: Qos and the culture of operational networking. In *RIPQoS '03: Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS*, pages 115–120, New York, NY, USA, 2003. ACM. ISBN 1-58113-748-6. doi: http://doi.acm.org/10.1145/944592.944595.

Ericsson. Erlang/otp efficiency guide. http://erlang.org/doc/efficiency_guide/users_guide.html, June 2010a.

Ericsson. Erlang/otp overload reference manual. http://erlang.org/doc/man/overload.html, June 2010b.

ESL. Jobs github repository. http://github.com/esl/jobs, September 2010.

Terry Gray. Why not atm? http://staff.washington.edu/gray/papers/whynotatm.html, November 2000.

Sverre Haver. Freak wave event at draupner jacket januar 1 1995. http://folk.uio.no/karstent/seminarV05/Haver2004.pdf, May 2003.

D. Richard Kuhn. Sources of failure in the public switched telephone network. *Computer*, 30(4):31–36, 1997. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.585151.

Frank L. Lewis. *Applied Optimal Control and Estimation.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1992. ISBN 013040361X.

J. H. Nyström, P. W. Trinder, and D. J. King. High-level distribution for the rapid production of robust telecoms software: Comparing c++ and erlang, 2008.

Markus Peuhkuri. Ip quality of service. http://www.netlab.tkk.fi/˜puhuri/htyo/Tik-110.551/iwork.ps, May 2010.

Paul H. Taylor, Thomas A.A. Adcock, Alistair G.L. Borthwick, Daniel A.G. Walker, and Yao Yao. The nature of the draupner giant wave of 1st january 1995 and the associated sea-state, and how to estimate directional spreading from an eulerian surface elevation time history. In *9th International Workshop on Wave Hindcasting and Forecasting*, 2006.

Theorem.net. On-line introductions to control theory and engineering. http://www.theorem.net/theorem/background.html, 2001.

Yaogong Wang. Bassoon: Backpressure-based sip overload control. http://research.csc.ncsu.edu/netsrv/?q=bassoon, April 2010.

Matt Welsh and David Culler. Virtualization considered harmful: Os design directions for well-conditioned services. *Hot Topics in Operating Systems, Workshop on*, 0:0139, 2001. doi: http://doi.ieeecomputersociety.org/10.1109/HOTOS.2001.990074.

Matt Welsh, Steven D. Gribble, Eric A. Brewer, and David Culler. A design framework for highly concurrent systems, 2000.

Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/502059.502057.

Ulf Wiger. Four-fold increase in productivity and quality - industrial-strength functional programming in telecom-class products, 2001.

Ulf Wiger. Extended process registry for Erlang. In *ERLANG '07: Proc. of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 1–10, New York, NY, USA, 2007. ACM.

Wikipedia. Poisson process. http://en.wikipedia.org/wiki/Poisson_process, July 2010.

Brian Wilson. Reddit case study. http://www.ski-epic.com/2008_reddit_case_study/index.html, May 2008.

Implementing a Multiagent Negotiation Protocol in Erlang

Álvaro Fernández Díaz   Clara Benac Earle   Lars-Åke Fredlund
Grupo Babel, Facultad de Informática, Universidad Politécnica de Madrid
{afernandez,cbenac,lfredlund}@fi.upm.es

Abstract

In this paper we present an implementation in Erlang of a multiagent negotiation protocol. The protocol is an extension of the well-known Contract Net Protocol, where concurrency and fault-tolerance have been addressed. We present some evidence that show Erlang is a very good choice for implementing this kind of protocol by identifying a quite high mapping between protocol specification and Erlang constructs. Moreover, we also elaborate on the added advantage that it can handle a larger number of agents than other implementations, with substantially better performance.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

General Terms Languages, Performance

1. Introduction

Multiagent negotiation is a very active area of research in the Autonomous Agents and Multiagent Systems communities. In a negotiation, each agent modifies its local plans in order to achieve an agreement with other agents in the system. Several negotiation protocols have been proposed, among them, the Contract Net Protocol (CNP) [3] is probably the most popular. CNP is based on the bidding mechanism of a human market. However, as a task assignment or resource allocation mechanism, CNP cannot optimise system performance, especially when scheduling several orders. Also, CNP cannot detect failures in the agents participating in a negotiation process. An extension of the CNP to address these two issues has been proposed in [1].

The concurrent and fault-tolerant aspects of that extension to CNP, make Erlang an ideal implementation candidate. Indeed we have implemented the protocol in Erlang, with very encouraging results so far. In particular, we have been able to work with much larger systems than those reported in [1], i.e., using 3000 agents compared to 40, and with much better performance.

The rest of the paper is organised as follows. In the following section we describe the extended negotiation protocol. The implementation of the protocol in Erlang is discussed in Sect. 3, and the results of some experiments are shown in Sect 4. Finally, the conclusions and future work are summarised in Sect. 5.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09...$10.00

2. The Contract Net Protocol extension

As mentioned in the previous section, the multiagent negotiation protocol presented in [1] is an extension of the Contract Net Protocol (CNP). The goal of the CNP protocol is to enable agents to negotiate the allocation of tasks among them in a fair way, avoiding the possibility of reaching a deadlock state. Such a deadlock state would prevent agents from performing other tasks they are committed to accomplish. The agents involved in the negotiation can be considered as self interested, as they are supposed to provide the best possible bids in order to get the task assigned to them. However, it is not important as the relevant outcome of the process depends only on the overall task allocation and on how efficiently it was performed. The CNP extension includes the addition of new negotiation phases, which: increase the probability for obtaining a more efficient allocation of tasks, permit an agent to handle negotiation of several tasks at the same time thus shortening the overall time required to finish the whole process, and support the detection of failures in agents and a mechanism for negotiation blockage avoidance.

The CNP extension protocol defines two roles for the agents involved in a negotiation process: the *Manager*, which announces the task to be accomplished, and the *Contractors*, which send bids to the manager.

The negotiation process is split into four phases:

1. **PreBidding:** first of all, the manager agent announces the task to all potential contractors. Then, the manager waits for bids from contractors. The contractor whose bid is the highest is promoted and considered as the *potential contractor*. A contractor whose bid is inferior to the highest bid is sent a *preReject* message from the manager. However, a contractor whose first bid failed can make new bids. The **PreBidding** phase ends once a time limit has expired, or all the informed contractors have bid at least once.
2. **PreAssignment:** in this phase, the manager informs the potential contractor that its bid was the highest one by sending a *preAccept* message.
3. **DefinitiveBidding:** once the potential contractor receives a *preAccept* message, it computes and sends a definitive bid, which can be different to its previous bid. If this definitive bid is lower than the bid of any other contractor, the current potential contractor is demoted and receives a *preReject* message. Then the contractor whose bid is now the highest is promoted to *potential contractor* and the negotiation protocol returns to *PreAssignment* phase. On the other hand, if the definitive bid is higher than the bids from all the other contractors, the *potential contractor* is assigned the task and, therefore, a contract is established.

4. **DefinitiveAssignment:** in this final phase, the manager sends a *definitiveAccept* message to the potential contractor and a *definitiveReject* message to all other contractors. After that, the negotiation for the task allocation has finished.

Despite the fact that there are four different phases, managers are the only agents that know the current phase for a negotiation. Hence, it is the only agent role that adapts its behaviour to the one associated to the negotiation phase. The behavior of the different contractors does not change with respect to the negotiation phase, as they just keep trying to get tasks assigned by improving their bids. Note that an agent can be simultaneously involved in several negotiations. This means that every contractor must define an ordering of the tasks it is negotiating.

Finally, in order to provide fault-tolerance to the negotiation process, the protocol defines an algorithm to end a multi-agent negotiation in case of a manager failure. In the first phase of this algorithm, all the contractors involved in the negotiation whose manager has crashed, namely fellow contractors, send each other a *manager_decision* message in which they specify the last answer received from the manager. Once a contractor has received a *manager_decision* from every fellow contractor, it infers its own definitive state for the task execution and forwards it again to all other fellow contractors. The decisions inferred by contractor agents are:

1. If any other contractor sends a *preAccept*, the agent infers a final decision of *preReject*.
2. If any other contractor sends a *definitiveAccept*, the agent infers a final decision of *definitiveReject*.
3. If all the other contractors send a *preReject*, the decision inferred for the task execution is *preAccept*.
4. If all fellow contractors send a *definitiveReject*, the agent infers a *definitiveAccept* final decision.
5. If any other fellow contractor is suspected of failure, and the contractor has received at least one *definitiveReject* message, the decision inferred is *definitiveReject*.
6. If any other fellow contractor is suspected of failure, and no other contractor has sent a *definitiveReject*, the decision inferred is *preReject*.

Note the contractors can only infer decisions by themselves when the manager has crashed. It occurs when this manager agent has not sent a message it is supposed to for a time that exceedes certain threshold, implementation dependent. The specification of this algorithm assumes that all the contractors know about a manager failure and that this failure really happens. Therefore, there is no need of an external agent deciding whether the manager failed or not, in order to start the negotiation termination algorithm.

Now that the main features of the protocol have been described, we will proceed to the details of its implementation in Erlang.

3. Implementing the Contract Net Protocol extension in Erlang

The implementation of a multi-agent environment performing the negotiation mechanism previously described seemed to be conceptually straight-forward. Nevertheless, some decisions had to be taken in order to obtain a fully working implementation of the CNP extension.

3.1 Process Orchestration

The first step was to define the process structure of the system. As the main goal of the implementation is to test the efficiency and performance of the negotiation protocol under Erlang, task execution is abstracted. We assume that each task completes successfully, and within any time limits specified in the contract. Consequently, the behaviour of each agent is relatively simple. Thus, the process structure of the system, depicted in Figure 1, has one Erlang process for each agent, manager or contractor, and a supervisor process which monitors the agents. The supervisor process receives a list containing several sets of tasks to be negotiated, and a set of contractor agents. The supervisor spawns a different manager agent for each set of tasks, and informs it of the contractors that should be involved in the negotiation. Finally, the supervisor is linked to all the managers, and the managers are linked to their associated contractors, to enable the system to be shut down cleanly and efficiently after a finished negotiation.

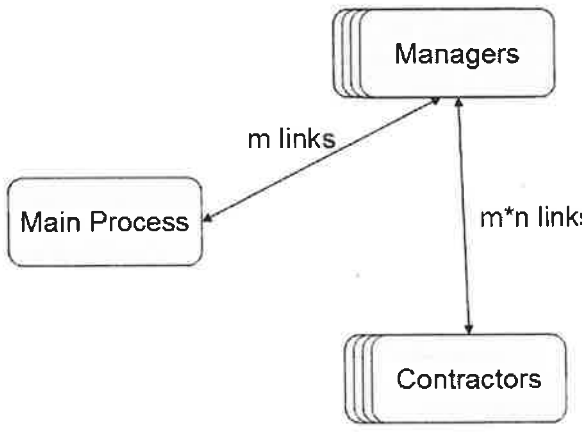


Figure 1. Process links orchestration for a system of *m* managers and *n* contractors

3.2 Information Representation

In order to represent a task, we decided to implement an Erlang record, named *task*, described in Figure 2. An instance of this record is generated by every manager for each task it handles and sent along to all contractors enclosed in an *announce* message during the start of each *preBidding* phase. The information required by a manager includes the name of the task being currently negotiated, the process identifier of the potential contractor, the bid of this potential contractor and the list of bids received from each contractor agent. This is represented by the *manager* record also shown in Figure 2. Analogously, every contractor maintains a record named *contractor* that contains information about the identifier of that agent, the next time it will be ready to execute new tasks according to its own schedule, a list of tasks it has committed to perform, a list of tasks it is currently negotiating and a list of tasks whose manager has crashed, for which the negotiation termination algorithm must be executed.

3.3 Simultaneous Negotiation

To complete the implementation of the protocol, that allows a contractor to be involved in several task negotiations, some decisions had to be taken. One such example was the "currency" used for computing task bids. As the negotiated tasks were not to be executed, but rather abstracted, we decided that a good currency candidate was the time an agent estimates that completing a task requires. Thus a bid b_1 is considered to be higher (or better) than another bid b_2, if $time(b_1) < time(b_2)$, meaning that bid b_1 promises to complete a task before the bid b_2 promises to complete the task.

```
-record(task,{
  id=0,              % task name
  startTime = 0,     % start time for task
  duration = 0,      % execution time for task
  endTime = 0,       % finish time for task
  manager=void,      % manager pid
  maxBid = 0,        % max bid received for task
  lastBid = 0,       % last bid received
  contractors=[],%   list of contractors
  status,            % protocol status
                     % {announced,preAccepted,
                     % preRejected,Accepted}
  info = []}).

-record(contractor,{
  id=0,               % agent identifier
  nextFreeTime=0,     % time last task would finish
                      % (if assigned to this agent)
  tasksExecuted= [],% tasks already committed
  nextExecTime=0,      % first available time slot
                       % after task execution
  offeredTasks=[],     % tasks being negotiated
  blockedTasks=[]      % tasks with failed manager
}).

-record(manager,{
  task=void,   % task being negotiated
  winner=void,% pid of potential contractor
  maxbid=void,% bid of the potential contractor
  bids=[]      % all bids received
}).
```

Figure 2. Erlang records used in the protocol implementation

Next, the behaviour of each agent (manager, contractor) was defined. A manager agent first receives a sequence of tasks to negotiate and a list of process identifiers that correspond to contractor agents. Then, it starts negotiating the first task in the sequence, beginning by sending an *announce* message to all the contractor processes. After that it waits for bids from the contractor agents (in the *preBidding* phase), until either all of them have answered or a timeout expires. At that moment, a *potential contractor* has been identified and the manager process sends this process a *preAccept* message and enters the *definitiveBidding* phase.

The following messages can be received in that state:

- **preBid:** if this bid, from an agent *a*, is higher than the bids from all the other agents, including the one from the potential contractor, the current potential contractor is demoted, by sending it a *preReject* message, and *a* is promoted to potential contractor, consequently receiving a *preAccept* message.
- **definitiveBid:** There are two cases:
  - *This bid comes from current potential contractor: a* if it is better than the bids from all other contractors, the task being negotiated is assigned to *a* and a contract is signed via a *definitiveAccept* message to *a* and a *definitiveReject* message is sent to every other contractor. If there is a contractor *b* whose bid is higher that this definitiveBid, *a* is demoted while *b* is promoted, via *preReject* and *preAccept* messages respectively.
  - *This bid comes from a past potential contractor:* as Erlang does not guarantee message ordering from different processes, it is possible for a *definitiveBid* from a potential contractor to arrive to the manager when this process has been demoted. When this situation happens, *definitiveBid* message is processed like a *preBid* one.

A manager agent similarly negotiates all the tasks it has, and when all tasks have been successfully assigned among the contractors, the manager process finishes.

A contractor agents listens and responds to incoming messages sent by manager agents:

- **announce:** when this message is received, the contractor adds this task to its list of offered tasks. The list of tasks is sorted according to the preference for the contractor to serve a particular task. As previously indicated, as we are primarily interested in evaluating the suitability of Erlang for implementing such a multi-agent system, with regards to performance, we here abstract away from the decision regarding task preference, and simply order the task list in arrival order. Finally, a contractor agent computes the time required to finish the task according to its own schedule, and sends a *preBid* to the manager of the task.
- **preAccept:** the contractor computes again the time required to finish the task, as the calculation may have changed from its first prebid, and sends a *definitiveBid* to the manager.
- **preReject:** if the bid for the task this message refers cannot be improved, the preference from the contractor to serve this this task should be decreased. In order to simulate this, the task is removed from the list of offered tasks and added to its last position. It the bid can be improved, a new *preBid* message is sent to the manager.
- **definitiveAccept:** the contract is signed, and the contractor commits itself to complete the task, so the agent removes it from its list of offered tasks and adds its to its list of executed ones.
- **definitiveReject:** as there is no possibility for the agent to accomplish the task, it is removed from its list of offered tasks, which means that the contractor leaves the negotiation.

As it is a simulated environment, we introduced a new message with the only purpose to let the contractor processes finish when there are no more tasks to negotiate. Then, when a contractor receives a *finish* message from the main process and its list of offered tasks is empty, it terminates. If that list is not empty, it resends the finish message to itself, in order to process it again later.

3.4 Fault Tolerance

In [1], the means for detecting a failure in an agent is based on the absence of an expected communication. Concretely an agent is suspected to be unavailable for the negotiation when no message arrives from it during a certain time interval. This behaviour can be implemented directly in Erlang by letting receive statements time out. However, as Erlang provides a better mechanism for error detection, process linking, we decided to use it instead. Nevertheless, in a large multi-agent systems linking every process to each other is not efficient as this can generate a huge number of termination messages to be generated. Because of that, the processes are linked to each other in a dynamic way that varies with respect to the status of the negotiations. As depicted in Figure 1, every contractor process is linked to the managers of the negotiations they are involved in. Then, as the links are bidirectional, we can treat failures by taking into account the state of the negotiating state, and the type of process that has failed:

- **Contractor failure:** the actions to be taken by the manager depends on the status of the contractor that failed
  - *Potential Contractor:* if a failure is detected in the potential contractor, the manager deletes it and its bid from the list of bids and possible contractors, respectively. Then, it chooses another potential contractor and continues the negotiation of the task.

- *Not potential contractor:* the manager erases the bid this contractor sent, if any, and removes it from the list of possible contractors.

- **Manager failure:** if a manager fails during a negotiation, all the contractors linked to it perform the following steps in order to avoid getting stuck in the negotiation:

  - Every contractor generates a link to all the other contractors whose process identifiers were included in the task announcement, and sends them the last answer received from the manager, labelled *manager_decision*. If none was received yet, it sends *unknown* and infers a definitive rejection for itself in order to avoid the possibility of having several processes inferring acceptance simultaneously.
  - A contractor waits for an answer from all the processes that were successfully linked to it. The rest are assumed to be in a failure state, as it is not possible to create a link to a process already terminated. If all the other processes send a *manager_decision* belonging to set {*preReject*, *definitiveReject* }, the contractor infers a final status of *definitiveAccept* for the task and, thus, it behaves as if a contract had been signed with the manager for the execution of the task. If it receives any *unknown*, *preAccept* or *definitiveAccept* message or if another contractor involved in the stuck avoiding process fails, detected through the existing link to it, the contractor infers a *definitiveReject* for the task, as avoiding simultaneous acceptance inference is more important that not being able to allocate a task, a procedure that could be retried.

3.5 Differences from Specification

During the implementation of the protocol, we realised the specification of the behaviour of the agents is not complete, as it is not able to handle some situations that can happen during the execution of the protocol. The first of them is a *manager_decision* of type *unknown*. As it resembles the absence of an answer from the manager, this allows the possibility of representing either a *preAccept* or *preReject* status. This situation is discussed in [1] although the expected behaviour of a contractor that receives the message is not specified. Therefore, we decided to treat such an occurrence as an inferred decision of type *definitiveReject* in order to avoid more than one contractor inferring an acceptance status. Another required decision was how to deal with failures in a contractor, when it is the potential contractor, as the specified algorithm only elaborates on what to do when a non potential contractor agent fails. The solution was quite obvious but it yet represents a shortcoming of the protocol specification. In addition, concerning the blockage avoidance algorithm, we decided to only allow the inference of *definitiveAccept* and *definitiveReject* negotiation status. The reasons for this decision derive from the fact that an inferred status of *preAccept* would provide the same semantics as a *definitiveAccept* one, while a *preReject* inferred status only allows to degrade a task, which still will remain in the list of offered tasks. Then, as the inference about the negotiation status for a task, is only computed once, the task will remain in the list of offered tasks of the contractors indefinitely. This problem can be solved by inferring a *definitiveReject* status, which removes the task from the list of offered tasks and provides the expected semantics. Finally, as the specification does not indicate what to do when a *contractor_decision* message arrives, and we found no use for it, we simply removed that mechanism from the blockage avoidance mechanism, leading to the savings of a huge amount of messages with no observable semantic changes.

4. Experiments

In the previous sections we have shown how the protocol specification could be easily implemented in Erlang, with no major changes necessary, thus greatly simplifying the design and implementation task. However, our chief motivation for implementing the protocol in Erlang was to assess whether the performance of the protocol implementation would be improved, with regards to the size of multi-agent systems that could be handled.

In order to evaluate this, we conducted a series of experiments, also present in [1], to determine if the re-implementation in Erlang provided any performance benefits. The original implementation was written in Java but the architecture over which the experiments were run is unknown to us. It modelled a transportation application. Nevertheless, it is still comparable to our implementation as the protocol is the same, thus involving the same kind of communication, and the computations are not heavier than the ones our implemented agents perform. Moreover, the original implementation includes a feature, not imposed by the protocol, that can shorten the overall process of task allocation. This feature implies that if a manager agent tries to allocate identical tasks, the bid from a contractor for certain task is also taken into account for its identical ones. For instance, assume a scenario where manager M announces identical tasks t_1 and t_2, and contractor C bids for them with bids b_1 and b_2, respectively. If $b_1 < b_2$ and C is preRejected for task t_1, the manager takes b_1 as bid from C for t_2. In our implementation, agent C has to bid again for t_2 in order to improve its bid, requiring more messages to be exchanged, thus increasing overall process. We decided not to implement that feature as it is implementation dependend and not derived from the procotol specification.

In order to obtain relevant results, every experiment was repeated at least a hundred times. All of them were run under Ubuntu Linux version 10.04 in a computer with two 2.53 GHz processors. Our execution time measurements cover the whole negotiation process: from the moment the different agents start to be created to the time instant the last agent finishes its execution. As explained in Sect. 3.1, task execution is just simulated in our implementation by committing a contract to a contractor which does not allow overlapping of its assigned tasks. In fact, tasks are just defined by a unique identifier and a number representing the time units that task completion requires. The same simulation is performed in the implementation by the protocol designers.

The experiments are divided in two groups. In the first group the experiments are run with the same number of agents as in [1], to compare the two protocol implementations. In the second group of experiments, a larger number of clients has been considered, for the Erlang implementation.

4.1 Small Agent Population

Here we present the results of a series of experiments which are directly comparable to measurements reported in [1], as they do not measure protocol behavior under failure conditions. Therefore, our deletion of *contractor_decision* message has no impact in the comparison of both implementations. We consider the experiments in this series to measure protocol behaviour in rather small agent groups. We present the results in tables due to the big difference between execution times from both implementations.

The first experimental group is composed of 4 agents, 2 contractors and 2 managers. The results obtained for the Erlang implementation, and the original results reported by the protocol designers are:

As we can see, the execution time required for the negotiation in our Erlang implementation is four orders of magnitude better than the timings reported by the designers of the protocol. As the results

Number of Tasks	Aknine et al. Average Execution Time (milliseconds)	Erlang Average Execution Time(milliseconds)
4	1320	0.122
10	2530	0.248
20	3460	0.518
30	5050	0.698
40	9140	0.974
50	14258	1.13

Table 1. 2 contractors and 2 managers

for such a small population are not very significant, we repeated the experiments with the maximum number of agents the protocol designers used to test their implementation. The results obtained for a population of 40 agents (14 managers and 26 contractors) were:

Number of Tasks	Aknine et al. Average Execution Time (milliseconds)	Erlang Average Execution Time(milliseconds)
4	3509	0.798
10	7901	2.027
20	10821	4.232
30	16941	7.292
40	18788	9.905
50	29872	1.2116

Table 2. 26 contractors and 14 managers

The results again indicate that our Erlang implementation runs approximately 1000 times faster. We repeated the experiment for another population of 40 agents, this time with 24 managers and 16 contractors. The results obtained were:

Number of Tasks	Aknine et al. Average Execution Time (milliseconds)	Erlang Average Execution Time(milliseconds)
4	3621	0.579
10	8189	1.374
20	10934	3.293
30	16991	4.743
40	18933	7.057
50	29978	9.686

Table 3. 16 contractors and 24 managers

The results for this scenario are consistent with early ones. Erlang clearly is an efficient platform for implementing multi-agent systems.

4.2 Large Agent Population

To measure the performance of the Erlang implementation of the protocol on larger, perhaps more realistic, multi-agent systems, we decided to repeat the previous experiments with much bigger agent populations. There is no possibility to compare the outcome of these experiments to previous works, as [1], for instance, contains no measurements for populations remotely similar in size to ours. In the following table, we show how the execution time correlates with an increase in the number of manager agents in a negotiation of 1000 tasks among 100 contractor agents:

Manager Agents	Erlang Implementation Average Execution Time(sec)
10	0.914946
20	1.231567
40	1.869366
60	2.515276
80	3.094301
100	3.910985

Table 4. 1000 tasks to 100 contractors

Then, we repeated the experiment for a variable number of contractors, while the number of managers remains constant with a population of 10, as depicted in the table below:

Contractor Agents	Erlang Implementation Average Execution Time(sec)
100	0.853911
200	2.212398
400	6.691916
600	12.915213
800	21.532578
1000	32.297314

Table 5. 100 tasks and 10 managers

Next, we measured the impact on execution time in environments with a constant number of contractors and managers (100 and 10 respectively), and a variable number of tasks:

Number of Tasks	Erlang Implementation Average Execution Time(sec)
100	0.087581
200	0.172625
400	0.353451
600	0.525856
800	0.708959
1000	0.873533
2000	1.75532
5000	4.396946
10000	8.620578

Table 6. 100 contractors and 10 managers

Finally, in order to test the performance of the algorithm in an environment with few tasks to assign, only 10, we run a series of experiments that involved 10 managers and a sharply increasing number of contractors:

4.3 Performance under Failure

In the previous sets of experiments, the execution time measured was taken from executions where there was no failure in the agents involved in the negotiation. To evaluate the performance of the protocol implementation in failure scenarios, we designed a set of experiments where failures were introduced in the agents at certain points of the negotiation. The first experiment set measures execution time when the *potential contractors* fail as soon as they

Contractor Agents	Erlang Implementation Average Execution Time(sec)
100	0.009974
200	0.026022
400	0.80698
600	0.158065
800	0.302604
1000	0.413268
2000	1.531921
3000	3.467723
3500	4.342805

Table 7. 10 tasks and 10 managers

receive a *preAccept* message, except for one agent of the population who will not fail and, thus, will commit all contracts. The results obtained for with 10 tasks, 5 managers and a varying number of contractors were:

Contractor Agents	Erlang Implementation Average Execution Time(sec)
100	0.007226
500	0.070008
1000	0.230252
2000	0.91076
4000	3.7773
5000	5.662974

Table 8. 10 tasks and 5 managers under failure

We can observe how the execution time is significantly lower than earlier measurements, as the number of contractors is constantly decreasing, as the tasks are being assigned, until only one contractor remains alive. Besides, we wanted to test the implementation when there are failures in the managers, as this will increase the number of communications taking place. For instance, in a negotiation between one manager and n contractors, with $n > 1$, the number of messages sent once the announcement of the task has been finished, is very close to 1. However, if there is a failure in the manager, this number grows exponentially, as in the better case there are $2n^n$ messages. The results obtained for the allocation of 5 tasks, where there are 5 managers and 2 of them fail, immediately before sending a *definitiveAccept*, under a variable number of contractors were:

Contractor Agents	Erlang Implementation Average Execution Time(sec)
100	0.069875
500	1.48867
1000	9.858859
2000	67.838057

Table 9. 5 tasks and 5 managers under failure

These results show the aforementioned exponential growth in the number of messages exchanged. Nevertheless, they are still surprising, as the time required for such a significant amount of computations is relatively small, especially if compared to the results provided by the protocol designers. Once again, Erlang has proven itself a very capable platform for this type of process and communication intensive systems.

5. Conclusion and Future Work

In this paper, we have shown the benefits of using Erlang for the implementation of a task allocation protocol in a multi-agent system. One such significant advantage is the fact that mapping the original protocol specification into an Erlang program took very little effort, approximately one man month, as the assumptions made in the protocol matches well with the actor process and communication model used in Erlang.

We have further shown how the information the different agents handle can be represented as Erlang records. Moreover, it is significant how easily an event-driven system, as multi-agent systems usually are, can be implemented using Erlang message passing. Another useful mechanism Erlang provides is process linking which, compared to the protocol specification, provides a high-level failure notification mechanism. The use of this high-level mechanism contributed to enabling the agents to become more "aware of their environment", in terms of their knowledge about the status of other agents, which is a very desirable, if not mandatory, feature in collaborative multi-agent systems.

Regarding the execution of multi-agent systems, [1] states "*We understand it is difficult to increase the number of agents and tasks, because of the computational complexity problem*", referring to experiments that performed a task allocation of 8 tasks which involved a population of 8 agents. In this document, we have shown that by using the Erlang/OTP runtime system, which has a superior implementation of processes management and message passing, it is possibly to dramatically increase the number both of tasks allocated, and the number of agents involved in a negotiation.

Because of these two observations, we strongly believe that Erlang is a very good choice as programming language for the implementation of multi-agent systems, allowing the generation of huge agent populations and, thus, increasing relevance of both empirical experiments and real world applications.

As a future line of work, we intend to formally verify that our protocol implementation behaves correctly, and moreover, that the original protocol specification is consistent. Specifically, we plan to model check our Contract Net Protocol extension implementation using the McErlang [2] model checker. Having implemented the protocol we remain suspicious of a number of features of the protocol, and we are of the opinion that a verification is required to increase the trust, and the quality, of the original protocol specification.

References

[1] S. Aknine, S. Pinson and M.F Shakun. An Extended Multi-Agent Negotiation Protocol. *In Int. Journal of Autonomous Agents and Multi-Agent Systems*. Volume 8, pages 5–45. Kluwer Academic Publishers, 2004.

[2] L. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International conference on functional programming (ICFP 2007)*, Oct. 2007.

[3] R.G. Smith and R. Davis. Frameworks for co-operation in distributed problem solving. *In IEEE Transaction on System, Man and Cybernetics*. Volume 11, number 1, 1981.

QuickChecking Refactoring Tools

Dániel Drienyovszky Dániel Horpácsi
University of Kent and Eötvös Loránd University
{dd210, dh254}@kent.ac.uk

Simon Thompson
University of Kent
S.J.Thompson@kent.ac.uk

Abstract

Refactoring is the transformation of program source code in a way that preserves the behaviour of the program. Many tools exist for automating a number of refactoring steps, but these tools are often poorly tested. We present an automated testing framework based on QuickCheck for testing refactoring tools written for the Erlang programming language.

Categories and Subject Descriptors D. Software [*D.2 SOFTWARE ENGINEERING*]: D.2.5 Testing and Debugging: Testing tools

General Terms Verification

Keywords refactoring, Wrangler, RefactorErl, Erlang, random program generation, QuickCheck, attribute grammar, yecc, property

1. Introduction

Refactoring [10, 15] is a process of rewriting program source code without changing its meaning whilst improving properties such as maintainability, clarity or performance. Refactorings range from simple ones, like renaming a variable, to more complex ones, like generalising a function. Refactoring transformations may affect large parts of the code base for a project and in particular may require modifications of more several different modules from a project. Applying such code-to-code transformations are common practice among software developers, consequently tool support for refactoring is widespread in mainstream programming languages. Refactoring tools/engines help to automate the routine aspects of certain code transformations. For Erlang there are three refactoring tools available: Wrangler [3] from the University of Kent, RefactorErl [2] from Eötvös Loránd University and last but not least, Tidier [6, 18], a completely automatic code cleaning tool from the National Technical University of Athens.

These tools are hard to test, as they require manually written test cases aiming to cover every corner case of the language being refactored. Evidently, by using only such case-based testing we never can provide a comprehensive check of the refactoring engines. To increase confidence in these tools, a more efficient testing approach has to be applied. We investigate automated testing of refactoring tools by generating random programs and verifying whether the refactorings preserve the meaning of these randomly-generated programs. We have chosen QuickCheck as our testing tool and two Erlang refactoring tools, Wrangler and RefactorErl as the tools to be tested.

In this paper we describe the difficulties of the testing of refactoring tools (Section 2) and we also present our contribution, which is to make the entire testing process fully automated. The method is composed of two separate parts.

- First, we have created a QuickCheck-based random generator for producing Erlang programs which are used as the input of the refactoring transformations. More precisely, we have formalised the programming language to be refactored (namely, Erlang) with a proper grammar class and then based on this description we have derived a corresponding QuickCheck generator for the syntactically and semantically correct programs of the language (see Section 3).
- Second, we have created a definition of equivalence between Erlang modules, which leans on the dynamic behaviour of the programs. Furthermore, we have formalised this equivalence relation by means of QuickCheck properties (see Section 4).

Using these two ideas together, we have built a fully automated testing framework for Erlang code-to-code transformation tools. The final sections of the paper comment on the results we have derived, on related work, and our conclusions, where we note that the tool we have built here is applicable to all the refactoring tools for Erlang, and that the approach we outline could equally well be applied to testing refactoring tools for other languages.

2. Validating refactoring tools

There are many ways of establishing program correctness, including formal proof mechanisms as well as several testing strategies. While developing software, evidently, we would like to be sure of our program's correctness. Since formal methods are mostly too difficult to apply, despite some preliminary work reported in [20], we focus on testing as a mechanism for validating and improving the quality of our programs.

2.1 Case-based testing

There are many testing approaches, which aim to check as many program parts as possible, as assessed in different ways. With testing, basically, we are not able to prove the program's correctness, but we can establish that in numerous cases the program does what we expect from it, and this can give us a degree of confidence that it indeed satisfies its requirements.

Commonly, programmers apply simple use-case based testing to check fundamental requirements. However, in the case of complex software it is impossible to cover all the most common and most interesting cases just by test cases written by hand. For instance, in our specific case, there always may be found unusual instances that are valid source code but that would seldom be writ-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09...$10.00

ten by human programmers. The latter sort of code should also be handled correctly by a practical refactoring tool.

Since refactoring tools can mess up or even corrupt our code base by accident, they have to be well-tested. They can only be useful accessories of the development process if they can be expected to perform the transformation steps properly, without making any mistakes. We apply such tools instead of performing transformations by hand because the refactoring software should be much more precise than the human programmer can ever be. An efficient and comprehensive testing method has to be used in order to achieve a reasonable confidence in the reliability of the tool.

Testing of refactoring tools is difficult due to the complexity of their input and output: both the input and the output of such programs are program source code. Such code is a complex data type, since it embodies not only syntactic well-formedness but also semantic correctness too. Furthermore, defining the semantics of the refactorings, namely, how a transformation has to be performed on different input data, is not straightforward task either. By using case-based testing on the transformation steps we can cover the main features of the functionality with a reasonable labour, however, more sophisticated approaches are also applicable, such as property-based testing methods.

2.2 Property-based testing in QuickCheck

Property-based testing makes a generalisation of usual test cases by eliminating the concrete input from the test case and replacing it with randomly generated test data. So then, the test cases describe only the properties (the main points) of the specified case rather than describing a concrete input-output pair. The properties can be efficiently checked on large number of randomly generated test inputs.

Such testing methods may be regarded as a fusion of the formal proof methods and the traditional test case based testing. When using property based testing, we do not define specific input-output pairs that describe the requirements, but we specify in a logical property the expected behaviour on inputs satisfying the specified conditions. The expected behaviour is drawn in terms of specification properties. During the test, these properties are checked in a large number of test cases. Usually, the test input is randomly generated by the framework, based on special data generators. The data generators describe the structure and the essential properties of the input data to be used for testing. Also, the distribution of the random data can also be controlled through the generators.

QuickCheck is a well-designed implementation of the property-based testing method for functional programs, including the Erlang programming language. QuviQ QuickCheck [1, 4, 5], the commercial QuickCheck implementation for Erlang, is a tool for automatic testing of Erlang programs against a user-written specification. The testing method known as 'QuickChecking' means the checking of specification properties (that is the expected functionality) in a large number of randomly generated cases. QuickCheck properties are expressed in standard Erlang code, using the macros and functions defined in the QuickCheck library.

As we mentioned already, property-based testing involves two kinds of activity.

- The first is the description of the testing data used as input for the program being tested. *Data generators* describe the way that the test data is generated, as well as the expected probability distribution of the randomly generated data.
- The other is the *specification of the properties* expected of the program. These are typically universally-quantified properties, and the data produced by the generators are used as the actual values of the universal variables.

Property-based testing can provide comprehensive testing of several kind of software. In this paper we present property-based testing of Erlang refactoring tools, which involves a definition of a data generator for Erlang module source code as well as a property for determining whether two modules, namely a module before and after refactoring, are equivalent.

3. Random program generation

While creating data generators we can use built-in data generators and in addition, QuickCheck allows the programmers to define their own data generators to create more complex random values. Generators for the built-in types are defined by the framework, so we have to create generators only for our own types by combining the built-in generators by using generator combinators.

In this paper, the term 'first-order generator' means simple data generators that are not parameterised with any other generators. On the other hand, the term 'higher-order generator' strands for the so-called generator combinators, which are generators that may take one or more generators as their arguments. First-order generators only take simple Erlang terms as parameters. They are the core of the generation (since they do not combine other generators but indeed create data values). On the other hand, higher-order generators combine other data generators and may result in arbitrarily complex data generators.

Despite the fact that QuickCheck generators provide a powerful toolkit for defining test data, in the case of larger programs operating on complex input, writing generators by hand is tedious and results in complicated source code, containing substantial 'boiler plate', that is hard to maintain. With a more general notation, that is, with a metalanguage more powerful than the QuickCheck generators, we can reduce the complexity of the description and the amount of the wasted coding time. In this approach QuickCheck generators are a low-level formalism and our meta-notation is a high-level formalism that eases the description of the test data.

Attribute grammars

A formal grammar is a set of rules, which describes a formal language [8], for example, the syntax of a programming language. Usually programming language syntax is formalised with EBNF (Extended Backus-Naur Form) [7], which is a meta-syntax notation used to express context-free grammars (CFG). However, also to describe the static semantics of a language – such as the binding structure of variables and other identifiers – a more expressive formalism is needed.

Attribute grammars (AG) [12, 16] are generalisations of context-free grammars, where the grammar rules are extended with semantic computation rules to calculate associated values or *attributes*. With attribute grammars both the syntax and the semantics are representable together. The attributes are divided into two groups: synthesised attributes and inherited attributes. The former are computed from constants and attributes attached to the children, the latter depend only on constants and the parents or siblings.

Synthesised attributes are used to calculate and store results like the value of an expression, whereas inherited attributes are used to carry the context of a node, such as an environment of variable bindings in scope at that point. In some approaches, synthesised attributes are used to pass semantic information up the parse tree, while inherited attributes help pass semantic information down and across it.

There are important subclasses of attribute grammars, which have some restrictions on the form of the attribute computation rules [13]. S-Attributed grammars involve synthesised attributes only, L-Attributed grammars allow attribute inheritance with the restriction that dependencies from a child to the child itself or to the child's right are not allowed. Formally, given a rule $A \rightarrow X_1 X_2 \ldots X_n$ in the L-Attributed grammar, each inherited attribute of X_j ($1 \leq j \leq n$) depends only on attributes of $X_1, X_2, \ldots, X_{j-1}$ and on inherited attributes of A. Furthermore, synthesised attributes of X_j may depend on its own inherited attributes. Synthesised attributes of A depend on inherited attributes of A and on any attributes of the right hand side symbols. This definition of L-attribution effectively means that there are no cycles in the attribute calculation, and that calculation can conclude in a single pass.

Grammars and testing

Test data may be defined by means of formal grammars. This kind of testing is usually called *grammar-based testing*[19]. In this concept, a test datum is a word of the language defining the domain of the tested function and this language can be given by means of formal grammars. We have created a grammar-based meta-notation for QuickCheck data generators and in our experience, data described in our notation usually is about 5 times more compact than writing the same data with standard QuickCheck generators. For example, in the case of a simple language ($a^n b^n c^n$), compiling some 10 lines of description results in about 55 lines of Erlang code containing the QuickCheck generators.

In QuickCheck for Erlang there is already a module with similar capabilities: `eqc_grammar` [1] is a library module of the QuviQ QuickCheck distribution. This tool is able to create QuickCheck generators from a yecc-like[1] grammar description, but in contrast to our work, it does not support attributes, EBNF notations and many other features that are covered in detail in Dániel Horpácsi's master's thesis [11].

The most significant difference from already available generator generators is that in our work the data generators are generated not from context-free grammars, but from L-attributed grammars. The latter formal grammar class is much more expressive than the former one. The notation of the `eqc_grammar` is based on context-free grammars and cannot be used to describe context-dependent data. Since the Erlang language is in the latter group, a more expressive metalanguage is needed. The Erlang syntax and static semantics can be conveniently described by using L-Attributed grammars, so we decided to design a QuickCheck generator generator for such grammars.

3.1 Generator metalanguage

A grammar-based generator generator takes a proper grammar description and produces data generators according to the grammar rules, or in other words, to the grammar nonterminals. Then, the generator belonging to the root symbol generates strings of the language described by the grammar. We have to create a metalanguage that can denote L-attributed grammars and can be efficiently compiled into data generators.

As already noted, the notation aims to express L-Attributed grammars, which are able to describe inheritance in the grammar. For those who are familiar with the usual attribute grammar notation, we present a very simple attribute grammar in both the usual and the new notation to illustrate the difference. The example describes the $a^n b^n c^n$ language, which is one of the simplest non-context-free languages. Figure 1 shows it in usual notation and Figure 2 shows it in our EBNF-like notation.

The main structure of the both descriptions are similar. The grammar is written as a group of rules and inside the rules there may be alternatives. The main difference lies in the place where the attribute computations are written. In the usual notation the semantic rules are separated from the common grammar rules. In

[1] Yecc is an LALR-1 parser generator for Erlang, similar to yacc.

$$
\begin{array}{lll}
\langle\text{abcSeq}\rangle & ::= & \langle\text{aSeq}\rangle\ \langle\text{bSeq}\rangle\ \langle\text{cSeq}\rangle \\
 & & \textit{InSize}(\langle\text{bSeq}\rangle) \leftarrow \textit{Size}(\langle\text{aSeq}\rangle) \\
 & & \textit{InSize}(\langle\text{cSeq}\rangle) \leftarrow \textit{Size}(\langle\text{aSeq}\rangle) \\
\langle\text{aSeq}\rangle & ::= & \mathbf{a} \\
 & & \textit{Size}(\langle\text{aSeq}\rangle) \leftarrow 1 \\
 & | & \langle\text{aSeq}\rangle_2\ \mathbf{a} \\
 & & \textit{Size}(\langle\text{aSeq}\rangle) \leftarrow \textit{Size}(\langle\text{aSeq}\rangle_2) + 1 \\
\langle\text{bSeq}\rangle & ::= & \mathbf{b} \\
 & & \textit{Condition: InSize}(\text{bSeq}) = 1 \\
 & | & \langle\text{bSeq}\rangle_2\ \mathbf{b} \\
 & & \textit{InSize}(\langle\text{bSeq}\rangle_2) \leftarrow \textit{InSize}(\langle\text{bSeq}\rangle) - 1 \\
\langle\text{cSeq}\rangle & ::= & \mathbf{c} \\
 & & \textit{Condition: InSize}(\text{cSeq}) = 1 \\
 & | & \langle\text{cSeq}\rangle_2\ \mathbf{c} \\
 & & \textit{InSize}(\langle\text{cSeq}\rangle_2) \leftarrow \textit{InSize}(\langle\text{cSeq}\rangle) - 1
\end{array}
$$

Figure 1. $a^n b^n c^n$ grammar in usual AG notation

```
1   abc_seq -> a_seq
2           ~> b_seq [@size = '$1'.size]
3              c_seq [@size = '$1'.size].
4
5   %% a_seq -> a           :: [@size = 1]
6   %%          | a_seq a :: [@size = '$1'.size + 1];
7   a_seq -> {a} :: [@size = length('$1')].
8
9   %% b_seq -> (when size_is_1) b
10  %%          | b_seq [@size = '$0'.size - 1] b.
11  b_seq -> {'$0'.size, b}.
12
13  c_seq -> {'$0'.size, c}.
```

Figure 2. $a^n b^n c^n$ grammar in our notation

contrast, in our notation the attribute computation rules are written on the spot, just after the entity to which the attributes belong.

This kind of formalism fits well with the constraints of L-Attributed grammars, where, due to the restrictions of the inheritance, attribute computations may refer only to their left. In our notation, the attribute computation section can refer only to symbols being on its left. This approach is similar to the sequential programming style, in which a statement may only refer already declared variables.

Due to the fact that attribute computations are written just after the symbol that they affect, the nonterminals do not have to be indexed on the right hand side, since the position of the attribute computation rule determines on which symbol it is defined. Synthesised attributes are given separately at the end of the rule.

In *line 7* (Figure 2) in comparison with the usual formalism we can see a useful simplification, that is, one can use repetition (lists) instead of primitive recursion, which will be shorter and easier to understand. In yecc (and therefore in eqc_grammar) rule alternatives and repetition are not supported, so with our formalism it is easier to express grammars, because it is closer to EBNF rather than to BNF.

Moreover, as can be seen in *line 9*, in order to make the notation express conditions based on attribute values, we added support for guards in rule alternatives. In *line 2* and *line 3* the setting of the inherited attributes is shown, and then in *line 11* and *line 13* it can be seen how the attributes can be accessed. To ease the attribute computation, one can use any kind of Erlang expression to compute the attribute value. As should be evident, the notation is concise and is similar to the usual AG formalisms.

Special grammar rules: embedded rules

Generating the right hand side belonging to a grammar rule theoretically is a single atomic step. That is, every value belonging to the symbols on the rule's right-hand side are generated simultaneously. While generating, for example, an Erlang function clause, apparently the generation of the clause patterns and of the clause body may not be simultaneous, since the body may well depend on the patterns, or formal parameters, in the function head. In such cases the right hand side values of the grammar rule may not be generated together in a single round. To denote this, we use a special arrow symbol in the grammar description, which separates the different parts of the rule. In other words, productions may be regarded as sequences of separately generated right hand side element groups, where the groups are separated by ~> symbols. After every such (possibly empty) group one can write any Erlang code and can manipulate the current attributes.

Consider a simple definition of Erlang clauses, in which a clause consists of a formal parameter list (patterns) and a body (expressions). Obviously, the formal parameters and the body of a clause are semantically interdependent, since the variables bound in the patterns might be used inside the clause body. Therefore, the generation of the expressions is embedded into the generation of the patterns. In other words, the two generation steps are performed in sequence. After generating the proper pattern and expression lists, a subtree is synthesised that accords to the function clause, like this:.

```
function_clause -> {~ N, pattern}
                ~> {~ M, bodyexpr}
                :: create_clause('$1', '$2').
```

The embedded rules are compiled into applications of the *bind* built-in QuickCheck generator combinator. By using this combinator, we get a monadic-style execution of the value generators (in Haskell QuickCheck this feature is implemented with monads).

Special grammar rules: recursive rules

As explained, earlier, formal grammars mainly consist of grammar rules. Basically, a grammar rule has a nonterminal on its left hand side and a list of either terminals or nonterminals on its right hand side. The rule defines the meaning, the way of production of the nonterminal being on the left. If that symbol also appears on the right, the rule is said to be recursive. Recursion might be indirect as well, that is, the rule's right contains a nonterminal whose definition refers to the current rule. Some of the recursive rules can be eliminated by using repetition, the others have to be handled or modified properly in order to avoid infinite recursions.

Repetition By applying EBNF-style repetition, many primitive recursive rules can be eliminated from grammar descriptions. Usually, when generating lists of entities, in BNF one has to create a primitive recursive rule, which has both a 'productive' and a 'base' alternative. Consider the following example which may generate lists of expressions. You can see that the recursion can be eliminated by using repetition.

The recursive description:

```
exprs -> expr exprs
       | expr
```

And the same rule by using repetition:

```
exprs -> {expr}
```

According to the actual context, one can use repetition in two different ways depending on the way of handling the attributes. Also, lists of entities may be generated with a given (fixed) size or else a randomly generated size. The generation of repeated parts is basically implemented by using QuickCheck's *list* and *vector* generator combinators, the former for variable sized lists and the latter for lists with a given size (the size parameter can be either a variable name, a constant or a macro/function call). However, if the generation of the list elements may be interdependent, that is, certain list elements may depend semantically upon each other, then the generation gets more difficult. In the latter case, special auxiliary generator combinators are included into the resulting source code, which help the generation of dependent lists.

While independent list elements are generated simultaneously and all elements inherit the same attribute list from their parent (in other words, every list element is generated over the same attribute list and cannot affect each other), in dependent list generation, elements are generated one after the other and each one inherits the attributes from the previous one. That is, the generated attributes flow through the list and the currently generated elements can affect the following siblings. The method and the notation is quite similar to rule embedding. In this case, we would say that all sublists are embedded. Dependent repetition symbols and embedded rules can be seen in the following example.

```
module   -> {attribute} {~ ?M, function}.
function -> {?N, clause}.
clause   -> {~ pattern} {~ expr}.
```

Controlling recursion While using a grammar description for parser generation, the alternatives are equivalent in the respect of applicability, since the input string determines which alternatives have to be used for reduction. During a random generation, in the case of rules built up from many alternatives the framework should somehow choose one of them. In our solution, the generator randomly makes a choice among the alternatives and the selected one is going to be used for generating the current subtree. Obviously, if the alternatives are equivalent, the mentioned choice is totally random, all the alternatives have the same chance to be chosen. However, in some cases it is expected to make a kind of priority order among the rule alternatives in order to control the structure and the properties of the randomly generated data.

In a rule, all rule alternatives may be associated with a frequency (or weight), based on which they will be chosen. Obviously, by adjusting the probability of the different alternatives the generated data structure accordingly changes. An alternative's weight can mean its relevance as well as the complexity of the subtree that may be generated by its application. In the case of primitive recursive rules (for example, generating list data structures) the weight of the rule alternatives may affect the size of the generated data.

Theoretically, recursive generation should terminate by a proper setting of probabilities. However, in practice, structurally recursive generation processes may not terminate, instead, infinitely enlarge the generated structure. To avoid infinitely recursive application chains, a recursion depth limit was injected into the generation process. The current depth of the recursion is registered during the rule applications, more precisely, a counter registers the number of the available recursive calls before hitting the limit. The counter is decreased every time when a recursive call is performed. If the limit is hit (in other words, the counter becomes zero), then only simple (usually non-recursive) alternatives can be applied. Thus, the generation terminates on the current subtree. This integer expression generator shows this in action:

```
intexpr(0) -> int :: erl_syntax:integer('$1').

intexpr(N) ->
    intexpr(0)
  | intexpr(decr(N)) infixop intexpr(decr(N))
  :: erl_syntax:infix_expr('$1',
        erl_syntax:operator('$2'), '$3').
```

The 'simple' rule alternatives could be found by analysing the recursiveness of the right hand side, but in our metalanguage the programmer has to mark the non-recursive, simple cases. In our formalism the rules are written in a function-like format and in particular they can have arguments. The mentioned counter registering the depth of the recursion is manually decreased and passed to the (directly or indirectly) recursive symbols. This solution gives the programmer full control over the recursive generation.

3.2 An Erlang grammar definition

The generator generator framework is applicable to produce any kind of data that can be described using formal grammars. In our case, we have used the framework to generate Erlang programs. Consequently, we have created a grammar definition of the Erlang language, more precisely, a definition of the sequential programming language elements.

First of all, we had to decide, at what abstraction level to generate programs, since source code can have many kinds of representation, such as the well-known textual representation, token stream, or abstract syntax tree (AST). This decision will in turn determine the further difficulties of the description, because different representations may introduce different problems during the generation process.

We decided to use the latter representation, namely Erlang Syntax Trees. Such trees can easily be represented and handled using the Erlang Syntax Tools library, which is included in the standard Erlang distribution and includes modules declaring useful types and functions helping in construction and pretty-printing such syntax trees. With the functions of the `erl_syntax` module it is simple to create ASTs in a bottom-up strategy. By using the Syntax Tools application we only have to focus on the generation of abstract syntax trees instead of the textual program code. Compiling the grammar description we can get a QuickCheck generator that can produce random, compilable Erlang source code. In the background, the generation method creates an Erlang syntax tree which is then pretty-printed.

Using the current language definition we can generate any number of modules containing random function definitions that may refer functions from another generated modules. Functions may have randomly one or more function clauses, which do not shadow each other and have randomly generated patterns. The function bodies may contain many kinds of Erlang expressions, including IO statements, case expressions and match expressions as well. Moreover, generated code may invoke library functions. Match expressions can bind variables, and other expressions may refer those variables (but cannot re-bind them) afterwards.

Every generated language element is well-typed, since types are managed by storing related informations in attributes. The types used during the generation are randomly generated as well.

Finally, many properties of the generated code can be parameterised, such as the number of the generated functions, the difficulty of the generated expressions and the maximum level of nested case expressions. By adjusting the grammar and the parameters we can make the generated code quite similar to real-world programs.

3.3 Transformation

We have implemented a compiler (a generator generator) for our grammar definition, which produces a single Erlang module containing functions returning QuickCheck generators for each production rule preserving its meaning. The compiler uses the standard Erlang scanner (with some extensions) and a yecc-generated parser. After scanning and parsing the grammar definition, firstly it checks some constraints on the grammar (for example, every declared nonterminal is defined as a rule, the are no symbol duplications, every right hand side symbol is defined in the file), then generates the

```
-module(prop).

-export([prop/0]).

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

prop() ->
    test:prop_beh_eqv(rename_mod,
                      fun gen_rename_mod_args/1).

gen_rename_mod_args(Filename) ->
    ?LET(Atom, test:gen_atom(),
         [Filename, Atom, [], 8]).
```

Figure 3. Example of rename module property

output Erlang module: if the input file is `abc.eyrl`, then the output file will be `abc.erl`, which is constructed of the generator functions belonging to the grammar rules, the attached Erlang code cut from the grammar definition file (without any changes) and further essential function and macro definitions being for the notation features.

The generated output file is checked whether it compiles or not (using the Erlang compiler strong validation). If the module is compilable, with the `erl_tidy` module (included in Erlang Syntax Tools) it is tidied (unused functions are removed from the code, some syntax constructs are rewritten in a more readable form) and then compiled again, for reasons of optimisation.

Despite the fact that in the generated Erlang module every function can be called from the outside (that is, they all are exported), only the function belonging to the root symbol can be used without any parameters (the others require parameters carrying information about the attributes). The return value of this function is a valid QuickCheck generator and passing this generator to QuickCheck results in the expected random data.

4. Properties

If a refactoring was performed correctly, the behaviour of the program should not have changed: it should return the same output for the same input, throw the same exceptions, send the same messages in the same order. We say that the original and refactored versions are behaviourally equivalent.

To test behavioural equivalency we follow these steps: generate random programs, perform the refactoring, pick a function, generate random arguments guided by type information, evaluate the function and then compare the result of the two versions.

Since Erlang programs contain many functions, and a refactoring may only modify a single function we could pick an irrelevant function to test and miss an error. This is natural, and the solution is to run many tests to minimize the chance that we miss the erroneous function. A similar thing happens with arguments when the function has multiple clauses, which is fairly typical.

To minimize the chance of missing an error, we have to run a large number of test, so test execution speed matters. The two slowest phases are program generation and the refactoring itself. By running the later phases more than once after every refactoring we can reduce the chance of missing errors without having to execute the slower steps repeatedly as many times.

Erlang is a dynamically typed language, which means that we can supply any term as argument to a function, and at the worst case we get a runtime exception. Arguably this doesn't help with catching real bugs. The dialyzer tool [17] can infer type information for functions from a codebase. This makes it possible to generate proper arguments for the function under test, so that we can avoid programs under test failing for reasons of data being ill-typed.

In Erlang I/O uses message passing under the hood, hidden from the user. The messages are in a certain format and are sent to an I/O device, which is a separate process. The format of the messages is well documented, and any process that can handle them can be used as an I/O device. The testing framework uses an I/O device which behaves like a ram file that is initially empty. This I/O device additionally keeps track of the messages received.

To test behavioural equivalency we evaluate both the old and the new version of a function, and compare the outputs and I/O traces. This can be done concurrently, saving us time. This is particularly true when the functions are non terminating due to the random nature of them. In this case we halt the evaluation after a given time. Concurrent execution means we only have to wait for this timeout only once, halving the time needed to test.

4.1 Example property

In order to define a behavioural equivalence property for a refactoring, the user has to call `test:prop_beh_eqv/2` with the appropriate arguments. The first argument is an atom, which is the name of the function in the `wrangler` module that implements the refactoring. The second parameter is a callback function, that given a filepath should return suitable arguments for the refactoring function. In the simplest case the callback returns a list containing the arguments. If the testing should be restricted to a specific function, the return value should be a three-tuple with the function name and arity for the said function and the argument list for the refactoring function.

There is another version of `test:prop_beh_eqv`, which takes an additional third argument. This is a callback function that receives the result of the previous callback function and returns a boolean indicating whether to proceed with the test or not.

The simplest property is for the rename module refactoring, the whole code is given in Figure 3.

5. Results

We have designed and implemented a notation for L-Attributed grammars and created a compiler for it. So far we have formalised a subset of the Erlang language with it and got promising results.

Moreover, by using random generation we have implemented the testing of four refactoring steps provided by Wrangler: rename variable, rename function, generalise function and tuple function arguments. We have found two bugs, one in rename variable regarding incorrectly handling patterns in function parameters, and one in generalise function regarding incorrectly transforming the function leading to compiler errors. Both of these errors are fixed in the latest release of Wrangler.

6. Related work

[9] is a similar study done for refactoring engines integrated in IDEs for mainstream languages. The program generator described is specific to the language they use and it can be parametrized by code fragments, so it would be difficult to adapt to other domains. They test the results of refactorings in a different way too: they test hand written structural properties as opposed to behavioural equivalence, and do this by testing the results of two different refactoring engines against each other, rather than testing the old and new code directly.

[14] describes previous work using QuickCheck for testing Wrangler. They did not use random program generation, refactoring a static codebase instead, and the only property formulated is successful compilation of the refactored program.

7. Conclusions and future work

We have demonstrated that automated, property-based random testing of refactoring tools is able to discover new bugs, and therefore it is a useful addition to the testing processes of tool developers.

How to check behavioural equivalence of arbitrary message passing programs, or refactorings which have wider ranging effects is still an open question.

Our main contribution is random program generation and behavioural equivalence testing, which together give much wider coverage, scalability and maintainability to the testing of refactoring engines.

References

[1] Quviq QuickCheck, June 2010. http://www.quviq.com/.

[2] RefactorErl, June 2010. http://plc.inf.elte.hu/erlang/.

[3] Wrangler, June 2010. http://www.cs.kent.ac.uk/projects/forse/.

[4] T. Arts and J. Hughes. Erlang/QuickCheck. In *In Ninth International Erlang/OTP User Conference*, November 2003.

[5] T. Arts et al. Testing Telecoms Software with Quviq QuickCheck. In *ACM SIGPLAN workshop on Erlang*. ACM Press, 2006.

[6] T. Avgerinos and K. F. Sagonas. Cleaning up erlang code is a dirty job but somebody's gotta do it. In *Erlang Workshop*, pages 1–10, 2009.

[7] J. W. Backus et al. Revised report on the algorithmic language ALGOL 60. 1997.

[8] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 1956. doi: 10.1109/TIT.1956.1056813.

[9] B. Daniel et al. Automated testing of refactoring engines. In *ESEC/FSE*, pages 185–194, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-811-4.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, July 1999. ISBN 0-201-48567-2.

[11] D. Horpácsi. Testing refactoring tools by generating random Erlang modules. Master's thesis, ELTE, Budapest, Hungary, 2010.

[12] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 1968. doi: 10.1007/BF01692511.

[13] P. M. Lewis et al. Attributed translations (Extended Abstract). In *STOC '73*. ACM, 1973.

[14] H. Li and S. Thompson. Testing Erlang Refactorings with QuickCheck. In *IFL*, pages 19–36, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-85372-5.

[15] W. F. Opdyke. Refactoring object-oriented frameworks. Technical report, 1997.

[16] J. Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 1995. ISSN 0360-0300.

[17] K. F. Sagonas. Experience from Developing the Dialyzer: A Static Analysis Tool Detecting Defects in Erlang Applications. In *Workshop on the Evaluation of Software Defect Detection Tools (Bugs'05)*, 2005.

[18] K. F. Sagonas and T. Avgerinos. Automatic refactoring of erlang programs. In *PPDP*, pages 13–24, 2009.

[19] L. P. Sobotkiewicz. A New Tool for Grammar-based Test Case Generation. Technical report, University of Victoria, 2008. MSc thesis.

[20] N. Sultana and S. Thompson. Mechanical Verification of Refactorings. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM Press, 2008.

Using Erlang to Implement a Autonomous Build and Distribution System for Software Projects

Tino Breddin
Erlang Solutions Ltd
tino@erlang-solutions.com

Abstract

The uptake of Open-Source Software (OSS) has led to new business models as well as software development practices. *OSS* projects are constrained by their limited resources both in time and manpower. In order to be successful such projects have to leverage tools to automate as many tasks as possible while providing usable results. One such set of tools used in software development are continuous build systems, which help teams to build and test their software whenever a change is published without manual interaction. The available systems have proven to be essential for any kind software project but are lacking real innovation.

This paper presents how Erlang, especially its distributed operation, fault-tolerance and lightweight processes, has been utilized to develop a next-generation continuous build system. This system executes many long-running tasks in parallel for any given change of the monitored software project, providing developers not only with the latest state of the project but also offers customizable software packaging and patch distribution.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing Tools; D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Verification, Measurement, Reliability

Keywords ci, continuousintegration, erlang, otp, testing, buildsystem

1. Introduction

Software projects, especially Open-Source Software projects, need to thrive for good software quality to satisfy both customers and users. Unfortunately any additional effort spend on testing might delay the project or increase its costs. Therefor tools, such as continuous build systems, provide ways to automate repetitive tasks, and thus help to improve software quality tremendously. But many projects don't have very specific definitions of their target platforms and distribution formats. Open-Source Software often requires as many well supported platforms as possible to attract new users, which in return requires many different distribution formats. This requirement cycle pushes any project's cost and available resourcesr to the limit, leaving much potential unused.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Erlang'10, September 30, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0253-1/10/09...$10.00

Continuous Build Systems The landscape of continuous build systems has improved to a point where *OSS* systems such as Hudson [1] and CruiseControl [2] deliver great advantages to their users. Their flexible plugin architectures have led to the addition of many new features based on external plugins. Nevertheless the core concepts of these systems are targeted towards projects with well defined specifications, making them somewhat inadequate for many more general-purpose software projects.

2. Background

A continuous build system implements Continuous Integration (CI) concepts to be used in software projects. Therefor section 2.1 explains the basics of this methodology. Further section 2.2 outlines the definition of Distributed Version Control Systems (Distributed VCSs) and their usefulness for agile software development. Because the system presented in this paper has been implemented using Erlang/OTP, section 2.3 provides a short description of its most important features. Finally the provided background is summarized in section 2.4.

2.1 Continuous Integration

The concept Continuous Integration was first defined by Beck [3] as part of a set of Extreme Programming (XP) best practices. Building and testing a software product every time a development task is completed was considered Continuous Integration. Beck [3] arguments that this regular cycle would instantly show whether new changes break the software and could be addressed much earlier than in traditional development models such as the *Waterfall Model* presented by Royce [4]. This simple concept has evolved from a suggested *XP* practice to a software development methodology, as Duvall et al. [5] argument, which is based on the set of core practices outlined next.

Build Automation Building complex software often involves many steps. For the convenience of software developers it should be possible to perform this procedure using a single command. This ensures that the software is build often since it doesn't require much manual work by a developer.

Build Fast The time which is required to build software tends to increase with the complexity of the software. This jeopardizes the usefulness of builds during development since one would have to wait too long to get feedback. Thus builds should be kept fast, if necessary the software should be split up into components which can be build and tested individually.

Visible Progress All information regarding the state of the software should be visible to all parties involved in its development. As a result all team members can make informed decisions when faced with bugs, tasks or a broken build.

Commits trigger Builds The software should be build in its entirety every time a change was applied. This should be done automatically without manual interaction, finally providing team members with the results of these builds instead.

Cloned Production Environment In order to utilize automated builds and tests further, these should be performed in an environment which resembles the target environment closely. The better this environment is cloned the more confident one can be that platform-specific bugs were addressed during development.

2.2 Distributed Version Control

Special purpose systems for controlling source code were first introduced by Rochkind [6] and Glasser [7]. Later Tichy [8] defined the stages and culprits involved in working on such systems. The first widely used *OSS* Source Code Management (SCM) system called Revision Control System (RCS), which was introduced by Tichy [9], was an important milestone because it nourished the development of many different approaches in the area of *SCM* as summarized by Royce [4]. Systems based on those early concepts are categorized as Centralized Version Control Systems (Centralized VCSs) because of their reliance on a central source code repository.

A change to this limitation has been provided by *Distributed VCSs*. Their definition of a repository allows every developer to have a full copy of a repository, empowering users to share changes freely. This flexibility makes it possible for teams to adapt their development workflow to their needs without being limited to a certain setup.

Git As a popular example of this kind of Version Control System (VCS), Git combines flexibility and exceptional speed of execution. Operations like *branching* and *merging* aren't only performed fast on a local repository, they also allow users to separate their development efforts into different channels for bugs, new features or releases.

2.3 Erlang/OTP

The Erlang/OTP project is comprised of the development of the programming language *Erlang* and a set of well supported applications referred to as Open Telecom Platform (OTP). The language itself was developed at the Ericsson CSLab in the 1980s and was made available as Open-Source Software in 1998 as outlined by Däcker [10]. The language features of *Erlang* allow easier development of concurrent applications, which is why it is also referred to as a *concurrent functional language*. Furthermore *OTP* adds functionality which provides a strong foundation for distributed and fault-tolerant systems. The features which are most important for this work are outlined next.

Virtual Machine *Erlang* source code is compiled to byte-code which can be executed by the *BEAM* Virtual Machine (VM). This allows pure *Erlang* applications to be highly portable to other platforms which are supported by the *VM*. Furthermore the *VM* provides efficient *SMP* support which improves the performance of highly concurrent applications.

Lightweight Processes Processes are fundamental language constructs. A single process has a very small memory footprint as well as fast creation and destruction performance. The Erlang *VM* can manage millions of processes while only being memory-bound. Moreover the *VM* schedules the execution of processes in parallel which makes writing parallel code much easier.

Message Passing Sending data between processes is realized by *Message Passing* and is the only way to share data, because no shared memory is provided for that purpose. This further improves *Erlang's* support for writing parallel applications since communication channels must be clearly defined.

Distributed Erlang Erlang nodes can be connected and execute applications location-transparently. A single running Erlang *VM* is referred to as a node. This location transparency makes running applications on many systems trivial. Moreover the execution of arbitrary Erlang code on other systems can be used to implement loosely coupled distributed systems.

2.4 Summary

Continuous Integration is a modern software development methodology which emphasizes sharing of information as a key requirement for any software project. Its agile concept is further supported by the growing popularity of Distributed Version Control Systems which allow development processes to by designed to fit a particular project rather than having to use a pre-defined setup. Continuous build systems usually provide support for such systems without utilizing their new concepts.

Erlang/OTP is a mature foundation for building concurrent applications which also require scalability and fault-tolerance. The language is based on functional concepts which allow users to implement parallel architectures more easily.

3. System Design

Based on the motivation outlined in section 1 the core requirements are gathered in section 3.1. Further section 3.2 describes the top-level architecture and its impact on non-functional requirements of the system. The following section 3.3 details the fundamental support for using multiple platforms for building and testing. *Package Generation*, another core feature, is explained in detail in section 3.4. Finally section 3.5 concludes the *system design* by giving a concise summary.

3.1 Requirements

Both *CI* and *Distributed VCS* as fundamental technologies already suggest a range of requirements for a continuous build system. What follows is a list of these and additional requirements which should be mandatory for any modern continuous build system.

Project Repository Monitoring As part of a project which is setup within a continuous build system, the respective source code repository should be monitored automatically. This allows the continuous build system to notice when the latest version within the repository changes.

Build Latest Version Furthermore the continuous build system should build and test the latest version of the source code, once it changed. The delay between the time when the change was published and the start of the build should be as small as possible.

Make Results Publicly Available Continuous Integration stresses the fact that any information about the state of the source code should be shared within a team. Thus the results of builds of a project need to be presented in a publicly readable form, once available. That includes that results from ongoing builds should be already presented.

Build in Target Environments The motivation for this work, which was presented in section 1, states that builds need to be run on many systems to improve software quality rather than having to identify a single production setup. This requirement especially holds true for *OSS* projects. Therefor a continuous build system needs to be able to build the source code on as many platforms as possible, which are distinct from the one it is running on.

Build Topic Branches Globally shared repositories aren't the first destination of changes within optimized development workflows, which rely on the use *Distributed VCSs*, anymore. Teams might decide to only submit verified changes to their central repository, while changes which still need to be verified are kept in the developers repositories, some other shared repository or within a code review system. Either way, part of the verification should be the building and testing of changes as soon as developers have finished working on these. A continuous build system should build such changes whether they live in a repository as topic branches, are kept as patches in a code review system or send as an email to a mailing list. The same benefits of building the latest version of some source code apply to building potential changes to the latest version, which is the next step of Continuous Integration.

Automatic Package Generation Any software needs to be distributed in a pre-defined format, which can be project-specific or follow a operating system package specification. Either way the task of creating such packages is repetitive and should be assisted by a continuous build system such that it generates packages based on given package specifications and the definition of trigger events.

Share Results An important part of the system integration is that results of builds for any change are shared with other systems within the build environment. This allows the setup of sophisticated processes which propagate the data to the correct communication channels. Thus a continuous build system should provide the foundation for sharing results with other communication channels.

3.2 Architecture

Swarm utilizes a component based design to achieve better results for its non-functional requirements. Each component is self contained as much as possible, which improves the ability to swap components which provide the same functionality for whatever reasons. Further the development of a single component doesn't impact other components, thus improving its overall stability. Moreover external components can be easily employed if they provide the required interfaces for other components.

Figure 1 depicts the main system components and their communication channels within the system. All information of the system state and builds is presented to users using a web-frontend by the *OAM* subsystem. Access to external servers is managed by the *Platform Management* which is explained in more detail in section 3.3. The coordination of all work done within Swarm is performed by the *Job Processor*, which therefor provides the core logic of the system. All other system components utilize the *Data Storage* subsystem for all persistent data access. In addition the use of a underlying *Distributed VCS* is masked by a *VCS Interface* which provides support for various such systems.

The overall system is self-contained with very few dependencies which very much improves its maintainability. No third-party software is used except the Version Control System which is used to keep source code locally. This also allows the system to be easily deployed and setup within minutes since only Swarm needs to be configured.

3.3 Multi-Platform Support

Swarm allows projects to build and test changes on many platforms at a time, which will help to improve software quality in the long run. These systems can be provided differently. Dedicated servers, which can be either virtual or physical machines, are the default options for platforms which can be added to Swarm. Using such servers one can support rather exotic combinations of hardware, operating systems and software. The bulk of supported platforms can be added through the use of computing clouds though. Swarm allows the utilization of *Amazon EC2* or the API-compatible *Eucalyptus* for creation of virtual machine instances. These instances will further be used to build and test individual changes. Thus one can easily create a range of virtual machine images, which represent combinations of operating systems and software, and register these with Swarm. Going forward these images will be used as well. Using computing clouds for supporting platforms provides the same advantages as these do for data processing of large data sets already. Because builds only need to be performed when changes to the source code have been published, the continuous build system wouldn't utilize dedicated servers in between such changes. But when builds are triggered, many platforms can be used to run those simultaneously.

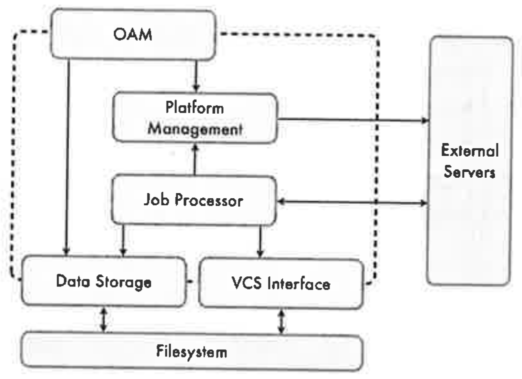


Figure 1. Top-level architecture for Swarm.

3.4 Package Generation

Software is generally distributed in packaged form instead of simply providing a snapshot of the source code. There is no standard format, instead most operating systems have their own package format which is used by some sort of package management tool. Popular examples are Debian's *deb package format* [11] and Red Hat's *rpm format* [12]. Because packages allow users to install software much more easily than using the source code directly, having many such packages available presents a great opportunity for software projects to reach a wider audience. Therefor Swarm assembles such packages automatically when new changes have been published, so that users are provided with up-to-date packages whenever possible. The requirement for package generation is a *package specification* defining how the package is supposed to look like and when it should be build. Furthermore its location is specified, which can be used to access generated packages.

3.5 Summary

Swarm utilizes a component-based architecture to achieve a high degree of flexibility while keeping the required maintainability effort to a minimum. Its novel support for using multiple platforms to build and test on allows software projects to extend their supported range of platforms easily. Furthermore packages for these platforms can be automatically generated based on a given specification, without requiringany manual interaction.

4. Workflows

Because a continuous build system is usually tightly integrated in a development process, many usage scenarios are imaginable and valid regarding their use in a production environment. To illustrate how the core components interact internally two workflows are further described in more detail. Section 4.1 explains how new

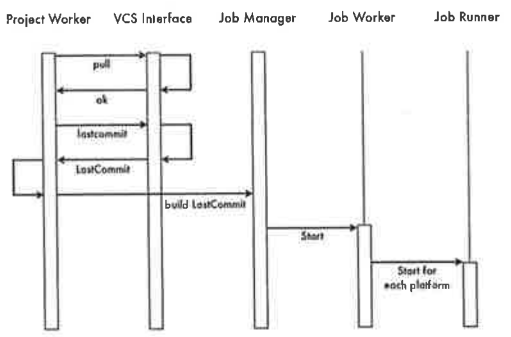


Figure 2. Activity diagram showing the workflow when the latest version changed.

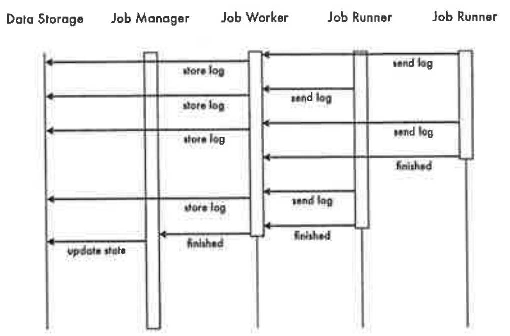


Figure 3. The workflow for result consolidation outlined in a activity diagram.

changes trigger builds while section 4.2 details how results are gathered from various builds.

4.1 Latest Version Changed

The standard event in any continuous build system is triggered when the latest version of a monitored repository changes. Upon such a change the system needs to start builds on all platforms a project wants to support. Figure 2 shows how the system receives a new change and propagates that information to the *Job Manager*, which coordinates all builds. This long-running process is the coordinator for all builds and manages the processes which are associated with individual builds. Any build requires a separate process which allows the *Job Manager* to implement fail-over for broken builds more easily. Once the manager has started all builds it will enter an idle state during which it will only ensure that builds are running.

4.2 Result Consolidation

Once builds are running, their respective results, which is most commonly stdio output, need to be stored for later inspection. This data is transfered from remote servers to Swarm as outlined in figure 3. All data is stored persistently even while builds are still running. This ensures that users can inspect the intermediate results of running builds. Once builds have finished the *Job Manager* needs to clean up its state to reflect the terminated processes and change the persistent build data accordingly.

5. Evaluation

The system still needs to be deployed in a production environment which fully exercises the core build execution, the user facing web-frontend and further system integration. Nevertheless its usage of Erlang/OTP can already be reflected upon which is done in section 5.1. Subsequently a set of scalability tests and their results are explained in section 5.2.

5.1 Using Erlang/OTP

Swarm's support for multiple platforms makes use of *Distributed Erlang* to execute builds and tests on remote systems. Because Erlang handles all communication transparently the integration of remote nodes was easily setup. Further the integration of those nodes into the main system supervision tree allows Swarm to react to node failures immediately without having to fallback on using polling mechanisms.

The *Job Processor* subsystem coordinates all builds and tests which are executed at any given time. This not only requires the execution of such jobs in parallel, but also the handling of all communication with remote nodes to gather log data and state changes. Because *Message Passing* and *Processes* are tightly integrated into the Erlang programming language, developing a working coordination system was possible without using other third-party libraries. The separation between communication channels and process state supports the fault-tolerant design of the core coordination system.

5.2 Scalability

The system is not meant to run on more than a single server, because uninterrupted availability is not required due to the nature of continuous build systems. Nevertheless a project should be able to make use of many platforms without incurring delays for the execution of builds. Because the communication effort which is performed for each running build is kept to a minimum using message caching on both ends, Swarm can be expected to perform well as long as *Distributed Erlang* performs as required. Further the computation-intensive work is performed on the build target machines which actually build the software.

The system was tested using a setup of two dedicated servers and eight Amazon EC2 images. Thus every change to the monitored source code triggered a total of ten builds to be executed in parallel. This didn't add any delay to the build execution other than the time required for transferring the source code to the build target, which depends on the size of the project and available network bandwidth. In further tests changes were applied in short intervals to trigger more builds in parallel. The system could sustain a total of 30 parallel builds without much CPU utilization. It should be noted that builds on dedicated servers are executed sequentially, because Swarm only runs one build at a time on any available server.

6. Conclusion

Continuous build systems have become a essential part of the development process of many software projects. The system presented here takes a different approach for some core concepts of such systems. By supporting multiple platforms for running builds and creating new software packages on-the-fly it enables projects to improve their support and thus increase their user reach. The system is designed to be easily dropped into established development processes without much effort required for an initial setup.

Although tests have shown that the system performs according to the author's expectations, it still needs to be deployed for a active software project to evaluate its proposed advantages and potential shortcomings. Nevertheless it shows that a novel approach to well-established systems such as continuous build systems can improve software quality tremendously.

References

[1] Open Source. Hudson - A extensible continuous integration server. http://hudson-ci.org/

[2] Open Source. Cruisecontrol - A continuous build system. http://cruisecontrol.sourceforge.net/

[3] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[4] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[5] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional, 2007.

[6] M.J. Rochkind. The source code control system. In *IEEE Transactions on Software Engineering, SE-1(4)*, pages 364–370. IEEE Computer Society Press, 1975.

[7] Alan L. Glasser. The evolution of a source code control system. *SIGSOFT Softw. Eng. Notes*, 3(5):122–125, 1978.

[8] Walter F. Tichy. Design, implementation, and evaluation of a revision control system. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 58–67, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[9] Walter F. Tichy. Rcs—a system for version control. *Softw. Pract. Exper.*, 15(7):637–654, 1985.

[10] Bjarne Däcker. Concurrent functional programming for telecommunications: A case study of technology introduction. Master's thesis, KTH Royal Institute of Technology Stockholm, 2000.

[11] Open Source. Deb - The Debian software package format. http://www.debian.org

[12] Open Source. RPM - The RPM package manager and file format. http://rpm.org

Author Index