September 5, 2009
Edinburgh, Scotland

Association for
Computing Machinery

Advancing Computing as a Science & Profession

Erlang'09

Proceedings of the 2009 ACM SIGPLAN Erlang Workshop

Sponsored by:

ACM SIGPLAN

Co-located with:

ICFP'09

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

ISBN: 978-1-60558-507-9

Additional copies may be ordered prepaid from:

ACM Order Department
PO Box 30777
New York, NY 10087-0777, USA

Phone: 1-800-342-6626 (US and Canada)
+1-212-626-0500 (Global)
Fax: +1-212-944-1318
E-mail: acmhelp@acm.org
Hours of Operation: 8:30 am – 4:30 pm ET

ACM Order Number 553092

Printed in the USA

Foreword

It is our great pleasure to welcome you to the Eighth ACM SIGPLAN Erlang Workshop – Erlang'09. This year's workshop continues the tradition of being co-located with the annual International Conference on Functional Programming (ICFP), and being a forum for the presentation of research theory, implementation and applications of the Erlang programming language.

All submissions to the workshop were reviewed by at least three program committee members. The program committee accepted 10 papers that cover a variety of topics, including language aspects, Erlang program development, testing and validation as well as teaching and learning Erlang.

Putting together Erlang'09 was a team effort. First of all, we would like to thank the authors and the invited speaker for providing such an interesting program. We would like to express our gratitude to the program committee and additional reviewers, who worked very hard in reviewing papers and providing constructive criticism for their improvement. We would also like to thank Christopher Stone and Michael Sperber, this year's ICFP Workshop Chairs, for their efforts in coordinating all workshops, and the ICFP local organizers for their hard work on local arrangements. As usual, special thanks go to Bjarne Däcker for sharing experiences of previous Workshops, maintaining the Erlang'09 website, and advertising the workshop. Finally we would like to thank ACM SIGPLAN for their continued support.

We hope that you will find this program interesting and thought-provoking and that the symposium will provide you with a valuable opportunity to share ideas with other researchers and Erlang practitioners from companies and academic institutions around the world.

Clara Benac Earle
Erlang'09 General Chair
Universidad Politécnica de Madrid

Simon Thompson
Erlang'09 Program Chair
University of Kent

Table of Contents

Erlang Workshop 2009 Organization

General Chair: Clara Benac Earle *(Universidad Politécnica de Madrid, Spain)*

Program Chair: Simon Thompson *(University of Kent, UK)*

Steering Committee Chair: Bjarne Däcker *(Independent Telecoms Professional, Sweden)*

Program Committee:
Laura M. Castro *(University of A Coruña, Spain)*
Francesco Cesarini *(Erlang Training and Consulting, London, UK)*
Torben Hoffman *(Motorola, Denmark)*
Zoltán Horváth, *(Eötvös Loránd University, Budapest, Hungary)*
Jan Lehnardt *(CouchDB, Berlin, Germany)*
Daniel Luna *(Kreditor, Stockholm, Sweden)*
Kenneth Lundin *(Ericsson, Stockholm, Sweden)*
Rex Page *(University of Oklahoma, USA)*
Corrado Santoro *(University of Catania, Italy)*
Tee Teoh *(Canadian Bank Note, Ottawa, Canada)*
Phil Trinder *(Heriot-Watt University, Edinburgh, UK)*

Additional reviewers:
István Bozó
Lilia Georgieva
Gudmund Grov
Peter King
Róbert Kitlei
Tamás Kozsik
Laszlo Lovei
Dominic Mulligan

Sponsor: 

Cleaning up Erlang Code is a Dirty Job but Somebody's Gotta Do It

Thanassis Avgerinos
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
ethan@softlab.ntua.gr

Konstantinos Sagonas
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
kostis@cs.ntua.gr

Abstract

This paper describes opportunities for automatically modernizing Erlang applications, cleaning them up, eliminating certain bad smells from their code and occasionally also improving their performance. In addition, we present concrete examples of code improvements and our experiences from using a software tool with these capabilities, tidier, on Erlang code bases of significant size.

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

General Terms Design, Languages

Keywords program transformation, refactoring, code cleanup, code simplification, Erlang

1. Introduction

Most programmers write code. Good programmers write code that works. Very good programmers besides writing code that works also rewrite their code in order to simplify it, clean it, and make it more succinct, modern and elegant. While there will probably never be any real substitute for very good programmers, one might wonder whether there is some intrinsic reason why certain code rewriting tasks cannot be automated and become part of the development tool suite so that even good programmers can readily and effortlessly employ them on their code.

This question has been bothering us for quite some time now, in Erlang and elsewhere. Rather than just pondering it, we decided to embark on a project aiming to automate the modernization, clean up and simplification of Erlang programs. We started by standing on the shoulders of `erl_tidy`, a module of the `syntax_tools` application of Erlang/OTP written by Richard Carlsson, but as we will soon see we have significantly extended it in functionality, features and user-friendliness. The resulting tool is called tidier.

Tidier is a software tool that modernizes and cleans up Erlang code, eliminates certain bad smells from it, simplifies it and improves its performance. In contrast to other refactoring tools for Erlang, such as RefactorErl [9] and Wrangler [7], tidier is completely automatic and not tied to any particular editor or IDE. Instead, tidier comes with a suite of code refactorings that can be selected by its user via appropriate command-line options and applied in bulk on a set of modules or applications. This paper provides only a bird's eye view of the transformations currently performed by tidier; a complete description of tidier and its capabilities is presented in a companion paper [11]. Instead, the main goal of this paper is to report our experiences from using tidier, shed light on some opportunities for code cleanups on existing Erlang source code out there and raise the awareness of the Erlang community on these issues.

The next section contains a brief presentation of tidier. The main section of this paper, Section 3, gives a captule review for each refactoring currently performed by tidier and shows interesting code fragments we have encountered while trying out tidier on various open source Erlang applications. Section 4 presents tables showing the number of opportunities for tidier's refactorings on several code bases of significant size and discusses tidier's effectiveness. Section 5 presents characteristics of Erlang code that currently prevent tidier from performing more aggressive refactorings, while at the same time preserving its main characteristics, and discusses planned future improvements. The paper ends with some concluding remarks.

2. Tidier: Characteristics and Overview

From the beginning we set a number of primary goals for tidier:

- Tidier should support a fully automatic mode, meaning that all the refactorings should be such that they can applied on programs without user confirmation.
- Tidier should be flexible. Users should be able to decide about the set of refactorings that they want from tidier and, if they choose so, supervise or even control the refactorings that are performed.
- Tidier should never be wrong. Due to its fully automatic nature, tidier should perform a refactoring only if it is absolutely certain that the transformations performed are semantics-preserving, even if this comes at the cost of missing some opportunity or performing some weaker but safer refactoring.
- Tidier's refactorings should be natural and as good as they get. The resulting code should, up to a certain extent, resemble the code that experienced Erlang programmers would have written if they performed these refactorings by hand.
- Tidier should be easy to use and not be bound to any particular editor or IDE.
- Tidier should be fast. So fast that it can be included in the typical `make` cycle of applications without imposing any significant overhead; ideally, an overhead that is hard to notice.

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.

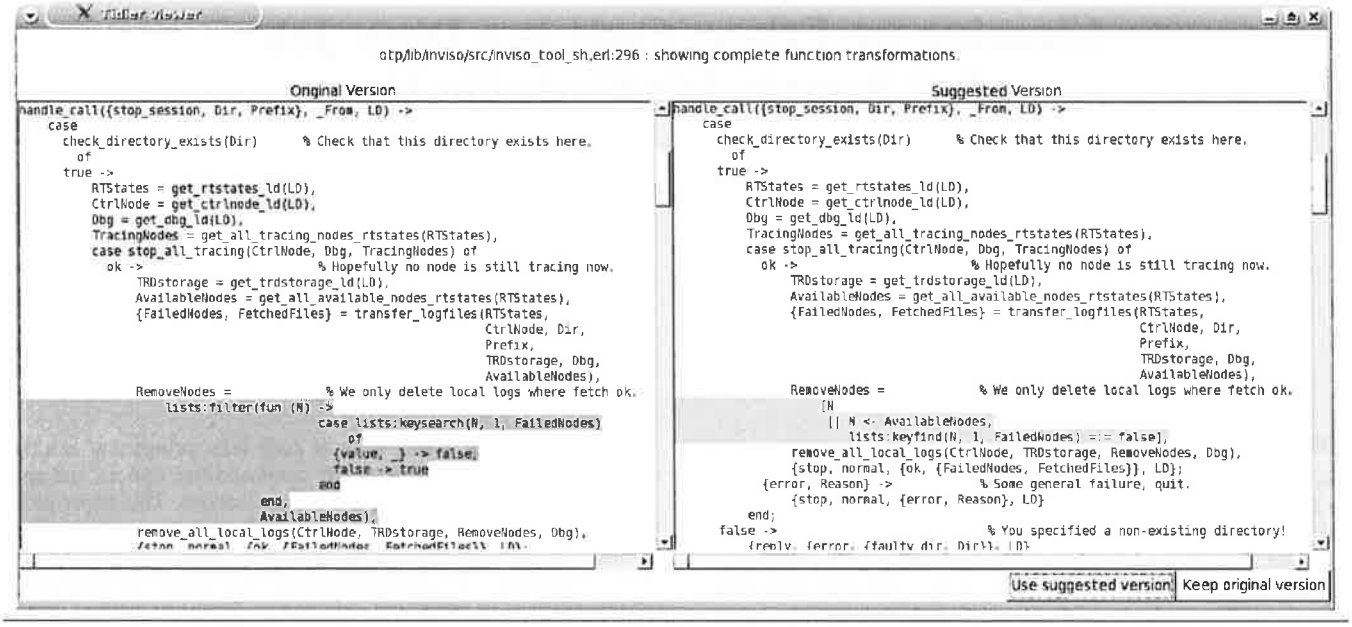

Figure 1. Tidier in action: simplifying the source code of a file from the `inviso` application of Erlang/OTP R13B.

Furthermore, we set a list of criteria that would serve as indicators of whether a specific refactoring should be performed by tidier. The transformations should result in code

modernizations: tidier should remove obsolete language constructs and use the most modern construct for the job.

simplifications: The resulting code should be shorter, simpler and therefore more elegant.

with fewer redundancies: The resulting code should contain fewer redundancies than the original version.

with the same or better performance: The new code should not deteriorate in performance and if possible become even faster.

Modes of using tidier One of our goals has been that tidier should be very easy to use. Indeed, the simplest way to use tidier on some Erlang file is via the command:

```
> tidier myfile.erl
```

If all goes well, this command will automatically refactor the code of `myfile.erl` and overwrite the contents of the file with the resulting source code (also leaving a backup file in the same directory). Multiple source files can also be given. Alternatively, the user can tidy a whole set of applications by a command of the form:

```
> tidier applic1/src ... applicN/src
```

which will tidy all `*.erl` files somewhere under these directories. Both of these commands will apply the default set of transformations on all files. If only some of the transformations are desired, the user can select them via appropriate command-line options. For example, one can issue the command:

```
> tidier --guards --case-simplify myfile.erl
```

to only rewrite guards to their modern counterparts (Section 3.1) and simplify all `case` expressions (Section 3.3) of `myfile.erl`. We refer the reader to tidier's manual for the complete and up-to-date set of command-line options.

A very handy option is the `-n` (or `--no-transform`) option that will cause tidier to just print on the standard output the list of transformations that would be performed on these files, together with their lines, without performing them. Alternatively the user can use the `-g` (or `--gui`) option to invoke tidier's GUI and perform refactoring interactively. We expect that novice tidier users will probably prefer this mode of using tidier, at least initially.

Let us examine tidier's GUI. Figure 1 shows tidier in action. In fact, the snapshot depicts tidier refactoring a file from the `inviso` application of Erlang/OTP R13B. Tidier has identified some code as a candidate for simplification and shows the final version of this code to its user. What the snapshot does not show is that that the simplification involves three different refactorings and that tidier has previously shown all these refactorings, one after the other, to its user. At the point when the snapshot is taken, Tidier's GUI shows the old code (on the left) and the new code (on the right); the code parts that differ between the two versions are coloured appropriately (with red color the old excerpt of the code and with green the new). At this point, the user can either press the "Use suggested version" button to accept tidier's transformation or the "Keep original version" button to bypass it. In either case, tidier will continue with the next refactoring or exit if this is the last one.

As a side comment, at some point during tidier's development we were thinking of giving the user the possibility to edit the code on the right (i.e., allowing the user to fine-tune tidier's refactorings), but we have given up on this idea as it requires dealing with too many issues which are peripheral to the main goals of tidier (e.g., how should tidier continue if the user inputs code which is syntactically erroneous, should there be an "undo" option, etc.). The user can and should better use an editor for such purposes.

3. Transformations Performed by Tidier

Let us now see the transformations that tidier performs.

3.1 Transformations inherited from `erl_tidy`

Some of tidier's transformations were inherited from the `erl_tidy` module of Erlang/OTP's `syntax_tools` application. They are all quite simple but, since they are part of tidier and the basis for our work, we briefly describe them here.

Modernizing guards

For many years now, the Erlang/OTP system has been supporting two sets of type checking functions: old-style (`atom/1`, `binary/1`, `integer/1`, ...) and new-style ones (`is_atom/1`, `is_binary/1`, `is_integer/1`, ...). All this time, the implicit recommendation has been that applications should gradually convert to using new-style guards, but not all applications have done so. Those that have not recently got one more incentive to do so: the compiler of the R13B release of Erlang/OTP stopped being silent about uses of old-style guards and generates warnings by default.

Note that the modernization of guards is both a rather tedious job for programmers and a task that cannot be automated easily. For example, it cannot be performed by a global search and replace without the programmer's full attention or by a simple `sed`-like script that does not understand what is a guard position in Erlang. Consider the following Erlang code which, although artificial and of really poor code quality, is syntactically valid. It is probably not immediately obvious to the human eye where the guard is.

```
-module(where_is_the_integer_guard).
-export([obfuscated_integer_test/1]).

obfuscated_integer_test(X) ->
    integer(X) =:= integer.

integer(X) when (X =:= infinity);
integer(X) -> integer;
integer(_) -> not_an_integer.
```

In contrast, for an automated refactoring tool like tidier, which understands Erlang syntax, the modernization of guards is a simple and straightforward task.

Eliminating explicit imports

This transformation eliminates all import statements and rewrites all calls to explicitly imported functions as remote calls as shown:

```
-import(m1, [foo/1]).
-import(m2, [bar/2]).

t(X) ->
   case foo(X) of
   ...
      bar(A, B),
   ...
```

$\Longrightarrow$

```
t(X) ->
   case m1:foo(X) of
   ...
     m2:bar(A, B),
   ...
```

Admittedly, to a large extent the *eliminating imports* refactoring is a matter of taste. Its primary goal is not to make the code shorter but to improve its readability and understandability by making clear to the eye which calls are calls to module-local functions and which are remote calls. In addition, in large code bases, it makes easier to find (e.g. using tools like Unix's `grep`) all calls to a specific `m:f` function of interest. Of course, it is possible to do the above even in files with explicit imports, but it is often more difficult.

Eliminating appends and subtracts

This is a very simple refactoring that substitutes all occurrences of calls to `lists:append/2` and `lists:subtract/2` functions with the much shorter equivalent operators `++` and `--`. The main purpose of this refactoring is to reduce the size of source code but also improve readability.

Transforming maps and filters to list comprehensions

This is a modernization refactoring that involves the transformation of `lists:map/2` and `lists:filter/2` to an equivalent list comprehension. The goals of this transformation are threefold: (a) reduce the source code size; (b) express the mapping or filtering of a list in a more elegant way and (c) increase the opportunities for further refactorings that involve list comprehensions as we will see.

Transforming fun expressions to functions

This is the Erlang analogue of the *extract method* refactoring in object oriented languages [5]. This particular refactoring removes `fun` expressions from functions and transforms them into module local functions. This transformation primarily aims at improving code readability but can also be used for detecting opportunities for clone removal as also noted by the developers of Wrangler [6].

3.2 Simple transformations

From this point on, all transformations we present are not performed by `erl_tidy`. We start with the simple ones.

Transforming coercing to exact equalities and inequations

In the beginning, the Erlang Creator was of the opinion that the only reasonable numbers were arbitrary precision integers and consequently one equality (`==`) and one inequation (`/=`) symbol were sufficient for comparing between different numbers. At a later point, it was realized that some programming tasks occasionally also need to manipulate floating point numbers and consequently Erlang was enriched by them. Most probably, because C programmers were accustomed to `==` having coercing semantics for numbers, comparison operators for exact equality (`=:=`) and inequation (`=/=`) were added to the language. These operators perform *matching* between numbers. Up to this point all is fine. The problem is that in 99% of all numeric comparisons, Erlang programmers want matching semantics but use the coercing equality and inequation operators instead, probably unaware of the distinction between them or its consequences for readability of their programs by others.

Tidier employs local type analysis to find opportunities for transforming coercing equalities and inequation with an integer to their matching counterparts. The analysis, although conservative, is often quite effective. The transformation itself is trivial.

Modernizing functions

As the Erlang language and its implementation evolve, some library functions become obsolete. These functions typically get replaced by some other function with similar functionality. Occasionally a new function which is cleaner and/or more efficient than the old one is added in the library and recommended as their replacement.

As a rather recent such example, we discuss in detail the case of the commonly used library function `lists:keysearch/3`. This function returns either a pair of the form `{value,Tuple}` or the atom `false`. Throughout the years, it was repeatedly noticed by various Erlang programmers that `Tuple` is a tuple, a whole tuple and nothing but a tuple, so wrapping it in another tuple in order to distinguish it from the atom `false` is completely unnecessary. As a result, R13 introduced the library function `lists:keyfind/3` which has the functionality of `lists:keysearch/3` but instead returns either `Tuple` or `false`. Notice that a simple function renaming refactoring and removing the `value` wrapper do *not* suffice in this case. To see this, consider the following excerpt from the code of Erlang/OTP R13B's `lib/stdlib/src/supervisor.erl:800`:

```
case lists:keysearch(Child#child.name, Pos, Res) of
  {value, _} -> {duplicate_child, Child#child.name};
  _ -> check_startspec(T, [Child|Res])
end
```

To preserve the semantics, this code should be changed to:

```
case lists:keyfind(Child#child.name, Pos, Res) of
  false -> check_startspec(T, [Child|Res]);
  _ -> {duplicate_child, Child#child.name}
end
```

and indeed this is the transformation that tidier performs, based on type information about the return values of the two functions. Moreover, notice that there are calls to `lists:keysearch/3` that *cannot* be changed to `lists:keyfind/3`. One of them, where the matching is used as an assertion, is shown below:

```
...
{value, _} = lists:keysearch(delete, 1, Query),
...
```

This particular transformation involving `lists:keysearch/3` is just one member of a wider set of similar function modernizations that are currently performed by tidier. Their purpose is to assist programmers with software maintenance and upgrades. Judging from the number of obsolete function warnings we have witnessed remaining unchanged across different releases, both in Erlang/OTP and elsewhere, it seems that in practice updating deprecated functions is a very tedious task for Erlang programmers to perform manually.

Record transformations

The *record transformations* refactoring refers to a series of record-related transformations that are performed by tidier. Detailed examples can be found in the companion paper [11] but briefly the refactoring consists of three main transformation steps: (i) converting `is_record/[2,3]` guards to clause matchings; (ii) generating fresh variables for the record fields that are used in the clause and matching them with the corresponding fields in the clause pattern; (iii) replacing record accesses in the clause body with the new field variables. Record transformations lead to shorter and cleaner code, improve code readability, may trigger further refactorings, and when applied *en masse* they can even improve performance.

3.3 Transformations that eliminate redundancy

Various refactorings specialize the code and remove redundancies.

Specializing the `size/1` function

Tidier employs this refactoring to find opportunities to specialize the `size/1` function. Since Erlang/OTP R12 there exist two new BIFs that return the size of tuples (`tuple_size/1`) and binaries (`byte_size/1`). By performing a local type analysis, tidier automatically performs this substitution whenever possible. Such a refactoring has a lot of benefits: (i) modernizes the code; (ii) makes the programmers' intentions about types clear rather than implicit; (iii) assists bug detection tools like Dialyzer [8] to detect type clashes with less effort; (iv) slightly improves the performance of programs; and (v) often triggers further simplifications.

Simplifying guard sequences

This refactoring removes redundant guards and simplifies guard sequences. Some examples are shown below (the `when` is not shown) where we have taken the liberty to combine guard simplifications with some other refactorings we have previously introduced.

```
is_list(L), length(L) > 42   ⟹   length(L) > 42
is_integer(N), N == 42       ⟹   N =:= 42
is_tuple(L), size(T) < 42    ⟹   tuple_size(T) < 42
```

Such refactorings reduce the code size (both source and object) and also improve performance.

Structure reuse

The *structure reuse* refactoring is quite similar to (and inspired from) transformations that optimizing compilers perform. Identical structures (tuples or lists) in the same clause containing fully evaluated terms (i.e., not calls) as subterms are identified by tidier and their first occurrences are assigned to fresh variables. When the identification phase is over, tidier simply replaces all subsequent occurrences of the identical structures with the new variables. This refactoring reduces the code size and also improves performance.

Straightening case expressions

We use the term *straightening* to describe the refactoring of a `case` expression to a matching statement. Such a refactoring can only be applied when the `case` expression has only one alternative clause. Tidier identifies those cases and performs this transformation provided that the body of the `case` does not contain any comments (presumably commented-out alternative `case` clauses or some message that the treatment in the `case` body is currently incomplete).

Temporary variable elimination

This is another refactoring inspired from compiler optimizations, namely from copy propagation. Temporarily storing an intermediate result in a variable to be used in the immediately following expression is actually commonplace in almost all programming languages. Tidier, by performing this refactoring, eliminates the temporary variable and replaces it with its value. This transformation, combined with the *straightening* refactoring of the previous paragraph can lead to significant simplifications. For example, consider the following fragment from the development version of Ejabberd's source code (file `src/ejabberd_c2s.erl:1951`, with one variable renamed so that the code fits here):

```
get_statustag(P) ->
  case xml:get_path_s(P, [{elem, "status"}, cdata]) of
    ShowTag -> ShowTag
  end.
```

by straightening the `case` expression and eliminating the temporary variable the code will be transformed by tidier to:

```
get_statustag(P) ->
  xml:get_path_s(P, [{elem, "status"}, cdata]).
```

However, if tidier applied this refactoring aggressively, we would end up with code 'simplifications' that would look completely unnatural and most probably would never be performed by a programmer. An example of unwanted behaviour from this refactoring is illustrated below:

```
get_results(BitStr) ->
  Tokens = get_tokens(BitStr),
  ServerInfo = get_server_info(Tokens),
  process_data(ServerInfo).
```

⇓

```
get_results(BitStr) ->
  process_data(get_server_info(get_tokens(BitStr))).
```

Since only few Erlang programmers would consider the resulting code an improvement over the original one as far as code readability is concerned, tidier does *not* perform such refactorings.

Instead, tidier performs the *temporary variable elimination* refactoring when:

- The variable that was used to store the temporary result is eventually used to return the result of a clause (as in the first example we saw).
- It is determined that such a refactoring can lead to further and more radical refactorings later on (such as the ones we will present in Section 3.4). In this case, to ensure that such refactorings are possible after the transformation, tidier has to perform a speculative analysis about the result of further refactorings after this transformation.

Simplifying expressions

While reviewing Erlang code fragments, we have come across a conglomeration of expression simplifications that could be achieved just by applying some simple transformations. Specifically, a very frequent case involved the simplification of boolean `case` and `if` expressions.

As an actual such example, the first transformation of Figure 2 shows the simplification of source code from Erlang Web (file `wparts-1.2.1/src/wtype_time.erl:177`). In this case the code will be simplified further by tidier when the `is_between/3` guard Erlang Enhancement Proposal [10] is accepted and by unfolding the `lists:all/2` call as shown in the second transformation of the same figure. This last step is not done yet.

```
is_valid_time({H1, H2, H3}) ->
  Hour = if (H1 >= 0) and (H1 < 24) -> true;
            true -> false
         end,
  Minute = if (H2 >= 0) and (H2 < 60) -> true;
              true -> false
           end,
  Sec = if (H3 >= 0) and (H3 < 60) -> true;
           true -> false
        end,
  lists:all(fun(X) -> X == true end,
            [Hour, Minute, Sec]).
```

⇓

```
is_valid_time({H1, H2, H3}) ->
  Hour = (H1 >= 0) and (H1 < 24),
  Minute = (H2 >= 0) and (H2 < 60),
  Sec = (H3 >= 0) and (H3 < 60),
  lists:all(fun (X) -> X == true end,
            [Hour, Minute, Sec]).
```

⋮

⇓

```
is_valid_time({H1, H2, H3}) ->
  Hour = is_between(H1, 0, 23),
  Minute = is_between(H2, 0, 59),
  Sec = is_between(H3, 0, 59),
  Hour andalso Minute andalso Sec.
```

Figure 2. A case of multiple `if` simplifications.

3.4 Simplification of list comprehensions

Although the list comprehension transformations that were inherited from `erl_tidy` are semantically correct, at times, the resulting code was not what an expert Erlang programmer would have written if she were transforming the code by hand. The refactorings in this section describe a series of transformations that are supported by tidier in order to improve the quality of the list comprehensions that are produced and at the same time simplify them even more by using the refactorings that were presented in the previous sections.

Fun to direct call

This is a very simple transformation. It is typically performed in conjunction with the refactoring that transforms a `fun` expression to a local function (Section 3.1), and transforms the application of a function variable to some arguments to a direct call to the local function with the same argument.

Inlining simple and boolean filtering funs

A simple `fun` within a `lists:map/2` or `lists:filter/2` which is defined by a match all clause without guards can be inlined when the `map` or `filter` call is transformed to a list comprehension. This simplifies the resulting code and simultaneously makes it more appealing and natural to the programmer's eye. We illustrate it:

```
lists:filter(fun (X) -> is_gazonk(X) end, L)
```

⇓

```
[X || X <- L, is_gazonk(X)]
```

One more case where it is possible to do a transformation similar to the above is when the `fun` is used in `lists:filter/2` and defines a *total* boolean function (i.e., a function that does not impose any constraints on its argument) as the code below (from Erlang/OTP R13B's `lib/kernel/src/pg2.erl:278`):

```
del_node_members([[Name, Pids] | T], Node) ->
  NewMembers =
    lists:filter(fun(Pid) when node(Pid) =:= Node -> false;
                    (_) -> true
                 end, Pids),
  ...
```

which tidier automatically transforms to:

```
del_node_members([[Name, Pids] | T], Node) ->
  NewMembers = [Pid || Pid <- Pids, node(Pid) =/= Node],
  ...
```

Deforestation in map+filter combinations

Some nested calls to `lists:map/2` and `lists:filter/2` are transformed to a single list comprehension by tidier, thus eliminating the intermediate list and effectively performing *deforestation* [13] at the source code level. (The companion paper [11] contains an interesting such example.) Whenever the calls to `map` and `filter` are not nested, tidier performs a speculative analysis employing the *temporary result elimination* refactoring from Section 3.3 to see if this can create further opportunities for deforestation. In either case, tidier will perform the deforestation *only* in cases it is certain that doing so will not alter the exception behaviour of the code (e.g., miss some exception that the original code generates). We will come back to this point in Section 5.

Zipping and unzipping

In general, type information (hard-coded or automatically inferred through analysis) can radically improve the resulting refactorings. For example, tidier has hard-coded information that the result of `lists:zip/2` is a list of pairs. This allows tidier to perform function inlining in cases that it would not have been possible without such information. It also prepares tidier for the possibility that *comprehension multigenerators* become part of the language.

Since tidier is treating calls to `lists:zip/2` specially, it felt natural that calls to `lists:unzip/1` would also receive special treatment. One very interesting case appears in the source code of `disco-0.2/master/src/event_server.erl:123`. We show tidier performing a non-trivial code transformation including this refactoring in Figure 3.

3.5 Transformations that reduce the complexity of programs

One of the blessings of high-level languages such as Erlang is that they allow programmers to write code for certain programming tasks with extreme ease. Unfortunately, this blessing occasionally turns into a curse: programmers with similar ease can also write code using a language construct that has the wrong complexity for the task.

Perhaps the most common demonstration of this phenomenon is unnecessarily using the `length/1` built-in function as a test. While

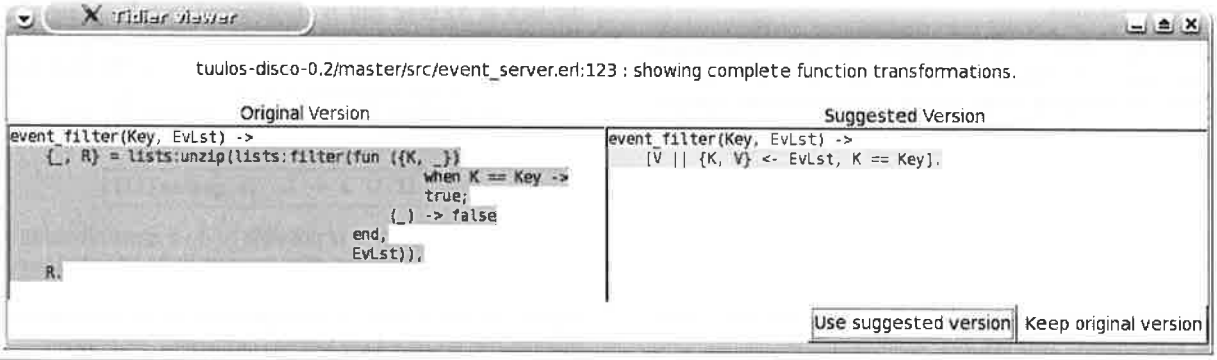

Figure 3. Tidier simplifying the code of `disco-0.2/master/src/event_server.erl`.

this is something we have witnessed functional programming novices do also in other functional languages (e.g., in ML), the situation is more acute in Erlang because Erlang allows `length/1` to also be used as a guard. While most other guards in Erlang have a constant cost and are relatively cheap to use, the cost of `length/1` is proportional to the size of its argument. Erlang programmers sometimes write code which gives the impression that they are totally ignorant of this fact.

Consider the following code excerpt from Erlang/OTP R13B's `lib/xmerl/src/xmerl_validate.erl:542`:

```
star(_Rule,XML,_,_WSa,Tree,_S) when length(XML) =:= 0 ->
  {[Tree],[]};
star(Rule,XMLs,Rules,WSaction,Tree,S) ->
  ... % recursive case of star function here ...
    star(Rule,XMLs2,Rules,WSaction,Tree++WS++[Tree1],S)
  end.
```

The use of `length/1` to check whether a list is empty is totally unnecessary; tidier will detect this and transform this code to:

```
star(_Rule,[],_,_WSa,Tree,_S) ->
  {[Tree],[]};
star(Rule,XMLs,Rules,WSaction,Tree,S) ->
  ... % recursive case of star function here ...
    star(Rule,XMLs2,Rules,WSaction,Tree++WS++[Tree1],S)
  end.
```

thereby changing the complexity of this function from quadratic to linear.

The above is not a singularity. Tidier has discovered plenty of Erlang programs which use `length` to check whether a list is empty. Occasionally some programs are not satisfied with traversing just one list to check if it is empty but traverse even more, as in the code excerpt in Figure 4. Tidier will automatically transform the two `length/1` guards to exact equalities with the empty list (e.g., `AllowedNodes =:= []`). Note that this transformation is safe to do because the two `lists:filter/2` calls which produce these lists supply tidier with enough information that the two lists will be proper and therefore the guards will not fail due to throwing some exception.

Tidier has also located a clause with three unnecessary calls to `length/1` next to each other. The code is from the latest released version of RefactorErl. Its refactoring is shown in Figure 5. Neither we nor tidier understand the comment in Hungarian, but we are pretty sure that the whole `case` statement can be written more simply as:

```
SideEffs =/= [] orelse UnKnown =/= []
                orelse DirtyFunc =/= []
```

```
choose_node({PrefNode, TaskBlackNodes}) ->
  ...
    % ..and choose the ones that are not 100% busy.
    AvailableNodes = lists:filter(fun({Node, _Load}) ->
                                    ...
                                  end, AllNodes),
    AllowedNodes = lists:filter(fun({Node, _Load}) ->
                                  ...
                                end, AvailableNodes),
    if length(AvailableNodes) == 0 -> busy;
       length(AllowedNodes) == 0 ->
         {all_bad, length(TaskNodes), length(AllNodes)};
       true ->
         % Pick the node with the lowest load.
         [{Node, _}|_] = lists:keysort(2, AllowedNodes),
         Node
    end;
  ...
```

Figure 4. Code with two unnecessary calls to `length/1` (from the code of `disco-0.2/master/src/disco_server.erl:280`).

thereby saving five lines of code (eight if one also includes the comments) and also avoiding the unnecessary tuple construction and deconstruction.

Similar cases also exist which check whether a list contains just one or more that one elements (e.g., `length(L) > 1`). Whenever relatively easy to do, tidier transforms them as in the case shown below (from the code of `lib/ssl/src/ssl_server.erl:1139`) where tidier has also eliminated the call to `hd/1` as part of the transformation.

```
decode_msg(<<_, Bin/binary>>, Format) ->
  Dec = ssl_server:dec(Format, Bin),
  if length(Dec) == 1 -> hd(Dec);
     true -> list_to_tuple(Dec)
  end.
```

⇓

```
decode_msg(<<_, Bin/binary>>, Format) ->
  Dec = ssl_server:dec(Format, Bin),
  case Dec of
    [Dec1] -> Dec1;
    _ -> list_to_tuple(Dec)
  end.
```

In some other cases though, the code also contains other guard checks which complicate the transformation. For example, consider function `splice/1` from the source code of ErlIDE (located in file `org.erlide.core/erl/pprint/erlide_pperl.erl:171`):

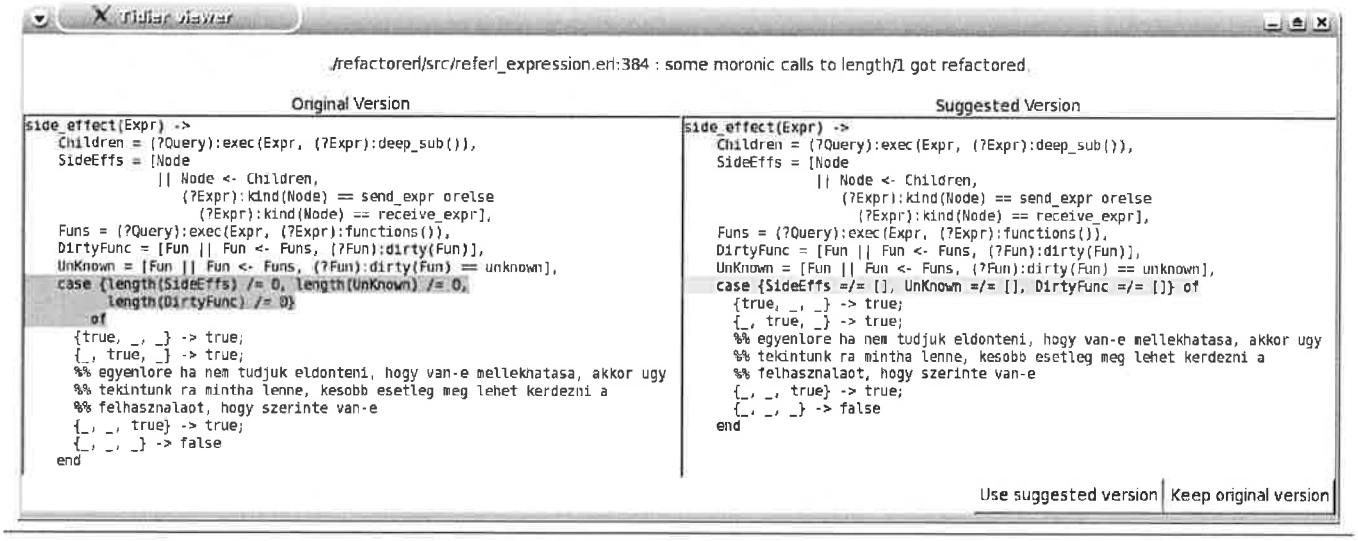

Figure 5. Tidier simplifying the code of `refactorerl-0.6/lib/refactorerl/src/referl_expression.erl`.

```
splice(L) ->
  Res = splice(L, [], []),
  case (length(Res) == 1) and is_list(hd(Res)) of
    true -> no;
    _ -> {yes, Res}
  end.
```

Automatically transforming such code to something like the following is future work:

```
splice(L) ->
  Res = splice(L, [], []),
  case Res of
    [Res1] when is_list(Res1) -> no;
    _ -> {yes, Res}
  end.
```

We intend to enhance tidier with more refactorings that detect programming idioms with wrong complexity for the task and improve programs in similar ways.

4. Effectiveness Across Applications

We have applied tidier to a considerable corpus of Erlang programs both in order to ensure that our tool can gracefully handle most Erlang code out there and in order to test its effectiveness. In this section we report our experiences and the number of opportunities for code cleanups detected by tidier on the code of the following open source projects:[1]

Erlang/OTP This system needs no introduction. We just mention that we report results on the source code of R13B totalling about 1,240,000 lines of Erlang code. Many of its applications under `lib` (e.g., `hipe`, `dialyzer`, `typer`, `stdlib`, `kernel`, `compiler`, `edoc`, and `syntax_tools`) had already been fully or partially cleaned up by tidier. Consequently, the number of opportunities for cleanups would have been even higher if such cleanups had not already taken place.

Apache CouchDB is a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API [1]. The CouchDB distribution contains ibrowse and mochiweb as components. We used release 0.9.0 which contains about 20,500 lines of Erlang code.

Disco is an implementation of the Map/Reduce framework for distributed computing [2]. We used version 0.2 of Disco. Its core is written in Erlang and consists of about 2,500 lines of code.

Ejabberd is a Jabber/XMPP instant messaging server that allows two or more people to communicate and collaborate in real-time based on typed text [3]. We used the development version of ejabberd from the public SVN repository of the project (revision 2074) consisting of about 55,000 lines of Erlang code.

Erlang Web is an open source framework for applications based on HTTP protocols [4]. Erlang Web supports both `inets` and `yaws` webservers. The source of Erlang Web (version 1.3) is about 10,000 lines of code.

RefactorErl is an refactoring tool that supports the semi-automatic refactoring of Erlang programs [9]. We used the latest release of RefactorErl (version 0.6). Its code base consists of about 24,000 lines of code.

Scalaris is a scalable, transactional, distributed key-value store which can be used for building scalable Web 2.0 services [12]. We used the development version of scalaris from the public SVN repository of the project (revision 278) consisting of about 35,000 lines of Erlang code. This includes the `contrib` directory of scalaris where the source code of Yaws [15] is also included as a component.

Wings 3D is a subdivision modeler for three-dimensional objects [14]. We used the development version of wings from the public SVN repository of the project (revision 608) consisting of about 112,000 lines of Erlang code. This includes its `contrib` directory.

Wrangler is a refactoring tool that supports the semi-automatic interactive refactoring of Erlang programs [7] within emacs or erlIDE, the Erlang plugin for Eclipse. We used the development version of Wrangler from the public SVN repository of the project (revision 678) consisting of about 42,000 lines of Erlang code.

[1] Throughout its development, we have also applied tidier to its own source code but, since we have been performing the cleanups which tidier were suggesting eagerly, we cannot include tidier in the measurements.

	lines of code	new guards	exact numeric equality	lists:keysearch/3	record matches	record accesses	size	simplifying guards	structure reuse	straighten + case simplify	map to comprehension	filter to comprehension	deforestations	zip + unzip	length
Erlang/OTP	1,240,000	2911	68	751	1805	2168	487	36	1467	77	564	115	4		12
CouchDB	20,500	22	9	8	6	27	31	2	88	3	38			1	
Disco	2,500	11	2	12		2	9		14		11	5		1	2
Ejabberd	55,000	2		78	18	26	6		70	11	134	40	2		
Erlang Web	10,000	7	11	37	1	12	1	1	15	6	35	7	1		2
RefactorErl	24,000		11	3		8			54	1	39	7		3	7
Scalaris	35,000			2	6	6			22		39	22	3		
Wings 3D	112,000	10	13	45	1	24	26		166	11	25	10			
Wrangler	42,000	6	28	141			1	1	110	7	236	47	5	14	2

Table 1. Number of tidier's transformations on various Erlang source code bases.

For all projects with SVN repositories the revisions we mention correspond to the most recent revision on the 12th of May 2009.

The number of opportunities for tidier's transformations on these code bases is shown on Table 1. From these numbers alone, it should be obvious that detecting, let alone actually performing, all these refactorings manually is an extremely strenuous and possibly also error-prone activity. Tidier, even if employed only as a detector of bad code smells, is worth the effort of typing its name on the command line.

Naturally, the number of opportunities for refactorings that tidier recognizes depends on two parameters: size and programming style of a project's code. As expected, the number of refactoring opportunities on the Erlang/OTP system is much bigger in absolute terms than on all the other code bases combined. This is probably due to the size of the code base and probably also due to the fact that some applications of Erlang/OTP were developed by many different programmers, often Erlang old-timers, over a period of years. But we can also notice that it's not only code size that matters. The table also shows smaller code bases offering more opportunities for refactoring than code bases of bigger size.

What Table 1 does not show is tidier's effectiveness. For some columns of the table (e.g., new guards, record matches) tidier's effectiveness is 100% by construction, meaning that tidier will detect all opportunities for these refactorings and perform them if requested to do so. For some other columns of the table (e.g., `lists:keysearch/3`, `map` and `filter` to list comprehension, structure reuse, `case` simplify) tidier can detect all opportunities for these refactorings but might not perform them based on heuristics which try to guess the intentions of programmers or take aesthetic aspects of code into account. For some refactorings, especially those for which type information is required, tidier's effectiveness is currently not as good as we would want it to be. (We will come back to this point in the next section.)

Table 2 contains numbers and percentages of numeric comparisons with `==` and `/=` that are transformed to their exact counterparts and numbers and percentages of calls to `size/1` that get transformed to `byte_size/1` or `tuple_size/1`. As can be seen, tidier's current analysis is pretty effective in detecting opportunities of transforming calls to `size/1` but quite ineffective when it comes to detecting opportunities for transforming coercing equalities and inequations. A global type analysis would definitely improve the situation in this case. (However, bear in mind that achieving 100% on all programs is impossible since there are uses of `==/2` or `size/1` that cannot be transformed to something else, even if tidier were guided by an oracle.)

	exact num. eq.	size
Erlang/OTP	68 / 577 = 12%	487 / 645 = 75%
CouchDB	9 / 15 = 60%	31 / 64 = 48%
Disco	2 / 11 = 18%	9 / 9 = 100%
Ejabberd		6 / 11 = 55%
Erlang Web	11 / 15 = 73%	1 / 1 = 100%
RefactorErl	11 / 35 = 31%	
Scalaris		5 / 6 = 83%
Wings 3D	13 / 46 = 28%	
Wrangler	28 / 54 = 52%	1 / 1 = 100%

Table 2. Effectiveness of tidier' refactorings requiring type info.

5. Conservatism of Refactorings

Despite the significant number of refactorings that tidier performs on existing code bases, we stress again that tidier is currently ultra conservative and careful to respect the operational semantics of Erlang. In particular, tidier will never miss an exception that programs may generate, whether deliberately or not.

To understand the exact consequences of this, we show a case from the code of `lib/edoc/src/otpsgml_layout.erl:148` from Erlang/OTP R13B. The code on that line reads:

```
Functions = [E || E <- get_content(functions, Es)],
```

Although to a human reader it is pretty clear that this code is totally redundant and the result of sloppy code evolution from similar code (actually from the code of `lib/edoc/src/edoc_layout.erl`), tidier *cannot* simplify this code to:

```
Functions = get_content(functions, Es),
```

because this transformation will shut off an exception in case function `get_content/2` returns something other than a proper list. To do this transformation, type information about the result of `get_content/2` is required. Currently, tidier is guided only by a function-local type analysis. Extending this analysis to the module level is future work.

Type information can also come in very handy in rewriting calls to `lists:map/2` and `lists:filter/2` to more succinct list comprehensions. Without type information, tidier performs the following transformation:

```
foo(Ps) -> lists:map(fun ({X,Y}) -> X + Y end, Ps).
```

⇓

```
foo(Ps) -> [foo_1(P) || P <- Ps].

foo_1({X,Y}) -> X + Y.
```

and cannot inline the body of the auxiliary function and generate the following code:

```
foo(Ps) -> [X + Y || {X,Y} <- Ps].
```

because this better refactoring requires definite knowledge that `Ps` is a list of pairs. Similar issues exist for refactorings involving `lists:filter/2`. Despite being conservative, tidier is pretty effective. In the code of Erlang/OTP R13B, out of the 679 refactorings of `lists:map/2` and `lists:filter/2` to list comprehensions a bit more than half of them (347) actually use the inlined translation.

We mentioned that tidier currently performs deforestation for combinations of `map` and `filter`. A similar deforestation of `map+map` combinations, namely the transformation:

```
L1 = lists:map(fun (X) -> m1:foo(X) end, L0),
L2 = lists:map(fun (X) -> m2:bar(X) end, L1)
```

⇓

```
L2 = [m2:bar(m1:foo(X)) || X <- L0]
```

as also shown in the arrow is *not* performed by tidier because this requires an analysis which determines that functions `m1:foo/1` and `m2:bar/1` are side-effect free. Again, hooking tidier to such an analysis is future work.

6. Concluding Remarks

This paper described opportunities for automatically modernizing Erlang applications, cleaning them up, eliminating certain bad smells from their code, and occasionally also improving their performance. In addition, we presented concrete examples of code improvements and our experiences from using tidier on code bases of significant size.

As mentioned, tidier is completely automatic as a refactorer but with equal ease can be used as a detector of opportunities for code cleanups and simplifications. Tools that aid software development, such as code refactorers, have their place in all languages, but it appears that higher-level languages such as Erlang are particularly suited for making the cleanup process fully or mostly automatic. We intend to explore this issue more.

Acknowledgements

We thank Richard Carlsson, Björn Gustavsson, and Kenneth Lundin for supportive comments and suggestions for refactorings. We also thank Dan Gudmundsson: without the use of his `wx` application, the user interface of tidier would have taken longer to write and would probably look less aesthetically pleasing.

Finally, we thank all developers of projects mentioned in this paper for publicly releasing their code as open source and giving us plenty of opportunities to find nice examples for our paper.

References

[1] The CouchDB project, 2009. `http://couchdb.apache.org/`.

[2] Disco: Massive data, minimal code (version 0.2), Apr. 2009. `http://discoproject.org/`.

[3] Ejabberd community site: The Erlang Jabber/XMPP daemon, 2009. `http://www.ejabberd.im/`.

[4] Erlang Web, May 2009. `http://www.erlang-web.org/`.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, Massachusetts, 2001.

[6] H. Li and S. Thompson. Clone detection and removal for Erlang/OTP within a refactoring environment. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 169–177, New York, NY, USA, Jan. 2009. ACM.

[7] H. Li, S. Thompson, G. Orösz, and M. Tóth. Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 61–72, New York, NY, USA, Sept. 2008. ACM.

[8] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In C. Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, Nov. 2004.

[9] L. Lövei, Cs. Hoch, H. Köllő, T. Nagy, A. Nagyné-Víg, D. Horpácsi, R. Kitlei, and R. Király. Refactoring module structure. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 83–89, New York, NY, USA, Sept. 2008. ACM.

[10] R. A. O'Keefe. Erlang Enhancement Proposal: `is_between/3`, July 2008. `http://www.erlang.org/eeps/eep-0016.html`.

[11] K. Sagonas and T. Avgerinos. Automatic refactoring of Erlang programs. In *Proceedings of the Eleventh International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, New York, NY, USA, Sept. 2009. ACM.

[12] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: Reliable transactional P2P key/value store. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 41–48, New York, NY, USA, Sept. 2008. ACM.

[13] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Comput. Sci.*, 73(2):231–248, 1990.

[14] Wings 3D, 2009. `http://www.wings3d.com/`.

[15] Yaws: Yet another web server, 2009. `http://yaws.hyber.org/`.

Automated Module Interface Upgrade *

László Lövei
Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
lovei@inf.elte.hu

Abstract

During the lifetime of a software product the interface of some used library modules might change in such a way that the new interface is no longer compatible with the old one. This paper proposes a generic interface migration schema to automatically transform the software in the case of such an incompatible change. The solution is based on refactoring techniques and data flow analysis, and makes use of a formal description of the differences between the old and the new interfaces. The approach is illustrated with a real-life example.

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

General Terms Design, Languages

Keywords Erlang, module interface change, interface transformation, refactoring

1. Introduction

Library modules may evolve during the lifetime of a software product. The usual approach is to keep library interfaces compatible with previous versions, but sometimes it makes sense to develop a new, richer, but incompatible interface for a library, which provides enhanced functionality, and gets rid of obsolete features. This paper proposes an approach to transform the code of a software product when such an incompatible interface change occurs. Our assumption is that the new interface retains the functionality of the old one, but makes it available in a different way. We show that by defining a generic interface migration schema, refactoring techniques combined with data flow analysis can help the automatic adaptation of a software product to an upgraded library interface.

Our approach will be illustrated by the real-life case of migrating from the Erlang/OTP `regexp` library to the novel, Perl compatible regular expression library, `re`. One of the interesting points is that characters are indexed from 1 in `regexp`, but from 0 in `re`.

RefactorErl [3] is a refactoring tool for Erlang. It turns out that this tool provides a convenient infrastructure to develop program transformations supporting our proposed generic interface migration schema. RefactorErl represents Erlang programs as *program graphs*. A program graph contains lexical, syntactic and semantic nodes and edges. The AST of the represented program is a subgraph of the program graph. Technically, the program graph has nodes labelled with the non-terminals of the abstract syntax, nodes that correspond to tokens either explicitly present in the represented program or generated by macro expansion, and nodes that describe semantical information, such as the binding structure of variables. The edges of the program graph are directed, labelled, and for each node the outgoing syntactic edges are ordered (as in an AST). RefactorErl supports a powerful query language to collect information by traversing the program graph, syntax based transformations (transformation that manipulate the AST) and semantic analysis plug-ins that automatically restore the consistency of the semantic information in a program graph after a transformation.

Our proposal to the interface migration problem is the following. First the differences between the old and the new interface have to be described with *change descriptions* written in an Erlang-like language. Then the program to be transformed is analyzed – in this paper we assume that the relevant syntactic and semantic information such as the binding structure or the function call graph is already available, e.g. by using the facilities of RefactorErl. Our goal is to find those parts of the program where transformations are the best to perform, e.g. where the transformations result in the smallest degradation of readability of the code. This is achieved by finding (using data flow analysis) the expressions affected by calls to functions from the library to be upgraded, and determining the very last points in the data flow where the necessary transformations can be safely applied.

The rest of the paper is structured as follows. In Section 2 a motivating example is given (the upgrade of the regular expression library). Section 3 introduces an abstract model of Erlang in which the interface migration problem can be studied. Section 4 describes how the necessary data flow analysis proceeds. Section 5 explains the way transformations are performed. Section 6 sketches some implementation issues. Section 7 presents related work, and Section 8 concludes the paper.

2. Interface upgrade example

In the following, we will use a real-life motivating example. Erlang/OTP contains an old regular expression library, called `regexp`, which has been made obsolete by a new, Perl compatible regular expression library called `re`. The new library covers almost every functionality of the old one, but there are small incompatibilities between them. For example, characters are indexed starting with 1 in the old library, and starting with 0 in the new one.

Here we show three old interface functions, and explain the details of migrating them to the new interface. The rest of the old interface is either compatible with the new one or has the same issues that are shown here. For details, refer to [10].

* Supported by TECH_08_A2-SZOMIN08, ELTE IKKK, and Ericsson Hungary

Erlang '09, September 5, 2009, Edinburgh, Scotland, UK.

2.1 The *match* function

The `regexp:match` function finds the first, longest match of a regular expression in a string. A simple example how this function is used:

```
case regexp:match(Str, RE) of
   {match, Start, Len} ->
      strings:substr(Str, Start, Len);
   nomatch ->
      ""
end
```

The function returns the starting index and the length of the match, or the atom `nomatch`. The same functionality is available using the `re:run` function, but its interface has a number of incompatibilities:

- By default, the return value contains the whole match and every subpattern match. This can be turned off with an option.
- The return value contains a list of match tuples, which always has a length of one after turning off subpattern matches.
- `regexp:match` starts indexing at one, but `re:run` starts indexing at zero.

These are small changes that can be compensated easily in the example by passing the necessary options to `re:run`, rewriting the pattern to the new structure, and adding 1 to the returned index:

```
case re:run(Str, RE, [{capture,first}]) of
   {match, [{Start, Len}]} ->
      strings:substr(Str, Start+1, Len);
   nomatch ->
      ""
end
```

Unfortunately, the return value of the `regexp:match` function can be used in other ways as well, which may require more complicated compensations. For example, the return value may be returned directly from a function, and in turn, used in many places:

```
word(Str) ->
   regexp:match(Str, "[a-zA-Z]+").

read_word() ->
   Line = read_line(),
   {match, S, L} = word(Line),
   string:substr(Line, S, L).

word_len(Str) ->
   {match, _, Len} = word(Str),
   Len.
```

In this example, the desired solution is to modify the pattern matching constructs used in functions `read_word` and `word_len`, as opposed to converting the return value of function `word`. This example involves only function calls, but generally data flow analysis has to be applied to find every code part that uses the return value of `regexp:match`.

Another possibility is that the return value is used in an expression that is not a pattern matching construct. There are some cases that can be handled in a specific way, for example `regexp:match(S,R) /= nomatch` can simply be rewritten to `re:run(S,R) /= nomatch`.

But in worse cases, the function call itself must be replaced with a `case` construct that restores the original return value. In general, this should be avoided as it leads to illegible code. For example, `tuple_to_list(regexp:match(S,R))` would look like this after such a transformation:

```
tuple_to_list(
  case re:run(Str, RE, [{capture,first}]) of
     {match, [{S, L}]} -> {match, S, L};
     nomatch -> nomatch
  end)
```

2.2 The *matches* function

The `regexp:matches` function finds all non-overlapping matches of a regular expression in a string. This functionality is provided by the same `re:run` function as previously, just the `global` option should also be passed to specify global matching. The basic problems and solutions are the same as in the previous case, but there is a complication: the return value contains a list that should be updated element-by-element. Also, in case of no match the old function returns an empty list, while the new one uses the `nomatch` atom.

A typical example of usage:

```
case regexp:matches(Str, RE) of
   {match, []} ->
      throw(no_match);
   {match, Matches} ->
      [string:substr(Str, S, L) ||
         {S, L} <- Matches]
end
```

The equivalent of this expression is easy to construct by hand, but automated transformation is much harder even in this simple example:

```
case re:run(Str, RE, [global,{capture,first}]) of
   nomatch ->
      throw(no_match);
   {match, Matches} ->
      [string:substr(Str, S+1, L) ||
         [{S, L}] <- Matches]
end
```

Data flow analysis is necessary to find the pattern `{S,L}` in the list comprehension that must be turned into a one-element list, and the usage of `S` that must be incremented. As a list is returned, which does not have a fixed length, a function call or a list comprehension is very likely to be used together with this function, so identifying where the list and its elements are used is essential here.

2.3 The *gsub* function

The `regexp:gsub` function substitutes every substring matching a regular expression in a string. The `re:replace` function can do the same (some options are needed), but this time the new return value contains less information than the old one: instead of a tuple, only the result of the substitution is returned, and the number of replacements provided by the old interface is completely missing from the new one. Fortunately it is rarely needed (that's why it has been removed), so a typical example is not too hard to migrate:

```
{ok, Result, _Count} = regexp:gsub(Str, RE, Repl)
```

If there are no references later to variable `_Count`, the same code with module `re`:

```
Result = re:replace(Str, RE, Repl,
                    [{return, list}, global])
```

If the missing value is used in the code, there is no generic solution for the migration, such a code must be rewritten by hand. Our task here is to detect this situation, and notify the user without actually changing the code.

V ::= variables (including _, the underscore pattern)
A ::= atoms
I ::= integers
$K ::= A \mid I \mid$ other constants (e.g. strings, floats)
$P ::= K \mid V \mid \{P,\ldots,P\} \mid [P,\ldots,P \mid P]$
$E ::= K \mid V \mid \{E,\ldots,E\} \mid [E,\ldots,E \mid E] \mid [E \mid\mid P \texttt{<-} E] \mid$
P `=` $E \mid E \circ E \mid (E) \mid E(E,\ldots,E) \mid$
`case` E `of`
P `->` $E,\ldots,E$;
$\vdots$
P `->` $E,\ldots,E$
`end` |
`fun`
$(P,\ldots,P)$ `->` $E,\ldots,E$;
$\vdots$
$(P,\ldots,P)$ `->` $E,\ldots,E$
`end`
$F ::= A(P,\ldots,P)$ `->` $E,\ldots,E$;
$\vdots$
$A(P,\ldots,P)$ `->` $E,\ldots,E$.

Figure 1. The used Erlang syntax subset

3. Model of Erlang programs

In the following discussion we do not consider the whole Erlang language. For the sake of simplicity, we omit some of the language elements, but the framework described here is capable of supporting the full Erlang syntax. The static graph construction rules of the missing syntactic constructs are easily added to those in Fig. 2, and no other additions are needed to support them.

The syntax of the Erlang subset that we use is defined in Fig. 1 is used. The simplifications are the following:

- Irrelevant language constructs are left out, for example, the results are completely independent of the module structure and the attributes of modules. Instead of a set of modules, we consider Erlang programs that consist of a set of named function definitions (F in Fig. 1).
- There are missing expression types which can be handled in the same way as one of the presented expressions, like records, which are analogous to tuples indexed with field names.
- The syntax contains further simplifications in the remaining expression types, for example, there are no guard expressions, because they would not be handled differently that other expressions.
- Finally, there are language elements that could be considered relevant, but analysis of industrial code shows that they are so rarely used in conjunction with the problem stated here that it does not pay off to support them. These include processes, message passing, and exceptions, as all of them would require control flow analysis in addition to the data flow analysis presented here.

We assume that the code to be transformed is available as an abstract syntax tree. The nodes of the syntax tree represent one of the above rules, they are uniquely identified, and attached to the corresponding source code part. Transformations will be expressed as rewriting parts of the syntax tree by providing the replacement structure for some of the nodes. This model is not elaborated further here, RefactorErl [3] and Wrangler [5] is capable of doing such transformations.

3.1 Semantic information

Information not available directly from the abstract syntax tree is also necessary for the analyses and transformations presented here. The information is based on standard Erlang semantics [1], so the exact definition of these concepts is omitted here. The presentation is also independent of how this information is represented or computed.

Internal function calls: for every function call expression that uses a constant function name, we need to know if the program contains the definition of the referred function. In this case, we call this an *internal function call*. The semantic model must make the function definition accessible.

External function calls: function calls that use a constant function name, but there is no corresponding function definition in the program, are called *external function calls*. These include built in functions (some of which are handled specially) and functions defined in libraries, but program parts which should not be modified during the transformation fall into this category too.

Variable bindings: every variable in a program must have at least one occurrence that binds the variable, and it is probably used elsewhere in the code, although there may be more bindings, and there may be no usages. Bindings are always patterns, but some patterns only use the value of the variable; expressions may only use the variable. Variables have a name, but there may be different variables with the same name in different scopes; they must be distinguished. The needed semantic information is the set of bindings and usages for any given variable occurrence.

4. Data flow graph

Data flow analysis is used to find expressions and patterns that are affected by changing a particular data in the program. The goal of data flow analysis is to inspect possible paths where data can travel during the execution of the program, and follow these paths to see what expressions can a particular data reach. The possible data flow paths are represented by a *data flow graph*.

The nodes of the graph ($n \in \mathcal{N} = \mathcal{E} \cup \mathcal{P}$) are the expressions ($e \in \mathcal{E}$) and patterns ($p \in \mathcal{P}$) of the program. Its directed edges represent single steps of data transfer. There are different kinds of data transfers, they are identified by edge labels. We use the notation $n_1 \xrightarrow{l} n_2$ for an edge from node n_1 to node n_2 with label l.

4.1 Direct flow information

Data flow graph edges represent direct data flow, later on we will derive information from these edges by traversing the direct graph. Data flow usually means copying data; when operations are used on data, it cannot be followed in general. In our interface upgrade problem when a modified data is used in a non-copying operation, we must convert it to the original value to preserve the original meaning of the program. The only exception is packing data into a compound term and later unpacking it: such an operation preserves the original data, so we can think of it as simple copying.

Based on these principles, the following kinds of edges are used in the graph to distinguish between different types of data flow steps:

Flow edges: $n_1 \xrightarrow{f} n_2$ means that the result of n_2 can be a copy of the result of n_1.

Constructor edges: $n_1 \xrightarrow{c_i} n_2$ means that the result of n_2 can be a compound value that contains n_1 at element i. In case of tuples, element labels are natural numbers, meaning the i^{th}

	Expressions	Direct graph edges
(a)	p is a binding n is a usage of the same variable	$p \xrightarrow{f} n$
(b)	e_0: $p = e$	$e \xrightarrow{f} p$ $e \xrightarrow{f} e_0$
(c)	p_0: $p_1 = p_2$	$p_0 \xrightarrow{f} p_1$ $p_0 \xrightarrow{f} p_2$
(d)	e_0: $e_1 \circ e_2$	$e_1 \xrightarrow{d} e_0$ $e_2 \xrightarrow{d} e_0$
(e)	e_0: (e)	$e \xrightarrow{f} e_0$
(f)	e_0: $\{e_1, \ldots, e_n\}$	$e_1 \xrightarrow{c_1} e_0, \ldots, e_n \xrightarrow{c_n} e_0$
(g)	p_0: $\{p_1, \ldots, p_n\}$	$p_0 \xrightarrow{s_1} p_1, \ldots, p_0 \xrightarrow{s_n} p_n$
(h)	e_0: $[e_1, \ldots, e_n \mid e_{n+1}]$	$e_1 \xrightarrow{c_e} e_0, \ldots, e_n \xrightarrow{c_e} e_0$ $e_{n+1} \xrightarrow{f} e_0$
(i)	e_0: $[e_1 \mid\mid p \leftarrow e_2]$	$e_1 \xrightarrow{c_e} e_0$ $e_2 \xrightarrow{s_e} p$
(j)	p_0: $[p_1, \ldots, p_n \mid p_{n+1}]$	$p_0 \xrightarrow{s_e} p_1, \ldots, p_0 \xrightarrow{s_e} p_n$ $p_0 \xrightarrow{f} p_{n+1}$
(k)	e_0: hd(e_1)	$e_0 \xrightarrow{s_e} e_1$

	Expressions	Direct graph edges
(l)	e_0: tl(e_1)	$e_0 \xrightarrow{f} e_1$
(m)	I is constant, e_0: element(I, e_1)	$e_0 \xrightarrow{s_I} e_1$
(n)	e_0: case e of $p_1 \rightarrow e_1^1, \ldots, e_{l_1}^1$; $\vdots$ $p_n \rightarrow e_1^n, \ldots, e_{l_n}^n$; end	$e \xrightarrow{f} p_1, \ldots, e \xrightarrow{f} p_n$ $e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_n}^n \xrightarrow{f} e_0$
(o)	e_0: f$(e_1, \ldots, e_n)$ f/n: f$(p_1^1, \ldots, p_n^1) \rightarrow$ $e_1^1, \ldots, e_{l_1}^1$; $\vdots$ f$(p_1^m, \ldots, p_n^m) \rightarrow$ $e_1^m, \ldots, e_{l_m}^m$.	$e_1 \xrightarrow{f} p_1^1, \ldots, e_1 \xrightarrow{f} p_1^m$ $\vdots$ $e_n \xrightarrow{f} p_n^1, \ldots, e_n \xrightarrow{f} p_n^m$ $e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$
(p)	e_0: $e(e_1, \ldots, e_n)$ e is not constant, or e/n undefined	$e_1 \xrightarrow{d} e_0, \ldots, e_n \xrightarrow{d} e_0$

Figure 2. Static data flow edge generation rules

element of the tuple. In case of lists, element label e is used for list elements (we cannot usually track their indexes).

Selector edges: $n_1 \xrightarrow{s_i} n_2$ means that the result of n_2 can be element i of n_1. Element labels have the same meaning as in constructor edges.

Dependency edges: $n_1 \xrightarrow{d} n_2$ means that the result of n_2 can directly depend on the value of n_1. This edge label is used when data flow cannot be followed, but data usage must be represented in the graph.

4.2 Static graph building rules

Most edges of the full data flow graph can be constructed based on the syntax tree and static semantic information. Every expression has a set of associated data flow edges, and the static graph is constructed by taking all of the edges generated by the expressions of the program.

In the following, graph edge construction rules are explained for every expression type. The summary of these rules is shown in Fig. 2.

Variable occurrences. Variable patterns in Erlang may bind a value to the variable, or may use the value; variable expressions always use the value. The value of a variable never changes during its life. This means that the only kind of data flow through a variable is copying the value from its binding occurrences to its usages. This is represented by creating flow edges from every binding occurrence to every usage occurrence (Fig. 2.a).

Expression results. There are expressions that return some of their subexpression's results without modifying it. An example is the `case` expression, which executes one of its clauses, and returns the result of the last expression from that clause. This is represented by flow edges from the last expressions of the clauses to the `case` expression itself (Fig. 2.n). Similar flow edges appear for match expressions and parentheses (Figs. 2.b and 2.e), except that there are no multiple clauses.

An alias pattern, which has the same syntax as a match expression, is a special construction that flows data in the opposite direction. Such a pattern means that two patterns are matched on the same value at the same time, which is represented by flow edges from the alias pattern to its subpatterns (Fig. 2.c).

Pattern matching expressions. During pattern matching, data is compared with a pattern, and free variables in the pattern are bound to values. Data is simply copied, which is represented by flow edges from the matched expression to every matched pattern. In case of simple match expressions there is only a single pattern (Fig. 2.b), but `case` expressions use multiple patterns, and each of them gets a flow edge (Fig. 2.n).

Compound data handling. Compound data structures (tuples and lists) preserve data, so putting a value in a tuple, copying the tuple, selecting the value from the tuple is the same as copying the value. To detect these copies, we use constructor and selector edges to connect the flow of embedded data and the flow of its container.

Constructor and selector edges are indexed to make a note of the position where the embedded data is stored. Only constructors and selectors with the same index are considered to be part of the same data flow. Tuple indexes are natural numbers, which enables a precise tracking of tuple elements. Typical use of lists makes tracking of list element indexes useless, so in this case only list elements are distinguished. The "tail" of a list do not need special handling, as almost every element of a list appears in its tail; approximating this by representing a list tail as a copy of the original list is good enough for our purposes.

Patterns are selectors, so they generate selector edges to their subexpressions (Figs. 2.g and 2.j). Expressions with the same syntax are constructor expressions with constructor edges (Figs. 2.f

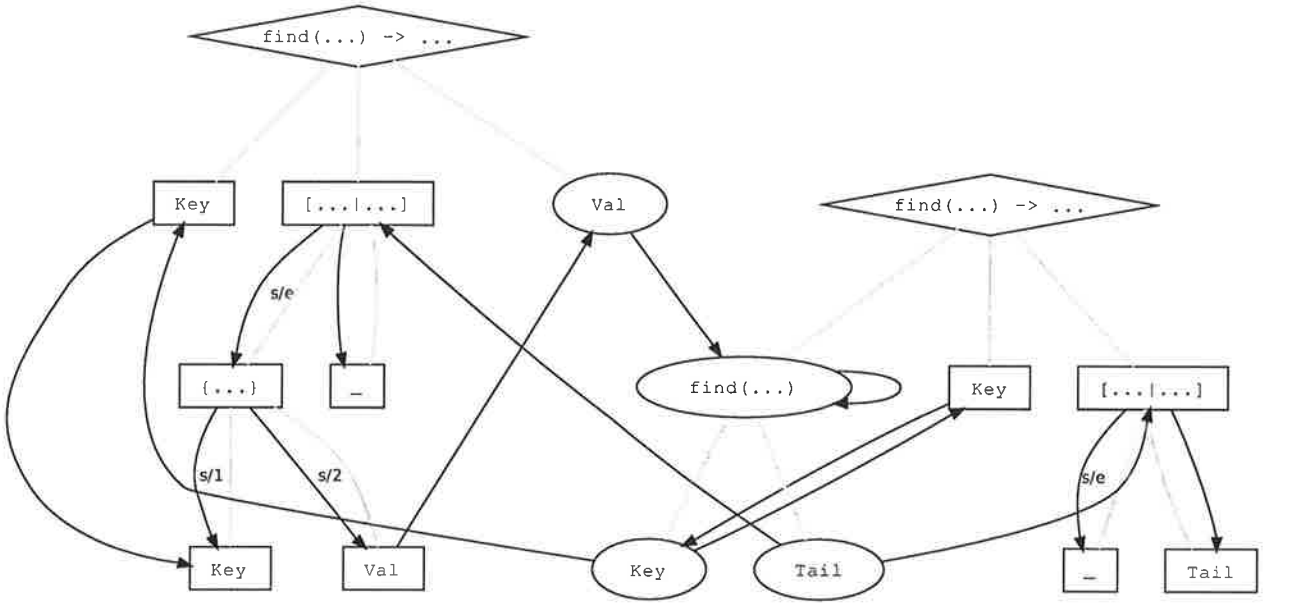

Figure 3. Data flow graph example.

and 2.h). Selector expressions are built-in functions, they generate selector edges (Figs. 2.m, 2.k, and 2.l).

List comprehensions are also selecting and constructing lists. The expression that provides the list values has a list element constructor edge to the whole list comprehension expression. The generator part, which means a pattern match for every element on its list expressions, produces a list element selector edge from the generator list expression to the generator pattern (Fig. 2.i).

Function calls. We have two essentially distinct case of function calls. First, calls to external functions cannot be represented in the flow graph, but they must be taken into account: such a construct means that data can get out of our hands, and it can be used in program parts that we cannot transform. We represent this situation by dependency edges from the function arguments to the function call expression (Fig. 2.p). Such edges signal a dead end, where data flow continues, but cannot be followed (as opposed to a node with no edges starting from it, which means there is no further data flow from that node). The same representation is used for expressions (usually operators) that do some computation with their arguments: data flow continues, but in an undefined way (Fig. 2.d).

On the other hand, functions with a known definition should use that definition in the flow graph. We don't even want to know that there is a function call involved, so the representation is very similar to a `case` expression, except that there are multiple arguments, therefore multiple pattern matches. Every clause of the function is considered, arguments from the function call are matched on the patterns in the function definition, and the possible results of the function are returned to the call expression (Fig. 2.o).

Example. As an illustration of these graph building rules, the data flow graph of the following simple function is presented in Fig. 3:

```
find(Key, [{Key, Val}|_]) -> Val;
find(Key, [_|Tail]) -> find(Key, Tail).
```

The syntax tree of the function is represented by grey lines. The diamond-shaped nodes are the function's clauses, these are not part of the data flow graph. Boxes are patterns, ovals are expressions, and the black arrows are data flow edges. Most of the edges are labelled with `f`, these labels are omitted from the drawing to make it clearer.

Rules applied to create this graph include variable usage rules, element selection for list and tuple patterns, and a function call rule. The last one is responsible for the edges between the two syntax tree parts and the looping edges that appear because of the recursive call.

4.3 Derived flow information

The static data flow graph represents only a part of every possible data flow in a program, because there are many dynamic constructs in Erlang. However, many of these constructs' data flow information can be approximated starting from the static graph.

Here we show the basic techniques to derive useful data flow information from the direct data flow graph. This information can be used to approximate the behaviour of some dynamic language constructs. Furthermore, the same techniques will be used to define the interface upgrade transformation.

4.3.1 Derivation techniques

The so-called "techniques" of this section are in fact simple sets or relations which are useful, and easy to calculate based on the flow graph.

Reaching. The most important relation we will use describes which nodes are connected by multiple-step data flow paths. We say that the value of node n_1 *reaches* node n_2, if during the evaluation of the program a value associated with n_1 may be passed to n_2. The notation $n_1 \rightsquigarrow n_2$ will be used for this concept.

The $\rightsquigarrow$ relation relation can be computed from the direct flow graph by following paths set out by edges. Simple flow edges are simply followed; dependency edges break the data flow. Constructor and selector edges are handled specially: when a data is packed into a structure, the compound data is tracked, and the corresponding unpacked nodes (pointed by selector edges with the same index as the constructor edge) are the next steps on the path. This computation involves graph traversals on f-labelled edges, executed recursively on c_i-labelled edges.

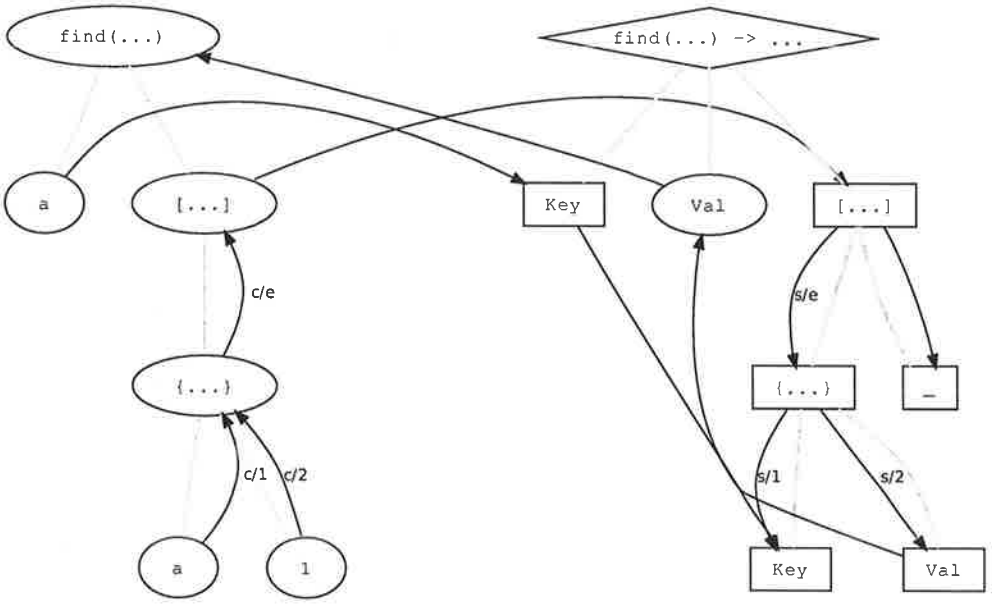

Figure 4. Data flow graph example.

The explanation above can be formulated by defining $\rightsquigarrow$ as the minimal relation that satisfies the following rules:

1. $n \rightsquigarrow n$
2. $n_1 \rightsquigarrow n_2 \wedge n_2 \xrightarrow{f} n_3 \Rightarrow n_1 \rightsquigarrow n_3$
3. $n_1 \xrightarrow{c_i} n_2 \wedge n_2 \rightsquigarrow n_3 \wedge n_3 \xrightarrow{s_i} n_4 \Rightarrow n_1 \rightsquigarrow n_4$

Example. Fig. 4 shows the graph of the first clause of the previous example's `find` function and expression `find(a, [{a, 1}])`. Using the above reaching definition, we can deduce that the value of expression `1` can be returned from the `find` function call:

1. $[\{\mathsf{a},1\}] \xrightarrow{f} [\{\mathsf{Key},\mathsf{Val}\},\_] \Rightarrow [\{\mathsf{a},1\}] \rightsquigarrow [\{\mathsf{Key},\mathsf{Val}\},\_]$
2. $\{\mathsf{a},1\} \xrightarrow{c_e} [\{\mathsf{a},1\}] \rightsquigarrow [\{\mathsf{Key},\mathsf{Val}\},\_] \xrightarrow{s_e} \{\mathsf{Key},\mathsf{Val}\}$
   $\Rightarrow \{\mathsf{a},1\} \rightsquigarrow \{\mathsf{Key},\mathsf{Val}\}$
3. $1 \xrightarrow{c_2} \{\mathsf{a},1\} \rightsquigarrow \{\mathsf{Key},\mathsf{Val}^p\} \xrightarrow{s_2} \mathsf{Val}^p \Rightarrow 1 \rightsquigarrow \mathsf{Val}^p$
4. $1 \rightsquigarrow \mathsf{Val}^p \xrightarrow{f} \mathsf{Val}^e \Rightarrow 1 \rightsquigarrow \mathsf{Val}^e$
5. $1 \rightsquigarrow \mathsf{Val}^e \xrightarrow{f} \mathsf{find}(...) \Rightarrow 1 \rightsquigarrow \mathsf{find}(...)$

One-step flow information. The following simple notation will be useful in other definitions. Starting from a given node, they describe the next (or previous) nodes directly accessible with a given edge type in the flow graph:

$$\begin{aligned} \mathrm{next}(l,n) &= \{n_n \mid n \xrightarrow{l} n_n\} \\ \mathrm{prev}(l,n) &= \{n_p \mid n_p \xrightarrow{l} n\} \end{aligned}$$

The input set of a node contains its direct predecessors on copying flow paths, taking data constructions into account. It's like taking one step backward on the $\rightsquigarrow$ relation.

$$\begin{aligned} \mathrm{input}(n) &= \mathrm{prev}(f,n) \cup \\ &\quad \{n_1 \mid n_1 \xrightarrow{c_i} n_2 \wedge n_2 \rightsquigarrow n_3 \wedge n_3 \xrightarrow{s_i} n\} \end{aligned}$$

4.3.2 Dynamic graph building

Data flow graph building rules introduced so far have all been static rules, which means they don't depend on run time data values, only on syntactic structures which are available at compile time. While our experiences show that these rules cover most of the interface upgrading situations which appear in practise, it should be pointed out that this data flow graph can be expanded to support some dynamic construct as well.

Our goal with this work is to create a simple framework for data flow analysis to support refactoring, and introducing sophisticated control flow analysis is not desired here. The static data flow graph itself contains enough information to make data flow analysis of some dynamic constructs possible.

Constant propagation. The static direct data flow graph can be iteratively extended with direct, but dynamic flow edges. Dynamic constructs use run-time values to control data flow, but in some cases these values can be derived from constants in the code.

The set of possible origins of an expression's value can be computed using the data flow graph, as we have all the static data flow paths:

$$\mathrm{source}(n) = \{n_1 \mid n_1 \rightsquigarrow n \wedge \nexists n_2 : n_2 \neq n_1 \wedge n_2 \rightsquigarrow n_1\}$$

This set may contain patterns, which means the information is not complete, or expressions that compute the value in any way. When $\mathrm{source}(n)$ contains only constant expressions, we have the finite set of the possible values of n.

Dynamic control flow. A dynamic function call is a function application where the function name is not an atom constant. In this case, the expression in place of the name must evaluate to a function object. If we have a dynamic function call $e_0(e_1, \ldots, e_n)$, then $\mathrm{source}(e_0)$ contains the function expressions that provide that function objects that may be called. This means that the same graph edges can be entered into the graph as in case of a static function call, using the function expression instead of the function definition. Extending the direct graph with these edges gives the same result as 0CFA [9].

Dynamic indexing. If we have the expression element(e_1, e_2) where e_1 is not an integer constant, the possible index values may be obtained from constant propagation. If source(e_1) contains only integer constant expressions, these integers can be used in the same way as in case of directly used integer constants.

5. Transformation

The goal of the transformation is to change calls of old interface functions to new ones. The interface change affects how the function is called and what is returned from the call.

Our experience shows that changes in function arguments are usually less complex. This is probably because the arguments themselves are usually less complex than the return values. However, if function arguments had to be transformed, the same approach could be used for them as for the return values. Still, in our main examples the only changes are some extra constant arguments.

In the following, we concentrate on the compensation of the changes in the return value of the function. The goal of the compensation is to modify existing code that expects the old return value to work with the new return value. The trivial compensation is to insert a run time conversion on the return value of the new function that produces the old value, and the unmodified old code will work. However, the approach presented here can provide a much better result.

5.1 Compensation spreading by data flow tracking

The run time conversion of the modified function's return value decomposes the returned data structure, and builds the old structure from the components. The old value is then probably decomposed again, and the same components are used somewhere. The idea is to skip the data structure conversion, and update the decomposition of the old value to work with the new structure.

To do this, we need to find code parts where the modified data can flow to. Ideally, data is copied through some expressions, and then a pattern is matched on it, so only the pattern should be updated. However, there are constructs that not only copy the modified data, but use it in an uncontrolled way. When an expression's value may be used in such a situation, compensation must be done on that expression at run time.

In the following, we formalise how to calculate the set of expressions that simply copy the modified data and should be left intact, and the set of expressions and patterns that terminate these data flow paths and should be transformed.

1. Let's start from a set of source graph nodes ($\mathcal{S}$). This set contains the function calls which return a modified value, or other data flow path start nodes which contain modified data. First, calculate the set of nodes that may get their value from this set:

$$\text{reach}(\mathcal{S}) = \{n \in \mathcal{N} \mid \mathcal{S} \rightsquigarrow n\}$$

2. Next, we must leave out every node from this set that may get its value from somewhere else, because it means that the node may receive values from unmodified sources, so it must not be transformed:

$$\text{strict}(\mathcal{S}) = \mathcal{S} \cup \{n \in \text{reach}(\mathcal{S}) \mid \text{input}(n) \subseteq \text{reach}(\mathcal{S}) \wedge \text{prev}(d, n) = \emptyset\}$$

   Element of $\mathcal{S}$ are explicitly included in this set, because they are the starting point of changes, so they are known to be modified, but the other conditions obviously do not hold for them.

3. We say that a node is *unsafe* if its value is used in an uncontrolled way, that is, copied to a node outside strict($\mathcal{S}$), put into a compound value, or used in an unspecified way. Nodes that safely copy their value are the following:

$$\text{safe}(\mathcal{S}) = \{n \in \text{strict}(\mathcal{S}) \mid \text{next}(f, n) \subseteq \text{strict}(\mathcal{S}) \wedge \text{next}(d, n) = \emptyset \wedge \text{next}(c_i, n) = \emptyset\}$$

4. Now we have to follow those flow paths which safely copy a modified value. These paths are terminated by non-safe nodes. We say that nodes found on these paths are *safely reachable*. Nodes that are safely reachable from set $\mathcal{S}$ are denoted with $\mathcal{S} \rightsquigarrow_s n$, and this is the minimal set that satisfies the following conditions:

   (a) $n \in \mathcal{S} \wedge n \in \text{strict}(\mathcal{S}) \Rightarrow \mathcal{S} \rightsquigarrow_s n$

   (b) $\mathcal{S} \rightsquigarrow_s n_1 \wedge n_1 \in \text{safe}(\mathcal{S}) \wedge n_1 \xrightarrow{f} n_2 \Rightarrow \mathcal{S} \rightsquigarrow_s n_2$

   Note that $\mathcal{S} \rightsquigarrow_s n$ implies $n \in \text{strict}(\mathcal{S})$.

5. Finally we have arrived at the scope of transformations. We modify the code so that safely reachable nodes will get the new data, and non-safe nodes will return the original data. This means that safe expressions, that only copy their value, remain unchanged; safely reachable patterns must be updated to reflect the structure of the new data; and finally, the return value of safely reachable but non-safe expressions must be converted at run time to the old form. Formally, the nodes to be transformed:

$$\begin{aligned} \text{trf}_p(\mathcal{S}) &= \{p \in \mathcal{P} \mid \mathcal{S} \rightsquigarrow_s p\} \\ \text{trf}_e(\mathcal{S}) &= \{e \in \mathcal{E} \mid \mathcal{S} \rightsquigarrow_s e, e \notin \text{safe}(\mathcal{S})\} \end{aligned}$$

Example. Let's have a look at how these concepts work on real code. We will use the `find` function shown on Fig. 3 and the function call show on Fig. 4, and a simple new function:

```
find(Key, [{Key, Val}|_]) -> Val;
find(Key, [_|Tail])       -> find(Key, Tail).
f()    -> find(a, [{a, 1}]).
dbl(X) -> 2*X.
```

As we have seen before, the value of expression `1` reaches the `find` function call in `f()`. If you check the conditions, this call is an element of strict(`1`), because it cannot get its value from anywhere else; and it is in safe(`1`), because its value is not used in an uncontrolled way (in fact, it's not used at all in this code).

Let's extend this code with the expression `dbl(f())`. Now the value of `1` can reach the call to `f()`, and variable `X` in `dbl` (both the pattern and the expression). These are elements of the strict(`1`) set, but expression `X` in the body of `dbl` is unsafe: it is used in expression `2*X` in an uncontrolled way. This is represented by edge `X` $\xrightarrow{d}$ `2 * X` in the graph, therefore next(d, `X`) is not empty, which violates the conditions of safe. In this case, trf$_p$(`1`) contains patterns `Val` in `find` and `X` in `dbl`, and trf$_e$(`1`) contains expression `X`.

If there is another call to `dbl`, e.g. we extend the code with `dbl(3)`, the situation changes: input(`X`) (for pattern `X` in `dbl`) now includes expression `3`, so `X` is not in the strict set anymore. This means that `f()` in `dbl(f())` is not safe, because its value is used in a non-strict expression, so it will become the only element of trf$_e$(`1`), and trf$_p$(`1`) now excludes `X` (as `X` in `dbl` is not safely reachable anymore).

5.2 Change descriptions

The goal of this section is to provide a formal description of data changes and a mechanism to reflect these changes in affected code. Changes are specified by a set of *change descriptions*. A change description can be applied to an expression or a pattern that works with the old data, and it results in two things: a new expression or pattern that works with the new data, and a (possibly empty) set of

$$P^c ::= K \mid V \mid \{P^c, \ldots, P^c\} \mid [P^c, \ldots, P^c \mid P^c]$$
$$E^c ::= K \mid V \mid \{E^c, \ldots, E^c\} \mid [E^c, \ldots, E^c \mid E^c] \mid A(V) \mid \texttt{map}(A, V)$$

Figure 5. Syntax of change patterns and expressions

induced transformations. Descriptions sometimes need to refer to each other, so each of them has a unique name.

Applying changes to a pattern means rewriting the pattern itself. This approach puts a restriction on possible changes, but changing only patterns leads to a much better result than changing expressions. This is because applying a change to an expression means doing run time data conversion, which is usually less readable and possibly slower.

Complete transformations are specified by a change description cd and a set $\mathcal{S}$ of source nodes. The transformation is executed by applying cd on $\mathrm{trf}_e(\mathcal{S})$ and $\mathrm{trf}_p(\mathcal{S})$; this may induce a set of other transformations on nodes. The source nodes of the same transformations are joined together, and the transformation is applied on the joined set.

5.2.1 Structural change descriptions

Changes in the data structures returned by a function can be conveniently handled through the pattern matching mechanism. When the changed data is used in a pattern matching expression, the pattern can usually be updated; when a compensating expression must be generated, it can be a `case` expression that decomposes the new data, and constructs the old one from the pieces.

Structural changes can be described quite naturally by providing the conversion from the old data structure to the new one, using Erlang patterns and data constructors, just as if we would implement a run time converter function. We will shortly see that such a description can be used to generate every kind of compensation that we need during the application of a change description.

A structural change description is formulated by a set of change patterns and change expressions (see Fig. 5). Change patterns are usual Erlang patterns, change expressions may additionally contain function call expressions (these refer to other change descriptions).

There are some restrictions on variables. We will use these descriptions to generate conversions between the old and new structures in both ways, so every variable must occur both in the change pattern and in the change expression exactly once. When a piece of data cannot be mapped to anything on the other side, it should be marked with the underscore pattern.

Matching patterns. When transforming a pattern, we need to find out which change patterns to apply. This decision is made based on the concept of *matching* between patterns. We use $\mathrm{subst}(p)$ to denote the set of substitution functions that map the free variables of pattern p to arbitrary patterns. The application of a substitution to p means replacing the variables in p with their mappings.

We say that pattern p_1 matches pattern p_2 when there is a substitution $s \in \mathrm{subst}(p_2)$ that satisfies $s(p_2) = p_1$. In this case, s maps subpatterns of p_1 to the free variables of p_2.

Example. A very simple interface change is migrating code that uses the `gb_trees` module to represent dictionaries with balanced trees to use the `dict` module instead, which represent dictionaries as hash tables. The changes in the return value of the `lookup` function are described this way:

$$\{\texttt{value}, \texttt{V}\} \mapsto \{\texttt{ok}, \texttt{V}\}, \quad \texttt{none} \mapsto \texttt{error}$$

Substitutions of the first change pattern, elements of the substitution set $\mathrm{subst}(\{\texttt{value}, \texttt{V}\})$, map `V` to patterns. Examples of patterns that match `{value, V}` are `{value, X}`, where the substitution is $\texttt{V} \mapsto \texttt{X}$, and `{value, [1,2]}`, where the substitution is $\texttt{V} \mapsto [1, 2]$.

Pattern transformation. When pattern p matches a change pattern cp, its transformation is straightforward. The new pattern for the data is given in the corresponding change expression ce, and we have the substitution function $s \in \mathrm{subst}(cp)$ so that $p = s(cp)$. All we have to do is apply the substitution s to ce, and replace the original pattern p with the result, $s(ce)$.

Applying a substitution to a change expression is similar to change patterns, that is, we replace the variables with their mappings in s. For example, let's transform the following code using the change description from the previous example:

```
case lookup(Key, Store) of
   {value, 0} -> inf;
   {value, N} -> 1/N;
   none       -> 0
end
```

Let n be the `lookup` function call node. $\mathrm{trf}_e(\{n\})$ is empty, $\mathrm{trf}_p(\{n\})$ contains the patterns of the `case` expression. Each of these pattern match a change pattern:

- `{value, 0}` matches `{value, V}`, the substitution is $\texttt{V} \mapsto \texttt{0}$. The result of applying this substitution to the corresponding change expression `{ok, V}` is `{ok, 0}`.
- `{value, N}` is similar, it should be changed to `{ok, N}`.
- `none` is exactly the same as the second change pattern, so it should be replaced with `error`.

After replacing the patterns (and the `lookup` function call), the result is the following:

```
case find(Key, Store) of
   {ok, 0} -> inf;
   {ok, N} -> 1/N;
   error   -> 0
end
```

Change description references. There is a change expression construct that has to be dealt with: references to other change descriptions of the form $\texttt{name}(v)$. This reference is substituted just as if only the variable was there, but this construct induces a new transformation. As we noted earlier, $s(v)$ refers to a subpattern of p, which is substituted into the place of the reference, and the induced transformation is applying the referred change description on the graph node of this subpattern.

For example, look at the following change description, where `lookup` is the name of the previous example's change description:

$$\{\texttt{A}, \texttt{B}\} \mapsto \{\texttt{lookup(B)}, \texttt{A}\}$$

Let's transform the expression

```
{X, {value, Y}} = f()
```

by changing the return value of `f()` with the above change description. $\mathrm{trf}_e(\{\texttt{f()}\})$ is empty, $\mathrm{trf}_p(\{\texttt{f()}\})$ contains only the pattern of the match expression, which matches the change pattern `{A, B}` using the substitution $\texttt{A} \mapsto \texttt{X}$, $\texttt{B} \mapsto \{\texttt{value}, \texttt{Y}\}$.

The first step is replacing the pattern with the substituted change expression, ignoring the `lookup` reference:

```
{{value, Y}, X} = f()
```

The second step is applying the induced transformation, which is `lookup` on `{value, Y}`. Repeating the same procedure as previously, the final result is:

```
{{ok, Y}, X} = f()
```

Generalised patterns. When a pattern does not match any of the change patterns, it is much harder to find a solution. Usually this means that the set of change descriptions is incomplete, in that case a warning should be given about the pattern that could not be transformed.

There is an exception: when a change pattern matches the pattern, there is a possibility for compensation. Consider the following change description: `{ok, {A, B}}` $\mapsto$ `{ok, A, B}`. How to transform the pattern in the following code?

```
{ok, Tup} = read(),
process(Tup).
```

In this case, the change pattern matches `{ok, Tup}`, the substitution is `Tup` $\mapsto$ `{A,B}`. A solution is to replace the pattern with the change expression, and replace the occurrences of the change pattern's variables with their substitution:

```
{ok, A, B} = read(),
process({A,B}).
```

When a change description reference occurs in the change expressions, the generated replacements should contain the variable compensated by the referred change description (as an expression). Note that this solution generates new variables in the program, which may introduce name clashes that must be resolved by renaming the variables in the change expression.

Multiple matches. Consider the following change description:

$$\{\texttt{ok}, []\} \mapsto \texttt{none}, \quad \{\texttt{ok}, \texttt{Lst}\} \mapsto \texttt{Lst}$$

In this situation the old, homogeneous data structure is replaced by at least two different constructs. This means that some of the patterns used in the old code cannot be used in the new code, because the same pattern cannot describe the various new structures.

In this case our approach is to duplicate the code that contains the pattern. In a `case` expression, problematic clauses are duplicated; in case of match expressions, the expression is turned into a `case` expression with one clause, which is then duplicated.

This situation can be recognised by a pattern that matches multiple change patterns. Every matching change pattern should get its own duplicate of the original code, and each duplicate is transformed according to one of the change expressions.

For example, let's transform the following expression by changing the return value of `f()` according to change description above:

```
case f() of
   {ok, L} -> length(L);
   error   -> 0
end
```

There are two possible ways to transform the first pattern: it is a generalisation of the first change pattern, and it matches the second change pattern. First the corresponding clause is duplicated:

```
case f() of
   {ok, L} -> length(L);
   {ok, L} -> length(L);
   error   -> 0
end
```

Then the first duplicate is transformed using the first change expression, and the second duplicate using the second change expression:

```
case f() of
   none  -> length([]);
   L     -> length(L);
   error -> 0
end
```

Transforming expressions. Expression transformation means that the return value of the expression should be converted at run time from the new structure to the original. This can easily be done by putting the expression into a `case` construct which uses patterns to decompose the return value according to the new structure, and rebuild the old structure using the components.

The patterns of the `case` expression should be generated from the change expressions. This means leaving the change description references out, only leaving their variables in the pattern; these variables will be converted at their usage places.

The results for the patterns are generated from the change patterns (these describe the original structure). The patterns are simply copied, except variables that have a change description reference in the change expressions: these are converted as expressions by the referred change description.

Note that the generated expression contains new variables, which may clash with existing variables names in the program; in this case, they should be renamed consistently.

Example. For an expression e, the following converter expression is generated from the `lookup` change description:

```
case e of
   {value, V} -> {ok, V};
   none       -> error
end
```

List element handling. There is a special change description reference with syntax `map(cd, Var)`, where `cd` is a change description name. We use this syntax to denote an element-wise application of a change description on a list. The meaning of this operator is simple, but cannot be described by the structural change elements introduced so far.

When such a change description reference is applied on a pattern, it makes no direct syntactic changes, because the outer data structure is unchanged: both the old and the new data is a list. Only the elements of the list are changed, this is reflected with a number of induced transformations:

- When the pattern has the form $[p_1, \ldots, p_n \mid p_{n+1}]$, the induced transformations are `cd` on $p_1, \ldots$, and p_i, and the same mapping transformation on p_{n+1}.
- Other patterns are not affected.

When this transformation is applied on an expression, it may be directly transformed, based on its type:

- $[e_1, \ldots, e_n \mid e_{n+1}]$ is converted by inducing transformation `cd` on e_1, ..., and e_n, and the same mapping transformation on e_{n+1}.
- `[`e_1 `||` p `<-` e_2`]` is converted by inducing transformation `cd` on e_1.
- Any other expression e is enclosed in the following compensation expression: `[case El of ... end || El <-` e`]`, where the clauses of the `case` expression are the same as described in the transformation of expressions.

5.2.2 Non-structural changes

In case of non-structural changes, like incrementing a value by one, we take a quite different approach. Instead of generating pattern and expression compensations from a common description, we provide two compensating expressions, one that converts old values to new values (used in updating patterns), and another that converts new values to old values (used on expression return values).

Such change descriptions are supposed to be used on atomic data. In case of patterns the compensation must result in a constant, so it makes sense to restrict the possible compensations to

side effect free expressions that can be evaluated during the transformation.

The syntactic representation of a non-structural change description is a pair of Erlang expressions. Both expressions may contain only one variable, that will be replaced with an expression. The first expression describes how to convert an old value to a new value, the second is the inverse of the first one.

Application on a constant pattern thus done by evaluation the expression on the constant, and replacing the original constant with the result. Variable patterns are not modified, their usages will be transformed (a safely reachable variable pattern is always safe itself).

Application on an expression is simply done by substituting the expression into the compensation expression.

Example. Transform the call to `f` in the following code using the non-structural change description (`Old-1`, `New+1`):

```
case f() of
   0 -> 0;
   N -> 1/N
end
```

trf_p(`f()`) contains the patterns of `case`, trf_e(`f()`) contains variable `N` in `1/N`. Pattern `0` is a constant, so it is substituted into `Old-1` and evaluated; pattern `N` is a variable, so it is not changed; expression `N` is substituted into `New+1`. The result is:

```
case f() of
   -1 -> 0;
   N -> 1/(N+1)
end
```

5.3 Change descriptions for the `regexp` module upgrade

Finally, the complete change description set that specifies the transformations proposed in Sec. 2 is provided here. They use all the features described in Sec. 5, which demonstrates that this rule set, in spite of being minimalistic, is strong enough to support real world applications.

When changing the calls to interface functions of module `regexp`, the new call expressions are to be transformed using the change description with the same name as the function. Applying these change descriptions on the examples in Sec. 2, the result is the solution proposed there.

match:

```
{match, St, Len}  ↦  {match, [{decr(St), Len}]}
         nomatch  ↦  nomatch
```

matches:

```
{match, []}  ↦  nomatch
{match, Ms}  ↦  {match, map(match_elem, Ms)}
```

match_elem: `{St, Len}` ↦ `[{decr(St), Len}]`

decr: (`Old-1`, `New+1`)

gsub: `{ok, Result, _}` ↦ `Result`

6. Implementation experiences

The main question about the real-world applicability of the presented approach is how large the data flow graph will be, and what the computational cost of the $\rightsquigarrow$ and $\rightsquigarrow_s$ relations is.

Experimental implementation in the RefactorErl system shows that the size of the presented static data flow graph is comparable to the size of the syntax tree. A syntax based transformation tool has to handle data structures of that size, so this should not be an issue.

The calculation of the relation based on the graph is a more delicate problem. These relations can be computed by either an iterative algorithm that finds new relation elements based on previously found elements and rules, or recursively applied graph traversals.

The former approach requires storage of the whole $\rightsquigarrow$ and $\rightsquigarrow_s$ relations, which is much more expensive than the direct flow graph, and its calculation time is proportional to its size. The latter approach with recursively called full-blown graph traversals seems to be no better.

Fortunately, using the graph traversal approach we don't have to calculate the whole $\rightsquigarrow$ and $\rightsquigarrow_s$ relations. In case of a large module, the data flow graph falls apart into many isolated components, because most data flow paths are not interconnected. The relations to be computed obviously cannot cross these graph component boundaries, which means we only have to compute them over the affected flow graph components.

In fact, we don't even compute the relations themselves, only the sets that are used during the transformation: reach, strict, safe, trf_p, and trf_e. These sets can be computed using graph traversal, starting from the initially changed node set, and the results will automatically restricted to the affected graph components, as the traversals won't cross component boundaries. This solution ensures that the run time cost of the transformation is proportional to the size of the affected code parts.

7. Related work

Restructuring software code while maintaining its consistency, known as refactoring, is a well known topic [2]. Tool support for refactoring Erlang programs exists for some years now [3, 5], but no support has been provided for automated data structure refactoring based on change descriptions. A scripted refactoring framework is built by Verbaere [11], targeting generic refactoring implementation for object oriented languages, it provides syntax-based manipulations opposed to declarative style descriptions used by this work.

Data flow analysis for functional languages has been studied by Shivers [9], but this work and its followers like [4] target optimising compilers with flow analysis. For Erlang programs, data flow analysis was used by Dialyzer for type inference, but that approach has been dropped in favour of success typing [6]. Control flow and data flow analysis has been successfully applied to improve testing of Erlang programs by Widera [12].

8. Conclusions and future work

A refactoring-based generic approach to introducing incompatible module interface changes to existing source code has been presented. The main contribution of this work is an automatic transformation mechanism that uses a simple, intuitive change description schema to describe interface changes.

Transformation is done applying data flow analysis based on simple data flow graphs. This is a cost-effective technique that produces a good static approximation of data flow paths in a program, and it is applicable for other data structure refactorings as well [7].

A complete real-life example of migrating regular expression module calls shows that while the presented approach is not capable of handling very complicated changes, it is still useful in practise.

An obvious area of necessary improvement is support for more language elements. The complete Erlang syntax can easily be supported by defining direct data flow edges for the missing constructs, this is straightforward to do in the same way as in Fig. 2.

A more complicated work that may improve the scope of data flow analysis is real support for those language features that require control flow analysis as well. These include processes and message passing, and exceptions. Analysing the process structure

and matching send and receive expressions is a really interesting, but hard problem.

Finally, it is also possible to extend the scope of the transformation by defining new change description schemes or improving the current ones. This can be done based on experiences with the current system, or by studying other interface migration cases which could be useful in practise.

References

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

[2] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[3] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A., Nagy, T., Tóth, M., and Király, R.: Building a refactoring tool for Erlang. Proceedings of the Workshop on Advanced Software Development Tools and Techniques, Paphos, Cyprus, 2008.

[4] Jagannathan, S. and Weeks, S.: A Unified Treatment of Flow Analysis in Higher-Order Languages Proceedings of the 22nd ACM Symposium on Principles of Programming Languages, 393–407, San Francisco, California 1995

[5] Li, H. and Thompson, S.: Tool Support for Refactoring Functional Programs. Proceedings of the Second ACM SIGPLAN Workshop on Refactoring Tools, Nashville, Tennessee, USA, 2008.

[6] Lindahl, T. and Sagonas, K.: Practical type inference based on success typings. Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, 167–178, Venice, Italy, 2006.

[7] Lövei, L., Horváth, Z., Kozsik, T., and Király, R.: Introducing Records by Refactoring. Proceedings of the 6th ACM SIGPLAN Erlang Workshop, 18–28, Freiburg, Germany, 2007.

[8] Muchnick, S. S. Advanced Compiler Design and Implementation. Morgan Kauffmann Publishers, 1997.

[9] Shivers, O.: Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University, 1991.

[10] STDLIB Reference Manual. `http://www.erlang.org/doc/apps/stdlib/`

[11] Verbaere, M., Ettinger, R., de Moor, O.: JunGL: a scripting language for refactoring. Proceedings of the 28th International Conference on Software Engineering, 172–181, Shanghai, China, 2006.

[12] Widera, M.: Flow Graphs for Testing Sequential Erlang Programs. Proceedings of the ACM SIGPLAN 2004 Erlang Workshop, 48–53, Snowbird, Utah, USA, 2004.

Automatic Assessment of Failure Recovery in Erlang Applications

Jan Henry Nyström
Erlang Training and Consulting Ltd.
henry.nystrom@erlang-consulting.com

Abstract

Erlang is a concurrent functional language, especially tailored for distributed, highly concurrent and fault-tolerant software. An important part of Erlang is its support for failure recovery. A designer implements failure recovery by organising the processes of an Erlang application into tree structures, in which parent processes monitor failures of their children and are responsible for their restart. Libraries support the creation of such structures during system initialisation.

We present a technique to automatically analyse that the process structure of an Erlang application is constructed in a way that guarantees recovery from process failures. First, we extract (part of) the process structure by static analysis of the initialisation code of the application. Thereafter, analysis of the process structure checks that it will recover from any process failure. We have implemented the technique in a tool, and applied it to several OTP library applications and to a subsystem of the AXD 301 ATM switch.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods; D.4.5 [*Reliability*]: Fault-tolerance

General Terms Reliability

Keywords Erlang, Fault-Tolerance

1. Introduction

Erlang is a concurrent functional language, especially tailored for distributed and fault-tolerant software, e.g., in telecommunications applications [Armstrong et al. 1996]. It has been used successfully in several commercial applications [Blau et al. 1999, Mullaparthi 2005, Stenman 2006] Prominent features of Erlang include support for light-weight processes, asynchronous message passing, and fault handling as integral parts of the language. The Open Telecom Platform (OTP) [Eri 2000] provides a number of libraries which support program design patterns that commonly occur in concurrent distributed software. Examples of such patterns, called "behaviours" in OTP, are event handlers, generic servers, and finite state machines.

The Erlang language supports implementation of failure recovery by a mechanism in which *links* can be created between processes. When a process fails, all process linked to it are notified, and can react either by failing themselves (thereby informing other linked processes), or by initiating a recovery action such as restarting a copy of the failed process.

The *supervisor behaviour* in OTP is used to program processes which monitor a set of children. Using links, it is notified about failures of its children, and can then restart new copies of failed children, possibly after some cleanup operations. This encourages designers to organise the processes of an Erlang system into *supervision structures*, i.e., trees of processes in which parent processes supervise their children.

The failure recovery mechanisms in Erlang and OTP can be seen as a way to catch exceptions caused by some anomalous condition in a process: it makes it possible to write clear code for each process, which is not obscured by defencive code. When using such a style of programming, it is important to ascertain that the global process structure of the system is set up in such a way that it recovers from arbitrary process failures. This can be done by extracting the process structure, and thereafter inspecting it to analyse the effect of any particular process failure, saying which processes will be affected and determining whether the process structure will be restored after recovery.

Currently, to obtain the process structure of an application, one must rely on external documentation or manual inspection of the source code. However, it is nontrivial to extract the process structure from source code since an Erlang program is structured according to modules and functions, whereas process creation and communication may occur anywhere in the code, and since the created process structure is not unique: it may depend on the system environment and configuration parameters.

In this paper, we present a technique for automatically detecting deficiencies in the failure recovery mechanism of Erlang applications, which are due to improperly designed supervision structures. The technique is structured in two phases.

- The set of possible process structures is extracted by static analysis of the source code. More precisely, we extract an over approximation of the set of possible static parts of process structures by symbolically executing the initialisation code of the application. By "the static part" we mean the processes started when the application is started and are to remain running (possibly restarted to handle failures) until the application terminates. The extraction assumes that the OTP libraries are used in the recommended way to set up the process structure; otherwise the precision will be poor.
- Each extracted process tree is analysed to determine the effect of process failures. We present a technique to determine the effect of a particular process failure on the entire process structure, which shows which processes will be terminated and restarted and whether the structure itself is restored to the situation before the failure.

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.

In addition to providing sufficient information for analysis, the extracted process structure is also of independent interest; it can be presented to the designer for visual inspection, with the possibility to choose different views depending on the information sought. As an example, the parts of the application that are affected by an abnormal process termination can be visualised.

We have implemented the techniques in this paper in a tool, which extracts sets of possible process static structures from source code, and which automatically checks the effects of a process failure in each process structure. The tool can also check that principles for construction of "good" supervision structures are followed. If the principles are not followed strictly, the tool can check the effects of process failures; this is useful when analysing legacy code applications, which may not have been designed using current design principles. We have applied the tool to several OTP-library applications and a subsystem of the AXD 301 ATM switch [Blau et al. 1999].

Overview In the remainder of the introduction section we survey related work. In Section 2, we introduce relevant features of Erlang and OTP libraries. In Section 3, we present our technique for analysing the effect of a process failure, based on relevant information in the process structure. In Section 4, we present how the possible process structures are extracted from source code by static analysis. Section 5 contains an extensive example of the information we are able to extract. Section 6 contains a report from application of our tool to several OTP-library applications and a subsystem of the AXD 301 ATM switch [Blau et al. 1999]. Finally, Section 7 presents our conclusions and directions for future work.

1.1 Related Work

We have not found any report of work that performs the same type of analysis as ours. This could partly be due to the fact that there are few other languages where failure recovery on the process level is supported by language and library primitives in the same way as in Erlang.

Analogous to our extraction of process structures is the extraction of call-graphs of a program in inter-procedural program analysis (e.g., by Agrawal in [Agrawal 2000]). There is a relationship, since an argument to a process creation statement in Erlang should be the function executed initially by the created function. A complication in Erlang is that this function may be computed in arbitrary ways, making it hard to obtain precision in the analysis.

Our aim is also to some extent related to the area of fault analysis, where one is also interested in the potential effects of faults in a system component (e.g., Sampath et al. in [Sampath et al. 1995]). However, most work in this area assumes that suitable models of system components are given (e.g., as state machines), and does not address the extraction of these models from source code.

1.2 Model extraction

The extraction of analysable models (e.g., finite state machines) of concurrent system from source code has recently attracted much attention in the model checking community. The aim is to extract control skeletons from source code.

C Holzmann extracts Promela models from C with threads in the tools FeaVer [Holzmann and Smith 2000] and `AX` [Holzmann 2000], This method relies on user defined abstractions, i.e. deciding what procedures and variables are unimportant, and consequently if the user definitions are incorrect executions that violates the properties under investigation might not be found.

Java and Concurrent ML An approach similar to Holzmann's, which does not rely on user definitions, is applied by Corbett for JAVA [Gosling et al. 1996] in the Bandera project [Corbett 2000] where shape analysis is used to determine what variables are only accessible from one thread. A method for the derivation of a finite-state control skeleton from Concurrent ML [Reppy 1993] programmes, abstracting values to their types, is presented in [Nielson et al. 1998].

Erlang Arts and Earle has investigated translation of Erlang programs into μCRL [Blom et al. 2001] models which can be model checked by the CÆSAR/ALDÉBARAN [Fernandez et al. 2000] tool set. Properties to proved are specified in alternation free modal μ-calculus and checked against state spaces generated by the μCRL tool set [Wouters 2001]. A problem of translating Erlang into μCRL is that Erlang requires process fairness, whereas the parallel composition of μCRL lacks this notion. The fairness assumptions of Erlang must be explicitly stated in the correctness properties. In [Arts and Earle 2001] they investigate a simplified version of a resource locking mechanism in the AXD 301 ATM switch [Blau et al. 1999]. This has been continued in [Arts et al. 2004a,b].

In [Leucker and Noll 2001] Leucker and Noll has continued this work and has implemented a prototype distributed model checker and proposes to employ the ELAN [Borovanský et al. 1998] environment for specifying and prototyping deduction systems as the next step. When using ELAN, they would have to specify the syntax and semantics of Erlang in terms of abstract data types and term rewriting rules similar to that of [Noll 2001]. The use of rewriting rules enables them to to develop higher level specifications in the same way as in [Arts and Noll 2001]. The ELAN work is presented in [Noll 2003, Amiranashvili 2002]. This work with rewriting logics has been continued using the Maude system [Clavel et al. 2003] in [Neuhäußer and Noll 2007].

1.3 Analysis of Erlang

The analysis of Erlang programs has received increased attention in the last years. This can be seen in the steadily increasing number of papers on the subject.

Static Type Analysis The first attempts made to analyse Erlang was to derive type information. It has been the approach with the largest practical impact with the Dialyzer tool [Lindahl and Sagonas 2004] included in the OTP release.

In [Lindgren 1996] is developed a soft-typing system for Erlang, where types are not declared but derived from the code. The constraint solver used was however deemed to be unsuitable for Erlang. That was followed by a paper [Marlow and Wadler 1997] describing another soft-typing scheme, in which however not all legal Erlang programs could be typed.

A third soft-typing scheme was presented in [Nyström 2003], where the type inference is based on data flow with optional user annotations, the intention being that the user would annotate all the interfaces, and warnings generated for all possible type clashes. The main drawback of this system is that it tends to generate too many false positives.

A slightly different approach is followed in [Lindahl and Sagonas 2004] where a data flow analysis is applied to the virtual machine byte code to derive the possible values of the live variables at each program point. This type information enables the tool to find: places that would raise exceptions, unreachable branches, and dead code. The work has be extended in [Lindahl and Sagonas 2006] to generate success typings. The success typings, unlike the other soft-typing schemata already described, allows for compositional bottom up type inference which has been shown to scale well in practice.

To provide the necessary information to provide optimisations using a new heap structure and allocation strategy, a variant of escape analysis is presented in [Carlsson et al. 2006].

Abstract Interpretation Huch has developed an approach for the formal verification of Erlang programmes using abstract interpretation and model checking. In [Huch 1999] an Erlang system is viewed as a set of expression evaluations in the context of the identity of the processes executing the expressions and their message queues. The abstraction consists of truncating the terms in the expression at a predefined depth. It is mentioned in the paper how one could tailor the interpretation so that for selected terms the terms are either kept as they are or truncated at a greater depth. The language used to specify the desired properties is a linear temporal logic [Manna and Pnueli 1992], expressive enough to state most interesting properties.

The interpretation can only handle tail recursive programs and does not handle exceptions, links, nor process termination. This work has been extended in [Huch 2001] where he handles non tail recursive calls, through a technique of jumps which makes his approach much more realistic for real life programs. The work has been further extended in [Huch 2003], where a technique to automatically generate an abstracted finite graph representation of the possible evaluations of a process. This graph is used to model check properties of the program.

A factor that hinders this method, in its current cast, from handling realistic applications is that the model checking does not scale to: dozen of processes and over 50'000 lines of source code of the AXD 301 subsystem analysed in Section 6.

Theorem Proving The Erlang Verification Tool(EVT) [Fredlund et al. 2003] is an interactive proof assistant with an embedding of the language in the proof rules. The specification logic used [Fredlund 2001] is a first order logic inspired by the modal μ-calculus [Kozen 1983] and extended with Erlang-specific features. The logic is quite powerful, with both least and greatest fix points, allowing the formalisation of a wide range of behavioural properties. The strength of the specification language however leaves the verification problem undecidable, although normally a considerable parts of a proof can be automatically produced.

There have been extensions to allow the tool to reason about Erlang code on an architectural level, where the specified behaviour of the OTP behaviours can be characterised by sets of transition rules [Arts and Noll 2001]. This enables the tool to reason about OTP behaviours without having to consider their concrete implementation.

There are two great differences between this work and my thesis. First that the proof of properties, using the proof assistant, are very much done by hand even if most of the tedious details are automated, whereas an analysis by my tool will performed automatically. When performing an analysis with my tool one may have to state an initial configuration, such as contents of database tables, but this constitutes only a small effort. Secondly, with the proof checker one can prove a wide range of properties, whereas if I were to examine anything else but the fault tolerance aspects of Erlang, I would have to redesign the model extraction and the subsequent analysis.

Conditional Term Rewriting Systems In [Giesl and Arts 2001] it is shown how techniques for analysing termination of conditional term rewriting systems, can be used to show properties for Erlang programmes. The correctness properties of the query lookup protocol of the Mnesia distributed database is transformed into a termination problem, and then it is shown that, using refinement of dependency pairs techniques [Arts and Giesl 1997], the transformed properties can be proved without manual intervention. The same properties have also been shown to hold using EVT [Arts and Dam 1999].

Model Checking Wiklander has in [Wiklander 1999] implemented a translation from a finite state subset of Erlang to Promela, the specification language of the model checker Spin [Holzmann 1991, 1997]. The main difficulties were to translate the dynamic data structures lists and tuples to Promela constructs. Without a good translation of these basic data structures of Erlang, it is hard to translate any program. Another major obstacle was, finding an effective encoding of Erlang's receive statement using Promela's different style of message passing.

Another process algebraic model is presented in [Noll and Roy 2005] and then refined in [Roy et al. 2006]. In these two papers a subset of Erlang is translated into asynchronous π-calculus with monadic communication. The model can be checked using existing tools for π-calculus. The advantage one gets from using π-calculus as opposed to μCRL used in [Arts et al. 2004a] is that name passing feature of π-calculus allows the direct representation of the sending of process ids between processes. The models deal with a subset of the language and do not handle distribution or concurrency at a level that is even near what is required to analyse real systems.

A distinctly new approach is examined in [Fredlund and Earle 2006], where model checking is achieved by running the program being checked in lock step with an automaton representing a safety property on a new run time system for Erlang. The result is a system where everything is Erlang. The system was only a prototype with the semantic model not fully completed, lacking elements of the semantics presented in [Claessen and Svensson 2005]. This has now been extended to support to the entire semantics, including the modeling of distributed Erlang as described in [Fredlund and Svensson 2007]. It is the approach that has most similarities with the techniques used in this thesis.

2. Erlang

Erlang[Armstrong et al. 1996, Barklund and Virding 1999] is a concurrent functional language especially tailored for distributed and fault-tolerant software. The language has explicit concurrency and distribution and is dynamically typed. It supports process creation, management, communication, and failure handling, through a number of built in functions (BIF for short) and through the OTP libraries.

Communication Communication in Erlang is asynchronous via message passing. The nonblocking function `!`, called as "`Pid ! Message`", transmits the message `Message` (which may be any Erlang term) to the mailbox of the process with process identifier `Pid`. The receiver's mailbox preserves the order of messages sent from the same process.

A process accesses its mailbox through the `receive` statement, which matches a number of clauses against the messages in the mailbox queue. The first message in the queue which matches some clause will be received. If there is no such message, the receive statement blocks; blocking may be avoided by an optional timeout clause with a specified waiting time.

Process Handling A process is created by calling a function in the `spawn` family of functions, with arguments specifying what function and with what arguments the created process should execute. Optional arguments determine on which Erlang node the process should be created. The `spawn` functions return the process identifier (pid) of the created process.

A process can terminate normally by returning from the last function call, or abnormally by not catching an exception. We use the term *failure* to denote abnormal termination. A process can force another process to terminate normally or abnormally by calling the BIF `exit(Pid, Reason)`, where `Pid` is the pid of the terminated process. There is also a mechanism for giving names to processes, which can be used instead of pids.

Failure Detection and Handling The basis for failure handling is that *links* can be created between cooperating processes. If a process terminates abnormally, by an uncaught exception, all processes linked to it will be informed by a special type of message. If the boolean process flag `trap_exit` is false, the informed process will also terminate abnormally and its linked processes be informed in the same way, i.e., the failure spreads. If the process flag `trap_exit` is true, the informed process is not terminated, and may use the received information to take some recovery action. We will say that a process *traps exit* when the process flag `trap_exit` is true. Supervisor processes use this mechanism to monitor termination of their child processes, and to restart them, possibly after performing some cleaning up actions.

A process creates a link by a call to the BIF `link` with the pid of the other process as an argument. Alternatively, the `spawn_link` functions behave in the same manner as the `spawn` functions, but also link the parent and child processes when creating the child.

2.1 Behaviours

To support the use of supervision structures, and other structures, the Open Telecom Platform[Eri 2000] provides a number of *behaviours*, which support design patterns addressing common problems of distributed concurrent systems. A behaviour is a library module that implements the generic parts that are shared by all processes with this behaviour. The behaviour is used by writing a callback module implementing the specific parts of a particular process. For example, in the *supervisor behaviour* the callback module will contain an `init` function which returns the list of children that should be started; the supervisor library module will handle the actual starting, supervision, restarting and stopping of the child processes.

The behaviours provided in OTP are:

application is a packaging of system components, and has a number of resources such as modules, registered names and processes. The processes can be loaded, started and stopped together and it can be checked that the needed resources are available when loading the application. Associated with an application is not only a callback module, but also a resource file which declares the resources needed by the application, such as the names that will be registered by the application, and what other applications have to be running before the application is loaded.

gen_event is a manager for a number of event handlers, each of which can be added and removed dynamically. The event manager applies every present event handler, via a call to the callback module, to a received call or notification.

gen_fsm is used to write finite state machines, where the callback module for each state has a function which describes the transitions made on events.

gen_server provides a simple way of writing the server part of client server applications, where the gen_server module handles debugging and termination of the parent.

supervisor is used to structure applications for failure recovery. A supervisor is a process which has a number of children which it monitors through links. When a child fails, one or several children are restarted, possibly after first shutting down one or several children. A child is shut down using the `exit` BIF with reason `shutdown`: this reason tells supervisors to shut down their children before terminating. The callback module's `init` function determines what children will be statically started by the supervisor whenever it starts or is restarted. Note that children of a supervisor can also be added later, but we will not regard such children as part of the static process structure.

The precise reaction to the failure of a child is determined by the supervisors *restart strategy* (returned by its `init` function), which is either

`one_for_one`: only the terminated child is restarted,

`one_for_all`: all children are shutdown and then restarted, or

`one_for_rest`: all children started after the terminated child are shutdown and restarted.

Each child of also has a restart strategy, being either

`permanent`: the process is always restarted if it terminates,

`transient`: it is restarted only if it fails, or

`temporary`: it is not restarted.

There is a limit to the number of times a supervisor will restart its children, given by two numbers `maxR` and `maxT`. If more than `maxR` restarts are made within `maxT` seconds the supervisor fails. This mechanism allows the system to attempt more global recovery action if restarting only the children of a particular supervisor is insufficient.

Apart from the restart strategy the children will have as parameters: a name, a triplet {`Module, Function, Arguments`} called as `apply(Module, Function, Arguments)` to start the child; shutdown time which is the time a child is given to terminate after being told to shutdown, after which it will be exited; and finally the type of child either a worker or supervisor.

supervisor_bridge Supervisor_bridge enables a subsystem, not originally intended to be part of a supervision hierarchy, to be connected to a supervision hierarchy. The process having the *supervisor_bridge behaviour* behaves as a bridge between the supervision tree and the subsystem.

2.2 Supervision Structures

An important pattern in Erlang/OTP is the supervision structure which consists of a tree of supervisors monitoring their children through links. The children processes normally execute one of the OTP behaviours or are connected to its supervisor via a `supervisor_bridge`.

The children are started with its behaviour's library function `start` or `start_link` with its call-back module as an argument. The call-back's `init` function is then responsible for creation of the process, and all other initialisation to be made. An exception is the `supervisor` behaviour for which the `init` function will only return the parameters of the supervisor; the library functions will create the supervisor and start its children. The children are started one after the other, and a child has to acknowledge that it has finished its initialisation before the next is started.

3. Analysis of Failure Recovery Through Supervision Structures

The main aim of the analysis is to ensure that the process structure built by the application is robust, meaning that it will recover from failures in an appropriate manner. In this section we will describe the restart mechanism in the supervisor behaviour, what problems may arise and how our tool detects these problems. We will also present conventions used in order to avoid these problems.

Our analysis will work if the extracted supervision structure is not altered by the dynamic behaviour of the application, and it will not alter the supervision structure if the OTP design principles are followed.

The effect of a process failure on the supervision structure is the following:

(1) All processes linked to the failed process are notified. Any process notified that does not trap exit will fail and the effect spreads.

(2) When a supervisor is informed that a child has failed, it will shut down some of the other children. What children are shut down depends on the supervisor's restart strategy. If a shutdown child is a supervisor that will cause it to shut down its children in turn, shut downs propagating down the supervision structure. Since shutting down is achieved using `exit` the process will fail and the effect will spread according to (1). A supervisor shuts down the children in the inverse order of creation and wait for a child to terminate before shutting down the next.

(3) When a supervisor has shut down all its children it will restart them, unless it was being shut down itself, in which case it terminates.

We will describe two properties that capture much of the core intentions of the supervision structures, and for which we can check automatically that the properties hold.

Property `P1`: Whenever a process that takes part in the supervision structure fails, the supervision structure returns to the process structure prior the failure after a reasonable delay.

Property `P2`: When the cause of a failure is not transient or sufficiently infrequent to let the application function acceptably, only a small number of recoveries should occur before the supervision structure fails.

An example supervision structure, illustrating violations of the properties `P1` and `P2` is shown in Figure 1. In the figure supervisors are depicted by rectangles, and all other processes by circles. Solid arrows indicate parent-child relationships and the dashed lines show links between what processes. Within process is either a registered name or a process id of the form `<number>`. In Associated boxes show parameters and flags values.

3.1 Analysis of Property `P1`

Property `P1` implies that in the recovery actions caused by a failure, **(a)** each processes that fails is replaced by an equivalent "restarted" process in the structure, and **(b)** each process replaced by a restarted process has indeed terminated. We now claim that, in fact, properties **(a)** and **(b)** are sufficient to guarantee property `P1`, under the two extra assumptions that **(c)** a non-supervisor that is linked to a failed process does not trap exits, and **(d)** the initialisation of a restarted process creates the same structure as the process it replaces. Property (c) mirrors the recommendation that only supervisors trap exits. Property (d) is true if the initialisation of a process is not dependent on configuration parameters that change during system operation; it holds in all but one of the applications that we have analysed.

Our claim can be "proved" as follows. Assume that a supervision structure satisfies properties (a) and (b). We then prove property `P1` by structural induction over the supervision structure, starting from its leaves. For a leaf process, the property is immediate. Consider a non-leaf process p, and assume that the substructures rooted at the children of p satisfy `P1`. If p fails, then it will be restarted (by (a)), and (when reinitialising, by (d)) recreate the entire substructure rooted at p, thus establishing `P1`. If p does not fail, then if no child fails, the connections between p and its children are preserved. If a child fails, then if p is a supervisor, it will restore its children and its connections to them, otherwise (by (c)) p must fail, and we are back to the previous case.

The possible violations of property `P1` are the following:

case 1: A process is terminated but not restarted, because it is not connected by a sequence of links to a supervisor, directly or in several steps, where no intermediate process traps exit.

In Figure 1 the process `<3>` might not be restarted if it fails since process `X_serv` traps exit. We can not determine if process `<3>` is restarted since this depends on the dynamic execution of `X_serv`.

case 2: A process is restarted without having terminated first, because its parent is a non-supervisor and it is either not linked to its parent or traps exit. In Figure 1 the process `<3>` might be restarted without having terminated if `X_serv` fails, since it traps exit.

Another reason is, if a supervisor has less time, to shut down its children, then the combined shutdown times of the children. In this case, when time expires for the supervisor all remaining children will be left unterminated. In Figure 1 processes `<1>` and `<2>` have the combined shutdown time of 4 seconds whereas the supervisor `Y_sup` has 2 seconds. Even if the `<2>` where linked to `Y_sup`, process `<1>` might not be terminated when `Y_sup` is shut down.

Should the supervisor with limited shut down time be deadlocked by a child the remaining children would remain. In Figure 1, when shutting down `Y_sup` process `<2>` will deadlock `Y_sup`, as described below, and `Y_sup` consequently will not shut down `<1>`.

case 3: A supervisor is deadlocked, when trying to shut down a child which is not linked to the supervisor, since the supervisor is never informed that the child has terminated. In Figure 1 the supervisor `Y_sup` is deadlocked when trying to shut down process `<2>`.

Another reason for a supervisor to deadlock is that a child with shut down time `infinity` that traps exit does not terminate, e.g., a deadlocked supervisor.

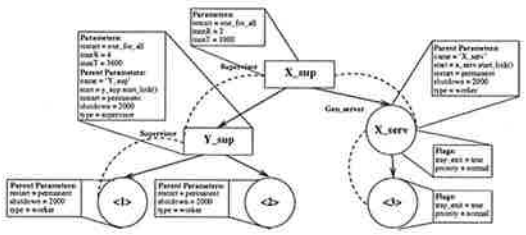

Figure 1. Supervision structure example.

In order to determine if property `P1` is violated by an application the tool extracts a set of supervision structures, and for each process records what the effects are of a failure.

As a first step the tool records processes that will fail by following links via processes that do no trap exit. As a second step, for each supervisor with a failed child, it records what children are shut down. Which children are shut down is decided by the supervisor's parameters, and for each shut down child we have to repeat step one and two. As a final step, the children that would be restarted by the supervisor are recorded.

Note that a supervisor can fail for two reasons. First it can have a maximal restart frequency of 0, and then the first and second steps are performed recursively up the supervision structure until reaching a supervisor that does not fail or until the entire supervision structure has failed. Secondly the number of restarts can exceeds the highest allowed restart frequency, in this case both the case where the supervisor fails and the case where it restarts its children must be recorded.

When all effects of the failure have been applied the tool can determine if property `P1` has been violated by checking that properties (a) and (b) are satisfied for each process. For example, if a process

has failed but not restarted, then property P1 is violated according to **case 1**.

3.2 Analysis of Property P2

Property P2: When the cause of a failure is not transient or sufficiently infrequent to let the application function acceptably, only a small number of recoveries should occur before the supervision structure fails.

A supervision structure can violate Property P2 in two ways: **(a)** a substructure fails and is restarted too many times before the supervision structure itself (represented by its root processes) fails, **(b)** the supervision structure never fails although there are an unbounded number of repeated failures in the structure.

Such situations can be detected as follows. The maximum rate of failures (MRF for short) that can occur in a structure before the structure itself fails, can be calculated from the restart strategies, the numbers `maxR` and `maxT`, as follows.

The MRF for a process with supervisors $s_1, \ldots, s_n$ above in the process structure, where the supervisors are numbered from the top down is given by the following equations

$$\begin{aligned} \text{MRF} &= \mathit{Restarts\ per\ TimeUnit} \\ \mathit{Restarts} &= \mathit{MaxR}(s_1) \cdot \prod_{i=2}^{n} (\mathit{MaxR}(s_i) + 1) \\ \mathit{TimeUnit} &= \min_{i=1}^{n} \mathit{MaxT}(s_i) \end{aligned}$$

where $\mathit{MaxR}(s_i)$ is the parameter `maxR` for the i:th supervisor and analogously for `maxT`.

Violations of type (a) occur if MRF is too high. The meaning of "too high", of course, varies from application to application. We suggest that a threshold for this number shall be provided for each application by designers or by company coding principles.

Violations of type (b) occur if the structure contains a supervisor for which the maximum time it takes to restart one child is larger than the least time between failures needed to cause failure of the supervisor itself.

The maximal time to restart a child is the combined shut down time of the other children that are shut down and the start times of all the terminated children. We can only determine the shut down times[1], but if this is already to large (in effect assuming a start time of 0) we are certain that it is too large. The least time between failure needed is simply supervisor parameter `maxT` divided by `maxR`.

In Figure 1 the shutdown time of `Y_sup` is 2 seconds but its supervisor `X_sup` will fail only if more than 2 restarts are performed within 1 second. If the shutdown of `Y_sup` actually takes 2 seconds then, even if the process `X_serv` fails immediately upon each restart the supervisor `X_sup` will never fail. This is a typical violation of (b).

When designing an application one wants to determine parameters of the entire supervision structure as well as over individual processes; restarts and shutdown times can be calculated automatically. Interesting Parameters include: (a) the maximum restart frequency for any process before the application restarts; (b) the largest shutdown time allowed for any process; (c) the largest shutdown time allowed for the entire application.

[1] There is a OTP library that allows the designer to set an upper limit on the time it takes to start and initialise process, if that was used throughout we could give an upper limit on startup and as a consequence of restart as a whole.

3.3 Conventions

There exists a number of coding conventions designed to prevent violation of properties P1 and P2, which catch a subset of these violations. The tool checks whether the conventions are followed:

All process should create its children using `spawn_link`, rather than `spawn`, and children should not `unlink` from their parents. If a process is started with `spawn` rather than `spawn_link` it can fail before it has time to link to its parent, in which case its parent will not be informed.

The maximum restart frequency of intermediate supervisors 0 in order to minimise the MRF of leaf processes in the supervision structure.

Only supervisors should have shutdown time infinity and all non-supervisor children of supervisors should have shutdown set to a limited time. This ensures that the supervisor has time to terminate its children and that no child can indefinitely block the shutdowns. This eliminates one of the causes of violation of property P1 in **case 3**.

4. Process Structure Extraction

Our tool extracts an over approximation of the set of possible static process structures of an Erlang application by symbolically executing its initialisation code. For each process that could be in the static process structure, the tool evaluates its initialisation code, in order to extract all possible combinations of children and relevant parameters and flags. The evaluation uses a symbolic representation[2] of the *set* of possible execution states, consisting of a set of triples of form `<value, side effect, state>` where

`value` is either a normal Erlang term, an exception, the element ($\top_{\text{Term}}$) denoting an unknown term, or the element ($\top_{\text{Exception}}$) denoting an unknown exception. The top element, i.e., an unknown value which is either a term or an exception, is represented by two triples, one containing $\top_{\text{Term}}$ and one containing $\top_{\text{Exception}}$.

`side effect` is a representation of the process tree, including relevant parameter and flags, which results from previous side effects. The tree may contain the element ($\top_{\text{SideEffect}}$), representing an unknown (possibly absent) side effect, meaning that we have no information about the behaviour of the subtree rooted at the position of $\top_{\text{SideEffect}}$.

`state` is an abstraction of the current state of the Erlang-node. This abstraction is tailored to contain those parts that are relevant for the current process; typically it contains representations of persistent data storages, such as the Erlang tabling system (ETS), which is a simple database.

Intuitively, a set of triples represent all possible states of the computation at a particular point in the execution of the `application:start` function. The elements at each position in triples are ordered in the natural way (e.g., $\top_{\text{Term}}$ is larger than any Erlang term). The ordering is extended pointwise to triples.

The evaluation is performed for each process in the static tree, starting from the root process, proceeding with its detected children, and so on. The evaluation of one process does not use all information about potential interaction with other processes: for instance, the value of a received message is always unknown and must be approximated by $\top_{\text{Term}}$.

Our symbolic evaluation is not performed directly on Erlang, but on Core Erlang [Carlsson et al. 2000]. Core Erlang is an intermediate format used, e.g., in the OTP Erlang compiler, where

[2] We use the terminology of abstract interpretation [Nielson et al. 1999].

some syntactic sugar has been removed and a restricted set of constructs and formats is used. It also has constructs that are not present in Erlang, such as `let` and `letrec` which are used to replace explicit matching and local functions generated by list expressions.

For our symbolic evaluator, we have defined a symbolic semantics for Core Erlang, based on the semantics for Core Erlang by Carlsson [Carlsson 2001]. Its key property is that

> if in standard (Core) Erlang the call `foo(args)` in state `state` returns `result`, new state `state'`, and has side effect `effect`,
> then a call to `foo` in the symbolic semantics, starting with a set containing a triple `<absargs, abseffect, absstate>` where `absargs` and `absstate` approximate `args` and `state`, will result in a set containing a triple `<absresult, abseffect', absstate'>` where `absresult` and `absstate'` approximate `result` and `state'`, and where `abseffect'` approximates the addition of `effect` to `abseffect`.

Space does not permit a presentation of the symbolic semantics. Let us here briefly describe parts that are not straight-forward, notably function calls, `receive` and `case`.

Function calls The treatment of a function is divided into three categories.

- Calls to local functions in the current module are evaluated symbolically in the extended value domain.
- Calls to functions that are not in the local module or a library module are evaluated as local functions, after ensuring the module is loaded.
- Calls to functions in library modules are treated in different ways, depending on the nature of the library:
  - Some library modules which influence the processes structure are supported in our tool. This means that behaviour of these modules is emulated by functions in the tool, which generate (an abstraction of) the same results and side effects as the library modules. Calls to the more important library modules are emulated by calls to the emulating functions in the tool. In the current implementation, we have introduced support for approximately 2% of the modules.
  - It is not reasonable to support all of the over a one thousand library modules. Unsupported library modules are treated in two ways:
    - Functions in library modules which do not influence the processes structure (e.g., the module `lists` which contains list manipulation functions) are handled as follows. For each triple `<value, side effect, state>` in the set, where `value` is a normal Erlang term, we call the function with argument `value`, and replace the triple by `<result, side effect, state>`, where `result` is returned by the function. For triples where `value` is $\top_{\text{Term}}$, we approximate the call by returning both $\top_{\text{Term}}$ and $\top_{\text{Exception}}$. We do not need to worry about the other elements of triples, since we know that the call can not generate side effects.
    - Functions in library modules which may influence the process structure are difficult to handle. An example of such a module is `mnemosyne`, an interface to `mnesia` a internal distributed database in OTP. We approximate their effect by returning $\top_{\text{Term}}$ and $\top_{\text{Exception}}$, together with adding the unknown side effect $\top_{\text{SideEffect}}$ to the tree.

      To determine if the approximated call actually performed side effects we have to inspect the call manually; in the future we will categorise all OTP library functions as having side effects or not. The supported libraries, together with the libraries without side effects, account for more than 99% of called library functions in the experiments presented in Section 6.

Receive The `receive` statement is handled specially since our evaluator does not spawn any processes, and consequently the evaluator has not created anyone who could send the message we are waiting for. As a consequence all the clauses of the `receive` statement must be evaluated, including timeout clause, with all free variables in the match of the clauses bound to $\top_{\text{Term}}$, and the set of all resulting triplets returned.

Case In the `case` statement, which is the only conditional statement in Core Erlang, the argument on which we switch may contain an approximation such as $\top_{\text{Term}}$. We must then evaluate all branches that could possibly match a value approximated by $\top_{\text{Term}}$, and return the set of resulting triplets.

Termination

In general, the symbolic execution may generate infinite execution paths. We use two mechanisms to avoid a potentially nonterminating analysis.

- There is a user definable bound on the depth of function applications allowed before terminating the execution and approximating the value of the execution branch with $\top_{\text{Term}}$ and $\top_{\text{Exception}}$.
- If in one branch of execution, a call to a function is repeated twice with the same parameters, the result is approximated by the top element of the value and state domains.

In both these cases of approximations a side effect could have occurred, and $\top_{\text{SideEffect}}$ is added to the side effects.

Limitations

The tool can handle applications that are syntactically correct, i.e., those that pass through the initial phases of the compiler which produces Core Erlang. Applications can be analysed even if only parts of the source code are available; missing parts are handled in the same manner as calls to functions in unsupported library modules.

The most noticeable limitation is that when many unsupported library calls are made, the number of possible executions become intractable. This is the case both for the OTP application `mnesia` and for the first tries to analyse applications from the real life system AXD 301[Blau et al. 1999]. Precision can be improved by supporting more OTP libraries and emulating a larger part of the Erlang runtime system, as has been done in order to analyse the AXD 301 applications.

Another limitation is that we can effectively analyse only applications that have been designed according to the suggestions of the OTP documentation and using the OTP library behaviours. This includes the restriction that the essential parts of the supervision structure should be set up by the initialisation code. This is a way to encourage use of standard coding idioms. It is in general intractable to analyse automatically the behaviour of arbitrarily structured code.

5. Example: Os_mon

As an example we will use the result of applying the tool to the OTP application `os_mon`, that monitors the underlying operating system

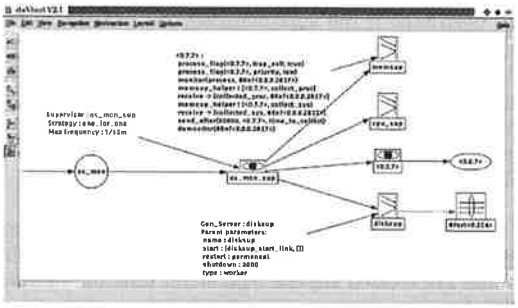

Figure 2. Creation tree for the OTP application `os_mon` with expanded information nodes.

for disk, memory and CPU usage. Snapshots from the tool after analysing `os_mon` is shown in Figure 2.

The `os_mon` application running on a Linux system will have seven processes and one external port. Below we describe each process in the order of creation.

os_mon The application master process registered as `os_mon`.

os_mon_sup The supervisor of the processes responsible for monitoring the operating system resources, this process is registered as `os_mon_sup`. From the parameters one can see that the highest frequency of restarts it will allow is 4 per hour, and the restart strategy is `one_for_one`.

disksup The generic server registered as `disksup` is responsible for monitoring the disk usage. From the supervisor's parameters associated with this process we can see that it will be started by a call to `disksup:start_link()`, its restart strategy is `permanent`, it will have 2000 milliseconds to shut down and type `worker`.

This process will during initialisation start an external port which communicate with an external process used to monitors the disk usage.

`<0.5.7>` This process, without registered name, starts process `<0.6.7>` of which the analysis can not determine anything.

memsup The generic server is registered as `memsup` is responsible for monitoring the memory usage by the Erlang-node. In this case we have highlighted the actions:

```
process_flag(<0.7.7>, trap_exit, true)
process_flag(<0.7.7>, priority, low)
monitor(process, ⊤Term)
memsup_helper!{<0.7.7>, collect_proc}
receive {collected_proc, ⊤Term}
memsup_helper!{<0.7.7>, collect_sys}
receive {collected_sys, ⊤Term}
send_after(60000, <0.7.7>, time_to_collect)
demonitor(⊤Term)
```

The two first actions set process flags, followed by the monitoring of an unknown process. After these initial setups, the process will communicate with the registered process `memsup_helper`, sending and receiving twice. After these calls and responses it uses a BIF that will send the message after 60'000 milliseconds `time_to_collect`. Finally it demonitors an unknown process.

The `monitor` BIF enables a process to monitor various aspects in the Erlang-node, getting messages from the runtime system when something happens which affects the process.

cpu_sup The generic server registered as `cpu_sup` is responsible for monitoring the cpu usage of the system.

When analysing the `os_mon` application our tool generated 200 different trees, depending on approximations as mentioned in Section 4. The great multiplicity comes from a receive statement where the `memsup` process either gets information back from the `memsup_helper` process or a timeout occurs, this is repeated 10 times giving rise to 100 different trees. The final step from 100 trees to 200 is taken by a previous process, namely the anonymous `<0.5.7>`, that may fail during initialisation, but in such a manner that the remaining processes can be started. The tree presented here is the simplest where no resends due to timeouts have to be done, and the process `<0.5.7>` initialises successfully.

The automatic check performed by the tool for each tree shows that `os_mon` respects properties `P1` and `P2` provided that process `<0.0.6>` behaves correctly. This clearly indicates that in order to be confident in the failure recovery we need to examine `<0.0.6>`.

6. Experiments

In order to test the tool we, have applied it to several OTP-library applications and a subsystem of the AXD 301 ATM switch. The OTP applications analysed were `os_mon`, `mnemosyne`, `sasl`, `megaco`, `inets` and `crypto`; in the AXD 301 subsystem, consisting of ten applications, we analysed `rcm`, `rcmInit`, `rcmKernel`, `rcmUtilities`, `rcmNtp`, `sys1` and `sysCmd`.

The OTP applications could be analysed simply by invoking the tool with the name of the application. When analysing the AXD 301 applications it was necessary to create and initialise a few ETS and Mnesia tables in the tool.

The analyses were made on a 450MHz Pentium III with 256MB memory. The statistics of the analysis of the applications are shown below in Table 1, with the exception of `mnemosyne`, `megaco` and `inets` which are trivial.

Name	Time	#Trees	#Lines	#Executed	#Total
OTP:					
os_mon	8.0s	200	260	620	2'588
sasl	1.1s	2	291	312	9'467
crypto	0.3s	1	52	65	337
AXD 301:					
rcm	180s	1'674	822	306'176	19'250 [3]
rcmInit	5.4s	1	650	5'320	3'744
rcmKern	582s	3'708	1'068	570'976	16'272
rcmUtil	1.7s	1	140	164	62
rcmNtp	1.9s	3	89	93	2'583
sys1	28.2s	8	897	58'395	33'227
sysCmd	0.2s	1	9	9	1'359

Table 1. Analysis Statistics

Int the table: Time = runtime of the analysis, #Trees = number of process trees extracted, #Lines = number of source code lines used by initialisation, #Executed = total number of lines executed, #Total = total number of lines in the entire application.

[3] The rcm application only contain 63 lines of source code, however it includes the applications `rcmKernel`, `rcmUtilities` and `rcmNtp`, and together with the it makes a total 19'250 lines.

From the results in Table 1 we draw conclusion regarding the following issues:

Code size: For larger applications our tool can symbolically execute in the order of 1000 lines of source code per second. Initialisation is normally only a small part of the application. As an example in the AXD 301 subsystem consisting of 57'310 lines of source code at most 1'068 out of these were executed in order to analyse one of its applications.

Language constructs Looking at the language constructs and library functions used by the OTP and AXD 301 subsystem application, we can conclude that they use all Core Erlang language constructs and many of the libraries.

Incomplete source code When analysing the AXD 301 subsystem parts of the code was missing, as was the remainder of the system. The tool could analyse all the applications after initialisation of configuration tables, with some increase of approximations made.

Precision For all OTP applications we have traced an execution of the application, and compared the process tree created with the trees created by the analysis. For all applications the trace generated tree was matched by one of the analysis generated trees.

For the AXD 301 applications we could not execute the incomplete subsystem. We have, however, showed the results to the senior engineers within the AXD 301 that provided us with the subsystem; they confirm that the analysis results correspond to the actual system.

The large number of trees generated by three of the applications (`os_mon`, `rcm` and `rcmKernel`) all arise from polling performed in the initialisation and results in a number of trees only differing in the number of sends and receives performed. If we ignore the number of differing send and receives only a few trees remain; in the case of `os_mon` only two out of the 200 trees would remain.

7. Conclusion

We have described a method for assessing failure recovery in Erlang applications by means of symbolic execution, were the process structure of the application is extracted automatically from the source code. The extracted process structure, although incomplete, is analysed automatically to find undesirable process structures. With these techniques we have automatically examined the supervision structures created by a number of industrial applications. The examination has either shown the supervision structures to be sound or where we should further investigate the application source code.

In the cases where we can not say anything conclusively, due to approximations, we can still use the result to guide us to places where we may have unwanted structures. The structures, conclusive or not, may be used to gain understanding of the application.

Future Work We are presently in the process of analysing applications from a real life system: the Ericsson AXD 301 ATM switch[Blau et al. 1999]. This work we believe will give inspiration not only on what properties to focus analysis but also how to perform the analysis.

Of the technical issues the most important is how to find information beyond the predefined OTP behaviours and in the dynamic part of the applications. We will first investigate the approach of combining evaluation of Core Erlang terms with finite state methods, where the dynamic parts are approximated by finite models extracted from the code.

References

G. Agrawal. Demand-driven construction of call graphs. In D.A. Watt, editor, *Proceedings of the* 9^{th} *International Conference on Compiler Construction (CC'00)*, volume 1781 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2000.

V. Amiranashvili. A rewriting logic formalization of core erlang semantics. Master's thesis, Aachen University of Technology, Germany, 2002.

J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in* ERLANG. Prentice Hall, 2nd edition, 1996.

T. Arts and M. Dam. Verifying a distributed database lookup manager written in erlang. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99–Formal Methods, Volume I, Proceedings of the* 1^{st} *World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 682–700. Springer-Verlag, 1999.

T. Arts and C.B. Earle. Development of a verified ERLANG program for resource locking. In S. Gnesi and U. Ultes-Nitsche, editors, *Proceedings of the* 6^{th} *International Workshop on Formal Methods for Industrial Critical Systems (FMICS'01), Paris*, 2001.

T. Arts and J. Giesl. Automatically proven termination where simplification orderings fail. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT:* 7^{th} *International Joint Conference on Theory and Practise of Software Development*, volume 1214 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

T. Arts and T. Noll. Verifying generic erlang client-server implementations. In M. Mohnen and P. Koopman, editors, *Proceedings of the* 12^{th} *International Workshop on the Implementation of Functional Languages (IFL'00)*, volume 2011 of *Lecture Notes in Computer Science*, pages 37–52. Springer-Verlag, 2001.

T. Arts, C. Earle, , and J. Derrick. Deveolpment of a verified erlang program for resource locking. *International Journal on Software Tools for Technology Transfer*, 5(2–3):205–220, March 2004a.

T. Arts, C. Earle, , and J. Penas. Translating Erlang to μCRL. In *In Proceedings of the International Conference on Application of Concurrency to System Design (ACSD2004)*, 2004b.

J. Barklund and R. Virding. Erlang 4.7.3 reference manual, draft (0.7). Ericsson, Computer Science Laboratory, www.erlang.org/download/erl_spec47.ps.gz, 1999.

S. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Buhrgard, T. Westin, and G. Wicklund. AXD 301: A new generation ATM switching system. *Computer Networks*, 31(6):559–582, 1999.

S. Blom, W. Fokkink, J.F. Groote, I. van Langevelde, B. lisser, and J. van den Pol. μcrl: A toolset for analysing algebraic specifications. In *Proceedings of the* 13^{th} *International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.

P. Borovanský, C. Kirchner, H. Kirchner, p. E. Moreau, and C. Ringeisen. An overview of elan. In *Proceedings of the International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

R. Carlsson. An introduction to core erlang. In *Proceedings of PLI'01 Erlang Workshop, Florence, Italy, September*, 2001. URL `http://www.erlang.se/workshop/carlsson.ps`.

R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core ERLANG 1.0 language specification. Technical Report 2000-03, Department of Information Technology, Uppsala University, Sweden, 2000.

R. Carlsson, K. Sagonas, and J. Wilhelmsson. Message analysis for concurrent programs using message passing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):715–746, July 2006.

K. Claessen and H. Svensson. A semantics for distributed erlang. In *In Proceedings of the ACM SIGPLAN 2005 Erlang Workshop, Tallinn, Estonia*, 2005.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

J.C. Corbett. Using shape analysis to reduce finite-state models of concurrent JAVA programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93, 2000.

OTP *Documentation*. Ericsson Utvecklings AB, 2000.

J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. Cadp: A protocol validation and verification toolbox. In *Proceedings of the 8^{th} International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer-Verlag, 2000.

L.-Å. Fredlund. *A Framework for Reasoning About* ERLANG *Code*. PhD thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, Sweden, 2001.

L.-Å. Fredlund and C. B. Earle. Model checking erlang programs: The functional approach. In *In Proceedings of the ACM SIGPLAN 2006 Erlang Workshop, Portland, USA*, 2006.

L.-Å. Fredlund and H. Svensson. Mcerlang: A model checker for a distributed functional programming language. In *Proceedings of the ICFP '07 conference*, volume 42 of *ACM SIGPLAN Notices*, pages 125–136. ACM Press, 2007.

L.-Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for erlang. *International Journal on Software Tools for Technology Transfer*, 4(4):405–420, 2003.

J. Giesl and T. Arts. Verification of erlang processes by dependency pairs. *Journal of Applicable Algebra in Engineering, Communication and Computing*, 12(1):39–72, 2001.

J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

G.J. Holzmann. *Design and Validation of Computer Protocol*. Prentice-Hall International, 1991.

G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proceedings of the 7^{th} International International SPIN Workshop (SPIN'00)*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–148. Springer-Verlag, 2000.

G.J. Holzmann and M.H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

F. Huch. Verification of ERLANG programs using abstract interpretation and model checking. In *Proceedings of the 4^{th} International Conference on Functional Programming (ICFP'99)*, volume 34 of *ACM SIGPLAN Notices*, pages 261–272. ACM Press, 1999.

F. Huch. Model checking ERLANG programs – abstracting the context-free structure. In *Proceedings of the 10^{th} International Workshop on Functional and Logic Programming (WFLP'01)*, 2001.

F. Huch. Model checking erlang programs – ltl-propositions and abstract interpretation. In *Proceedings of the 12^{th} International Workshop on Functional and (Constraint) Logic Programming (WFLP'03)*, 2003.

D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

M. Leucker and T. Noll. A distributed model checking tool tailored erlang. In *Proceedings of PLI'01 Erlang Workshop, Florence, Italy, September*, 2001.

Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In Chin Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, November 2004.

Tobias Lindahl and Konstantinos Sagonas. Practical subtype inference based on success typings. In *In Proceedings of the Eight ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 167–178. ACM Press, July 2006.

A. Lindgren. A prototype of a soft type system for erlang. Master's thesis, Computing Science Department, Uppsala University, Sweden, 1996.

Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 2nd edition, 1992.

S. Marlow and P. Wadler. A practical subtyping system for erlang. In *Proceedings of the 2^{nd} International Conference on Functional Programming (ICFP'97)*, volume 32 of *ACM SIGPLAN Notices*, pages 136–149. ACM Press, 1997.

C. Mullaparthi. Third party gateway. In *Proceedings of the 11^{th} International* ERLANG*/OTP Users Conference (EUC'05)*. Ericsson Utveckling AB, 2005.

M. Neuhäußer and T. Noll. Abstraction and model checking of CORE ERLANG programs in MAUDE. In G. Denker and C. Talcott, editors, *Proceedings of the 6^{th} International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 147–163, 2007.

F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

H. R. Nielson, T. Amtoft, and F. Nielson. Behaviour analysis and safety conditions: A case study in CML. In *Proceedings of the 1^{st} International Conference on Fundemantal Approaches to Software Engineering (FASE'98)*, volume 1382 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1998.

T. Noll. A rewriting logic implementation of erlang. In M. van den Brand and D. Parigot, editors, *Proceedings of the 1^{st} International Workshop on Language Descriptions, Tools and Applications (ETAPS/LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

T. Noll. Term rewriting models of concurrency: Foundation and applications, 2003.

T. Noll and C.K. Roy. Modeling erlang in the π-calculus. In *In Proceedings of the ACM SIGPLAN 2005 Erlang Workshop, Tallinn, Estonia*, 2005.

S.-O. Nyström. A soft-typing system for erlang. In *In Proceedings of the ACM SIGPLAN 2003 Erlang Workshop, Uppsala, Sweden*, 2003.

J.h. Reppy. Concurrent ml: Design, application and semantics. In P.E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 165–198. Springer-Verlag, 1993.

C.K. Roy, T. Noll, B. Roy, and J.R. Cordy. Towards automatic verification of erlang programs by π-calculus translation. In *In Proceedings of the ACM SIGPLAN 2006 Erlang Workshop, Portland, USA*, 2006.

M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Tekenekekzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, 1995.

E. Stenman. Betting on fp (and winning?). In *Proceedings of the 13^{th} International* ERLANG*/OTP Users Conference (EUC'06)*. Ericsson Utveckling AB, 2006.

C. Wiklander. Verification of erlang programmes using spin. Technical report, Department Of Teleinformatics, Royal Institute of Technology, Sweden, 1999.

A.G. Wouters. Manual for the μcrl toolset (version 2.07). Technical Report To appear???, CWI, Amsterdam, 2001.

Teaching Erlang using Robotics and Player/Stage

Sten Grüner
University of Kent, UK
sten.gruener@gmail.com

Thomas Lorentsen
University of Kent, UK
tom@thomaslorentsen.co.uk

Abstract

Computer science is often associated with dull code debugging instead of solving interesting problems. This fact causes a decrease in the number of computer science students which can be stopped by giving lectures on an interesting context like robotics. In this paper we introduce an easily deployable and extensible library which allows programming a popular robot simulator in Erlang. New possibilities for visual, simple and attractive teaching of functional languages are open.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; I.2.9 [*Robotics*]: Commercial robots and applications; K.3.2 [*Computers and Education*]: Computer and Information Science Education

General Terms Experimentation, Languages

Keywords Erlang, Player/Stage, Teaching

1. Introduction

Computer science students of the 21st century are not willing to write "Hello World" applications any longer and demand for a more fascinating and innovative learning environment. Several years ago Patterson (2005) drew attention to the falling number of CS students despite increasing number of career opportunities. The fact of falling student interest has even led to the closure of several CS departments in the United Kingdom. Rashid (2008) points to the same fact in 2008 suggesting a solid association of computer science with debugging code in front of a computer screen all day instead of solving real-world problems to be responsible for the downturn of the student interest.

This development is not surprising since similar observations have already been made by Fisher and Margolis (2002) in the early 2000s when a lack of female interest in studying computer science was associated with women's wish to have a "context of computing" instead of "hacking for hacking's sake".

Robotics, a field which can inspire young people quickly, has been suggested by Blank (2006) as one of the key-fields to restore the personality of computer science by giving students a real hands-on experience and discarding the classical pedagogical "correct answer" paradigms. Patterson (2006) points out several challenges for the CS education in the upcoming century which unfortunately are still missed by many CS departments. The key points among these are concurrency, the usage of cutting edge libraries from the first educational year and open source programming.

In this paper we will introduce our idea for solving these problems in terms of:

- Introducing the Erlang-based Kent Erlang Robotic Library (KERL), available from `http://kerl.sf.net`, for the open source robot middleware Player (Section 3). The library can be used in connection with both physical robots as well as a simulation framework – Player/Stage.
- Describing the created infrastructure such as installation scripts, examples and assignment ideas (Section 4) which can be used as an out-of-a-box environment for a functional programming course.
- Discussing alternative approaches (Section 5) as well as future improvements (Section 6).

With our approach we try to reach several educational goals proposed by Patterson. The usage of a functional, highly-concurrent language, which becomes more and more popular in the industry, is a crucial part of the whole educational stance and should help students not only to learn the syntax and semantics of Erlang more quickly, but also to work in Erlang's natural environment, concurrent real-time systems, from the very beginning. The presented library was developed during an undergraduate final-year project at the University of Kent in the academic year 2008/09.

2. Background

2.1 Erlang

In the 1980s Ericsson started to search for a new programming language to be used in telephone exchanges. The research showed that descriptive languages were better suited for programming concurrent telecommunication tasks than imperative languages. Erlang was developed for this purpose. It is a functional language, influenced by logical and imperative languages (Armstrong 1997). Designed for concurrent, fail-tolerant systems, Erlang's main running component is a process which can be understood as a lightweight thread which is cheap to initialise and run. Processes do not share any resources and interprocess communication is achieved by the use of message parsing. Furthermore, processes can monitor and restart each other in case of a failure. This philosophy makes developing distributed applications simple and comfortable.

Erlang syntax is convenient and allows a smooth transition from well-known imperative languages. Another important aspect is the concurrency specialisation of the language, which makes Erlang perfectly suitable for robotic applications because they are inherently concurrent.

2.2 Player

A robot is built up of devices for sensing and manoeuvring in an environment. Different devices provide different interfaces, e.g.

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.

movements require a speed and direction. Some devices may be used for gathering information such as laser sensors that produce an array of readings. The wide range of proprietary devices makes porting applications difficult.

Player[1] has been developed to solve this problem (Gerkey et al. 2003). It provides a hardware-independent API allowing rapid development of distributed systems (Gerkey et al. 2001).

Each device is represented by an independent proxy. These can be shared by different applications, e.g. an application may log data while another displays camera feeds on a screen.

Player is effectively language independent since it uses TCP for communication and most modern programming languages support sockets.

Clients have already been written for Java and Python (Collett et al. 2005). LibPlayerC, utilised in our project, is a widely used C library providing functions to implement a Player client.

2.3 Stage

Stage compliments Player by providing a graphical robot simulation in a 2D indoor environment. Robots from different vendors (e.g. Pioneer 3-DX[2]) equipped with various sensors (e.g. laser, sonar and fiducial) can be simulated. Multiple robots operate inside of a black and white image, often denoted as world. A world has several parameters, e.g. the number and type of the simulated robots and their starting positions. Stage also allows user interaction during runtime, e.g. allowing to monitor and modify the positions of robots.

One goal of the Stage development was to produce "good enough fidelity" (Gerkey et al. 2003) meaning that the user can not usually tell the difference between a simulated and a physical device making it a perfect tool for testing robotic application. As reported by Gerkey et al. (Gerkey et al. 2003), a program that performed well in Stage could be transferred onto a physical robot with little or no modification. Stage is considered to provide an accurate real world simulation. We think that the simulation realism is likely to draw more student interest, as well as a wide range of applications using KERL. Another feature making Stage perfect for educational purposes is the fact, that it was designed to simulate multi-robot environments and has linear computational scaling respective to population. Therefore Stage allows a simulation of multiple robots controlled by KERL in real time.

The Player/Stage Project is used for teaching and research in various computer science departments as well as laboratories around the world (Gerkey et al. 2003). We hope, that KERL can increase interest in this great project.

3. KERL

KERL consists broadly of two major components: the linked-in driver and several Erlang modules. Together they harness a simple interface to the Player server. They were designed with the following goals in mind.

Teachable – the amount of information students are required to know in order to be able to start developing concurrent programs using KERL should be minimal.

Maintainable – the time and effort required to install the library should be small.

Featureful – even if a simplified interface for the beginners is provided, KERL should be able to support all the features of Player.

Extensible – the code should be modular making it simple for developers to extend KERL with additional features.

Portable – KERL should be usable on different platforms supporting Erlang.

Fast and Concurrent – KERL should be able to handle multiple robots with various devices without major performance problems; real-time performance is vital for robotic applications.

Open Source – KERL should avoid the use of any proprietary technologies and encourage further development.

3.1 Wall Follower Example

Controlling a robot in KERL is very simple. The minimalistic example in Listing 1 shows how to make a robot navigate around a world following convex walls. Using two processes, one for controlling the motors and another to detect the proximity of the walls, the robot can be navigated successfully around a world. When the collision detection process detects an obstacle, it passes a message to the movement process. When this happens the movement process will let the robot rotate until a message is passed from the collision detection process telling it is safe to move forward again. The robot being controlled in Stage using the example code is shown in Figure 1.

```
-module(wallfollow).
-export([start/0, movement/1, collision/2]).

% Runs the wall follower with a single robot
start() ->
    init(rih:init(mrh:start(), 0)).

% Detects an error when initialising
init({error, Error}) ->
    io:format("Error: ~p~n", [Error]);

% Spawns the collision and movement processes
init(Rid) ->
    Pid = spawn(?MODULE, movement, [Rid]),
    spawn(?MODULE, collision, [Pid, Rid]),
    Pid.

% Movement process
% Controls the robot's movements
movement(Rid) ->
    receive
        {collision, true} ->
            mvh:rotate(Rid, speed, 10);
        {collision,false} ->
            mvh:move(Rid, speed, 0.5)
    end,
    movement(Rid).

% Collision process
% Reads the lasers and checks for collisions
collision(Pid, Rid) ->
    % 360 laser results representing 180 degrees
    {_, Results} = dvh:results(Rid, lasers),
    Front = lists:sublist(Results, 90, 180),
    is_collision(Pid, lists:min(Front)),
    timer:sleep(50),
    collision(Pid, Rid).

% A proximity check using the smallest laser
% measurment
is_collision(Pid, Min) when Min < 1 ->
    Pid ! {collision, true};
is_collision(Pid, _) ->
    Pid ! {collision, false}.
```

Listing 1. wallfollow.erl

We would like to emphasise the fact, that even this small example already makes use of Erlang's concurrent task philosophy and not of the common sense-think-react approach used in robotic applications. The details of the robot initialisation and the movement control will be reviewed later in this section.

[1] http://playerstage.sourceforge.net/

[2] http://www.activrobots.com/

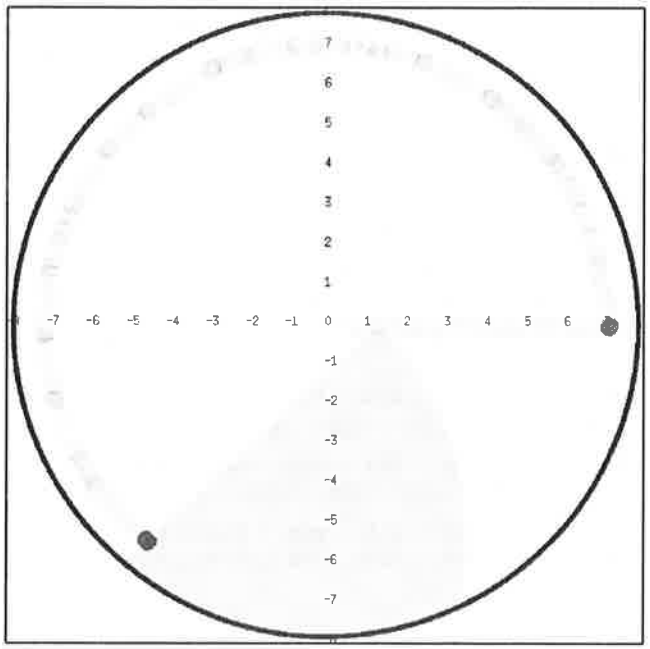

Figure 1. Stage shows the trail of the robot being controlled by the wall follower. The shaded area represents the robot's laser sensor.

3.2 Architecture

An overview of the architecture can found on the Figure 2. Dashed lines mark the parts of the KERL library – the low-level module communicating to LibPlayerC through a linked-in driver and some high-level modules providing a programming interface.

3.2.1 KERL Driver

The driver went into a few development cycles where we started with a very simple driver and then built more complicated functionality into it. The driver is modular to help with refactoring and makes adding functionality simple. This also allows the same code to be used to support other robotic libraries or even other languages with little effort.

We had studied LibPlayerC and decided to utilise it rather than to implement the Player protocol itself. This pragmatic decision was based on the fact, that the understanding of the protocol involved exploration of Player's source code while the C library is documented and referenced in the Player/Stage manual. Another advantage of using LibPlayerC is the forward compatibility – in case of a protocol update only the library needs to be replaced. We also decided that this common library is known to many developers who may want to support KERL. A disadvantage of our approach is the handling of concurrent TCP connections by the C side regardless the better suitability of Erlang for this task. Another disadvantage is the need for interfacing with the C library which was very problematic to implement. Looking back, we think that the implementation of the Player protocol directly in Erlang over TCP would be more elegant than using LibPlayerC. However we doubt if it is worth losing the mentioned advantages.

In order to integrate the library into Erlang we used Erlang's foreign function interface (FFI) known as Ports. A port appears to Erlang as a normal process which can pass and receive messages. Messages are passed to the C driver as a serial byte stream.

Using Erlang Interface (EI) Erlang terms are encoded from and decoded into C primitives. We initially used this to determine which function to call by passing a string that identified the function that

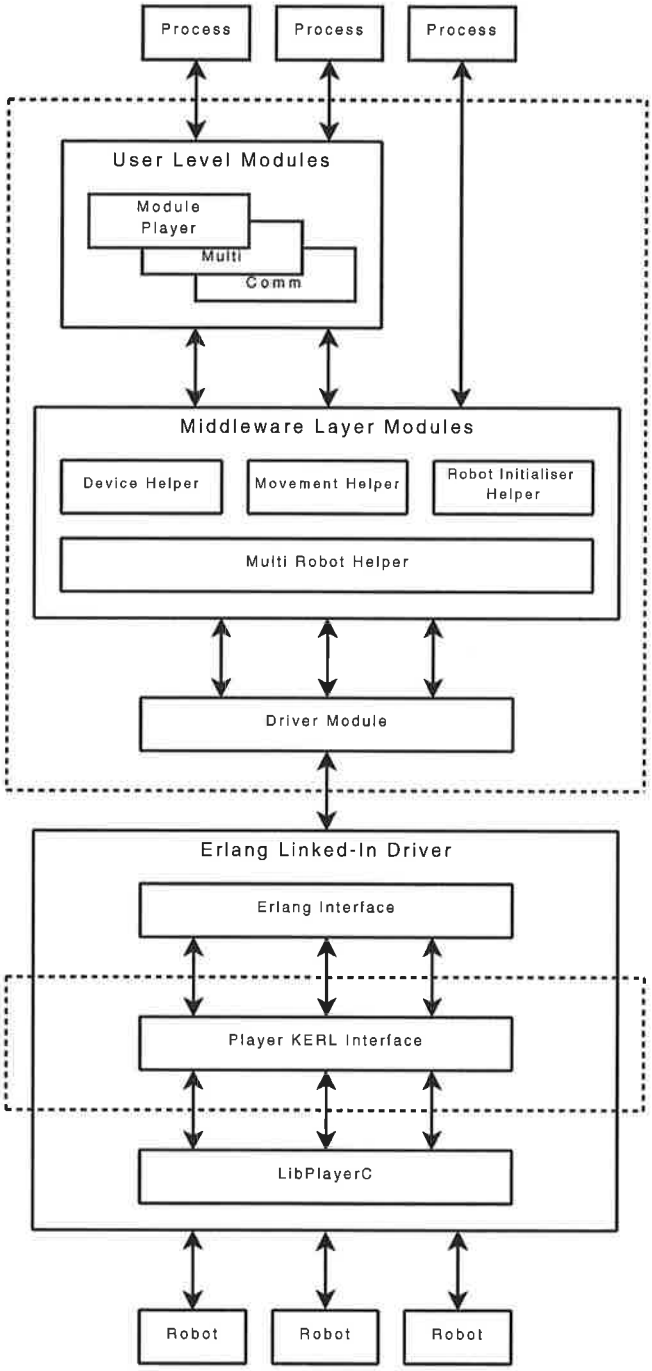

Figure 2. KERL's architecture. Dashed lines show the components we developed.

should be called. The function that was called would then decode the rest of the passed terms.

Initialising a robot and its devices with Player is simplified by our driver. For every initialised robot an instance of the Player client is stored in a C++ Map located in shared memory. This approach allows accessing every robot from different driver instances. Once the connection has been made, our driver will then automatically detect what devices are available and make them ready for use. Each robot is associated with a unique hash which is used as the key to the map and tracked by an Erlang module.

Sensor results are returned in the form of a list of variable length. EI makes it easy to return a fixed size list, but dealing with dynamic sizes is a much more complicated issue. The data is obtained from Player first and saved in an auxiliary array. A sufficient amount of memory is then allocated for an Erlang list term that is passed to Erlang. Failing to determine the exact amount of needed memory would cause the driver to crash. Every value passed is paired with an Erlang term type. The amount of required memory can then be calculated from this.

Some performance issues were discovered while working with multiple robots. For example, robot initialisation was slow because it was done sequentially. We noticed that robots' movements were not synchronised which was caused by commands not being passed in a timely manner. A concurrency problem in the Erlang driver caused this performance bottleneck.

In order to solve this issue the synchronous driver was replaced by an asynchronous one. There are no major differences between the synchronous and the asynchronous driver; the program flow and the memory management were the only parts modified. The asynchronous driver helped to move KERL towards a more complete and usable state and allowed us to refocus our attention on high-level features.

3.2.2 Accessing the Driver from Erlang

While a linked-in driver is used to communicate with LibPlayerC, the *driver* module in Erlang carries out the message handling. The module locates the driver first. After the driver is loaded a process is started which passes messages from KERL modules to the port and vice versa.

The synchronous driver was able to be spawned multiple times so our first approach was to spawn a new driver for each robot. A message could be passed and the expected data would be returned because the single driver could only do one task at a time. This also allowed the driver itself to save the robot's ID, which is the key to the robot map stored in the driver. A few minor changes were required to support the asynchronious driver. We modified the driver to take the robot ID as a parameter and moved the handling of robot IDs to a separate part of KERL.

In order for the driver to work concurrently we needed to keep track of the original caller. The best way achieve this was to pass the caller's process ID (PID) into the port driver instead of keeping track of it internally. Only the process that spawned the instance of port driver can pass messages to it, which is why we only run a single *driver* module process.

Messages can be continually passed into the driver without bigger impacts on performance. There were, however, performance issues found when spawning large numbers of threads in a virtual machine (like the one installed in Section 4).

3.2.3 High-level KERL

In this section we will place special emphasis on the high-level KERL modules written in Erlang. The modules philosophy has to differ from the driver since they are directly accessible by the user.

We divided the modules into two layers: the middleware layer and the user layer. The middleware provides the required functions to control robots, such as initialisation, sensor readings and movement. These functions are then utilised at a higher level by the user layer.

This approach has many benefits, e.g. the modularity and the extensibility of the code. The modularity allows to provide a simple interface that helps students to learn KERL quickly.

Middleware Layer The middleware layer consists of following modules:

- Multiple Robot Helper (*mrh*) which provides control of multiple robots via a single asynchronous driver instance,
- Robot Initialisation Helper (*rih*) which provides functions to initialise robots,
- Movement Helper (*mvh*) which provides simple functions for robot control, such as movement and rotation,
- Device Helper (*dvh*) which allows controlling and reading data from robot devices such as the laser sensors.

The best way to understand the usage is to consider an example: connect to Player server on the local host, initialise a robot and move it for one meter.

```
Driver = mrh:start().
Robot = rih:init(Driver, 0).
mvh:move(Robot, distance, 1).
```

The first command to be executed is the `mrh:start()` command of the multiple robot helper module which starts the asynchronous driver and returns its PID. This is used to spawn a process for each robot. All the middleware functions accept the PID of the robot process. In order to initialise a robot we need to call `rih:init()`. The last function is the `mvh:move()` function which moves the robot for one meter in the simulated world.

Please note that the last call will unblock instantly while the robot is moving since all the middleware functions are non-blocking.

The middleware layer modules provide simple and modular access to the main robot's functions. It extends on the basic functions to give more or less control depending on user's requirements. For example the Robot Initialisation Helper provides a very simple or a highly configurable method of robot initialisation. A sequential and concurrent method of initialising multiple robots is also provided.

User Layer The user layer provides abstractions and extended functions needed for a comfortable process interaction and device handling. It consists of two main modules:

- Player Module (*player*) which provides a simple interface to the middleware functions,
- Comm Module (*comm*) which is used to manage a mailbox of a single process,
- Multi Module (*multi*) which provides a framework of interprocess communication.

Similarly to Armstrong's book (2007), we provide a gentle introduction to Erlang before moving towards concurrency concepts. The *player* module is intended to be the first module which a user encounters in KERL. It provides almost all the functions of all the middleware layers in one module. An important feature is the PID binding – once a robot has been initialised its PID is bound to the process and the user will not have to remember the PID again. As a careful reader may have noticed this simple approach has a restriction: only one robot can be controlled by a process. While all the movement functions in *mvh* are non-blocking, many of the movement functions in *player* are artificially made to block until the robot reach its position and stops. Let us consider the usage of

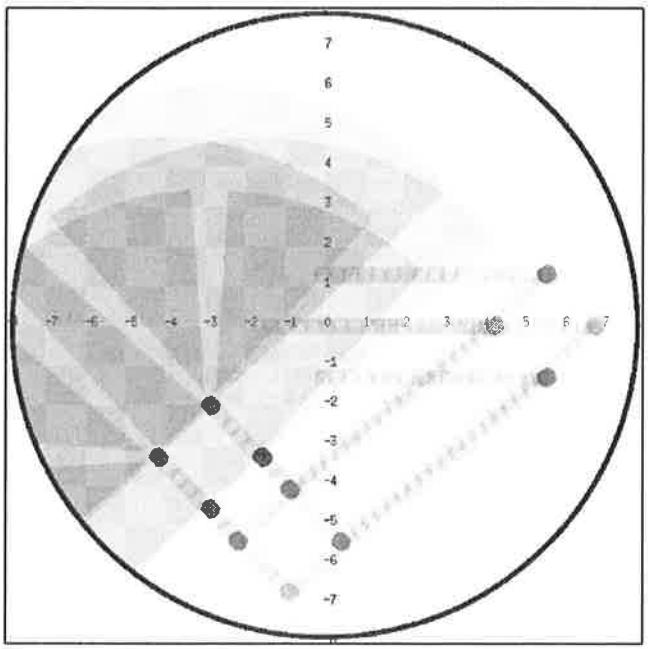

Figure 3. A group of four robots bouncing off the walls in a diamond formation.

the module briefly; the selected example has the same functionality as the previous one – initialise the robot and let it move for one meter.

```
Driver = player:start().
player:init(Driver, 0).
player:move(distance, 1).
```

We observe that the function signatures are shorter and more convenient to use since the user does not need to track the PID of the robot anymore. The simplifications of the *player* module provide an ad-hoc usage of KERL. The first working application can be created very quickly by using it.

The *comm* module currently provides only a function called `comm:discard()` which allow the process to discard undesired messages from it's mailbox. An example of usage of this function is given in the case-study below.

The *multi* module is designed to provide a framework for interprocess communications and extends the features of Erlang with some useful functions. The main entity of the module is the dispatcher process which controls the group of processes associated with the robots. The dispatcher process itself can be part of a process group. This possibility allows the creation of hierarchical tree structures of processes. We will show the usage of this function in the next section.

The high-level modules have been designed to provide both a simple and detailed user interface to KERL. The modules are kept as simple and intuitive as possible to help teaching Erlang's programming philosophy.

3.2.4 A Case Study: Multibouncer

In order to illustrate the usage of user layer modules let us consider the following task: a group of robots should move in a world while staying in their starting formation. This behaviour is illustrated in Figure 3.

All the robots share the same code, which allows any of them to accept and give commands when a wall is sensed. They all need to be commanded by the robot that senses a wall first in order to avoid collisions. In this example called *multibouncer* we want to demonstrate how the *multi* module is used to achieve this goal.

```
main(Dispatcherid, Driver, Playerid) ->
        % Initialise the robot
        player:init(Driver, Playerid),
        loop(Dispatcherid).

loop(Dispatcherid) ->
        % Time-synchronisation
        multi:barrier(Dispatcherid),
        % Travel to a wall and rotate
        travel(Dispatcherid),
        % Infinite loop
        loop(Dispatcherid).
```

Listing 2. Listing main() and loop() functions.

We use the *player* module for the robot control in this case study. The usage of this user layer module will allow only one process to control the robot in contrast to the wall follower example. This reduces the complexity of dealing with individual devices allowing us to concentrate on interprocess communication.

The first steps are: initialise the driver, initialise a dispatcher process to handle the interprocess communication and at last, spawn 4 control processes using the `multi:create_robot()` function.

```
Driver = player:start(),
Disp = multi:start(),
% Spawn four processes running the main function
% and give them the Dispatcher's and Driver's PID
lists:map(fun(X) -> multi:create_robot(Disp,
    ?MODULE, main, [Disp, Driver, X]) end,
  lists:seq(0,3,1)).
```

Let us take a more precise look at the `main()` function (refer to Listing 2) which receives the PIDs of the dispatcher, the driver and the ID of the Player robot controlled by it. The function remains in a loop which is time-synchronised by using `multi:barrier()` function. The barrier will only unblock if all processes belonging to the same group call it.

The `travel()` function sets the formation in motion until one of the robots locates a wall, then the group stops and rotates so that they can safely continue their journey. The basic structure of this function is listed in Listing 3. Every robot senses the environment every 40ms and alerts the group if it detects an obstacle by broadcasting a warning: all the robots stop when the wall is sensed. The `multi:broadcast()` function allows a process to send messages to all group members except itself. The originator of the warning becomes the master, the recipients become slaves for the current turn.

A naïve implementation of the `master()` and `slave()` functions relied on the time synchronisation of the robots' rotations. The master announced the start and the end of the rotation with a broadcast. As expected due to the nature of the concurrent system this approach failed – after the first rotation robots did not move parallely anymore and therefore collided.

How can we remedy this problem? Our idea was to use the robot's odometer sensor which can be accessed by calling the `player:get_position()` function. It returns the robot's position in a local coordinate system with its starting point as the origin. Every robot saves its initial position (the one before calling the `travel()`) function. After a rotation the master then computes the difference between the saved and the actual position which is broadcasted. The slaves receive the position difference and add it to their initial position. In order to cope better with the measurement errors of the odometer only the differences are transferred.

```
travel(Dispatcherid) ->
 receive
   {_, {stop}} ->
     player:stop(),
     % Do the slave part
     slave()
 % Sense the wall every 40 ms
 after 40 ->
   % Read lasers
   {_, Results} = player:results(lasers),
   case lists:min(Results) < 1 of
    % No obstacle found -> keep moving
    false ->
       player:move(speed, 0.35),
       travel(Dispatcherid);
    % I see an obstacle!
    true ->
       % Alert the group and stop
       multi:broadcast(Dispatcherid, {stop}),
       player:stop(),
       % I am the master for the turn
       master(Dispatcherid)
   end
end.
```

Listing 3. The travel() function without voting.

The modified functions resulted into a stable system where robots were able to stay in formation.

However, this system was not perfect: occasionally, two of the robots sensed the wall at the same time and both tried to rule the group which resulted in chaos. In order to solve this problem the `multi:voting()` function was introduced. The modified body of the `travel()` function is listed in the Listing 4. Similarly to a barrier the `multi:vote()` function needs to be called by all the processes in the group in order to unblock. After unblocking a randomly chosen PID of one of the participants (the processes which set the participation flag) is returned. We also make use of `comm:discard()` function in order to discard the duplicates of the warning message sent by more the one potential masters. The modified system works stable and completes the task.

We hope that the presented case study demonstrate the user level functions provided by KERL. Describing all the features provided by the modules is beyond the scope of this paper – please refer to the documentation and tutorials packaged with KERL.

4. Infrastructure

Making KERL suitable for teaching means not only to have a student friendly programming environment, but also to help teaching staff to establish this environment quickly. The provided infrastructure consists of:

Installation scripts We provide a script which installs only the dependencies, required by Player/Stage, and configures an Ubuntu Server creating a minimal distribution using 2.5GB space. This distribution can be run as a virtual appliance inside of freeware VMware Player supported by Windows. A second script automatically installs Player/Stage for GNU/Linux distribution. This reduces the time consuming installation to 30 minutes. A tutorial on Player/Stage installation and getting started with KERL is also provided. These components create an environment suitable for teaching with KERL.

Live CD A modified Ubuntu Live CD with preinstalled Player/Stage and KERL is available. It provides a possibility to try KERL without installing anything on the hard drive.

```
travel(Dispatcherid) ->
 receive
   {_, {stop}} ->
     player:stop(),
     % I do not want to become a leader
     multi:vote(Dispatcherid, false),
     % Discard received duplicates
     comm:discard({stop}),
     % Do the slave part
     slave()
 % Sence the wall every 40 ms
 after 40 ->
   % Read lasers
   {_, Results} = player:results(lasers),
   case lists:min(Results) < 1 of
    % No obstacle found -> keep moving
    false ->
       player:move(speed, 0.35),
       travel(Dispatcherid);
    % I see an obstacle!
    true ->
       % Alert the group and stop
       multi:broadcast(Dispatcherid, {stop}),
       player:stop(),
       % Discard received duplicates
       comm:discard({stop}),
       % Save my PID
       MyPid = self(),
       % Participate as a master-candidate
       Leader = multi:vote(Dispatcherid, true),
       case Leader of
         MyPid ->
           % I have won the election
           master(Dispatcherid);
         _ ->
           % I have lost the election
           slave()
       end
   end
end.
```

Listing 4. The travel() function using voting.

Examples Several examples are included which show the usage of different KERL modules. We provide examples of using simple, as well as multiple robots communicating with each other. The examples should allow an easy ad-hoc KERL learning.

Tutorials We made tutorials, explaining difficult examples, which should help students to understand advantages and disadvantages of several high-level techniques.

Assignment Idea The task for a lecture is to extend the presented wall follower to cope with non-convex shapes. Additionally it can be extended into a maze solver which which makes use of the right hand rule or the Pledge algorithm.

5. Related Work

There is a lot of scope for using robotics as an effective vehicle for teaching Computer Science, with a lot of research taking place in this area (Blank 2006). However, complicated, incompatible hardware and software, with high set up costs, make it difficult to work effectively with robots. Also, universities often lack the facilities to provide every student with a chance to work with robots, especially when it comes to working in their own time.

Many of these problems cannot be resolved with physical robotic hardware, therefore a growing interest in simulators has been observed. Player/Stage has been found to fit successfully

into teaching applications (Anderson et al. 2007). The learning curve is low due to having a very simple interface. Imperative languages like Java, usually taught during the first year, can be used with Player/Stage. For this reason it is possible to focus more on teaching robotic control algorithms and paradigms using a familiar language.

An example of the usage of the Player/Stage is a concurrency module taught at the University of Kent which explored programming robots using the Occam-π language. The module went through several phases before it matured. The Lego Mindstorms NXT Transterpreter (Simpson et al. 2007) was used in the early stages. It was, however, noticed that better documentation and a more complete implementation was required for teaching purposes. The latter problems led to a development of RoboDeb – a VMware based Linux distribution with built in Player/Stage (Jacobsen and Jadud 2007). Simulating robots enabled students to control multiple robots what would have been impossible for the discussed reasons.

Robotics is used in conjunction with functional languages for education. Wakeling (2008) sees clear advantages in this problem-based learning approach allowing students to learn Haskell in an exciting manner. Therefore, he proposes to use a Haskell interpreter on a Lego Mindstorms NXT robot. One of his proposed assessments for the module is the building and programming of a line-following robot. One of the disadvantages are the high setup expenses, due to the high costs of the Lego Mindstorms kits. We do not know if the interpreter is currently used by any university.

Erlang has been used for teaching highschool pupils interested in computer science at the University of Kiel, Germany (Huch 2007). Multikara[3], a program simulating ladybugs (simplified robots) in a discrete space, was modified to support Erlang as a programming interface. Erlang was preferred to finite state automate and Java, because the recursion concepts seemed to be more intuitive for pupils. An example of this superiority is the fact that many pupils were able to implement backtracking during their first day. The overall results were very positive; pupils learned a new functional programming concept very quickly with help of Kara's visual environment.

Erlang controlled robots can compete with traditional imperative language based systems. In his work Santoro (2007) introduces a framework for controlling autonomous mobile robots. The latter provides various layers of control starting with hardware abstraction and ending with a complicated reasoning system. A robot controlled by the framework achieved 14th place among more than 50 competitors at the Eurobot 2007 competition. In contrast to C/C++ based systems, the Erlang based framework allowed the rapid development of reusable and maintainable code less prone to concurrency related problems. Unlike Santoro's framework, our library utilitises Player for hardware abstraction and does not currently provide any intelligence. However, users can benefit from the same advantages as described above while using KERL.

6. Further Work

KERL matured to a stable, usable, concurrent library which is ready to use for teaching Erlang. However, further work and several improvements of a different nature can be done.

The limited amount of time available for our project made it impossible to implement all features we wanted. However, we tried our best to make KERL easy to extend and hope it would not be a big problem to add support of features Player offers like fiducial or sonar sensors. We spent most of the development time working on the C code needed for Erlang–Player communication. The project would benefit from more user layer modules providing complex algorithms, e.g. navigation or robot recognition. The provided examples are very artificial and explain the usage of single functions – more real-world related examples are substantial.

Better error handling in our Erlang modules would be appreciable. We would like to provide two types of error management. One to be used by developers to provide details about what went wrong, reporting back on states of the robots. A second type would be for students, hiding problems from them, but attempting to seek solutions intelligently. For example, if the robot connection was lost then we would like to attempt a reconnection automatically.

Our original aim was to support Windows, Linux and OSX. Our solution to support Windows by providing a VMware image of Ubuntu resulted in speed issues. This was due to the virtual machine not providing fast enough support for OpenGL required by Stage. KERL itself can run on every platform supporting Erlang.

Some other features were considered but not implemented, mostly because of technical limitations in Stage. Such a feature is the spatial synchronisation, e.g. critical regions of the world or object interaction used to provide more challenging tasks.

We hope the described features will be implemented in future.

7. Conclusions

Despite problems during the development, mostly related to the poor documentation of both EI and Player, we have managed to reach the level of functionality and stability required for a public release. However, KERL's potential has not yet been fully realised. The modular structure of the library allows extension with minimal effort, encouraging ongoing support from the open source community.

KERL successfully connects the world of robotic applications with a powerful concurrent language. By being simple and transparent to the end user, it can be a valuable tool in teaching Erlang.

We hope that our work will be used to improve teaching of functional languages, to spread Erlang in the educational domain and to inspire the next generation of computer scientists. KERL has yet to be tried and tested for teaching Erlang and we welcome CS departments to use it. Let us know your opinion about it, as we eagerly await your feedback.

Acknowledgments

We would like to thank Allen Brooker and Mathew Champ for their contribution. Without their help the project would have not been accomplished on time. We also thank Dr. Olaf Chitil for his patience and supervision throughout the project.

References

Monica Anderson, Laurence Thaete, and Nathan Wiegand. Player/stage: A unifying paradigm to improve robotics education delivery. In *Robotics: Science and Systems: Workshop on Research in Robots for Education*, 2007.

Joe Armstrong. The development of Erlang. *SIGPLAN Not.*, 32(8):196–203, 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/258949.258967.

Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007. ISBN 193435600X, 9781934356005.

Douglas Blank. Robots make computer science personal. *Commun. ACM*, 49(12):25–27, 2006. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1183236.1183254.

Toby H.J. Collett, Bruce A. MacDonald, and Brian P. Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proc. of the Australasian Conf. on Robotics and Automation (ACRA)*, Sydney, Australia, Dec 2005.

[3] http://www.swisseduc.ch/compscience/karatojava/multikara/

Allan Fisher and Jane Margolis. Unlocking the clubhouse: the carnegie mellon experience. *SIGCSE Bull.*, 34(2):79–83, 2002. ISSN 0097-8418. doi: http://doi.acm.org/10.1145/543812.543836.

B.P. Gerkey, R.T. Vaughan, K. Stoy, A. Howard, G.S. Sukhatme, and M.J. Mataric. Most valuable player: a robot device server for distributed control. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 3, pages 1226–1231 vol.3, 2001. doi: 10.1109/IROS.2001.977150.

Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *In Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

Frank Huch. Learning programming with Erlang. In *ERLANG '07: Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 93–99, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-675-2. doi: http://doi.acm.org/10.1145/1292520.1292534.

Christian L. Jacobsen and Matthew C. Jadud. Concurrency, Robotics, and RoboDeb. In *Proceedings of AAAI Robotics and Education*, Palo Alto, CA, USA, March 2007. American Association for Artificial Intelligence. URL `http://www.cs.kent.ac.uk/pubs/2007/2875`.

David A. Patterson. Restoring the popularity of computer science. *Commun. ACM*, 48(9):25–28, 2005. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1081992.1082011.

David A. Patterson. Computer science education in the 21st century. *Commun. ACM*, 49(3):27–30, 2006. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1118178.1118212.

Rick Rashid. Image crisis inspiring a new generation of computer scientists. *Commun. ACM*, 51(7):33–34, 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1364782.1364793.

Corrado Santoro. An erlang framework for autonomous mobile robots. In *ERLANG '07: Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 85–92, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-675-2. doi: http://doi.acm.org/10.1145/1292520.1292533.

Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. A Native Transterpreter for the LEGO Mindstorms Rcx. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 339–348. IOS Press, Jul 2007. ISBN 978-1-58603-767-3.

David Wakeling. A robot in every classroom: robots and functional programming across the curriculum. In *FDPE '08: Proceedings of the 2008 international workshop on Functional and declarative programming in education*, pages 51–60, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-068-5. doi: http://doi.acm.org/10.1145/1411260.1411268.

Development of a Distributed System Applied to Teaching and Learning

Hugo Cortés Mónica García Jorge Hernández Manuel Hernández Esperanza Pérez-Cordoba Erik Ramos *

Universidad Tecnológica de la Mixteca

{hugoe,mgarcia,jahdezp,manuelhg,mapercor,erik}@mixteco.utm.mx

Abstract

The emergence of networked computers has originated new technologies for teaching and learning, particularly, the technology of learning management systems. We have applied Erlang to deal with the concurrent part of a distributed system to support teaching and learning tasks. We have also employed declarative programming together with some formal tools to elaborate the specification and the conceptual model of the system and some extreme programming techniques to deal with some issues of software development. We show how Erlang supports the transition from the specification to the implementation, and the whole concurrent and computational process of our distributed system.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Concurrent, Distributed and Parallel Programming; H.5.3 [*Information System*]: Computer-supported cooperative work; K.3.1 [*Computers and Education*]: Computers uses in Education—Computer-assisted Instruction (CAI)

General Terms Erlang, Distributed Programming, Learning, Functional Programming, Extreme Programming

Keywords Concurrency, Erlang, Passing Messages, CCS, UML, Bloom's Taxonomy

1. Introduction

In this paper we report an experience based on using Erlang to deal with the concurrent part in the specification and a partial implementation of a distributed system. This distributed system is intended to support some of the concurrent tasks of teaching and learning. With this proposal we want to complement the traditional classroom environment through a computerized teaching-learning tool. We use Erlang to help us in some concerns, including the fact that concurrent programming is complex by itself. Erlang ameliorates this complexity by using built-in functions to establish communication among processes via message-passing and by omitting shared resources. In addition, Erlang is a functional programming language with automatic processes management and built-in fault-tolerance mechanisms. Erlang gave us a realistic prototype of our system in a short time, hiding low-level implementation details, and allowing us to concentrate on the logic of the system.

To implement our distributed system we experiment with a method of software development: the combination of extreme (Bec99) and declarative programming (Mac90; Bir98). When using extreme programming we attempt to produce code by involving programmers and customers from the beginning of a project. Because Erlang code is so clear it can be seen as an "executable specification," extreme programming acts as a method for rapid prototyping. Departing from traditional techniques of rapid prototyping, the prototype is not disposable; instead, the prototype can be enhanced in some aspects like fault-tolerance, improved in its efficiency, and enriched with new characteristics.

In this sense, modeling is helpful for all phases of system development: from establishing and expressing the specifications to systematically deriving a correct-by-construction implementation. As asserted in (Bro07), when engineering software systems, the quality of the resulting product depends strictly on the quality of the models used explicitly or implicitly in the engineering process. Prototype information can also be commented and explained, and, by taking advantage of the declarative parts of Erlang, the code of the prototype can be modified following strict methods, like the transformational or those based on refinement (BD77; PP96; Bro97).

We have also designed a formalized distributed architecture. After proposing some commands we have detected actors executing them. Thus, we specify our system from a pair of complementary viewpoints: on the one hand, we model the action flow of activities carried out for each actor by using the Calculus of Communicating Systems (CCS); on the other hand, we model the external behavior of each actor by using the use case model, a technique belonging to the Unified Modeling Language (UML). We have divided the specification and implementation of the system into complementary layers; these layers are illustrated in Fig. 1.

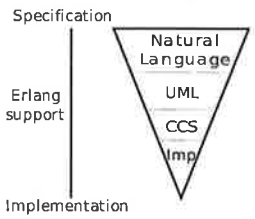

Figure 1. Our general methodology.

* Members of the research group *Redes y Sistemas Distribuidos* (Networks and Distributed Systems), *RESDI*, Instituto de Electrónica y Computación, Universidad Tecnológica de la Mixteca.

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.

To manage complexity, our system has only one shared resource: evaluation tests, which are intended to be applied to bachelor students. This resource is planned to be shared by several teachers, so that they can write or read common databases. To design our tests we have used Bloom's taxonomy. Each test incorporates some skill evaluation according to this taxonomy. Next, we have formalized a set of commands to interface with these tests. The interfaces are intended to be distinct for the case of a teacher or a student.

This writing is organized as follows: In Section 2 we describe Erlang as well as a software development methodology; in Section 3 we propose a distributed architecture and describe in detail the case of a student process; In Sections 4 and 5 we describe our system from two complementary viewpoints: internal and external. In Section 6 we deal with tests. Finally, we present some conclusions in Section 7.

2. Erlang: extreme and declarative programming

In this section, we give a succinct description of Erlang. Further or more detailed information about Erlang can be found on the Internet (Sit09b) or in the book (Arm07). This section also deals with the advantages of applying extreme programming techniques and declarative programming to a system like ours.

2.1 Erlang and concurrence

Erlang is a functional programming language oriented to concurrency. To support concurrency, Erlang is described as follows: An *Erlang node* is an activation of a Erlang system. A *process* is a program being executed on an Erlang node. A *host* is a computer machine hosting nodes. A *distributed system* consists of several hosts, nodes and processes. In a distributed system based on *message-passing* every communication among processes is based on messages. Erlang has two basic mechanisms to send or to receive messages: `receive` and send (written as the asymmetric binary operator `!`). In Erlang, we can send or receive any valid term (even a process). Messages are stored in *mailboxes*. A message is *attended* in an asynchronous way when the messages matches a *pattern* of reception. Erlang has also other characteristics to support the construction of distributed systems: a fault-tolerant mechanism, industrial capacity (many processes can be alive without a sensible decrement in the efficiency of the overall system), and asynchronous communication. Moreover, Erlang has many libraries to solve specific problems (from interfacing with other languages, including Java, to managing databases), is open-source and excludes an explicit treatment of share-resourcing, facilitating the reasoning about the properties of a distributed system (San07).

In the part of functional programming, Erlang follows an eager evaluation computational model. Erlang also has higher-order functions, comprehension lists, and a module system; these characteristics place Erlang within the mainstream of other well-matured and well-crafted functional programming languages, such as Haskell (Sit09c) or Clean (Sit09a).

2.2 Extreme and Declarative Programming

We have used extreme programming methodology together with Erlang because our customers are our teachers and students. Furthermore, we are teachers (and, sometimes students) an so application intended to support teaching/learning activities is within our knowledge and direct experience. Moreover, our application is intended to be a medium-sized one and our team is small; thus we only want to apply lightweight formal methods.

Other reasons to adopt some techniques from extreme programming are the following. We want not only review each other's code (which is highly valuable), but also to review to general design decisions at specification level. Sharing high-level ideas along with every stage of implementation is also a good idea, as well as reasonably changing the specification on demand. In fact, a good specification should also be a good guide of the overall development process; in our case, this is possible because Erlang code is close to specification. We want to strengthen extreme programming with formal methods, like stepwise refinement, refactoring, detailed documentation and transformational programming, or with techniques taken from other methods of Software Engineering, such as UML. Prototypes are seen as naive programs which must be enhanced and developed to get final, efficient, and correct implementations.

To summarize, extreme programming is a useful lightweight software development methodology, but this methodology must be complemented with other useful practices of programming; particularly, with a good and topic-oriented programming language and with techniques based on code and library reuse (where Erlang stands out by its vast set of reusable OTP libraries). The main purpose in using our tools is to establish whether the actual behavior of the system is in agreement with its intended behavior. In developing a complex distributed system of emergent interacting components, it is important to be able to describe the behavior of the system in a precise way at various levels of detail, and to analyze and to enrich it on the basis of descriptions.

In this work, in fact, we try to use an eclectic collection of methods that work together such as transformational techniques (Par90), mathematical models of concurrence (Mil89), and modeling languages (BR98)) having Erlang as a pivot. Erlang programs are the acid-test of our ideas. Moreover, declarative programming should be viewed as a good complement to formulate rapid prototyping and transformational programming.

3. A basic instance of our model

In this section we describe some basic aspects of our architecture, a classification of users, some examples of actions and events attached to a student, and some steps to declare and implement a student command. We also sketch the actions and events associated with a teacher.

3.1 Basic architecture

To describe our basic architecture, we need the following terminology. A *server* is a process in execution from a node. A *client* is a process requiring the services or resources from the server. A *centralized distributed system* is a distributed system having many clients but only one server. A *request* is a message from the node to the server. An *answer* is a message from the server to the node. Message reception is managed by *mailboxes*. A mailbox receives messages. Message reception is *asynchronous*: the sender will not wait for an answer from the receiver.

A *configuration* of a distributed system is a diagram showing nodes and connections at a given time. This diagram describes the essential components within a distributed system as well as the current existent connections. These connections are shown through continuous lines connecting nodes (in a Graph Theory sense), and are called *direct connections*. In the configuration showed in Fig. 3, clients are able to communicate each other in an *indirect* way: an intermediate node is used. This type of communication channel is indicated by a dashed line. Server S, in this case, works as a receiver and transmitter of messages. Other tasks can be directly implemented in the server, but this basic scheme allowed us to begin our architecture.

Let S be a *server* and C_1, C_2 be clients. Suppose the server S is activated at time t_0; from time t_0 on, clients may open a *communication channel* with the server (see Fig. 2). Once activated, the server S is able to accept or reject *requests* formulated by the clients. The most basic request formulated by a client to the server

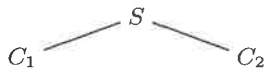

Figure 2. Clients connect to server at time t, with $t > t_0$.

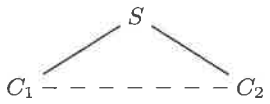

Figure 3. Clients communicate each other via an intermediate node.

is *connect*. The most basic answers formulated by a server to one request *connect* is either *accept* or *reject*. A request *connect* is always done before an answer *accept* or *reject*.

Let us suppose that server S accepts both requests formulated by nodes C_1 and C_2. The current distributed system is represented in Fig. 2. In this figure, both clients have been accepted by the server. Nodes execute client processes. These processes can be accepted or rejected by the server.

Our model incorporates some functions for interacting with users. Modern interfaces are based in *Graphical User Interfaces* (GUIs) systems. Erlang has a basic GUI to do interface with humans. The package is named `gs`, a component of the OTP system. To follow a disciplined and uniform architecture, interfaces should be considered as processes. Given the Erlang capacity to create and to control processes, this does not represent any problem. However, interfaces should be designed, developed and maintained quite apart from the fundamental system. To achieve this goal, we prefer to create a GUI process, and open communication channels between this and a client process, or between a GUI process and a server process; see Fig. 4. This "separation of concerns" (Dij76) is a ubiquitous and useful technique in developing programs. In particular, we can extend our architecture by including final users linked with the system via mobile devices.

Following this design, we would have a system like that of Fig. 4. After activating the communication channels, the configuration given in Fig. 4 is seen like that of Fig. 5.

Graphical interfaces can also be considered to be final user interfaces, not necessarily related to Erlang. User interfaces can be implemented on a mobile device. We propose *hybrid systems*, where Erlang is a process manager, but interfaces are perhaps implemented using another programming language. Java and its virtual machine is a good candidate, because implementations of Java virtual machines on mobile devices are common right now. The flow of messages between an Erlang node and a Java ME virtual machine could be carried out through pipeline techniques or through binary flow data (reduced to the minimum size). The design we propose includes using Bluetooth or Wi-Fi technology, in such a way that students can use mobile phones or PDAs as learning tools.The most important part of this proposal is that the topology of the distributed system is considered highly dynamic.

Following, we describe the basic architecture and model of our distributed system. We have classified users into three basic types: teachers, students, and an administrator. In the next section, we describe some representative cases of clients, as well as some of the functions or actions associated with these clients.

3.2 Clients

To continue, we should classify clients according to their characteristics. This classification is described in the next.

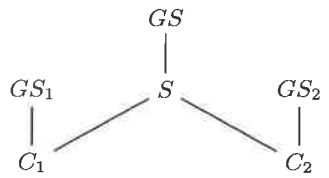

Figure 4. Activation of graphics processes.

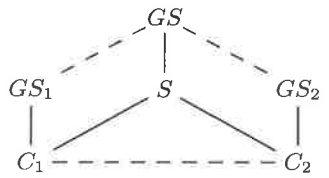

Figure 5. Communication channels through intermediate nodes.

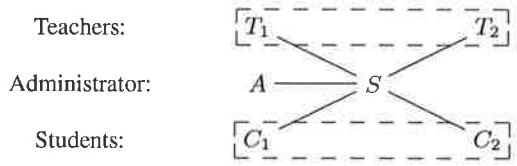

Figure 6. A classification of users.

First, to ensure level maintenance, our architecture only includes an *administrator*. Next, we consider a classification of nodes as either students or teachers or administrators. In Fig. 6 the nodes C_1 and C_2 are classified as students, nodes T_1 and T_2 are classified as teachers, and the node A is an administrator.

The system given in Fig. 6 can be improved to fault tolerance as showed in Fig. 7 by using a *link*. This is a built-in mechanism of Erlang to implement fault tolerance. From now on, we fill in this skeleton to implement a distributed system of learning and teaching where fault-tolerance is optional. Learning is emphasized according to the proper characteristics of students; teaching is emphasized according the requirements and functions from the point of view of teachers.

If the server accepts the connections with these users, this generates new system configurations (i.e., the topology connection is dynamical). We represent a generated configuration in Fig. 7, where we have added an *administrator*, and a basic mechanism to support fault tolerance: we *link* a server S with a server S'. If the server S fails, S' is ready to support all the system by using specific built-in functions of Erlang.

A possible and ideal configuration of our distributed system is given in Fig. 8, where MF denotes a mainframe computer (a high capacity computer), DPC a desktop computer, MD a mobile device, and the protocols of communication are represented by LAN (local access networks), Wi-Fi and BT (to represent Wi-Fi and Bluetooth wireless technology, respectively). Bluetooth and Wi-Fi technologies can support the communication between mobile devices and computers (HR07; Smi03) by using Java (Knu03; KK03).

3.3 Student actions and events

Having split the set of users of the system into three subsets, we need to specify what functions or activities are available for each subset. We should note that, in spite of having locally referential

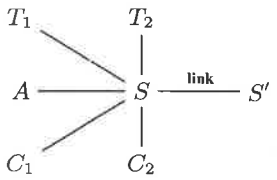

Figure 7. A distributed system with a basic fault-tolerance system.

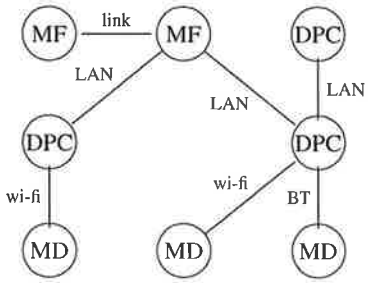

Figure 8. An example of an ideal distributed system for teaching and learning.

transparency, globally Erlang depends on the flow time when the system is in performance. Users have a behavior described by a set of actions carried out through *events*. For us, the interesting events are those caused by interaction with the rest of the distributed system, with processes as well as with other events. We think of events in some temporal reasoning setting: events can have associated either a logical clock or events can be embedded within an absolute time.

Students have the following available activities:

1. The command **subscribe**: the student can try to belong to the system; this request is sent to the server; after filling a form, the answer from the server can be: **accept** or **reject**;
2. The command **logon**: A student can log on the system; server tries to log the user on; answers are either **You are logged** or **You are not accepted**;
3. The command **test**; an answer can be: **Type of test?**, and the student can select a kind of test; later, the test is sent to the student from the server; we propose two kind of tests: either simulated or real; after the selection, the student receives a test of the selected kind;
4. The command **do_test**: A student does an test within the interval (absolute time) $[t_1, t_2], t_1 < t_2$; time can be monitored from the main server or directly from the local computer;
5. The command **chat(friend)**: a student can chat with other student named `friend` if she is not occupied (for example, if she is not doing an test);
6. The command **invitation(friend)**: this command is to request communication with another student named `friend`. The answer can be **accept** or **reject**;
7. The command **cgroup(name)** is a command to create a work group: students can decide to join to this group; the leader of this group is whomever initiated the creation of the group;
8. The command **who** reveals who is logged to the system, the student can know who is available to chat (including other students or teachers);
9. The command **groups** shows the current existent groups;
10. The command **chat(teacher)**: a student can chat with teachers if a teacher is available;
11. The command **exit**: a student can exit from the system, and terminate his session;
12. The command **resources**: this command indicates the available resources (notes, books, and so on);
13. The command **history_resources**: this command allows one to obtain a history of the obtained resources;
14. The command **update_resources** accesses a local repository to get the latest resources.

Some commands already implemented are shown in Fig. 9, with a simple proposal of graphical interface. Further commands can be available to students and teachers, but these commands can be standard or local to the machine. We illustrate how to define commands in the next subsection.

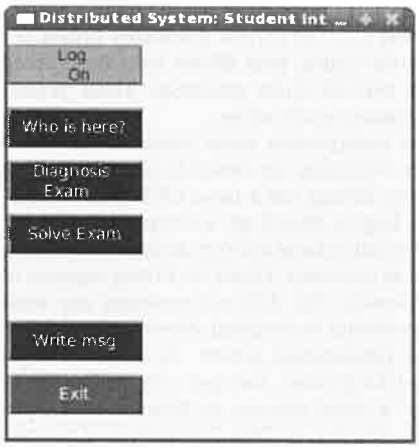

Figure 9. Student interface.

3.4 User-command definitions: student case

Now, we explain how to add a command to be used by a student. First, we name the command. Suppose we want to obtain a homework assignment from the server. Supposing that the server is already activated and the `studentId` exists (and a `studentId` exists within the system when the user has logged on the system, following the use-case model and the CCS description).

The `studentId` process must receive a message as follows:

```
homework() ->
        studentId ! homework.
```

The `homework()` function gives the homework of the current day. If we want to get a homework with a distinct date, we would add an additional time parameter. The `homework()` function is either accessed directly from the prompt or indirectly by using a button, linked with the `homework()` function, within a graphical interface. This is so because we allow (and encourage) to experiment with prompt-based commands before giving its graphical versions.

From here, the `studentId` process notifies the request of the homework to the server:

```
homework_request ->
        {server, sv_node()} ! {self(), homework};
```

Where `homework_request` is a constant pattern (a word) within a `received` construction (please note that we give only segments

of code and incomplete definitions on purpose). Now, the server receives the request, and calls a function named `sv_homework`:

```
receive
        {From, homework_request} ->
                sv_homework(From),
```

Depending on the requester identity, as filtered by the flow of events, the server begins to dispatch the request.

```
sv_homework(From) ->
        Homework=homeworkR(),
        From ! {server, homework, Homework},
        io:format("Ready: Some homeworks were sent.~n",[]).
```

Where `homeworkR()` is the available local resource. The user now receives the homework, and from now on the homework is a data available within her node.

```
{_FromServer, homework, Homework} -> %5E, response
        io:format("Homework:~w~n",[Homework]);
```

This finishes the construction of a new command. Optionally, if we want to deal with a graphical interface to manage a homework request, we proceed as follows. Suppose we have already activated a graphical process. We add a button to the visible graphics layout:

```
gs:create(button,buttonGetHomework,mainWindow,
        [{label, {text,"Get homework"}},{y,190},
                {bg,blue},{fg,yellow}]),
```

We link the `buttonGetHomework` with a specific function; in this case, with the `homework()` function.

```
   {gs, buttonGetHomework, click, _Data, _Args} ->
        homework(),
        loop_student();
```

The `homework()` function can now use the received data.

3.5 Teachers actions and events

Having described some student's commands, we proceed to clarify some teacher's commands (to complement our treatment, see (Vra04)). Teachers can use the following commands:

1. The command **subscribe**: the teacher can try to belong to the system; this request is sent to the server; after filling a form, the answer can be: **accept** or **reject**;
2. The command **logon**: A teacher can log on the system; server tries to log the user on; answers are either **You are logged on** or **You are not accepted**;
3. The command **design_test**: A teacher can collaboratively design a test;
4. The command **public(test)** A teacher can publish a `test`;
5. The command **chat(friend)** A teacher can chat with a friend (a teacher or a student);
6. The command **who** reveals who is logged into the system;
7. The command **exit**: a teacher can exit from the system, and terminate his session;
8. The command **resources**: this command indicates the available resources (notes, books, and so on);
9. With the command **update_resources** we access a local repository to get the latest available resources.

Server dispatches each request and coordinates the overall performance. Examples of other server tasks would be: blocking communication with the external world, logging (in a hidden and special file) services used by students, and limiting the time for test solving.

4. Formal specification at internal level

To characterize the *internal* flow of actions of each user, we use *CCS calculus* (Mil89) (however, recognizing that the most serious restriction of CCS is that for any particular system its connection topology is static, other proposals are possible, like that in (Hen07)). To express the sequential activation of processes (following the guidelines given in (NR05; RNRC06)); here we must understand process definition as:

$$\mathsf{Proc} \stackrel{\text{def}}{=} \mathsf{signalIn.Proc1} + \overline{\mathsf{signalOut}}.\mathsf{Proc2}$$

where the dot notation means *sequence* and the plus sign is an exclusive *choice*. The bar over the signalOut indicates an output signal; in contrast, signalIn has no bar because it is a signal input. *Signals* are generated by processes for synchronization purposes. We differentiate between a *signal* and a *process* by inspecting the first letter on the name, using initial lowercase letters for signals and initial uppercase letters for processes. We may also include an *idle* state as follows: $\mathsf{Idle} \stackrel{\text{def}}{=} \mathsf{Idle.Something}$.

We have a model consisting of three special processes: Admin, Student and Teacher. The process Admin coordinates the overall system by allowing or blocking other processes. The process Admin will generate (spawn) other processes when needed by user interaction. The processes Teacher and Student directly interact with a human, teacher or student. Any time Student or Teacher needs a system resource, it will be managed by Admin. This approach takes into account the Erlang capacity to generate processes as well as its message-passing technology. Besides, we can use a socket to connect to non-Erlang processes (transmitting data in a binary format).

The process Student models activities of a student, and hides communication interchanges issues. In Fig. 10 we show the Student definition. An analogous model holds for Teacher and Admin. The process Admin coordinates education activities, accepts online and offline user activities, and controls the accesses and privileges. The startEnvironment and startInterface both open communication channels with the final user. The clients could be portable devices with low resources such as PDAs or a mobile phones, via intermediate nodes. We can think of a course activity as a set of constrained events by time and space.

As you can read on lines 4 and 5 in the text in Fig. 10, Bad and NoMoreAttempts processes are in charge of conducting system's behaviour on the event of wrong conditions. These processes trigger timerOut or maxCount signals to match with the Login process. So we may think of Counter and Clock as independent processes that communicate with Login as stated in lines 14 and 18 through Counter | Login | Clock. As a partial example of how we can implement on Erlang the login process we use the existence of a studentId as a guard for the execution of a valid command on the system. For sake of simplicity, we show the case of a good login and how the logoff, test and who commands need a prior validation of the user through the sequential reading of the validCommand function.

```
validCommand(Id,Command) ->
    case whereis(Id) of
        undefined ->
                io:format("You are not logged on~n",[]);
            _ -> Id ! Command
    end.
logoff() ->
    validCommand(studentId,logoff).

test() ->
    validCommand(studentId,test).

who() ->
    validCommand(studentId,who).
```

```
1  Student ≝ Student.Login

2  Login ≝
3      ReqAdminAut(user,pswd).(Ok +
4      Bad +
5      NoMoreAttempts +
6      NoGoodClosing.Recover +
7      cancel.Student)

8  Recover ≝
9      LoadStudentLastGoodState.startEnv(stateInfo)

10 Ok ≝
11     goodLogin.LoadStudentPreferences.StartMenu +
12     goodLogin.LoadStudentLastActivity.startEnv(lastActivity)

13 Bad ≝
14     timerOut.LockStudentAccount +
15     wrongUser.increaseCounter.Login +
16     wrongPasswd.increaseCounter.Login

17 NoMoreAttempts ≝
18     maxCount.LockStudentAccount

19 LoadStudentLastGoodState ≝
20     GetAdminConnection(stateInfo).SearchStudentLog(stateInfo)

21 LoadStudentPreferences ≝
22     StartStudentCommModel.CheckSchedule.
23         JoinCourse.SendSyncInfo

24 StartMenu ≝
25     LoadOptions.StartInterface

26 LoadStudentLastActivity ≝
27     SearchStudentLog(lastActivity)

28 StartEnv(x) ≝
29     InitProperState(x).StartInterface

30 GetAdminConnection(command) ≝
31     RequestSafeMode.SendCommand(command).GetSystemResponse

32 SearchStudentLog(x) ≝
33     GetAdminConnection(x).QueryDataBase(studentId, x)

34 StartStudentCommModel ≝
35     LoadGroupInfo(courseAtendantList).SetPrivilegies.StartCommSetMenu

36 CheckSchedule ≝
37     GetAdminConnection(scheduleInfo).
38     queryDataBase(studentId, scheduleInfo).
39     LoadProperActivity

40 JoinCourse ≝
41     RequestCourseServConnection(courseSelection).LoadCourseMaterial
```

Figure 10. A student characterization via a flow of actions.

5. Formal specification at external level

Having described how to deal with successions of actions, now we present some techniques belonging to UML to specify the external behavior of the system, including extension handling.

To specify the system behavior from the external point of view, we have applied the technique called *use-case model* (Coc97) of the Unified Modeling Language (UML). This model represents the interaction among distinct *actors* (students, teachers or administrator) as well as the interaction of the actors with the system under design. A use-case model describes the characteristics of the system under design, without any compromise with implementation details, although in our concurrent application, we intermix some implementation and specification stages. Extreme programming methodology suggests to write *user stories* for the same purpose as use cases; but use cases and user stories are not the same specification technique: first, user stories are written with the format of about three sentences of text; second, user stories are written by a user; further user stories are used to define time estimates for the release planning and, finally, they are also used instead of a large requirement document. However, user stories can fail to serve as reliable documentation of the system and can also be difficult to scale them to large projects. Instead of applying user stories for specifying the system external behavior, we have adopted use cases. Moreover, use cases allow us to have a basis for estimating, scheduling, and validating programming efforts. To summarize, we have adopted the use-case model technique as a replacement of the user stories proposed by the extreme programming methodology.

The adopted use-case model is basically a text where an actor is endowed with actions and events. Each action can be modelled by a command, whereas an event modifies the current system state. We must build a use case for each activity carried out by an actor, to create an overview of the functions that the system should provide to the users. The Fig. 11 shows the dressed[1] use case specification for the **login** command when it is executed by a user of the system; other cases are omitted for sake of brevity. To summarize, we have focused on extension handling analysis through use cases for detecting rare, irregular or extreme behaviour of the system.

Extension handling is a major contribution of the use-case specification to the system: extension handling enforces the planning of all alternative ways or potential abnormal events that can occur during the execution of the main successful scenario, thus capturing bad behaviors and so improving system robustness. With respect to exception handling, though Erlang has a *let-it-crash* philosophy, we are cautious and our proposal is to be preventive.

An alternative view of this use-case specification can be built using UML sequence diagrams; in this way, sequence diagrams describe the logic of events between an actor and the system in a visual style, enabling both documentation of the flow of messages as well as validation of Erlang's logic of message passing. Sequence diagrams show the use-case main successful scenario, plus one or more alternative scenarios of the extensions. Currently, the use-case model is helping us to detect, in a user-oriented way, some strange misbehaviour of the system.

Due to the multidisciplinary character of our project, the next sections deal with educational concerns and the case of the design and the implementation of tests.

6. Designing and managing tests

Now we focus our attention on the only collaborative part we currently have: tests; in (JMRE02) there are related concerns.

Sometimes, teachers have the experience of doing boring, tiring and repetitive tasks, like revising homework and grading tests: these tasks can be achieved by an computerized wizard. Tests can have several formats: from multiple-selection (Mar08) to free text (Cra01). One point to take into account in our system is the degree of desirable automation, but without neglecting the pedagogical issues. A well-designed test should go along with a good evaluation method: Students are impeded from cheating, and their evaluations are fast and precisely delivered. Except in courses requiring an ad hoc resources classroom (perhaps where, for example, the evalu-

[1] Dressed case: A detailed use case, following the UML terminology.

Use Case: User login.
Brief Description: User proceeds to log into the system.
Scope: System.
Level: User goal.
Preconditions: User must previously be registered (through the **subscribe** command).
Post-conditions: User is successfully authenticated and log into the system.
Main Successful Scenario:
1.User introduces data for authentication by the system.
2.System notifies the user that he or she has been accepted.
3.System loads user preferences.
Extensions:
2a System detects that the authentication is incorrect:
2a.1 System notifies to the user that has been rejected.
2a.2 System logs the event.

2b System detects that the user has try a number n of unsuccessful attempts:
2b.1 System blocks user account.
2b.2 System logs the event.

2c System detects that the time for entering data has expired:
2c.1 System blocks user account.
2c.2 System logs the event.

2d System detects that the user last session was not correctly closed:
2d.1 System loads the last good state.

2e System detects that the user is already login:
2e.1 System notifies to the user that he or she has been rejected.

2f User cancels the authentication procedure:
2f.1 System notifies the user that the authentication procedure has been canceled.
2f.2 User returns to step 1 (line 8).

3a System detects that there are pending activities from the last session of user:
3a.1 System loads the pending activities from the last session of user.

Figure 11. Use case specification for the **login** command.

ation of projects could hardly be done by computers), multiple-choice tests would be a good source of designing and automation.

Each evaluation question should associate with an acquired skill: knowledge, comprehension, application, analysis, synthesis, and evaluation, as described by Bloom's taxonomy (Bar08). This taxonomy categorizes levels of abstraction of questions, and gives us general guidelines in the formulation of tests. The taxonomy also provides a useful framework to categorize test questions, supposing teachers ask questions with respect to particular or mixed levels. Finally, the taxonomy is officially recognized in México and is used in some universities abroad (SitUV09). Tests and questions are the topic of the following two subsections.

6.1 Elements of tests

We can divide tests into two kinds: either *action-research* or *real*. An action-research test diagnoses the state of students' proficiency, giving to student a major preparation and to teachers valuable pieces of pedagogical information. But these tests do not contribute to the final evaluation. In contrast, we can find real tests. Grades obtained in these tests by students give us an accumulative final evaluation (perhaps like an average).

Managing tests is one task of a learning content management system. Now we describe some commands for managing tests, as a possible set of tools for authoring and re-using or re-orienting content:

1. **createTest**(`subject,testType,testNum,time`). This command creates the test container within a database. The command has the following parameters: `subject` declares the subject or topic of the test; `testType`, the type of test, either automatic or not; `testNum` is an identifier of the test; and, finally, `time`, a parameter given by the teacher to solve the test. Because the automatic nature of the system, we prefer questions oriented to be answered without supervision's teacher.
2. **openTest**(`identifier`). This command allows us to open the container of a test.
3. **searchQuestion**(`subject,type,level`). With this command we look for questions to be inserted into the test. The `subject` parameter is the topic of the test; `type` is the type of the question, and `level` is the assigned level according to Bloom's taxonomy. The command returns the properties and the identifier of a question.
4. **selectQuestion**(`id, score`). This command selects the question that will be add to test. This command requests an `id` (identifier) question and a `score` assigned by the teacher.
5. **deleteQuestion**(`id`. This command erases the question of a test container. Only the test creator would erase questions.
6. **checkup**(`container, group`). This is a broadcasting message, and the command sends a test to be checked by a specialized team.
7. **closeTest**. This closes the test container for both checkup or application. After issuing this command, the test can no longer be modified.
8. **application**(`container,group,date`). This command schedules the application of a test for a specific group or student on the specified date.
9. **scoring**. The command calculates and shows the grade obtained for the group or people that took the test.
10. **supervision**(`container,student`). This command opens the answers container for a supervised evaluation test.
11. **grade**(`question,score`). This command returns the chosen value to the answer valuated by `score`. This command requires the previous opening of a student test.
12. **total**(). This command calculates and shows the obtained grade by the student in a supervised evaluation test.
13. **grade**(`container,student`). This command shows the obtained grade for a specific student.

In the following subsection we give some details about test designing.

6.2 Questions

Tests consist of questions. We create a database of questions. The following list of commands returns `accept` or `reject` within our system mainly by integrity constraints of the database (other reasons to accept or reject a question can be defined in the correspondent use-case specification but are omitted). These commands, similarly to the previous ones, are intended to be accesible only to teachers. The level `levelBloom` is given according to Bloom's taxonomy.

1. **questionTF**(`question,answer,levelBloom`). This command stores `True` or `False` questions.
2. **questionMO**(`question,answer, dis1,dis2,dis3,levelBloom`). This command stores a multiple option question. Typically, four possible options are representative of this kind of question. At least three options are named *distracters* and only one is named the *answer*: the correct answer.

3. **questionSA**(`question,answerComplement,levelBloom`). This command stores a question to be complemented with a short answer; typically, a word or a number.

Tests under supervision are created by the command **questions**(`template`). This command creates and stores templates used for open questions.

Other commands to insert information into the database are the following.

1. **cSubject**(`subject`) allows the creation of a new theme. It returns `accept` or `reject`.
2. **oSubject**(`subject`) opens the question database for searching or updating. It returns `accept` or `reject`.
3. **close**(`subject`) closes an open database.

Currently we are using a simple database to maintain a list of users logged on and textual representations of tests, by using ETS and DETS (direct OTP tools for database management), but a database like Mnesia would give us a good support for this part of our proposal. Through ODBC we can interact with some other databases, like PostgreSQL[2] and MySQL[3].

Erlang is not only suitable to deal with the distributed part of our system, but also to deal with common programming tasks from a declarative view. We exemplify this point with the implementation of tests. Suppose we need a test consisting of a finite succesion of optional questions. Each time the teacher selects only two, three or four options (the recommended real questions should have four options). The student must answer the question by selecting one option. The pattern of the test includes a solution when a teacher accesses it. The student receives only the question and the possible answers. The server node stores the complete test, including the correct answers.

To illustrate our example from the point of view of the server, we show the following test template, where each question has the following structure:

```
question(Question,optionsN(Options)),sol(Solution).
```

where `optionsN` can be `options2` with only two options or `options3` or `options4`for three or four options, respectively.

```
testTemplateServer() ->
    testExample(comics, %Topic
    [
    question("Which is the color of Homer Simpson?",
         options3("Yellow","Green","Blue"),sol(1)),
    question("What animal is Donald?",
         options4("A mouse", "A duck","A dog","A pig"),sol(2)),
    question("What is the favourite meal of Bugs Bunny?",
         options2("Artichokes", "Carriots"),sol(2))
    ]).
```

A student learning the *comics* topic receives the test as follows:

```
testExampleStudent() ->
    [
    question("Which is the color of Homer Simpson?",
          option3("Yellow","Green","Blue")),
    question("What animal is Donald?",
          option4("A mouse", "A duck","A dog","A pig")),
    question("What is the favourite meal of Bugs Bunny?",
          option2("Artichokes", "Carriots"))
     ].
```

In a graphical format, the first question of this test looks like Fig.12. When the student clicks over a button, the student's answer is stored. After the final answer, the student is notified of his or her grade by the server.

We use the following raw interface to indicate the obtained grade, where the empty list indicates that the test has finished:

```
loop_answers(Ls,[]) ->
            Ls1 = lists:reverse(Ls),
            io:format("Got Ls=~w~n",[Ls1]),
            Ls2 = sols(testExampleStudent()),
            io:format("Original=~w~n",[Ls2]),
            Total=match(Ls1,sols(testExampleStudent())),
            io:format("Points: ~w~n",[Total]),
            L=length(sols(testExampleStudent())),
            Result=Total*(10/L),
            io:format("Grade result: ~w~n",[Result]).
```

The simple algorithm to calculate the grade is as follows: We extract the solutions from the test, and compare them with those given from the student. After normalizing, the number of successful comparisons divided by the total number of questions gives us the final grade. Other statistical pieces of information can be obtained if we give qualitative characteristics to the questions.

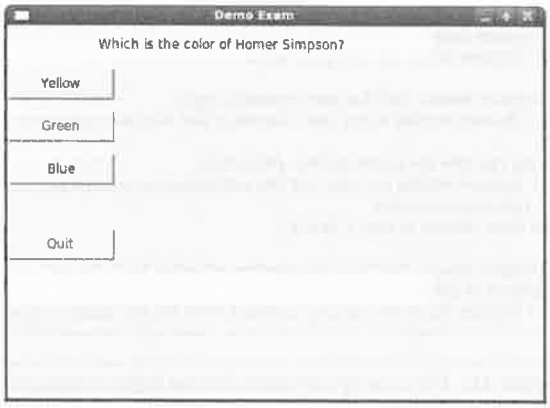

Figure 12. Test interface.

7. Conclusions

In this work we have tried to extend the traditional classroom environment through a computerized teaching-learning tool. New educational technologies, software support, and hardware are constantly emerging, and they present unique opportunities for the application of teaching and learning (WM08). Educational activity is the result of a distributed and concurrent system performed by several actors collaborating together. In a methodological setting, we have tried to complement extreme programming by using declarative programming and Erlang. Two key points make Erlang a good tool for experimenting with the development of such a computerized educational system: First, concurrent events and collaborating activities involve a heavy communication load. And second, as a methodology we have chosen extreme programming to show that functional languages can fit the restrictions associated with this methodology. Trying to maximize comprehension and design of the system, we have used several tools supporting our system. In our efforts to produce the system's specifications we have relied on some formal methods like CCS, and on some specification techniques like UML designs.

Some decisions are still being specified and implemented by our research group to produce a system that incorporates our experience as both teachers and students. This report shows that some fine tuning is necessary to accomplish the goals we may want to be included in the final design, but the tools we have chosen to work

[2] `www.postgresql.org`

[3] `www.mysql.com`

with are flexible enough to make changes that will not have negative repercussions.

7.1 Future work

Currently, our system is operative although incomplete with respect to the actual specification. The parts already specified are valuable documentation to guide the implementation. At the same time, the parts already implemented contribute to improvement of the specification. The actual system has been used only by our team. As future work, we hope the system will be used by students within our university.

We have selected Erlang because this programming language incorporates many facilities for doing concurrent programming. We have identified concurrence as a fundamental part of a learning-teaching system like ours. In the short term, our goal is to integrate Erlang and some other languages heavily used by industrial producers of mobile devices. Also, another goal is to link the semantics of UML and CCS, adapting it to our case. Other goals include: to incorporate multimedia tools, to connect Erlang and other languages, to use Erlang to control mobile devices, and to attach specialized databases to the system. We would also be happy for incorporating Artificial Intelligence in our system: eventually, teachers would be completely independent of human supervision, automatized and "smarts," or we could provide an administrator as a rational agent. A major research on the important topic of shared-resources management is necessary. Finally, we are interested in applying our framework to develop ubiquitous systems seen as distributed systems specified through UML and CCS, and implemented through Erlang.

Acknowledgements

We thank reviewers of the Seven and Eight Erlang Workshops for their valuable comments. We also thank *Universidad Tecnológica de la Mixteca* (UTM) for giving us a nice working environment for doing research. Manuel Hernández thanks *Consejo Nacional de Ciencia y Tecnología* (CONACYT).

References

[Arm07] Joe Armstrong. *Programming Erlang. Software for a concurrent world.* The Pragmatic Programmers, 2007.

[Bar08] William M. Bart. *Encyclopedia of Educational Psychology*, chapter Bloom's taxonomy of educational objectives, pages 110–111. SAGE Publications, 2008.

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[Bec99] Kent Beck. *Extreme Programming Explained. Embrace change.* Addison-Wesley Professional, 1999.

[Bir98] Richard Bird. *An introduction to functional programming using Haskell.* Prentice–Hall, second edition, 1998.

[BR98] Grady Booch and James Rumbaugh. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[Bro97] Manfred Broy. Compositional refinement of interactive systems. *Journal of the ACM*, 44(6):850–891, November 1997.

[Bro07] Manfred Broy. From "formal methods" to system modeling. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, Lectures Notes in Computer Science, pages 24–44. Springer-Verlag, 2007.

[Coc97] Alistair Cockburn. Goals and use cases. *JOOP*, 10(5):35–40, 1997.

[Cra01] Edward F. Crawley. Creating the CDIO Syllabus, a universal template for engineering education. Vol. 3, 32nd Annual Frontiers in Education (FIE'02), ASEE/IEEE, November 2002.

[Dij76] Edger W. Dijsktra. *A discipline of programming.* Prentice–Hall, 1976.

[Hen07] Matthew Hennesy. *A distributed Pi-Calculus.* Cambridge University Press, 2007.

[HR07] Albert S. Huang and Larry Rudolph. *Bluetooth Essentials for Programmers.* Cambridge University Press, 2007.

[JMRE02] Mike Joy, Boris Muzykantskii, Simon Rawles, and Michael Evans. An infrastructure for web-based computer-assisted learning. *J. Educ. Resour. Comput.*, 2(4):4, 2002.

[KK03] James Keogh and James Edward Keogh. *J2ME: The Complete Reference.* McGraw-Hill/Osborne, 2003.

[Knu03] Jonathan Knudsen. *Wireless Java: Developing with J2ME.* Apress, 2003.

[Mac90] Bruce J. MacLennan. *Functional programming. Practice and theory.* Addison–Wesley, 1990.

[Mar08] Kim Marshall. The use of multiple choice options in law. In Reggie Kwan, Robert Fox, F.T. Chan, and Philip Sang, editors, *Enhancing Learning Through Technology. Research on emerging technologies and pedagogies*, pages 263–276. Scientific World, 2008.

[Mil89] Robin Milner. *Communication and concurrency.* Prentice-Hall, 1989.

[NR05] Thomas Noll and Chanchal Kumar Roy. Modeling Erlang in the Pi-Calculus. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 72–77, New York, NY, USA, 2005. ACM.

[Par90] Helmut Partsch. *Specification and Transformation of Programs.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[PP96] Alberto Pettorossi and Maurizio Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[RNRC06] Chanchal Kumar Roy, Thomas Noll, Banani Roy, and James R. Cordy. Towards automatic verification of Erlang programs by π-calculus translation. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 38–50, New York, NY, USA, 2006. ACM.

[San07] Nicola Santoro. *Design and Analysis of Distributed Algorithms.* Wiley-Interscience, 2007.

[Sit09a] Clean Site. `www.cs.kun.nl/~clean`, 2009.

[SitUV09] Bloom's taxonomy and University of Victoria Web Site. `http://www.coun.uvic.ca/learning/exams/blooms-taxonomy.html`, 2009

[Sit09b] Erlang Site. `www.erlang.org`, 2009.

[Sit09c] Haskell Site. `www.haskell.org`, 2009.

[Smi03] Raymond Smith. *Wi-Fi Home Networking.* McGraw-Hill/TAB Electronics, 2003.

[Vra04] Charambolos Vrasidas. Issues of Pedagogy and Design in e-learnnig Systems. In *SAC2004*, Nicosia, Cyprus, March 2004. ACM.

[WM08] Leonard Webster and David Murphy. Enhancing learning through technology: Challenges and responses. In Reggie Kwan, Robert Fox, F.T. Chan, and Philip Sang, editors, *Enhancing Learning Through Technology. Research on emerging technologies and pedagogies*, pages 1–16. Scientific World, 2008.

ECT: An Object-Oriented Extension to Erlang

Gábor Fehér
Budapest University of Technology and Economics
Budapest, Hungary
feherga@gmail.com

András G. Békés
Ericsson Hungary
Budapest, Hungary
andras.gyorgy.bekes@ericsson.com

Abstract

To structure the code of an Erlang program, one can split it into modules. There are also available some basic data structures: lists, records and tuples. However, there are no fully functional classes that encapsulate data and functionality, and can be extended with inheritance. We think these features could promote code reuse in certain cases, therefore we decided to extend the language with object-oriented capabilities. A strong evidence of the usability of this is the fact that part of the program itself was rewritten using our newly created language elements, and the new version was simpler and cleaner than the original Erlang one.

Our main goals were to preserve the single-assignment nature of Erlang and to keep method-call and value-access times constant. It was also a priority to make the extension easily installable, to reach as much developers as possible. For this, we avoided changes in the Erlang compiler itself. Instead, we created the extension as a parse transformation[1]. In our implementation a class is a module that contain the methods, and a record type whose instances are the object instances. Both methods and fields can be inherited.

We also examined the currently available other object-oriented extensions for Erlang, and compared them with ours. Our implementation has strong advantages, but it also lacks some features. Compatibility with records and speed are the main advantages. In this paper – among describing and comparing our extension – we also show the possible ways of adding the missing features.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Object Oriented Programming, Erlang

1. Introduction

Computer programs can be large and complex. However the amount of complexity a programmer can handle is limited. To bridge this gap, several techniques emerged to structure programs into isolated segments, so that programmers can deal with them one by one. Structuring has two aspects: structuring the data the program works on, and structuring the instructions of the program.

[1] The Erlang compiler interface for language extensions

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.

Lifespan of programs might be long as well, and during that maintenance is required. This is especially important in the field where Erlang was developed. Reusing the already written segments can decrease the cost of maintenance, and also the number of programming faults. If the segments have been used before, it is less likely that they contain errors. One of the advantages of Object Oriented Programming (OOP in the sequel) is that it promotes code reuse.

The possibilities of segmenting a program are determined by the programming language. In the Erlang community, there is a constant debate about whether Erlang should be extended with features of OOP, or Erlang is capable of providing all the advantages of OOP, as is. We think that although most advantages of OOP can be reached in pure Erlang, some aspects of OOP are laborious to achieve. We have developed an OOP framework as an extension to Erlang, and in this paper we discuss our extension.

1.1 Object-oriented programming and Erlang

Object-oriented programming is a way to structure a program. In an object-oriented system, the segments of the program are objects. An object encapsulates data and functionality. Objects are grouped into classes, where instances of a class have the same data structures and functionality, but might have different values in their data structures. Classes can be extended with data structures or functionality to create new classes. An important aspect of OOP languages is polymorphism, so that objects of an extended class can be used in any place where objects of the original class can be used. This makes reusing code written in objects possible. We highlight some costs and benefits of OOP based on (Mössenböck 1993):

The advantages and disadvantages are:

+ If a suitable algorithm works with an object class, it can be used with extended objects and this way even its behaviour can be altered.
+ If a program contains suitable classes, new functionality can be added to the program without the modification of these classes, just by extending them.
- Programmers need to learn applying the concepts of classes, inheritance and polymorphism.
- Programmers need to learn when to use classes and when not to use them.
- Inefficiency: run-time overhead and storage-overhead.

Erlang has some features that provide the main advantages of OOP, however more work from the programmer is needed to achieve these. With the introduction of objects our aim is to generalise and simplify these solutions. It is important to note, what is not our intention: we are not saying that the use of objects will always improve a program written in Erlang. This can be stated in

parallel with the processes of Concurrency-Oriented Programming. Processes, when used carelessly, provide no benefits: e.g. evaluating each function call in a new process is too resource-hungry. In this paper we will not deal with the questions of object-oriented design. Most of the examples will be so short that they can not show the benefits of using objects.

1.2 The outline of the paper

The paper is structured as follows.

In Section 2 we describe the available object-oriented extensions to Erlang and OO-related techniques offered by Erlang. In Section 3 we specify the exact objectives for our object-oriented system and describe the new language elements in detail. In Section 4 we show the architecture of the implemented system. We measured the speed of the implemented system and other object-oriented systems for Erlang. In Section 5 we describe the process of measurement and present the results. In Section 7 we conclude the paper and show possible development directions.

2. Related work

We collected and tested the following other object-oriented, and semi-object-oriented approaches for Erlang. We found that there are basically two approaches.

According to the first approach, an object is represented by a data structure that contains field values and type information. The methods of the object are functions and the type information is used to find these functions.

In the second case, not only a data-structure, but one or more Erlang processes represent the instance of a class. In their general pattern, when the client creates such an object, a process is created and its PID is handed to the client. This newly created process contains the field values of the object, and executes the loop waiting for messages from clients. A client then can send a message to query or update a field or to call a method of the object. As a response, the server sends the value back, updates the field, calls the method or terminates the loop.

Both approaches have advantages and disadvantages. Our framework tries to make switching from one approach to the other easier: The source code of a class can be used for creating data-based objects and process-based objects as well.

2.1 Programming objects with funs

In Erlang, just like in other functional languages, functions can be stored in data structures. One can store the methods of an object (`funs`) in a data structure, together with the fields of the object. Users of an object need to call the methods through these `funs`. This makes inheritance and overriding possible: an object of an inherited class can have a `fun` at the same place of its data structure as its superclass, but either with the same functionality or with overridden functionality. The most serious problem with this approach is that it makes code upgrade almost impossible[2].

2.2 Behaviours and parameterized modules

The behaviour concept of Erlang provides similar functionality to interfaces of OOP. An Erlang behaviour simply defines a set of functions. A module that wants to implement a behaviour (a *callback module*) has to define these functions. In practice, behaviours are used just like virtual methods of OOP. The *behaviour module* defines an abstract class, while the callback module defines the functionality that was omitted from the behaviour module.

Behaviours use the fact that functions can be called from a module that's name is only known at runtime. The name of the callback module is passed as argument to the functions of the behaviour module, and the functions of the callback module are called from the behaviour module, when needed.

Behaviours do not provide methods for inheritance and for structuring data.

Parameterized modules (Carlsson 2003) address these issues. They are already built into the Erlang distribution, but not yet officially documented, and programmers are not recommended to use them. Therefore it is possible that their behaviour changes from release to release. Because of this, we are describing the behaviour we experienced in Erlang/OTP R12B-4, released at 3rd September, 2008.

Parameterized modules provide a way to define parameters for a module. Functions of such a module can not be called directly. Instead, an instance of the module should be initialised with parameter values, and functions of that instance can be called. These parameter values are then implicitly available for the functions.

Erlang modules also support virtual function inheritance (Carlsson 2007), but this is also an unofficial new feature. Exported functions in a base module that are not implemented in the extension module are inherited to the extension module. Field inheritance is also available, but not fully supported.

The problem is because of the fact that fields of objects are stored in a nested structure. Let us assume that we have three classes: `beta` extends `alpha` and `gamma` extends `beta`, moreover `alpha` has two fields, and the other classes both have one. The following examples show the underlying data structure of an instance of `gamma`:

```
AlphaInstance = {alpha,Field1Val,Field2Val}
BetaInstance  = {beta,AlphaInstance,Field3Val}
GammaInstance = {gamma,BetaInstance,Field4Val}
```

In the first line, `alpha` is just like a record. However, the subsequent two module instances contain the fields of superclasses in nested instances. This implies that fields can only be accessed or updated if we know which base module defines them. Another problem is that a function in a base class has only access to a "stripped-down" module instance: it only sees that part of the nested structure that corresponds to that module. This has troublesome consequences. First, the base module can not invoke the overridden implementation of its functions, however this is the desired behaviour in object-oriented languages.

Another consequence is a problem with the update of fields in base modules. By update, we mean creating a copy of the module instance, which has some of its fields changed. The problem is that a field update in the base module must be implemented in all extension modules, because the inherited function from the base module can not see and copy those data parts of the instance, which belong to an extended module. The diagram in Figure 1 shows the case when a function in a base class updates some variables and returns the module/object instance.

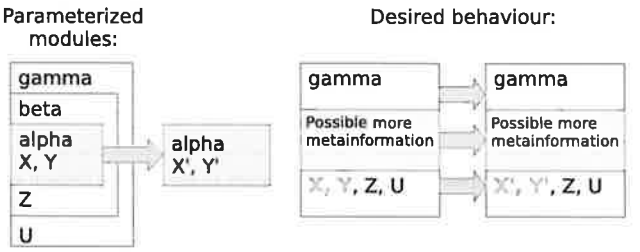

Figure 1. Field updating with Parameterized modules

[2] Processes holding `funs` from an old module are killed when the old module is replaced.

Apart from implementing functions involving field updates in all extension modules, we could not find other ways to work this problem around.

2.3 eXAT Objects

eXAT (Stefano and Santoro) (Santoro 2004) is an experimental agent-programming platform written in Erlang. It provides object-oriented programming features for the programming of agents. It supports virtual methods and methods can be extended by adding new clauses (before the inherited ones). Each object runs in separate processes. Fields are declared per instance, the first time they are given a value (much like a dictionary, mapping field names to values). eXAT classes can be defined as modules. According to (eXAT website), every instance of an object has two associated Erlang processes: one of them serves the method call requests, and the other the field manipulation requests. This implies that every function call and field manipulation involves inter-process message sending. For the programmer, this is hidden: the methods and fields of an object can be accessed through function calls.

2.4 WOOPER: Wrapper for OOP in Erlang

WOOPER (Boudeville) is also a process-per-object class implementation for Erlang. It supports virtual methods and methods can be extended by adding new clauses (before the inherited ones). For fields, it uses a similar directory approach like eXAT. An advantage over eXAT is the possibility of multiple inheritance. A possible disadvantage is that the programmer needs to send messages for method calling by hand. However this can be useful in certain cases, for example, more than one method calls can be initiated at a time, and the the response messages can be processed asynchronously in a batch. To create a class, its main parameters – exported methods, constructor parameters, etc. – should be defined as Erlang macros. Macros should also be used in methods to perform some basic tasks. These are the getting and setting of field values, and specifying the return value of the method.

3. The Erlang Class Transformation Extension – Specification

In this section we describe the object-oriented extension we created for Erlang. Its name is ECT, which stands for Erlang Class Transformation. In the first subsection we state the goals for ECT. In the consecutive subsections we describe our design decisions that satisfy these goals and their results: the new syntax elements and their semantics.

3.1 Motivation and design goals

In the following list, we summarise the necessary features for our object-oriented system. We also highlight some goals that have been motivated by the related works.

- field[3] inheritance
- *Method inheritance and virtual methods*
- *Polymorphism* – if class B is a descendant of A, than it should be possible to use instances of B any place where A can be used.
- *Uniform access to fields* – inherited parameterised modules showed that it is inconvenient when access of a field depends on the class where it was defined.
- *Single-assignment behaviour of fields.* – one problem with the object-as-process paradigm is that it breaks the single-assignment nature of Erlang. We wanted to integrate objects into Erlang as smooth as possible: objects are immutable, just like records.
- *Provide objects-as-processes optionally.* For certain uses, supporting this is useful (see (Stefano and Santoro; Santoro 2004; Boudeville)).
- *Only $O(1)$ runtime overhead.* Our measurements show that the method-call times of certain OOP systems increase with the depth of inheritance. We should keep this constant.
- All this should be achieved without modifying the Erlang compiler.

We don't implement the following possible features:

- Single-clause overriding. We show that it can be simulated easily by the use of full-method overriding.
- Multiple inheritance. According to section 8.6 of (Mössenböck 1993), multiple inheritance has several issues, and can be left out from an object-oriented language.

3.2 Overview

To create a system satisfying the above requirements, we have chosen an Erlang technology named Parse Transformation. See page `erl_id_trans` of (Erlang Manual).

There are two options to extend a language with a parse transformation. One can use certain syntactically correct elements, and give them new semantics by the transformation. Second, one can find syntactic elements that are invalid for some reason, but accepted by the parser, and give semantics to them. An example for the former will be pattern-matching with objects, and method-calls for the latter.

When using our framework, the programmers write their programs using our extended syntax, and the parse transformation generates code that is standard Erlang. Behind the scenes, for each class a record type is generated, and each object instance is represented by a record instance. These records contain all the fields of the class, including the inherited ones. The name of the record type is the same as the module that implements the methods of the class. The methods are ordinary Erlang functions. They receive the object instance record in their first parameter. Methods are virtual, and the function of the method is always called from the module with the most specific implementation of the method.

The novel feature of our approach is that object instances can be pattern-matched, and the pattern can also be a pattern of an ancestor class. There is no such feature in other OO extensions of Erlang.

3.2.1 Source files

Our extension distinguishes two types of modules. Class modules begin with `-class(classname)` instead of the original `-module(modulename)`, and each such module defines exactly one class. Client modules begin with the usual `-module(modulename)`. They differ from class modules in that they do not define classes, only use them. They are more similar to normal Erlang modules. Class-aware pattern matching, expressions and method calls are available in both module types.

Modules of both types use the same parse transformation, which detects the `-class` attribute and performs appropriate – class module or client module – transformations. Software developers using this system do not need to know about this parse transformation: the preferred way of using it is to include the `ect.hrl` header file in each module.

[3] In other terminologies these are called attributes, or the state of the object. We will use the term field, because we build objects on records that have fields. The other reason is that in the Erlang terminology, attributes usually refer to module attributes.

3.3 Defining a class

A class is a bundle of a module and a record having the same name. Some of the functions of the module are the methods of the class, and all of the fields of the record are the fields of the class. This means that a module defines exactly one class. When defining a class, the `-module(modulename)` attribute at the beginning of the file should be replaced with `-class(classname)`. This can be followed by three possible types of optional class specific attributes:

`?SUPERCLASS(superclass_name)` Defines the superclass by its name.

`?FIELDS(field1 = value1, field2, ...)` Fields of the class in the same format as in record definitions. After transformation, they will be added to a record, which has the same name as the class. Its definition will also contain the fields of the superclasses and an extra field for administration.

`?METHODS([methodname/arity, ...])` List of methods in the usual Erlang export syntax. (Methods are always automatically exported.)

The transformer generates a new record type and creates an includeable header file with the definition of it, and with some other administrative attributes that are necessary for compilation of client modules. It has the same name as the module, but with a `.class.hrl` extension. This header file must be included in a module if one wants to use class-aware pattern matching for the class (see later).

Let us see an example of a simple class definition:

```
1  -include_lib("ect/include/ect.hrl").
2
3  -class(class1).
4  ?FIELDS({field1,field2 = 2,field3}).
5  ?METHODS([method1/2, method2/2]).
6
7  -export([notmethod/1]).
8
9  method1(This, X) ->
10     {This, X+1}.
11 method2(This, a) ->
12     1;
13 method2(This, b) ->
14     2.
15 notmethod(A) ->
16     A+1.
```

In the above code, we defined a class with three fields. Field `field2` has a default value. There are also two methods: they are almost the same as ordinary functions. The only difference is that non-method functions – such as `notmethod/2`, which is not in the `?METHOD` list – will not be inherited to subclasses, while methods will be. This class has no superclass, therefore it will not inherit any methods.

When calling a method of an object, the object is automatically passed as the first parameter. It is analogous to *this pointers* in OOP languages. The name of the first parameter can be any valid variable name, however we will refer it by the name `This`, and we recommend the use of this name. Variables are immutable in Erlang, therefore if a method changes the value of a field, it must return the updated `This` object. Method `method1/2` shows the preferred way to return the modified object along with a return value. Note that the fields of the object are not used in this example.

3.4 Method calls and -inheritance

To understand method overriding, we must first discuss our method calling mechanism. When an object – an instance of a class – has a method, it can be called from any module using our transformation, with the following syntax:

```
1  -include("class1.class.hrl").
2  ...
3      %% create a class instance:
4      Instance = #classname{initialisers,
5                            exactly like for records},
6      %% method call:
7      {Instance}:methodname(Param2, Param3, ...)
```

Class instances can be created like record instances. This is not surprising since we represent classes with records.

A method call[4] is expanded into the following expression:

```
1  (element(1, Instance)):methodname(Instance,
2                                    Param2,
3                                    Param3, ...)
```

This shows that the object is always passed as the first parameter of the method. From the method's point of view, this is the equivalent of *this pointer*. Ordinary function calls can play the role of static method calls.

The other thing to notice here is that the module name of the called method is determined by the first element of the tuple of the class, which is in fact its name. It is extracted at runtime to ensure that having an object, method calls are always directed to the module of its most specific class. The methods implemented in the module of that class are called therefore directly. But this does not make it possible to call a method implemented in one of the superclasses – an inherited method. To achieve this, a so called *stub function* is generated in the module of the class for each inherited method. The name and arity of this function is the same as the inherited methods' , but it has only one clause with only one expression: a call to the inherited method. Consider the following example: let us suppose that `area/2` is a method in a class named `geometry`:

```
1  area(This, {rect, W, H}) when W > 0, H > 0 ->
2      W*H;
3  area(This, {square, A}) when A > 0 ->
4      A*A.
```

And also let us suppose that another class named `xgeometry` extends this class, and does not define a method with name and arity of `area/2`. However, users of this class should be able to call `{Obj}:area/2` on its instances, therefore the following stub is generated:

```
1  area(Arg1, Arg2) ->
2      geometry:area(Arg1, Arg2).
```

This stub calls the implementation. Creating stubs is optimised in the sense that always the direct implementation is called. For example if a third class would extend `xgeometry` and does not implement `area/2`, its stub method would call directly `geometry:area/2` and not `xgeometry:area/2`. Thus we limit the number extra function-calls per method call to one. We used guards and patterns in `geometry:area/2` to emphasise that no guards and patterns are copied to the stub. Choosing the clause remains the responsibility of the implementing class.

[4] The value of the `{Instance}` expression can never be a valid module name, but this construct is accepted by the parser. We use this syntax to denote a method call.

3.5 Defining a subclass

In the following example we extend `class1` to `class2`.

```
-include("class1.class.hrl").

-class(class2).
?SUPERCLASS(class1).
?FIELDS({x, y, z}).
?METHODS([method2/2]).

method2(_, _) -> ok.
```

It is essential to include the header file of the superclass, because that contains all necessary information about it.

First, let us see the generated source code:

```
-module(class2).
-export([method1/2,method2/2]).
-include("class1.class.hrl").
-include("class2.class.hrl").
method2(_, _) ->
    ok.
method1(Var2, Var1) ->
    class1:method1(Var2, Var1).
```

As the code shows, a stub function is generated for `method1/2`, because it must be available through the call of `class2:method1/2`, since it is inherited. No stub is generated for `method2/2`, since its implementation in `class2` overrides the implementation in `class1`.

It is time to examine the generated header file as well:

```
-include_lib("/ect/include/ect.hrl").
-record(class2,{'_types' = {class1,class2},
                field1,
                field2 = 2,
                field3,
                x, y, z}).
-classlevel({class2,2}).
-vmt({class2,[{method1,2,class1},
              {method2,2,class2}]}).
```

The role of the different things are the following:

- The `class2` record contains the fields of both `class1` and `class2`. The new fields are always appended at the end, in the same order as defined in the `?FIELDS` directive of the `erl` file. This is an unambiguous representation, because we do not support multiple inheritance.
- Attribute `-classlevel` and the record field `_types` are used for $O(1)$ time runtime type checking. (See section 3.8.)
- The attribute `vmt`[5] stores for each method, which is the most specific class that implements it. Methods are represented by their names and arities. This information will be used when compiling a subclass of `class2`. It makes possible to create stubs that redirect in one step to the implementation functions, instead of calling through a chain of stubs.

  The `vmt` for a class can be generated from its methods, and the `vmt` of the superclass.

3.6 Method inheritance

An important principle of OOP is the principle of substitutability. It says that if A is a subclass of B, then instances of B can be used in any place where instances of A can be used. By usage, we mean method calls and field manipulation. Field manipulation is a more complex matter and will be discussed in the next section. As for method calls, the principle is followed, because a subclass always inherits all the methods of the superclass.

The following example demonstrates possible method-call scenarios. It also introduces a new syntax element for method calls: calling a method of the superclass. Note that in this simplified source code, all functions are methods.

```
% extract of class1.erl
method1(This) -> 2.
method2(This) -> {This}:method1().
method3(This) -> method1(This).
```

```
% extract of class2.erl
?SUPERCLASS(class1).
method1(This) -> {{This}}:method1()+40.
```

This example introduces the following new syntax:

```
{{Instance}}:methodname(Arg2, Arg3, ...)
```

It can not be used in client modules, only in such class modules that have a superclass. The meaning is: "call the method named `methodname` in the superclass of the current class (and not in the superclass of `Instance`)".

The key point in this example is `method1`. We demonstrate the issues of method-calls on it. For this, let us suppose that we have an instance of `class1` named `O1`, and an instance of `class2` named `O2`. There are six possible calls:

{O1}:method1() is directed to `class1:method1(O1)`.

{O1}:method2() is directed to `class1:method2(O1)`. The call in its body is directed to `class1:method1(O1)`.

{O1}:method3() is directed to `class1:method3(O1)`. The call in its body is directed to `class1:method1(O1)`.

{O2}:method1() is directed to `class2:method1(O1)`, because it overrides `class1:method1/2`. It calls `class1:method1(O1)`, and adds 40 to its return value.

{O2}:method2() is directed to `class1:method2(O1)`, because it is an inherited method. The call inside it is directed to `class2:method1(O1)`, because it is overridden. This latter is a desired behaviour, and can be utilised for example to create easily extendable algorithms.

{O1}:method3() is directed to `class1:method3(O1)`, because it is an inherited method. However, the call inside it is always directed to `class1:method1(O1)`, because the ordinary (static) function-call syntax is used.

Now let us return to the syntax of superclass-calls. The implemented one is:

```
{{Instance}}:methodname(Arg2, Arg3, ...)
```

And an alternative one is:

```
?SUPERCLASS:methodname(Instance, Arg2, Arg3, ...)
```

The alternative one stresses better that the destination of the call is static – it is the superclass. However, the implemented one emphasizes better that this is a method-call of an object.

3.6.1 Overriding a single clause

In eXAT, it is possible to override the behaviour of a method by adding new clauses before the clauses of the method of the superclass(2.3). There is no such feature in ECT, however it is easy to program such overriding:

[5] virtual method table

```
-class(beta).
?SUPERCLASS(alpha).
?METHODS([m/2]).

m(Pattern1,Pattern2) when Guard -> do_something(...)
m(This, X) ->
    % call implementation of superclass.
    {{This}}:fact(This, X).
```

3.7 Field manipulation

The following example recalls the possible ways of record field manipulation in Erlang:

```
-record(example, {a, b, c}).
demo() ->
    % instance creation:
    X = #example{b = 6},
    % field extraction (with pattern match):
    #example{a = A, b = B} = X,
    % field update:
    Y = X#example{a = 1, b = 2},
    % single field extraction expression:
    C = Y#example.c,
```

The important thing here is that all operations explicitly state the name of the record. When a record pattern match, single field extraction or record update is executed, it is checked whether the input value is a valid record instance. This type check passes if two conditions are met:

- The input value is a tuple with the correct size (number of record fields + 1)
- and the first field of the tuple is the record name.

However, similar operations on objects should behave such a way that instances of any descendant class of a superclass can be matched and updated with the type of the superclass. The following example shows what should work:

```
%...definition of classa and a descendant class classb
demo() ->
    % instance creation:
    X = #classb{b = 6},
    % field extraction (using own class):
    #classb{a = A1, b = B1} = X,
    % field extraction (using superclass):
    #classa{a = A1, b = B1} = X,
    % field update (using own class):
    X1 = X#classb{a = 1, b = 2},
    % field update (using superclass):
    X1 = X#classa{a = 1, b = 2},
    % single field extraction (using own class):
    C1 = X#classb.c,
    % single field extraction (using superclass):
    C1 = X#classa.c,
```

Lines 6, 10 and 14 work with the standard record operations. Lines 8, 12 and 16 would fail with standard record operations, but should work in ECT.

There is no easy way to turn off type-checking of records[6]. However switching type checking completely off would have an undesired side-effect: it would became possible to access a field of a class with the pattern of an unrelated class. For a correct behaviour, we need a type check that answers the following question: "Is `X` an instance of class `Y` or one of its descendants?" The implementation of this check is discussed in section 3.8.

In the previous example we showed class-operations with the same syntax as record-operations. Objects are records with an extra field, so the only difference is the method of type-checking. However it is not obvious that using the same syntax is the best choice. We have two options:

Use a new syntax – a new syntax that the parser accepts, but was not legal in Erlang. An example for this is
`obj#classname{field1 = Pattern1,...}`
for patterns, and
`{Instance}#classname{field1 = Value1,...}`
for field-updating expressions.

Extend record syntax – the semantics of the current record-syntax can be extended: record manipulations behave ordinarily when the stated name is the name of a record, but behave according to class-semantics when the stated name is the name of a class. The parse transformer should know the set of class-definitions, locate all record-manipulating constructs and transform those that work on classes.

Until a point in the development, we planned to create a syntax that differs from record manipulation for these behaviours, but according to the following arguments, we have changed the design.

Possible arguments about having the same syntax for classes as records:

- \- Misunderstandable, because the same expressions has a different meaning.
- \+ The meaning is different, but strongly related to the original: "Check the type of an entity and take/update a field of it". This makes the usage intuitive.
- \+ Simpler source code: easier to learn and understand.
- \+ In common imperative languages like C++ and C# record/struct and object fields are accessed the same way. There are examples where only one construct exists for the role of records and classes: classes in Java and records in Oberon.

Therefore we have chosen to implement field manipulations for objects with record syntax. To emphasise the differences of object field-handling in the following texts, we will distinguish the patterns and expressions when they deal with classes from any other patterns or expressions by the terms: *class-patterns* and *class-expressions*.

3.8 Type checking

The current implementation transforms a record-pattern or -expression, if and only if the stated record type is the type of a *known class*. Known classes in a module are the included classes and the current class if the module is class module. The generated code does the type checking instead of the built-in record type check. All the field extractions and updates are decomposed into code using the following Erlang guards and BIFs: `element/2`, `setelement/3`, `tuple_size/1`. The `begin-end` construct is also used.

Before any object-field manipulation, the class of the object will be checked, like the type is checked for record instances. This can be done in $O(1)$ time, with the following rule(Mössenböck 1993): the class of object O is subclass or equal to the class C if and only if $O.types[C.classlevel] = C$, where

- $O.types$ is an array containing the classes of O from the most general to the most specific. The most general superclass is at index 1, and the last element is the most specific class of O.

[6] For single-field extractions, it can be turned off with compiler option `no_strict_record_tests`

- *C.classlevel* is the depth of *C* in the class hierarchy. In other words, this is the number of superclasses + 1.

It is easy to see that *C* can only appear at the *C.classlevel*th position in the array of any object. Our objects carry the *types* array as a tuple in their `_types` field. This is the second field in the tuple of the object.

The formula can be rewritten as a guard:

```
ClassName =:= element(ClassLevel, element(2, Object))
```

or as a pattern match:

```
ClassName = element(ClassLevel, element(2, Object))
```

As we mentioned above, the `_types` record-field contains the names of the superclasses and the class of `Object`. This information is in fact the property of the class of `Object`. We store it per object, just like the record name is stored per record instance.

Note that in the above code piece, `ClassSize`, `ClassLevel` and `ClassName` are written as Erlang variables, but in fact all of them are known at compile-time, and are literal constants in the generated code. The value of `ClassName` is the stated name – the name after the hash-mark (`#classname`). The level and size for the stated class can be extracted at compile-time from the header file of the class.

Generated source code examples after this point will be written as if there would be a BIF for the previous guards, named `is_object(Object, ClassName)`. The name choice is based on the similarity to Erlang BIF `is_record(Record, Name)`. When it appears in a guard, it stands for
`ClassName =:= element(ClassLevel, element(2, Object))`.
When it appears as an expression, it stands for
`ClassName = element(ClassLevel, element(2, Object))`.

3.9 Query expressions

A query expression extracts one field from an object.

```
Expression#classname.field
```

is converted to

```
1 begin
2     X1 = Expression,
3     is_object(X1, classname),
4     element(NNN, X1)
5 end
```

where `NNN` is a constant number, representing the position of field in class `classname`. First, the expression is evaluated and stored into the variable `X1`, this is to avoid evaluation more than once. After that, a type check is done, which when fails, causes a runtime error. The result of the third expression is the field in question. Naturally, `Expression` can also contain class field queries, therefore they are converted recursively if needed. An example for a nested query expression can be found in (Fehér 2008).

The name of the variable `X1` is generated by the transformation. In the current implementation, to minimise the risk of name clashes, these variables in fact named `_CCTRANS_1`, `_CCTRANS_2`, etc. For better readability, we use `X1`, `X2`, etc. in this text.

3.10 Update expressions

Update expressions create a new object from an existing one, changing some fields of it. The existing object, and the new field values can of course be results of expressions. So

```
1 Expression#classname{
2     field1 = ValueExpr1,
3     field2 = ValueExpr2
4 }
```

is converted to

```
1 begin
2     X1 = Expression,
3     is_object(X1, classname),
4     X2 = ValueExpr1, X3 = ValueExpr2,
5     setelement(NNN1, setelement(NNN2, X1, X3), X2)
6 end
```

`NNN1` and `NNN2` are the positions of `field1` and `field2`, respectively. After the type-check, all the fields are evaluated. According to section 3.5 of (Erlang Efficiency Guide), the Erlang compiler optimises `setelement` calls when there are no other function-calls, or access to the tuple between them, and the indices are variable literals in descending order. This optimisation causes that only one copy of `X1` is made during the `setelement` chain, instead of making one copy per one `setelement` call. The values are evaluated before the `setelement` chain to trigger this optimisation. The order of the calls is also rearranged to update fields in descending order.

Due to the use of `begin-end` blocks, nesting is also possible here. So an expression of a field can contain further class-expressions.

3.11 Pattern-matches

A pattern-match expression has the following form:

```
Pattern = Expression
```

If a pattern is (or contains) a class-pattern, the pattern-match is converted, as in the following example:

```
#class1{field1 = 5, field2 = B} = Obj
```

is transformed to

```
1 begin
2     X0 = Obj,
3     is_object(X0, class1)
4     5 = element(3, X0),
5     B = element(4, X0),
6     X0
7 end
```

- Line 2 binds a variable to the matched object. The matched object is used several times in the next lines. If it is an expression, it is important to evaluate it only once.
- Line 3 is the type-check.
- Line 4 and 5 test the value of field1 and field2 with a pattern-match.
- Line 6 sets the result of the begin-end to the value what the pattern-match would have.

More examples of complex pattern matches can be found in (Fehér 2008).

3.12 Patterns in clause heads

Besides pattern match expressions, patterns show up in clause-based constructs (case, try-catch and receive expressions, functions

and funs). These patterns need to be handled differently than pattern matches. In the following subsections, we describe some of the interesting issues.

We must consider that:

- Any value-tests derived from the class-pattern must be placed in the guard, to maintain correct program behaviour, which is: skipping to the next clause instead of raising a runtime error on match failure.
- It is not possible to assign values to variables in the guard.

These imply that the class-patterns must be split into two parts: a value-testing, which goes into the guard, and a variable assignment, which goes into the beginning of the body. Luckily, every pattern can be converted into a single-variable pattern plus a guard and variable assignments in the body. In the following subsections we describe each case to consider when doing this transformation.

3.12.1 An unbound variable in a pattern

An unbound variable means that its value is to be determined by the pattern-match; in case it is in a class-pattern, it means that its value will be extracted from the object, against which the pattern is matched. (If it is not in a class-pattern, the transformation keeps it untouched.) What is done is the following: class-patterns in the head are substituted with variables, a type-check is added to the guard, and the values are extracted at the beginning of the body:

```
1 clausedemo1(A = {_, #class1{field2 = B}}, C)
2     when C > 5 ->
3     A+B+C;
```

is transformed to

```
1 clausedemo1(A = {_,X0}, C)
2     when is_object(X0, class1), C > 5 ->
3     B = element(4, X0),
4     A+B+C;
```

3.12.2 Constants in the pattern

When the pattern contains constant expressions, they can not be checked in the body, because then the match-failure will cause a runtime error, instead of the desired behaviour which is to step to the next clause of the function. The solution is to do the check in the guard:

```
1 clausedemo1(
2     A = {_, #class1{field2 = B, field3 = xyz}},
3     C) when C < 5 ->
4     A + B + C;
```

is transformed to

```
1 clausedemo1(A = {_,X1}, C)
2     when
3         is_object(X1, class1),
4         xyz =:= element(5, X1),
5         C < 5 ->
6     B = element(4, X1),
7     A + B + C;
```

3.12.3 Bound variables in a pattern

When the pattern contains a variable which already has a value, it is treated similarly to constants: checked in the guard. The boundness of a variable tells whether to put its value-extraction into the guard as a test, or into the body as a variable binding. Because of this, ECT needs to know if a variable is bound or unbound. To achieve this, it detects variable creations as it parses a source code, and maintains the set of bound variables.

3.12.4 Variables present in a pattern and also in the guard

Another case is when a variable appearing in a class-pattern, also appears in the guard. This does not affect the transformation of the pattern in any ways, but since the variable will only be bound in the body, the guard can not use it. Instead, it can use an expression whose result is the value of the variable. For example, in:

```
1 clausedemo1(A = {_, #class1{field2 = B}}, C)
2     when B =:= 42 ->
3     A + B + C;
```

is transformed to

```
1 clausedemo1(A = {_,X3}, C)
2     when
3         is_object(X3, class1),
4         element(4, X3) =:= 42 ->
5     B = element(4, X3),
6     A + B + C;
```

In line 5, `element(4, X3)` extracts the value of `B` for the guard.

3.12.5 Variables appearing multiple times in patterns

One more case remains, when a variable appears more than once in a pattern. In this case, binding the variable in the function body multiple times is not acceptable, because in this case different values trigger a runtime error, while the desired behaviour is that the clause is not selected. Therefore the guard must check that all occurrences of the variable get the same value, and the variable is bound in the body.

3.12.6 Composite sub-patterns for field extraction

Until this point, we only discussed single value extractions from objects into variables, and single value tests. Further complications appear when a class-pattern contains complex sub-patterns i.e. not variables or constants, but tuple, record or list structures with variables inside.

Because the class pattern matching happens in the guards, the sub-patterns found in the class pattern must also be transformed into guard tests and variable assignments in the body. This always can be done, because there are selector functions for all the compound data types, and they can be used in guards:

`hd(List)`: The head of `List`.

`tl(List)`: The tail of `List`.

`element(N,Tuple)`: The `N`th element of `Tuple`.

All variables appearing in class-patterns will be extracted with the combination of these. For example, the transformation of a clause with complex patterns, and special cases:

```
1 clausedemo3([_,#class1{field1 = A,
2                        field2 =
3                          #class2{z = #rec{b = Z}}},
4               #class1{field3 =
5                         [_, {_,_,Z}]}], Z) ->
6     A+Z.
```

is transformed to

```
clausedemo3([_,X0,X1], Z)
  when
    is_object(X0, class1),
    is_object(element(4, X0), class2),
    %% test record type:
    is_record(element(8, element(4, X0)), rec, 4),
    Z =:= element(3, hd(tl(element(5, X1)))),
    %% test tuple size:
    3 =:= tuple_size(hd(tl(element(5, X1)))),
    is_object(X1, class1),
    Z =:= element(3, element(8, element(4, X0)))
    %% test list closure:
    [] =:= tl(tl(element(5, X1))) ->
  A = element(3, X0),
  A + Z.
```

Note that Z occurs (once) outside of a class-pattern, so it gets its value in the head, it can be used in the guard. The other two occurrences of Z are inside a class-pattern, so their values are extracted with nested `hd/tl/element` calls in the guard.

One might ask how efficient are these nested value extraction functions in the guard. We examined the compiled assembly code of our generated class-matches with "normal" Erlang record pattern matches, and found that they are very similar or the same. However, we have not examined this in detail.

3.12.7 Summary of transformation

When processing a pattern, the following steps are done in the following order:

1. The class-patterns are substituted with variables in the head.
2. These class-patterns are converted into sequence of value-extractions, and stored.
3. Based on the list of bound variables, these extractions are sorted into two groups: (1) real extractions, and (2) extractions that are in fact value-tests, which access constants or bound variables.
4. Those variables in the guard, which appear in class-patterns, substituted with their corresponding value extractions.
5. Members of group (1) are appended to the beginning of the function body.
6. Members of group (2), along with necessary type-checks are appended to each guard sequence of the current guard.
7. Equality tests for variables that are the result of multiple extractions – group (1) – are appended to each guard sequence of the current guard.

3.13 Remote objects

In our approach, an object is an Erlang record, and a method invocation is a function that calculates some return value and possibly a modified copy of the record. Sometimes it is desirable to be able to call the methods of an object from several different processes. To achieve this, an object can have a process that receives method invocation requests, sends the return value of them, and stores the state object record in the process loop data. WOOPER and eXAT only allow this approach. With ECT, this approach is also available. Currently, we only have a prototype implementation. It was implemented in order to be able to compare the object-as-process performance of ECT to WOOPER and eXAT, but in the future it will be part of ECT.

We added some extra generated code to class modules and created a new module `ectremote` which is the API for handling objects remotely. Currently, the following operations are available for a remote object.

get Queries a field by its name.

set Sets the value of a field.

query_call Invokes a method of the object. The return value will be sent back to the client unchanged, and the fields of the object are not changed.

update_call Invokes a method of the object. The return value will be sent back to the client unchanged. The state of the object is updated.

delete Terminates the server process.

copy Sends (a copy of) the internal object to the client.

To make remote objects efficient, the server-loop is generated at compile-time into the class module, and the processing of incoming messages is done entirely by pattern matching. This is most notable in the case of get/set operations, where in the `receive` construct a separate clause is generated for each field's get and set operations.

4. Implementation

When a module is processed by ECT, first it checks if it is a class module. If this is the case, its class specific parts are compiled into native Erlang code. After this, the module can be treated as a client module: resolving of class-patterns and class-expressions is done by the client transformation.

Our parse transformation has to satisfy two equally important requirements:

- It must reach every expressions in the *Abstract Syntax Tree* (AST in the sequel) of the transformed module, even at deeply nested places such as "in a case construct in a fun in an if construct in a function".
- Our transformation is not context-free, so some state information must be maintained while traversing the AST. Examples for state information: the class name, the record definitions found so far, the set of bound variables, a counter to guarantee that no two variables with the same name are generated, etc.

The first requirement can be satisfied, if we take `erl_id_trans.erl`[7] from the Erlang distribution, and replace the necessary function clauses with our custom processing. This is how we created our first transformation (currently used as bootstrapper).

For the second requirement, some additional arguments must be appended to the argument list of all clauses, and every return value must contain their updated instance. We solved this by a single extra argument, a record with the state variables as fields. For this, return values were turned into 2-sized tuples that contain the state record and the original return value of the function. The main problem with this is the lack of flexibility and readability, because clauses responsible for traversing normal Erlang AST, and clauses responsible for the transformation of our modified semantics AST are mixed in a module. Also, this module must be edited, whenever there is a change in the Erlang AST specification.

Our idea to improve the situation was the use of the Template method pattern (Gamma et al. 1995) with our new OO-syntax. `erl_id_trans.erl` is turned into a class named `idtrans`, with no fields. Then we extended this class, with fields that store the state information that are maintained as the AST is traversed (class `cctrans`). To change the behaviour of a certain clause in the identity transformation, there's no need to rewrite it, it can be overridden as a virtual method in `cctrans.erl`. When something in the official `erl_id_trans.erl` changes, there is a good chance that

[7] `erl_id_trans.erl` is an identity parse transformation, which traverses the syntax tree and returns it unchanged.

only `idtrans` needs to be updated, and `cctrans` can be kept unmodified.

This way we separated the transformation-logic-specific and Erlang tree traversal specific aspects of the transformation. This is a good general pattern for writing parse transformations. It could also be extended with a third class between `idtrans` and `cctrans`, which provides general, domain-independent services for parse transformations, for example: unique variable name generation, error handling, etc. We did not implement this because then the bootstrap compilation would be more difficult.

5. Performance analysis

In this section we describe the performance tests we made on our and on other OO extensions for Erlang. In the first subsection, we describe what properties we measured. In the second subsection, we highlight some of the technical issues that were raised during the measurements. In the third subsection, we present and analyse the results.

5.1 Measured properties

We measured and compared the speed of object-operations in the following systems: ECT, Remote ECT, eXAT, Parameterized Modules, WOOPER. We also measured the corresponding non-OO operations of Erlang: records, static function call (`module:function()`), dynamic function calls (`Module:function()`) and `funs`. The measured operations were: method calls, field queries and field modification. The term field modification means different things for different systems. For ECT and Erlang records it is making a copy of an object with some fields changed. For Remote ECT, eXAT and WOOPER, it is destructively updating a field.

We examined the how the execution time depends on the depth of the inheritance chain. We created the class hierarchy seen in Figure 2 for each system, and measured method call times on an instance of `alpha`, `beta`, `gamma` ... etc. The body of each method is just the atom `ok`, which they return as a result.

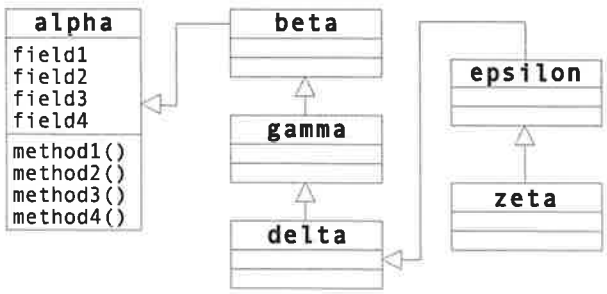

Figure 2. Class hierarchy used for benchmarking

This way we could examine how the distance of the method implementation and the object instance in the inheritance chain affects the time of a call.

We set the following parameters to be constant:

- The number of defined fields: 4.
- The number of defined methods: 4.
- The number of accessed fields per operation: 1.

  Records and ECT support pattern matching, and more than one field-extractions can appear in a pattern-match. An update-expression for records can also update more than one field. We chose to measure one-field operations for two reasons. First, for comparison with systems that support only one. Second, because in theory, this is the worst-case scenario, because each field operation might trigger a type-check. With the use of multiple fields, the overhead of the type checks per field operation could be smaller.

5.2 Technical issues

A method call or a field read is a rather short operation. Their execution times are usually in the range of nanoseconds or microseconds. This is such a short time that operating-system interrupts can significantly influence it. The timer function for Erlang also works in the range of microseconds. To measure such a short time, we ran the operation several times, and divided the total time by the number of operations. We also made some steps to minimize the effect of the loop itself that runs the operations.

To get a picture on the stability of these measurements, we repeated each of them several times, and the average was taken as final result. The point of this was that the difference of single measurements from the average characterises the stability: for 77% of the cases, the difference was under 1%; for 20% of the cases, it was between 1% and 5%, and for the remaining 3% it was between 5% and 7%.

The computer we used for the measurements had an Athlon64 X2 4200+ CPU running at 2210MHz, with the second core turned off.

5.3 Results

The following table summarises the results of the performance tests. Each row with a bold face denotes the beginning of the section of a system. Each remaining row represents a measured operation of the system they belong to. Columns denote the class, whose instance was tested. The values are in microseconds.

	alpha	beta	gamma	delta	epsilon	zeta
Erlang static function call						
Call	0.010			n.a.		
Erlang dynamic function call						
Call	0.088			n.a.		
Erlang fun call						
Call	0.042			n.a.		
Erlang records						
Read	0.026			n.a.		
Write	0.102			n.a.		
Parameterized modules						
Call	0.090	4.283	8.451	12.626	16.773	20.848
ECT classes						
Call	0.094	0.105	0.104	0.104	0.104	0.104
Read	0.047	0.047	0.047	0.047	0.047	0.047
Write	0.107	0.107	0.107	0.107	0.107	0.109
ECT remote classes						
Call	0.839	0.885	0.884	0.889	0.891	0.879
Read	0.715	0.702	0.702	0.702	0.701	0.698
Write	0.436	0.436	0.437	0.439	0.444	0.440
eXAT classes						
Call	1.394	7.301	13.254	19.635	25.453	31.533
Read	1.759	1.761	1.768	1.767	1.766	1.769
Write	2.736	2.750	2.759	2.747	2.740	2.739
Wooper classes						
Call	1.915	1.977	1.968	1.968	1.971	1.979
Read	2.342	2.354	2.337	2.337	2.335	2.342
Write	2.012	2.017	2.015	2.014	2.016	2.012

In the following subsections we highlight some facts from the above table.

5.3.1 Dependence on the depth of class hierarchy

- Field manipulation times are near-constant functions of the depth in all systems.
- Method calls for ECT, Remote ECT and WOOPER are near-constant functions of the depth.
- Method calls for eXAT and Parameterized Modules are near-linear functions of the depth.

5.3.2 ECT versus Erlang

- Method calls of ECT are about 10 times slower than statically bound, global function calls.
- However if they are compared to dynamic function calls, the picture is better: only 2.5 times slower than `fun`-s and 1.2 times slower than dynamic calls.
- Field read is about 2 times slower than record field read.
- Field update is less than 1.1 times slower than record field update.

5.3.3 ECT versus other OOP

When considering object-as-process systems, Remote ECT outperforms both eXAT and WOOPER. The narrowest difference is against eXAT, when comparing the times of method calls with no inheritance: ECT is 1.6 times faster.

The only object-as-data OOP system we measured was Parameterized Modules. We did not measure field access times because – as we stated in section 2.2 – we did not find field handling fully functional. The results for method calls:

- When the called function is not inherited, Parameterized Modules are faster by about 1.04 times.
- When the called function is inherited, the speed of parameterized modules decreases linearly. In this case ECT is at least 40 times faster.

6. Future work

While ECT already provides the main functions of an object-oriented extension, there are areas to improve:

- Generating proper error messages when invalid source code is found.
- Implementing `is_object(Object, Class)` to be available for programmers.
- Improving the currently available ad-hoc automated tests.
- Check for name-clashes of variables generated by the compiler.
- Integration with Erlang behaviours: if a class implements a behaviour, subclasses should inherit this feature so that they will also implement it.

6.1 Use of upcoming Erlang features

There is a proposal (O'Keefe 2008) for the Erlang language to allow binding variables in guards. If it will be implemented, the transformation can be simplified: no separate value-extraction will be needed in the beginning of clauses, and there will be no need for equality test of these values.

6.2 Better integration with the compiler

One possibility is to modify the Erlang compiler to create code immediately from our syntax. However records – which are similar to our classes – are also implemented by a preprocessor transformation. Another possibility is to propose new elements for Erlang language that make code generation for classes easier:

Support Object:method()-style calls: The Erlang language already supports `Object:method()`-style calls for parameterised modules. This syntax is similar, and possibly ECT should be switched to this. However semantics differ, because the instance variable is passed as the last argument – and not the first, as in ECT.

Better support for class-patterns and class-expressions: The following concept would made class-matching very simple. (We have not examined its feasibility.) Let us consider the following tuple-pattern:

```
{SubPattern1, SubPattern2, ..., SubPatternN, *}
```

It matches any tuple that has at least `N` elements, and the corresponding `SubPatternss` match the corresponding elements. For example:

```
{A, _, B, ok, *} = {1, 2, 3, ok, 4, 5, 6, 7},
A = 1, B = 3    % the results of the match
```

This would allow the following transformation of class-patterns[8]:

```
#class2{b = 4, field2 = Z} = Obj
```

to

```
{class2, {_, class2, *}, _, 4, _, _, Z, _, *}
```

New syntactic elements: Some of the current syntax elements are ugly or clumsy. This is because a parse transformation can only work on a source file if it can be parsed. If the parser of the Erlang compiler is modified, any new syntax can be introduced.

One example is the `This` parameter of method functions. It is possible to automatically insert the first argument – which is the "This" object – for methods, like for Parameterized Modules. This might seem convenient, but the problem is that pattern-matching on `This` in the head would be impossible.

In Oberon-2 (Mössenböck 1993), functions that are methods have a different syntax from plain functions:

```
PROCEDURE (VAR obj: Object) Name(param1: Type1, ...)
```

The this-pointer – `obj` – is defined separately from the other arguments. This would seem the following in ECT:

```
(This)method(Arg1, Arg2, Arg3) -> ...
% or
method(This)(Arg1, Arg2, Arg3) -> ...
```

The intention behind this concept is to separate the `This` instance from other arguments. It would also make it possible to mark methods, instead of the currently used marking: `?METHODS([method1/N, method2/N, ...])`.

7. Conclusions

In this paper, we discussed the design, the implementation and the performance of ECT, our object oriented extension to Erlang. The new language elements we added provide extensible objects that encapsulate data and functionality, while preserving the single-assignment nature of Erlang. Fields and methods can be inherited. The objects instances of ECT are in fact records, which implies that the related features and tools of Erlang can be used with them: they can be matched with record patterns, can be pretty-printed in

[8] `class1` and `class2` are defined in sections 3.3 and 3.5

the shell, can be stored in MNESIA, etc. The other approaches that we studied do not have this feature. We believe storing objects as records is the main advantage of our solution compared to others. The main difference from eXAT and WOOPER is that in our solution the programmer is not forced to create processes for each class instance. Although it is still possible to do so when necessary.

We also measured the performance of other implementations. To compare ECT with other OOP extensions that store objects in processes, we added a similar feature to ECT, which we named Remote ECT. When ECT is compared to standard Erlang constructs, it has a slowdown of maximum 2.5 times. However when compared to other OOP extensions for Erlang, it outperforms them in all situations by at least 1.6 times, with one exception: in a very special case ECT and Parameterized Modules perform equally.

ECT is free software and its source code can be downloaded from `http://ect.googlecode.com`.

We demonstrated the applicability of ECT by using it in the client transformation part of the system itself. By using ECT, the client transformation code received a better structure and will be easier to maintain.

Acknowledgments

We would like to thank Péter Szeredi his valuable ideas, advises and all his help in writing this paper.

References

Olivier Boudeville. WOOPER: Wrapper for OOP in Erlang. `http://ceylan.sourceforge.net/main/documentation/wooper/`.

Richard Carlsson. Inheritance in Erlang. Erlang/OTP User Conference, November 2007. `http://www.erlang.se/euc/07/papers/1700Carlsson.pdf` (slides only).

Richard Carlsson. Parameterized modules in erlang. 2nd ACM SIGPLAN Erlang Workshop, August 2003. `http://www.erlang.se/workshop/2003/paper/p29-carlsson.pdf`.

Erlang Efficiency Guide. *Erlang Efficiency Guide*. `http://www.erlang.org/doc/efficiency_guide/part_frame.html`.

Erlang Manual. *Erlang/OTP R12B online manual*. `http://www.erlang.org/doc/man/`.

eXAT website. eXAT download site. `http://www.diit.unict.it/users/csanto/exat/download.html`.

Gábor Fehér. Object-Oriented Extension to Erlang. Student Conference (TDK). Budapest University of Technology and Economics, 2008. `http://ect.googlecode.com/files/tdk2008.pdf`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

Hanspeter Mössenböck. *Object-Oriented Programming in Oberon-2*, chapter Appendix A Oberon-2 Language Definition. Springer-Verlag, 1993.

Richard A. O'Keefe. EEP14: Guard clarification and extension, 2008. `http://www.erlang.org/eeps/eep-0014.html`.

Corrado Santoro. *Erlang eXperimental Agent Tool. Reference Manual*, 2004. `http://www.diit.unict.it/users/csanto/exat/exat.ps`.

Antonella Di Stefano and Corrado Santoro. Designing collaborative agents with exat. 2nd International Workshop on Agent-based Computing for Enterprise Collaboration (ACEC 2004) at WETICE 2004. `http://www.diit.unict.it/users/csanto/exat/ACEC-Santoro-eXAT.ps` Modena, Italy, June 14-16, 2004.

Implementing an LTL-to-Büchi Translator in Erlang

a ProTest Experience Report

Hans Svensson

Department of Applied IT, Chalmers University of Technology, Gothenburg, Sweden
hanssv@chalmers.se

Abstract

In order to provide a nice user experience in McErlang, a model checker for Erlang programs, we needed an LTL-to-Büchi translator. This paper reports on our experiences implementing a translator in Erlang using well known algorithms described in literature. We followed a *property driven development* schema, where QuickCheck properties were formulated before writing the implementation. We successfully implement an LTL-to-Büchi translator, where the end result performs on par (or better) than two well known reference implementations.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Testing tools

General Terms Algorithms, Verification

Keywords LTL-to-Büchi translator, QuickCheck, property driven development

1. Introduction

Correctness of concurrent or distributed software is a well known, and immensely complicated problem. It is also a fact that during the last couple of years, the problem has become more important due to the introduction of multi-core processors. This has led to an increased interest in the problem, and a variety of good work has made the problem a lot more tractable.

Model checking is one of the most successful and mostly used techniques to prove correctness of a (hardware or software) system. The standard model checking problem consists of a model system expressed as a *finite state machine (FSM)*, and a specification given by a *temporal logic formula*. Model checking for temporal logic formulas was pioneered by Clarke et al. (1986) and as well by Queille and Sifakis (1982). The main obstacle in model checking is the famous *state space explosion* problem, due to a combinatorial blow-up of the state space (the size of the FSM). This combinatorial blow up is very problematic when dealing with concurrent and fault tolerant systems (meaning that most Erlang software falls into this category). However, there are some different techniques that try to deal with the problem such as:

- Symbolic model checking, where the state space is instead represented symbolically in terms of logic formulas.
- Partial order reduction techniques, where equivalent paths through the FSM are grouped together and reduced to a single path.
- Abstraction techniques, where the system is simplified by an abstraction function, thus reducing the size of the FSM.

One tool that aims to simplify the verification of correctness for distributed Erlang programs is McErlang, a model checker for Erlang. McErlang was first presented in (Fredlund and Svensson 2007), and has been used in a couple of successful projects (Fredlund and Sánchez Penas 2007; Earle et al. 2008). Since 2008 McErlang is an important part of the ProTest project and the release of the first public open source version in March 2009 is a result of this.

McErlang (Fredlund and Svensson 2007) is a fairly standard explicit state model checker, although through plugins more advanced concepts such as state abstraction can easily be used. The interesting part of McErlang is that it is a model checker for Erlang written in Erlang. As a result of this, side-effect free parts of a system can be evaluated as is, without tedious translation. This means that one can focus on the complicated parts such as distribution and fault tolerance, and the result is that McErlang supports a large subset of Erlang. McErlang implements the full distributed Erlang semantics (Svensson and Fredlund 2007) and all the important OTP components (`gen_server`, `gen_fsm`, ...).

The first (non-public) versions of McErlang encoded correctness properties (specifications) as simple automata programmed directly in Erlang. Having very little support for higher level constructs, this meant that writing properties was both tedious and error prone. For more complex properties (such as *fairness* properties) McErlang also supported Büchi automata, but they still needed be hand written.

To make McErlang more accessible before its first public release it was decided to simplify the specifications by adding the possibility of using LTL properties. This is fairly straightforward, since LTL expressions can be automatically translated into Büchi automata (Wolper et al. 1983). However, for model checking to be efficient it is important to produce as small an automaton as possible, thus a good translator was needed. The obvious solution was to use an existing implementation. However, no Erlang implementation could be found and although it is a simple task to wrap an existing Java implementation (such as Giannakopoulou and Lerda (2002)) it did not appeal to us aesthetically having advocated the all-in-Erlang aspect of McErlang. From a distribution point of view it is also simpler to have a native implementation, we avoid the licensing aspect as well as the problem of missing external components, while being in control of new releases and bug fixes. In the end, this made us decide to implement an LTL formula to Büchi automata translator ourselves.

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.

This paper describes the implementation of an LTL to Büchi automata translator in Erlang. The implementation includes some state-of-the-art technologies and it performs quite well as seen in Sect. 5. The paper focuses quite a lot on the verification of the correctness of the implementation, and could be seen as an experience report on *property driven development* and *property based testing*. We used Erlang QuickCheck (Hughes 2007) to formulate properties and run test cases randomly generated from the properties. Property driven development is the QuickCheck dual of *test driven development* (TDD) (Beck 2003), where the test cases drive the implementation forward. In practice it consists of three stages; (1) tests are written that do **not** pass for the implementation, (2) the implementation is extended so the tests pass, and (3) the code is refactored to produce the end result. The development process is iterative, and it worked well for our implementation. Using QuickCheck makes phase (2) even more productive. Whenever a test fail, QuickCheck provide a minimal counter example, and in many cases the counter example shorten the time to diagnose an error. In our work it was very useful to get small LTL formulas for which the implementation produces an incorrect automaton.

Paper organization – The paper is organized as follows; in Sect. 2 we introduce the theoretical background as well as the problem of testing an LTL to Büchi automata translation. Sect. 3 describes the properties that were defined during the implementation, and how various model checking concepts are mapped to QuickCheck. The implementation is described in Sect. 4, the implementation is evaluated and compared to other implementations in Sect. 5 and the paper is summarized in Sect. 6.

2. Theoretical background

In this section, we provide some theoretical background to the concepts discussed in later sections.

2.1 Linear temporal logic – LTL

To write specifications for reactive systems, we need to be able to precisely describe how the system behaves for (possibly) infinite executions. Temporal logic (Pnueli 1977) has become a de facto standard formalism for this kind of specifications, and there are quite a few flavors of temporal logic, here we focus on linear temporal logic. LTL is an extension of *propositional logic*. If AP is a (non-empty and finite) set of atomic propositions, then the LTL formulas are:

- All $p \in AP$ are LTL formulas.
- If φ and ψ are LTL formulas, then $\neg\varphi$, $(\varphi \wedge \psi)$, $\mathbf{X}\,\varphi$, $(\varphi\ \mathbf{U}\ \psi)$ are also LTL formulas.

The standard semantics for an LTL formula φ is defined in terms of an infinite sequence (ξ) over 2^{AP}. Let ξ^i be the infinite subsequence of ξ that begins at the ith position of ξ ($i \geq 0$). Now the relation ($\models$) is defined as:

- For $p \in AP$, $\xi \models p$ iff $p \in \xi_0$. (ξ_0 is the first element of ξ).
- $\xi \models \neg\varphi$ iff $\neg(\xi \models \varphi)$, usually written $\xi \not\models \varphi$.
- $\xi \models \mathbf{X}\,\varphi$ iff $\xi^1 \models \varphi$.
- $\xi \models (\varphi\ \mathbf{U}\ \psi)$ iff $\exists\, i \geq 0.\ \xi^i \models \psi$ and $\forall\, 0 \leq j < i.\ \xi^j \models \varphi$.

If $\xi \models \varphi$, then ξ is called a *model* of φ. The set of all models $\{\xi \in (2^{AP})^\omega \mid \xi \models \varphi\}$ is called the *language* of φ and is denoted by $\mathcal{L}_\varphi$.

2.2 Büchi automaton

Büchi automata were introduced by Büchi (1960). A Büchi automaton accepts infinite input sequences if and only if there exists a path that visits an accepting state infinitely often. A Büchi automaton is a tuple $BA = \langle \Sigma, Q, \Delta, Q^0, F \rangle$ where:

- Σ is the *alphabet*,
- Q is the finite set of *states*,
- $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*,
- $Q^0 \subseteq Q$ is the set of *initial states*, and
- $F \subseteq Q$ is the set of *accepting states*.

An *execution* of BA for an infinite sequence $\xi = \langle x_0, x_1, \ldots \rangle \in \Sigma^\omega$ is the infinite sequence of states $\sigma = \langle q_0, q_1, \ldots \rangle \in Q^\omega$ where $q_o \in Q^0$, and for all $i \geq 0$, $(q_i, p_i, q_{i+1}) \in \Delta$. Let $\mathit{inf}(\sigma) \subseteq Q$ denote the states that occur infinitely often in an execution σ, then σ is an *accepting* execution if and only if $(\mathit{inf}(\sigma) \cap F) \neq \emptyset$.

We say that the automaton *accepts* $\xi \in \Sigma^\omega$ exactly when there exists an accepting execution of BA on ξ. If there is no accepting execution, then BA is said to *reject* ξ. The *accepted* language of an automaton BA is the set of sequences ξ accepted by the automaton BA, it is denoted by $\mathcal{L}_{BA}$.

There is a close connection between languages induced by an LTL formula and the accepted language of a Büchi automaton, in fact any LTL formula can be represented as a Büchi automaton. This connection was explicitly shown by Wolper et al. (1983) and this property was used to develop the automata-theoretic approach to LTL model checking (Vardi and Wolper 1986).

2.3 LTL model checking

The LTL model checking problem answers the question of whether a specification (in the form of an LTL formula) is satisfied in a finite model of a system. The system is often (translated into) a state-transition graph where each state is augmented with exactly the atomic propositions that hold in the state. Such a transition system is known as a *Kripke structure*; more formally defined as a tuple $M = \langle S, \rho, \pi \rangle$, where:

- S the finite set of *states*,
- $\rho \subseteq S \times S$ is the *transition relation*,
- $\pi : S \to 2^{AP}$ is the *labeling function*, which means that $\pi(s)$ is the set of propositions that hold in a state $s \in S$.

The model checking problem can thus be formulated as: "For a Kripke structure $M = \langle S, \rho, \pi \rangle$, a state s and an LTL formula φ, is it true for all infinite paths $x = \langle s_0, s_1, \ldots \rangle \in S^\omega$ (with $s_0 = s$) that $\xi = \langle \pi(s_0), \pi(s_1), \ldots \rangle \models \varphi$". For practical reasons this is often reformulated as looking for *any* x such that $\xi = \langle \pi(s_0), \pi(s_1), \ldots \rangle \models \neg\varphi$, i.e. $M \models \varphi$ holds if no such x can be found.

By treating the set of paths starting in a specific state s as a language $\mathcal{L}_{M,s}$ the problem can compactly be expressed as checking whether $\mathcal{L}_{M,s} \cap \mathcal{L}_{\neg\varphi} = \emptyset$. This means that the LTL model checking problem can be solved by constructing $BA_{\neg\varphi}$ and $BA_{M,s}$ from the LTL formula φ and the Kripke structure M respectively, and then checking the automaton $BA_{\neg\varphi} \otimes BA_{M,s}$ for *emptiness* ($\otimes$ is the synchronous composition of two automata, and the resulting automata recognizes exactly the infinite sequences recognized by both automata). This check can be done by an algorithm that checks whether the constructed automata can reach an accepting cycle from any of its initial states (Clarke et al. 2000). The time complexity is linear in the size of the new automata.

2.4 Testing an LTL to Büchi automata translation

The theoretical aspect of how to test an LTL formula translation into Büchi automata is thoroughly covered by Tauriainen and Heljanko (2002). Here we briefly describe the most important aspects from Sect. 3 of (Tauriainen and Heljanko 2002). The methods described do not aim to *prove* the correctness of the implementation, but instead perform tests to detect inconsistencies in the implementations. Most of the ideas test the results given by an LTL-to-Büchi translator with the results from using other implementations. One particular situation could be to test different optimizations against

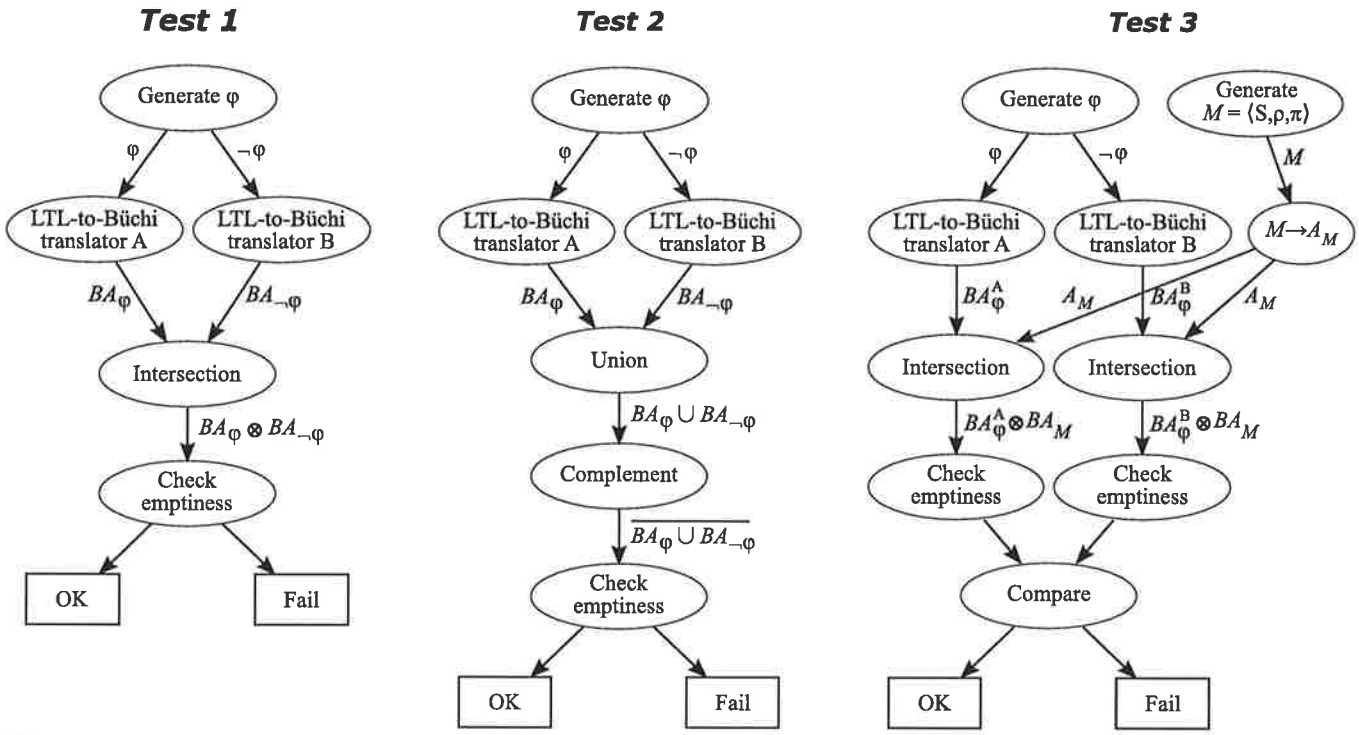

Figure 1. Testing an LTL-to-Büchi translation

(a stable) reference implementation. This is something that we have used frequently. We also used two reference implementations WRING by Somenzi and Bloem (2000) and LTL2BUCHI by Giannakopoulou and Lerda (2002).

Interestingly, we found that WRING fails for some particular input. Therefore, in order to be able to use WRING as a reference implementation, we implemented a pre-condition check that guarantee that the generated formula is one that does not crash WRING[1].

2.4.1 Language property tests

The first two tests use properties of the languages $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$, in particular we use the fact that for each $\xi \in (2^{AP})^\omega$, $\xi \in \mathcal{L}_\varphi$ if and only if $\xi \notin \mathcal{L}_{\neg\varphi}$. Thus we can conclude that:

- The languages $\mathcal{L}_\varphi$ and $\mathcal{L}_{\neg\varphi}$ are disjoint, which means that $\mathcal{L}_\varphi \cap \mathcal{L}_{\neg\varphi} = \emptyset$
- Each $\xi \in (2^{AP})^\omega$ belongs to either $\mathcal{L}_\varphi$ or $\mathcal{L}_{\neg\varphi}$, and therefore $\mathcal{L}_\varphi \cup \mathcal{L}_{\neg\varphi} = (2^{AP})^\omega$

The first test is easy to realize in practice, we just need a way to generate random LTL formulas. For each formula φ we produce BA_φ and $BA_{\neg\varphi}$ using different translators and check that the intersection $(BA_\varphi \otimes BA_{\neg\varphi})$ is empty. If the intersection is non-empty we know that one (or both) translator did not produce the correct behavior.[2]

The second test is harder to utilize directly. The sequences involved are infinite (and infinitely many), thus we can not generate them all, and to algorithmically check for universality is PSPACE-complete (as shown by Vardi (1996)). The test can (as shown in Fig. 1) be reduced to emptiness checking, by *complementing* $BA_\varphi \cup BA_{\neg\varphi}$. However, the complementation automata of a (non-deterministic) Büchi automata may have an exponential blow-up in size. (Currently the upper bound is $(0.97n)^n$ states and the lower bound is $(0.76n)^n$ states, see Vardi (2007).) Therefore, instead of using complementation, we simplify this test into something which can easily be used in QuickCheck, as shown in Sect. 3. Since we are only testing for defects, a simplified test is of interest.

2.4.2 Model checking result tests

For a given model (Kripke structure) M and a given LTL formula φ the value of $M \models \varphi$ is well defined by the semantics of LTL. This means that we can use the result of the model checking problem to test the LTL-to-Büchi translations against each other. Using two different LTL-to-Büchi translators we can produce BA^1_φ and BA^2_φ. Then we translate the model M into another automaton BA_M. The test then compare the results of the emptiness checks for $BA^1_\varphi \otimes BA_M$ and $BA^2_\varphi \otimes BA_M$. The three tests are graphically described in Fig. 1.

3. Properties

In this section the QuickCheck specification for the testing of the LTL-to-Büchi translation is introduced. Most of the section is devoted to explaining how the tests from Sect. 2.4 are implemented in terms of QuickCheck.

3.1 LTL formula generator

Random LTL formulas are generated by the QuickCheck *generator* presented in Fig. 2. The generator is recursively defined. In the base case we pick an element from the alphabet, and in the recursive case we pick one of the seven different ways to produce an LTL formula from sub-formulas. Note, here we use the more common operators *always* ($\Box\varphi$) and *eventually* ($\Diamond\varphi$). These operators are defined in terms of the until-operator ($\Box\varphi \equiv \text{true}\ \mathbf{U}\ \varphi$ and $\Diamond\varphi \equiv \neg(\text{true}\mathbf{U}\neg\varphi)$). In the generator, the `LETSHRINK`-macro is used to be

[1] In the documentation, it is claimed that the reason for the crashes is an incompatibility between WRING and more recent versions of Perl

[2] We should also note that this test itself is not enough since a translator that produces the empty automaton for all LTL formulas would pass the test.

able to *shrink* generated formulas when a counter example is found. (Refer to Hughes (2007) for a thorough introduction to QuickCheck and its shrinking facility, and Arts et al. (2008) for an explanation of `LETSHRINK`.) The divisors (2 and 4) are rather arbitrarily chosen since they result in reasonably complex LTL formulas. An alphabet is a sorted non-empty list of propositional variables (chosen from `a-f`). The `SIZED`-macro is used to grab the implicit *size*-parameter in QuickCheck and pass it to the formula generator.

```
lprop() ->
    elements([{lprop,X}
              || X <- [a,b,c,d,e,f]]).

alpha() ->
  ?LET(Lst, [lprop() | list(lprop())],
           lists:usort(Lst)).

ltl_formula() ->
  ?SIZED(Size,
         ?LET(Alpha, alpha(),
              ltl_formula(Size, Alpha))).

ltl_formula(0, Alpha) ->
  elements(Alpha);
ltl_formula(Size,Alpha) ->
  Small = ltl_formula(Size div 2,Alpha),
  Smaller = ltl_formula(Size div 4,Alpha),
  oneof(
   [ltl_formula(0,Alpha),
    ?LETSHRINK([Phi,Psi],[Smaller,Smaller],
               {land,Phi,Psi}),
    ?LETSHRINK([Phi,Psi],[Smaller,Smaller],
               {lor,Phi,Psi}),
    ?LETSHRINK([Phi],[Small],{next,Phi}),
    ?LETSHRINK([Phi],[Small],{eventually,Phi}),
    ?LETSHRINK([Phi],[Small],{always,Phi}),
    ?LETSHRINK([Phi],[Small],{lnot,Phi})
   ]).
```

Figure 2. LTL formulas generator

3.2 Test 1

The first test is implemented in the property `prop_test1`, listed in Fig. 3. The property is parametrized by the LTL-to-Büchi translation functions to use `B1` and `B2`. The rest of the property is straightforward, we create the two Büchi automata, calculate the intersection and check for emptiness.

3.3 Test 2

The second test is implemented in the property `prop_test2`, listed in Fig. 4. This property is also parametrized by the LTL-to-Büchi translation functions to use `B1` and `B2`. Since it is not computationally feasible to check exactly what is presented we have simplified the test a bit. We have introduced a simple deterministic Kripke structure, which we denote a *witness*. A witness is in effect an infinite sequence of sets of labels produced by a *prefix* and a (non-empty) *loop*. The generator for witness is also presented in Fig. 4. The test checks that exactly one of the two automata BA_{φ} and $BA_{\neg\varphi}$ accepts the witness.

3.4 Test 3

The implemented property for the third test is very similar to the second test. Instead of generating BA_{φ} and $BA_{\neg\varphi}$, we generate BA^1_{φ} and BA^2_{φ} and check that their model checking results agree for different witnesses W. The third test property is listed in Fig. 5.

```
prop_test1(B1, B2) ->
    ?FORALL(Phi, (ltl_formula()),
      begin
        Bu1 = B1(Phi),
        Bu2 = B2(negate(Phi)),
        Bu1Bu2 = buchi:intersection(Bu1, Bu2),
        buchi:is_empty(Bu1Bu2)
      end).
```

Figure 3. Test 1 – QuickCheck property

```
witness(Alpha) ->
    #witness{alpha  = Alpha,
             prefix = list(lbl_set(Alpha)),
             loop   = [lbl_set(Alpha)
                       | list(lbl_set(Alpha))]}.

prop_test2(B1, B2) ->
  ?FORALL(Alpha, alpha(),
    ?FORALL(
      {Phi, W},
      {ltl_formula(Alpha), witness(Alpha)},
    begin
      Bu1 = B1(Phi),
      Bu2 = B2(negate(Phi)),
          is_witness(W, Bu1) =/=
            is_witness(W, Bu2)
    end)).
```

Figure 4. Test 2 – QuickCheck property

```
prop_test3(B1, B2) ->
  ?FORALL(Alpha, alpha(),
    ?FORALL(
       Phi, ltl_formula(Alpha),
       begin
         Bu1 = B1(Phi),
         Bu2 = B2(Phi),
         ?FORALL(W, witness(Alpha),
           is_witness(W, Bu1) ==
             is_witness(W, Bu2))
       end)).
```

Figure 5. Test 3 – QuickCheck property

3.5 Additional tests

The properties listed above cover all of the high level properties for the LTL-to-Büchi translation. In order to help during development we used many more, more fine grained, properties that only covered small parts of the translation. We also wrote some tests to check the `intersection` function used in the tests listed above. One example of a property for the intersection function is listed in Fig. 6. This test checks that for all Büchi automata `B1` and `B2`, if the intersection `B1B2` accepts a witness, then also `B1` as well as `B2` should accept the same witness.

When using this test, we ran into one of the standard problems with QuickCheck; namely that randomly generated data is not good enough. For a random (non-empty) Büchi automaton `B`, only about 2 out of 100 random witnesses are accepted by `B`. For the more complex `B1B2` in the intersection property, the numbers are even worse: only in about 1 out of 250 generated combinations of `B1B2` and `W`, was `W` accepted by `B1B2`.

The general solution to this problem is to use a better (more specific) data generator. In our case the problem is two-fold; (1) the intersection of two random non-empty Büchi automata is often

```
prop_intersection() ->
  ?FORALL(A, alpha(),
    ?FORALL({B1, B2}, {buchi(A), buchi(A)},
      begin
        B1B2 = buchi:intersection(B1, B2),
        ?FORALL(W, witness(A),
          (not is_witness(W, B1B2)) orelse
           (is_witness(W, B1) andalso
            is_witness(W, B2)))
      end)).
```

Figure 6. Intersection – QuickCheck property

```
prop_intersection() ->
  ?FORALL(A, alpha(),
    ?FORALL({B1, B2}, {buchi(A), buchi(A)},
      begin
        B1B2 = buchi:intersection(B1, B2),
        ?IMPLIES(
          not buchi:is_empty(B1B2),
          ?FORALL(W, witness_for_buchi(B1B2),
             (is_witness(W, B1) andalso
              is_witness(W, B2))))
      end)).
```

Figure 7. Intersection – optimized QuickCheck property

an automaton that do not accept any witnesses and (2) a random witness is not very likely to be accepted by a non-empty Büchi automaton. Therefore, we added a filter (using the `IMPLIES`-macro in QuickCheck) for empty automata and implemented a new generator `witness_for_buchi(B)`, that given a non-empty Büchi automaton `B` produces a witness that is accepted by `B`.

The trick is to search for an accepting cycle (picking one randomly if there are several) in the automaton, and use this together with a suitable prefix leading up to the cycle as a witness. For a non-empty automaton this generator, by construction, *always* gives a witness accepted by the automaton. This might not always be desired, but for the intersection property we are interested in the case when `W` is witness for `B1B2` and `witness_for_buchi` is the perfect generator to use. In Fig. 7 the optimized property is presented.

4. Implementation

After having all the QuickCheck properties available, as well as some already tested Büchi automaton manipulation functions (`intersection`, `is_empty`, etc), we were ready to implement the LTL-to-Büchi translator. Most of what is implemented has already been described elsewhere, we have looked for inspiration in several different places and combined many bits and pieces. Most well performing LTL-to-Büchi translator consist of the following three parts:

1. A rewrite engine, which aims to simplify the LTL formula. It normally uses a fixed set of (heuristically chosen) rewrite rules. One example is well documented by Somenzi and Bloem (2000).
2. Core translation algorithm – Construction of the Büchi automaton from the re-written LTL formula. There are two main algorithms for this phase: the tableau-based algorithms (for example described by (Gerth et al. 1996)), and algorithms based on *alternating automata* as introduced by Gastin and Oddoux (2001).
3. If needed, a translation of the result in phase 2, into a standard Büchi automaton. (Many translations works with intermediate automata formats, such as *generalized Büchi automata*, *alternating automata*, *transition-based Büchi automata*, etc.) Thereafter, reductions and optimizations, such as *simulation reductions* (see for example Etessami and Holzmann (2000)) and removal of *non-reachable* and *non-accepting* states, are applied to the Büchi automaton.

4.1 Rewrite LTL formula

Implementation of a rewrite functionality is fairly straightforward. You have to choose which rewrite rules to use and implement the application of a rewrite rule. We chose to use the rewrite rules described by Somenzi and Bloem (2000), these rules are the result of some thorough experiments and have also been used in (Giannakopoulou and Lerda 2002). The rules aim to simplify the LTL formula in a way that is (according to the heuristics) favorable in terms of the size of the resulting Büchi automaton. For example the LTL formula $(\mathbf{X}\,\varphi)\,\mathbf{U}\,(\mathbf{X}\,\psi)$ is re-written into $\mathbf{X}\,(\varphi\,\mathbf{U}\,\psi)$. Testing the rewrite facility using the properties described in Sect. 3 (by using one translator function with rewriting and one function without) quickly removed some (rather silly) implementation errors.

4.2 Core translation algorithm

As indicated above, there are two main alternatives for this phase: the tableau-based algorithms and algorithms based on alternating automata. We have chosen to use a tableau-based algorithm. We implemented an algorithm in the style described by Giannakopoulou and Lerda (2002). The main reason for choosing a tableau-based algorithm was non-technical, we simply were more familiar with this style of algorithm. We believe that a core translation based on alternating automata would have performed at about the same level.

The translation algorithm generates *transition-based generalized Büchi automata*, which carry labels on transitions instead of the normal state-based automata. The benefit of using a transition-based automata for the core translation is that more states can possibly be merged during translation. The result is a potentially smaller resulting automaton. The algorithm is tableau-based and works by expanding $\varphi\,\mathbf{U}\,\psi$ into $\psi \vee (\varphi \wedge \mathbf{X}\,(\varphi\,\mathbf{U}\,\psi))$. Step by step the automaton is build up, while keeping track of equivalent states as well as acceptance conditions.

By using the QuickCheck properties and the (small) counter examples it produced it was a rather painless process to get the core translation algorithm correctly (at least to the level of passing a very large number of tests) implemented.

4.3 Degeneralization

Since the core translation algorithm produces a (transition-based) *generalized* Büchi automaton, while McErlang only supports non-generalized automata, we needed to implement degeneralization. Again we follow what is described by Giannakopoulou and Lerda (2002), with some additions. A generalized automaton has (possibly) more than one set of accepting states, and an infinite sequence is accepting only if it passes through *all* accepting sets infinitely often. To convert a generalized automaton into a non-generalized automaton we need to translate the concept of visiting all states into a single accepting state set. The approach taken is to use a second specialized automaton called a *degeneralizer*. The degeneralizer has a size (and shape) that corresponds directly to the number of accepting sets in the generalized automaton. The degeneralizer 'counts' the visits of accepting sets, and all accepting cycles in the degeneralizer visit all accepting sets. To produce the final degeneralized automaton the *synchronous product* between the generalized automaton and the degeneralizer is computed. (For a thorough explanation of degeneralization, refer to (Gastin and Oddoux 2001).)

Although the size and shape are fixed, the order in which the accepting sets are counted can be varied. If the number of accepting

sets is n, there are $n!$ variations. Normally a heuristic, for example based on the sizes of the accepting sets, is used to select an order. Using some additional QuickCheck properties that compare the size of the result, we performed some experiments with different ordering heuristics. We tried to avoid having to calculate (and use) all possible degeneralizers, but unfortunately we could not find a heuristics that was consistently better than the random choice. However, since we are not worried about the performance of the translator, we decided to settle for a computationally more expensive solution. That is, we generate all possible automata and pick the best result after computing the synchronous product. This ensures that we get the smallest final automaton. We should note, that in many cases the reductions described in the next section actually produces the same final automaton regardless of the degeneralizer used. Finally, the justification for this more expensive solution is simply that more is hopefully gained in the model checking phase by having a smaller automaton, than what is spent in translation.

4.4 Reductions and optimizations

In automata theory there is a multitude of different reduction techniques and optimizations proposed. Some perform well on some structures, while others work best in completely different cases. We have chosen to implement some reduction algorithms that have proved useful for others, see for example (Etessami and Holzmann 2000; Giannakopoulou and Lerda 2002; Somenzi and Bloem 2000).

We have particularly opted for algorithms that reduce the size of the automaton, there are other algorithms that for example tries to make the automaton more deterministic (but also larger), see (Sebastiani and Tonetta 2003). Although a more deterministic automaton is sometimes preferable, we have chosen not to consider it in our implementation.

4.4.1 Simple reductions

We have implemented a couple of simple reductions:

- **Remove unnecessary transitions –** Unnecessary transitions are removed. For example two transitions from state X to state Y with the labels $a \wedge b$ and $a \wedge \neg b$ can be merged to one transition with the label a.
- **Remove non-reachable states –** States that cannot be reached from an initial state can be removed (together with their outgoing transitions).
- **Remove never accepting states –** States, from which it is impossible to reach an accepting state (or rather an accepting cycle) can be removed.
- **Reducing the number of accepting states –** Not technically a reduction, but it is favorable for the particular simulation reductions we have implemented to have few accepting states. Thus, accepting states that are not part of a cycle are removed from the set of accepting states.

It is worth pointing out that these reductions are usually not necessary for the initial result of the translation algorithm. However, after performing other reductions, also these simple reductions can be useful.

4.4.2 Bi-simulation reduction

Bi-simulation reduction is a standard reduction algorithm (see Kanellakis and Smolka (1983)). The reduction algorithm is based on a color-refinement partitioning of the states. The algorithm is adapted to the fact that transitions are labeled by conjunctions of propositional variables rather than just variables, we followed the algorithm presented by Etessami and Holzmann (2000).

Most problems with the implementation occurred due to the fact that most algorithm descriptions are rather imperative and thus took some time to convert into something not-so-ugly looking in Erlang. Again having the possibility to quickly find small counter examples when things went wrong helped a lot.

4.4.3 Strong fair simulation reduction

The most elaborate reduction algorithm we implemented was a *strong fair simulation* reduction. There are many different versions of fair simulation (see, e.g. Henzinger et al. (1997)). The version we implemented is described by Etessami and Holzmann (2000), it is rather similar to the bi-simulation reduction algorithm. However, more details are considered and a more fine grained ordering makes it possible to perform some more complicated reductions.

We had some difficulties implementing the strong fair simulation algorithm, mostly due to a misinterpretation of the algorithm description. (The *i-dominates* relation should be seen as a total order, if transition A i-dominates transition B then B **cannot also** i-dominate A. This is not clear from the definition.) While looking for this bug we were actually not helped by the QuickCheck tests, rather the opposite. Since errors occurred quite infrequently (and for rather complex automata), we were for a long time looking for a *less fundamental*(!!) error. Eventually we found the error, thanks to more basic debugging techniques, and could quickly verify that the algorithm implementation was working by running a large number of tests.

5. Results

Translator	Max size	Avg. size	Total size
erl_ltl2buchi	416	17.202	17202
erl_ltl2buchi+red	32	6.15	6150
erl_ltl2buchi+red+rew	31	6.005	6005
WRING	74	10.997	10997
LTL2BUCHI	31	6.066	6066

Table 1. Test results - 1000 random LTL formulas

To measure the performance in terms of size of the resulting automata we used a simple QuickCheck property, utilizing the built-in measure functions. An alternative would have been to use something readily available, like LBTT (Tauriainen 2001), but still, quite a bit of work would have been needed to write wrappers for the compared implementations. Therefore, since we had already used WRING and LTL2BUCHI as reference implementations during testing and thus had all the plumbing in place, we settled for the more lightweight QuickCheck measuring approach. The property listed in Fig. 8, is all that is needed to measure the performance for randomly generated LTL formulas. We test three different versions of our implementation: only the basic translation, translation + reductions and rewriting + translation + reductions.

Running the property for 1000 tests (1000 randomly generated LTL formulas) gives a result as presented in Table 1. The results vary slightly due to the random nature of the tests, but the numbers presented are representative. The table presents the maximal size of an individual translated automaton, as well as the average size of the automata.

We see that for random LTL formulas, we perform a lot (about 45%) better than WRING and also slightly (about 1%) better than *LTL2Buchi*. It is a result that we are satisfied with, since the alternative to writing a new implementation was to wrap *LTL2Buchi* and use it in McErlang. Nevertheless, we believe that there is still some room for improvement, and with the properties in place it should be easy to test new ideas for reduction algorithms in the future.

```
%% List of five different translations,
%% used in prop_compare_size.
translations() ->
    [{"erl_ltl2buchi",
      fun ltl2buchi:translate_basic/1},
     {"erl_ltl2buchi+red",
      fun ltl2buchi:translate_norew/1},
     {"erl_ltl2buchi+red+rew",
      fun ltl2buchi:translate/1}
     {"wring",
      fun wring_wrap:run/1},
     {"ltl2buchi",
      fun ltl2buchi_wrap:run/1}
    ].

prop_compare_size() ->
    ?FORALL(Phi, (ltl_formula()),
      %% Filter formulas crashing Wring
      ?IMPLIES(wring_ok(Phi),
        begin
          Trs = [{N, F(Phi)} || {N, F} <-
                translations()],
          nested_measure(Trs, true)
        end)).
```

Figure 8. Performance measurement property

For two reasons we have not measured the speed of the translators; (1) it is impossible to fairly compare a native implementation with wrapped implementations called externally, and (2) the model checking connection for the translator makes size much more important than speed.

6. Summary

Using a property driven development for the implementation of an LTL-to-Büchi translator in Erlang turned out to be a nice experience. We had a fun time, and the implementation quickly stabilized into a mature translator. We believe that by first formulating the properties, we saved time both in testing and debugging and also in having a clearer picture of what to implement. Also, the shrinking facilities of QuickCheck meant that we usually got a fairly simple LTL formula for which the translator produced an incorrect automaton, which in the end probably reduced the time spent debugging the translator.

However, we should also note that we were helped (quite a lot) by the implementation task being very well defined. This also meant that the properties were not too hard to formulate. So, for situations where a clearly defined and well described algorithm should be implemented, this style of development is especially well suited. Having the paper about testing LTL-to-Büchi translations by Tauriainen and Heljanko (2002) was also very helpful.

In the end we managed to implement an LTL-to-Büchi translator that performs at least as well as our reference implementations WRING and LTL2BUCHI.

To further improve the QuickCheck experience it would be good to directly generate and *effectively* shrink Büchi automata. Especially when working with reduction algorithms it would be nice to not get the smallest LTL formula that translates into an automaton where the reduction fails, but rather the smallest automaton. We did implement a generator for Büchi automata, but shrinking them in an intelligent way was deemed a too complicated task and we did not have time to investigate this further.

Finally we should also comment on our focus to trade a smaller resulting automaton for a longer translation time. The run time of the LTL-to-Büchi translator is usually some couple of hundred *milliseconds*, while a model checking run could easily spend some hundred *seconds*. Thus spending a factor ten longer time in generation for gaining 10% of the model checking time is still a good trade-off.

Acknowledgments

This research was sponsored by EU FP7 Collaborative project *ProTest*, grant number 215868. Many thanks to the McErlang team (Lars-Åke Fredlund and Clara Benac Earle) for their efforts in including the translator in the McErlang release. Also thanks to Koen Claessen for helpful comments on testing and optimization of Büchi automata.

References

Thomas Arts, Laura M. Castro, and John Hughes. Testing Erlang data types with Quviq QuickCheck. In *ERLANG '08: Proc. of the 7th ACM SIGPLAN workshop on ERLANG*, pages 1–8, New York, NY, USA, 2008. ACM.

Kent Beck. *Test-driven development : by example*. Addison-Wesley, Boston, MA, 2003.

J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci.*, 1960.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

C. Benac Earle, L-Å. Fredlund, J. A. Iglesias, and A. Ledezma. Verifying Robocup teams. In *Proc. 5th International Workshop, Mochart 2008*, pages 34–48. Springer, 2008.

K. Etessami and G. Holzmann. Optimizing Büchi automata. In *CONCUR '00: Proc. of the 11th International Conference on Concurrency Theory*, pages 153–167, London, UK, 2000. Springer-Verlag.

L-Å. Fredlund and J.J. Sánchez Penas. Model checking a VoD server using McErlang. In *In proceedings of the 2007 Eurocast conference*, Feb 2007.

L-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2007.

P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV '01: Proc. of the 13th International Conference on Computer Aided Verification*, pages 53–65, London, UK, 2001. Springer-Verlag.

R. Gerth, D. A. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd.

D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *FORTE '02: Proc. of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, pages 308–326, London, UK, 2002. Springer-Verlag.

T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. In *Information and Computation*, pages 273–287. Springer-Verlag, 1997.

J. Hughes. QuickCheck testing for fun and profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *LNCS*, pages 1–32. Springer-Verlag, Berlin Heidelberg, 2007.

P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *PODC '83: Proc. of the second annual ACM symposium on Principles of distributed computing*, pages 228–240, New York, NY, USA, 1983. ACM.

McErlang – https://babel.ls.fi.upm.es/trac/McErlang/. (Web page, 2009).

A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Nov 1977.

ProTest Project – http://www.protest-project.eu. (Web page, 2009).

J-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

R. Sebastiani and S. Tonetta. "more deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *12th IFIP WG 10.5 Advanced Research Working Conference, CHARME*. Springer-Verlag, 2003.

F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV '00: Proc. of the 12th International Conference on Computer Aided Verification*, pages 248–263, London, UK, 2000. Springer-Verlag.

H. Svensson and L-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Erlang '07: Proc. of the 2007 SIGPLAN workshop on Erlang Workshop*, pages 43–54, New York, NY, USA, 2007. ACM.

H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. *STTT*, 4(1):57–70, 2002.

Heikki Tauriainen. Ibtt 1.0.0 – an LTL-to-Büchi translator testbench. Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2001. Software.

M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, pages 238–266, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

Moshe Y. Vardi. The Büchi complementation saga. In *STACS 2007*, volume 4393 of *Lecture Notes in Computer Science*, pages 12–22. Springer Berlin / Heidelberg, 2007.

M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 332–344, New York, 1986. IEEE Computer.

Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 185–194, Nov. 1983.

Model Based Testing of Data Constraints

Testing the Business Logic of a Mnesia Application with Quviq QuickCheck

Nicolae Paladi
IT University of Gothenburg, Sweden
paladi@ituniv.se

Thomas Arts
IT University of Gothenburg and Quviq AB, Sweden
thomas.arts@ituniv.se

Abstract

Correct implementation of data constraints, such as referential integrity constraints and business rules is an essential precondition for data consistency. Though most modern commercial DBMSs support data constraints, the latter are often implemented in the business logic of the applications. This is especially true for non relational DBMS like Mnesia, which do not provide constraints enforcement mechanisms. This case study examines a database application which uses Mnesia as data storage in order to determine, express and test data constraints with Quviq QuickCheck, adopting a model-based testing approach. Some of the important stages of the study described in the article are: reverse engineering of the database, analysis of the obtained database structure diagrams and extraction of data constraint, validation of constraints, formulating the test specifications and finally running the generated test suits. As a result of running the test suits randomly generated by QuickCheck, we have detected several violations of the identified and validated business rules. We have found that the applied methodology is suitable for applications using non relational, unnormalized databases. It is important to note the methodology applied within the case study is not bound to a specific application or DBMS, and can be applied to other database applications.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging;

General Terms Verification

Keywords Business rules, model based testing, formal specifications, Erlang, Mnesia, QuickCheck.

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.

1. Introduction

Referential integrity is the database-related practice of ensuring that implied relationships between tables are enforced. Most modern Database Management Systems (DBMSs), especially *relational* DBMSs have built in mechanisms for defining and ensuring basic data constraints [24]. However, in practice far from all constraints are defined in the database management system itself, but many are rather encoded in the application using the data. For example, an application supporting an internet shop would impose a relation between a customer willing to make a purchase and the credit balance of that customer. In a purely relational database, one could hard code that the credit must at least be the purchase amount, but this is hardly ever done, since this constraint is based on a business strategy that may well change or is different for different customers.

Many database applications have a layered architecture, part of the data constraints are hard coded as relational constraints in the DBMS, other constraints are implemented in the business logic of the application. There are several reasons for implementing constraints in the business logic rather than in the DBMS. For example, the above mentioned situation in which one wants to get flexibility in the relationship; either in the future or for special subset of the customers. Other reasons may be purely social, such as lack of developer time or required expertise and insight, or strictly technical. An excellent example of the latter case is Mnesia [26], a distributed DBMS, appropriate for telecommunications applications and other Erlang [2] applications which requires continuous operation and exhibit soft real-time properties. According to Mattsson et al, [26], Mnesia employs an *extended relational model*, which results in the ability to store arbitrary Erlang terms in the attribute fields. However, Mnesia is not a relational database and does not have any mechanisms for ensuring database constraints other than ensuring them in the business logic of the application.

Problem: ensuring the constraints

When data constraints cannot be ensured by a DBMS alone, then those constraints are much less visible in the software

design. As a consequence, constraints are often implicitly defined and certain parts of an application may accidentally violate a constraint. Constraint violation may result the database to be in an inconsistent state and software assuming certain properties of the data may crash. In addition, violation of constraints may impact the business, since it may be possible to perform actions that the business disallows.

When software needs to be reliable, the constraints implemented in the business logic need to be satisfied and therefore it should be tested that they cannot be violated by executing the application.

Defining and enforcing database constraints within relational (and to a lesser extent object oriented) databases has long been in the focus of both academia and industry. The SQL implementation [17] of database constraints is currently supported by most relational DBMSs. A simple example is that the SQL standard for ensuring referential integrity (which is a typical example of a database constraint) is supported by DBMS like MySQL, Microsoft SQL Server, DB2, Oracle and even MS Access (through its graphical relational tool) [9]. However, this does only solve simple relations and not the more dynamic relations captured by the business rules.

There has been little research done on the topic of ensuring business rule constraints, especially when we consider databases which do not use SQL. Furthermore, previous research assume the database to be in at least the 3rd normal form [22] or higher, and do not consider the case of unnormalized databases [14], [12], [15]. Nevertheless, non-relational unnormalized databases are rapidly becoming more popular.

The case: a large Mnesia application

Recently, Castro and Arts [4] have developed a method for testing business logic constraints with Quviq QuickCheck. They used their method to verify business logic constraints in an application on top of a normalized relational database. In this paper we show that the method is also applicable to unnormalized, non-relational databases and that we are able to identify business logic violations in existing applications.

Kreditor AB is a Stockholm-based financial organization which has developed for its operations a database application implemented in Erlang using Mnesia for data storage. The application, further referred to as the *Kred application*, uses an unnormalized database supported by a non relational database system. ***In this paper we show that the method of Arts and Castro is applicable to Kreditor's database application. We show that we can identify violations of the constraints. Therefore, it will help improve the existing solutions for testing data constraints for non relational database systems, and minimize the occurrence of situations when invalid data input can lead to data corruption.***

As part of this study, we have reverse engineered the Kred application to create its database schema and the corresponding entity-relationship (ER) diagram. Besides, we have identified a number of data constraints that are implemented in the business logic of the application.

The presented approach is generally applicable to non-relational databases, but in particular to Erlang applications build upon Mnesia.

2. Related Research

We base our case study on the method developed by Castro and Arts [4], which is a general methodology for testing data consistency of database applications. In this approach, the system under test is modeled as a state machine, the state of which is examined after consecutive calls to database interface functions. In the method, the focus lies on keeping the state as simple as possible and not making the state a copy of the database; only data generated by the interface functions (such as unique keys, etc), should be stored in the state, the rest is assumed to be correctly stored. The state machine model is tested against the real application with QuickCheck, (cf. [5]). The novelty of the method of Castro and Arts is that business rules are formulated as data invariants and are checked after each test.

The method is applied to a normalized, relational database and invariants are described and executed as SQL queries.

2.1 Other approaches

Chan and Cheung support the idea that current "traditional" approaches in software testing cannot reveal many of the software errors which can lead to database corruption. Therefore, they suggest the idea of extending the white box testing approach with the inclusion of SQL statements that are embedded into the database application. In order to do that, they suggest to convert the SQL statements to the general programming language in which the application is implemented and include them into the white box testing framework [10].

In addition, Chan *et al* propose to integrate SQL statements and the conceptual data models of an application for fault-based testing. In their paper, they propose a set of mutation operators based on the standard types of constraints used in the enhanced entity-relationship model. The operators are semantic in nature and guide the construction of affected attributes and join conditions of mutated SQL statements [11].

Chays *et al* have developed a framework for testing relational database applications called AGENDA. AGENDA has a strong reliance on the relational model and SQL and its use has not been described for non-relational databases [12].

Dietrich and Paschke describe a test-driven approach to the development and validation of business rules [16]. They propose a way to develop JUnit test cases based on formal rules, however they propose a manual implementation of the test cases.

As a complement to the above method, Kuliamin's description of the UniTestK test development technology [25] contains some practical advice on using models to test large complex systems. In particular, the author describes the use of well known software engineering concepts such as modularization, abstraction and separation of concerns in order to manage the stages of determining the interface functions, development of the model, and finally the development of the test scenario.

3. Research Approach

The project has been carried out with an emphasis on quantitative post positivist approach, focused on a combination of qualitative in-depth analysis of the database application under examination, and empirical observation of the results of an extensive set of randomly generates test instances.

In the light of Boudreau's claim that "Field experiments involve the experimental manipulation of one or more variables within a naturally occurring system and subsequent measurement of the impact of the manipulation on one or more dependent variables" [8], this study heavily relies on field experiments which will focus on studying the change of the variables in the Kred application as a response to certain alterations of the database. Furthermore, during the study we have not only observed and measured the occurrence of changes, but also compared them with the *expected* alterations. Based on the outcome of the latter comparison, we have been able to draw conclusions on whether the business logic of the system conforms to the requirement of maintaining the data in a consistent state. Other methods used include interviews [23], data analysis, and heuristic estimations of the functionality limits of the system under examination for test design purposes [27].

In the course of the project we had to answer several questions concerned with database representation, identification of database constraints, as well as their codification. This section will describe the tools used, as well as the steps taken to conduct this study.

3.1 Limitations of the study

This study focused on database applications implemented in Erlang and which use Mnesia as data storage. Despite the fact that both Erlang and Mnesia have highly concurrent and distributed properties, such aspects have not been taken into consideration in the current study.

3.2 Tools

3.2.1 Test generation tools

In order to fully leverage the power of the formal verification approach adopted for testing the business logic, we choose QuickCheck to generate and execute the tests. Quviq QuickCheck is a specification based testing tool [5] which tests the software with randomly generated test cases, which follow a formal specification expressed in Erlang. QuickCheck has several libraries for expressing higher level system specifications like the state machine library that we used.

There are several other test generation tools available, which are listed below [1]:

- TVEDA, a tool developed by France Telecom CNET [28], which generates tests against formal specifications written in TTCN, which is an ISO test suit notation standard [31], [34]. This tool is used by France Telecom, mainly for testing telecommunication protocols.
- TorX is an *on-the-fly* testing tool. i.e. which offers support for test generation and test execution in an integrated manner. It generates tests against specifications expressed in PROMELA and LOTOS [30].
- Blom and Jonsson describe a case study of automatic test generation for a telecom application implemented in Erlang. In their detailed paper, the authors also describe the test generation algorithm, as well as a formal specification language, Erlang-EFSM [7]. However, it is not fully developed and has not moved further from the concept state described in the article.

QuickCheck's support of Erlang and library for state machines, together with a larger number of previous case studies, has made it the preferred tool for our research project. However, it is important to note that the method followed in our case study is generalizable, and it is not strictly bound to either QuickCheck or the Kred application.

3.2.2 Structure visualization

As a consequence of the compelling lack of suitable database reverse engineering tools as well as of database structure visualization tools that could be used for Mnesia, Dia has been used to visualize the database structure, both the ER diagram and the Database schema. Dia is a lightweight open source tool that has been chosen particularly for its relatively extensive capabilities [35]. While Dia may not be a specialized database visualization tool, its capabilities allow to plot the structure of databases as complex as the database used by the Kred application.

3.3 Examination of the database structure

One of the first steps in our work has been the manual examination of the database structure. Reverse engineering of relational databases has been in the focus of research, and several approaches are available.

To mention a few, Premerlany and Blaha offer a generic approach to reverse-engineering legacy relational databases [29]. In their paper the authors describe a manual process of analysis, deconstruction and visualization of the database model, which consists of seven steps. Premerlany and Blaha support the idea that reverse engineering of legacy databases

[1] Far from being a complete list, this is an example selection of test generation and execution tools

should be carried out in a flexible, interactive approach. The authors also claim that an approach based on rigid, batch-oriented compilers will most likely fail [29].

Similarly, Andersson describes the process of Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering [1]. In his approach, Andresson also use an ER model extended with multi-valued and complex data, as well as multi-instantiation. This makes the latter approach suitable for reverse engineering of the database under assessment.

The method described by Premerlany and Blaha has numerous similarities with the current study, and in particular the focus on large unnormalized databases, the consideration of the lack of enforcement for foreign keys, as well as consideration of the "optimized or flawed schemas which are often found in practice" , [19]. Furthermore, this approach is suggested by the authors as suitable for large legacy databases with little or no semantic advice available [29]. However, this method is not entirely applicable, mainly due to its focus on *relational* database systems, as well as due to the large effort required to reverse engineer the database structure, especially since reverse engineering of Mnesia databases is *not* a central issue for this project.

Following the above idea, reverse engineering of the Kred database has been performed in an iterative process consisting of the steps described below.

- *Determine Candidate Keys*, a step focusing on identifying the primary keys of the tables. In case of Mnesia, this is facilitated by the peculiarities of record definition, where the *first* element of the record serves as a key to the record.
- *Determining Foreign-key Groups* by observation of the tables' elements, search for synonyms, homonyms, and fields with the same name.
- *Discovering associations* by revealing additional links of all types between the tables, with the help of code comments and other semantic information. However as Premerlany and Blaha state, "one should be careful at this stage, since reverse engineering produces hypothesis, which must nevertheless be validated with the help of semantic understanding".
- *Performing the transformation* by transferring of the discovered information, based on decisions on the exact representation of certain components. For example, in many cases a one-to-one relationship implies that the element can be represented as an attribute (even perhaps a complex attribute) rather than entity. Furthermore, N-ary associations should be decomposed into binary (rarely ternary [29]) relationships, for a more realistic visualization of the database structure. Other similar steps must be considered as well.

An additional consideration to be added to the process is the earlier mentioned ability of Mnesia to store Erlang terms of arbitrary complexity in the attribute fields, for example a record that in itself would candidate for being an entity.

The goal of the described method is to obtain a visual representation of the database schema, the corresponding ER diagram of the most important components of the database, as well as getting acquainted with the overall structure and functioning of the database. The iterative approach allows to gradually add entities, based on their relevance to the identification of data constraints.

3.4 Identification of data constraints

In order to test the business logic, we need to find the data constraints. However, it is often the case that data constraints are not explicitly documented, and identifying them is not a trivial task. There have been several case studies focusing on the extraction of business rules from COBOL programs [21] and applications using object oriented databases [6]. Unfortunately these efforts resulted in very narrow automated solutions, suitable for the specific purpose of the respective studies. Therefore, in the current project we approach the identification of database constraints from two directions: an analysis of the database reverse engineered into a visual structure and individual in-depth interviews with several of the developers of the Kred application.

The analysis of the visual representation of the database structure focused on the key elements of the database structure, such as the schema tables and primary keys, entities within the ER diagram as well as the relationships between them. We used Chen's notation for entity-relationship modeling [13], particularly since in this notation *relationships* are first class objects and can thus have attributes of their own. The latter is important for modeling Mnesia databases, where the relationship between two tables can be expressed in a table containing several additional elements. We express such elements as attributes of the relationship in the entity-relationship diagram. The first step in identifying constraints is to note the primary and foreign keys of the entities in order to establish the relations between the entities. For example, if two entities have a relationship between them, and share a set of foreign keys (which are primary keys in other tables), we expect the values of the foreign keys to be equal in all cases. Multiplicity will not be a deciding factor in this process, since it cannot be precisely determined without additional semantic information. This approach will help identify a part of the referential integrity constraints within the application.

The identified constraints have been validated during a presentation to the system developers. Furthermore, interviews with the developers have identified additional semantic information determining domain specific data constraints. The goal has been to identify an initial set of constraints that were recognized by the developers, rather than identifying all of constraints, which would require a lot of effort for a non-trivial large scale system. The initial set has been used to

perform testing and to evaluate whether our approach could find inconsistencies in the data.

The ER diagram, together with the database schema produced as a result of the above mentioned reverse engineering of the database should yield enough information to determine part of the constraints. We have identified the following two categories of data constraints: referential integrity constraints, and domain specific data constraints.

3.4.1 Referential integrity constraints

As mentioned above, Mnesia does not provide support for referential integrity constraints. Therefore, referential integrity should be embedded in the business logic implementation. Referential integrity checks are easiest to discover, by examining the ER and the schema representation of the database as previously described. An example of a referential constraint identified in the current project. It is expressed as an SQL query, and should return NULL in case the constraint is satisfied. This particular example describes the relation between the tables **ptrans** and **pbal**:

```
SELECT 'ptrans'.'ano'
FROM ptrans, pbal
WHERE
  (('ptrans'.'pbal_key' ='pbal'.'key')
  AND NOT
   ('ptrans'.'invno' ='pbal'.'invno' ))
```

This example constraint ensures that the **pbal** events (i.e. events that influence the payments balance of the account) and the **ptrans** events (i.e. events that are related to a personal account but do not influence the payment balance of the account) refer to the same invoice number, in case the **ptrans** table contains the key of the pbal event. Since there is a simple direct relationship between the two tables, it is likely that the constraint has been identified and checked by the developers, hence the probability of revealing an inconsistency error is quite low.

3.4.2 Domain specific data constraints

Business rules are domain- and business-specific constraints which are expected to be expressed in the business logic. Identification of domain specific data constraints is difficult, especially in the situation when semantic information about the system is not available. This task requires a combination of the above mentioned analysis of the schema representation and ER model of the database, code analysis and finally interviews with the developers familiar with the system. Code analysis includes tracing the events generated by the execution of the interface functions, examination of event logs and static code review. Below follows an example constraint, which is similarly to the previous one expressed in SQL and should return NULL in case the constraint holds.

```
SELECT 'pbal'.'key'
FROM 'pbal', 'invoice'
WHERE
  (('invoice'.'invno'  = 'pbal.invno')
AND
  ('invoice'.flags' = ?FI_IS_PACC))
```

This slightly more complex domain specific business rule ensures that the invoices that have their 'pstatus' flag set to "?FI_IS_PACC", which means that they belong to a personal account and therefore should have at least one payment balance (pbal) entry. Certainly such a constraint should not be incorporated into the implementation of the DBMS and is therefore left to the business logic implementation.

3.5 Testing data constraints

Before actually testing the identified data constraints, they have first been validated by the developers familiar with the system. This is needed in order to avoid errors in formulating constraints as a result of the lack of familiarity with the system. The earlier mentioned methodology of Castro and Arts is used to test the data constraints. Below are the stages of the methodology, adapted to the specifics of the project. A more thorough description can be found in [4].

Since the system under test uses Mnesia as data storage, the identified data constraints will have to be converted to Query List Comprehensions (QLC), which is Mnesia's query language. This query is written as an invariant to validate that the business rules hold before and after test execution. For example, the business rule presented above is expressed using QLC as follows:

```
invariant_pbal() ->
   QH = (qlc:q([Pb#pbal.key ||
                Pb <- mnesia:table(pbal),
                Inv <- mnesia:table(invoice),
                Pb#pbal.invno == Inv#invoice.invno,
                Inv#invoice.flags == ?FI_IS_PACC])),
   {atomic, Response} =
       mnesia:transaction(fun() -> qlc:e(QH) end),
   Response /= [].
```

The state of the database will be checked against the invariant both at the start and end of the test case, thus ensuring that the business rule is respected. A very important aspect at this point is the correct design of the test cases that will be run. A few test cases selected by the developers will be insufficient, because of the developers assuming system constraints that may not hold. Instead, a large number of test cases, that are valid operations but are extremely unlikely to ever happen during system operation, has to be generated.

The next step will be to identify the available interface functions to modify the states of the system. Depending on the architecture and the implementation of the system under test, this stage can be very time consuming. In examining the choice of the interface functions it is important to note their position relative to the implementation of the business logic.

If the interface functions are determined, generators are written for sequences of interface calls to the system. The generators will produce only the minimal set of data which is

needed for the interface calls, in order to produce valid state transitions. At the same time, the generated data sets should be as varying as possible, in order to explore any potential non standard behavior of the application.

We present a few of the generators we developed to show what they look like and how similar they are to Erlang functions. The following generator would generate lists of items that can be used as an argument to an interface function. The generator shown below will produce sequences of varying length containing fairly different item sets. The generators for *artno*, *vat* and *discount* will produce small natural numbers:

```
list(#item{artno = nat(),
           description = list(char()),
           vat = choose(nat),
           flags = 0,
           discount = nat(),
```

The generator for *price* will produce large numbers with two decimals. Finally, the generator for *quantity* will produce either very large values, or small values for the quantity parameter. This will produce values at the boundaries rather than obtaining a normal distribution of number, as the use of `choose/1` would yield.

```
price = ?LET({H,F},{nat(),  choose(0,99)},
             (H*100)+ F\ 100),
quantity = ?LET({N,T,I},{nat(),choose(0,1),largeint()},
                N + T*abs(I))})
```

It can be argued that the values of these sequences do not affect the business logic, and are artificial, hard coded sequence of goods would suffice. However, this depends on the implementation of the business rules and the price paid for random generation is extremely low.

For the selected interface functions, a local function is written in order to validate that the response from the interface function corresponds to the expected result. For example, the interface function via the `estore_server` module that is used to activate a reservation gets a local variant as follows:

```
activate_reservation(Reservation, Items, Pno) ->
  Result =
     estore_server:handler(
        'undefined',
        {call, activate_reservation,
        [Reservation, Items, Pno, (...)]}),
  Person = person:read_d(Pno),
  Blacklisted =  (Person#person.blacklisted == 1),
  case Result of
       {false,{response,[{array, ["no_risk",Invno]}]}}
           when not Blacklisted -> Result;
       {false,{response,{fault, -4,"blocked"}}}
           when Blacklisted -> Result;
       _ -> exit(unexpected value)
  end.
```

First the function is called and the result is stored, after that, the result is compared to the expected outcome.

After having added all the interface functions, QuickCheck will create test cases by running sequences of generated interface calls. The results will be validated through the expected values, and finally the invariant will be checked. A situation in which the invariant evaluates to 'false' would mean that the business rule has been invalidated, and the database is in an inconsistent state. The available test sequence will make it possible to observe the exact actions that have invalidated the data constraints. Furthermore, QuickCheck will automatically shrink the test sequence to a minimal failing case in order to show the exact cause of the error.

3.6 Analysis of the test results

The results collected by running the tests developed according to the above described methodology will be used to evaluate the way the business logic *actually* enforces the data constraints in contrast to the *expected* enforcement of data constraints. Furthermore, the data will be used to verify whether the approach is fully applicable to database applications which use non-relational unnormalized databases.

4. Results

4.1 Reverse engineering

One of the obtained results is the reverse engineered ER diagram and a raw representation of the schema of the database. The obtained ER diagram is a useful artifact for Kreditor, and together with the initial set of defined and formalized constraints will contribute to the current system documentation. At the same time, it is an essential document for our test approach, since we extract constraints from this ER diagram that we use to test against.

We used the data in the schema files, the table descriptions to obtain our first rough estimate of the ER diagram. A schema is a set of record definitions, each record has a name, the *table name*, and a number of fields, corresponding to the *table fields*. We initially assumed each record to correspond to an entity and the fields to attributes. After that we assume equal field names (attributes) to symbolize relations. That is, if a entity **pbal** has an attribute *invno* and the entity **ptrans** also has an attribute *invno*, then we assume that these entities are related and the attributes are replaced by a relation symbol. The kind of relation is unknown, it can be one-to-one, many-to-one, or something else, but that is impossible to infer from the schema file.

In the second iteration of the reverse engineering process, 18 of the entities were transformed into relationships. This was done in order to both reduce the complexity of the ER model, as well as bring the ER model close to the actual structure of the database. For example, the entity *personal_email* was converted to a relationship between

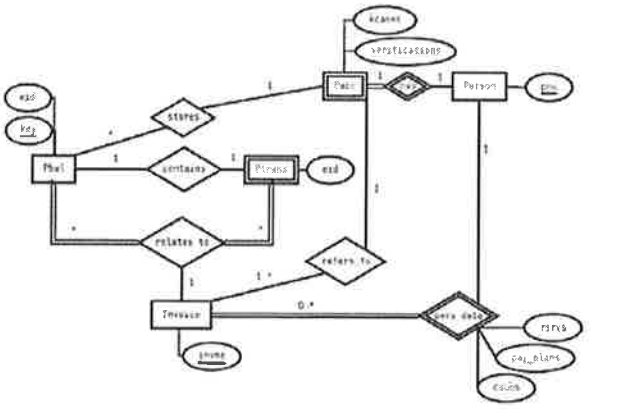

Figure 1. Fragment of the ER model of the database

estore and **person**. The other elements contained in the *personal_email* were noted as the attributes of this relationship.

The primary key of each table is assumed to be the first field of the record definition, since that's the standard in Mnesia. In this way we visualized the ER diagram using 43 of the 87 tables that the developers considered as most relevant.

Of course, we identified entities that had more than one attribute in common with each other. For example, the entity **pbal** and **ptrans** have 2 attributes in common: *invno* and *key*. Since *key* also occurred in a third entity, viz. **pacc**, we created two relations between the entities, as depicted in Figure 1.

There is, of course, a risk that certain attributes have the same name, but do not identify a relationship. Similarly, it may be the case that there is a relationship between fields that have different names. In our case for example the attributes *invno* and *ocr* expressed a relation, where *ocr* is a non-standard name for the invoice number. In addition, it is totally unclear what kind of relation the attributes symbolize. Therefore, we consulted the domain experts to look at the ER diagram and provide us with feedback.

This revealed a number of unclarities in the author's model of the database, for example the already mentioned relation hidden behind *invno* and *ocr*, but also more sophisticated issues. For example, the *Pno* which is used as an alias for *Personal Number* throughout the database implementation, can be used to denote both the Personal Number for physical persons, as well the Organization number for legal entities. Therefore, the relation between two entities is context dependent and a zero-to-many relationship.

After consulting the developers, the resulting ER diagram contained 23 entities and 36 relations and a total of 250 attributes. Obviously, attempting to discover all data constraints that can be found in the ER diagram would be a daunting task, therefore we only selected a subset of possible constraints for the five weeks we had left for our case study.

4.2 Constraints Identified

For the purpose of the project, 24 constraints have been identified and recorded for further testing. The constraints were initially expressed in SQL in order to be validated during individual interviews with developers. Unexpectedly, of the 24 identified constraints only 16 have been considered as valid, while the other 8 were considered as either irrelevant, or not true for the system.

The reason for such large number of invalid constraints can be explained either by the misinterpretations and miscellaneous errors in the process of reverse engineering the database, or by the often stated *lack of familiarity with the system*. However, the identification of invalid, or false constraints should not be considered as a waste of time. This is simply because formulating and discussing these *incorrect* constraints made it possible to both learn that not all relations are relevant at some point in the business process, despite their apparent semantic similarities.

In any case, even a reduced set of constraints is important, since no other constraints of the Kred application have been recorded earlier. The above mentioned constraint

```
SELECT `ptrans`.`ano`
FROM ptrans, pbal
WHERE
 ((`ptrans`.`pbal_key`=`pbal`.`key`)
  AND NOT
  (`ptrans`.`invno`=`pbal`.`invno` ))
```

was remarked as particularly relevant, since there have been situations in the past when this constraint was not respected.

4.2.1 Failing constraints

We used QuickCheck to generate random sequences of calls to the interface functions, or in other words, have users of the system "go wild on it. After each such sequence we validated the identified constraints. Surprising enough, we detected that two of them could be violated.

Contrary to the earlier expectations that the referential constraints are most likely to hold (in contrast with the domain specific business logic, the errors in which are more difficult to spot), the constraint provided earlier as an example, did not pass the test (the output details are ignored):

QuickCheck has found a counterexample when **ptrans.invno** is *not* equal to **pbal.invno**, when pbal.pno is equal with ptrans.pno. This data constraint has been detected through the analysis of the ER diagram and later confirmed by several developers as correct. However, in some *apparently* rare cases this referential integrity constraint does not hold. QuickCheck's shrinking technique made analysis of this rare case an easy task.

A second failing constraint that has been discovered was a "domain specific data constraint" (according to the above classification). It will not be described further, however its discovery demonstrates that the applied methodology allows us to discover both failing referential integrity data constraints, and domain specific business rules.

4.3 Constraints testing results

4.3.1 Validation of the test specifications

When constraints are violated by a sequence of interface calls, one needs to ensure oneself that indeed the constraint should hold and the sequence of calls introduces an error. When recognizing this, the error can be fixed and the same sequence can be executed again, now not resulting in a violation.

However, what do we know if a constraint is not violated? Probably we simply formulated a database query that is always satisfied and does not really describe the constraint we wanted to validate. For each constraint we also wanted to check that this constraint expressed what we intended. This is problem similar to ensuring that your test suite is correct. Several methods to do so exist.

- Mutation testing, which involves changing the source code of the system under test [36], [33].
- Fault injection, a method involving altering the source code to test code paths that might not be visited [3].
- The use and probation of various test design approaches: using classification trees [18], the Z method [20], or even a combination of the two [32].
- Deliberately alter a constraint so that it *must* fail, and test that it actually fails during test execution.
- Introduce a change in the implementation of the system – the change should produce a controlled fault that would invalidate a constraint (that otherwise holds) during test execution.

We believe that combining all (or several) of the described methods would yield the highest certainty that the test specifications are correct. However, this section will describe the application of the second method, and namely the deliberate introduction of a software fault that would invalidate a certain constraint during test execution.

In order to apply this approach, the following constraint has been chosen:

```
SELECT `invoice`.`pno`
FROM `invoice`, `estore_data`
WHERE
 (`invoice`.`eid` = `estore_data`.`eid`
AND NOT
(`invoice`.`pno` in `estore_data`.`customers`))
```

This constraint ensures that whenever a new customer makes a purchase in the estore, they are added to the list of customers in the estore_data table of the corresponding estore.

This example has been chosen both for its simplicity (which becomes highly valuable in an unknown and complex system) and for the relatively low effort of adding a fault that would violate the constraint. To produce this fault, the code is altered to that the customers list of estore_data is emptied each time a new invoice is added. Though it might seem rather raw, the constraint is guaranteed to fail once there are some invoices added.

Once the tests are run, QuickCheck quickly spot an example sequence in which this property is violated. It might be worth noting that despite the apparent triviality of the described bug, other existing test suites did not discover it.

5. Discussion

The results of the project show that overall, the method described in the paper of Castro and Arts [4] and applied to the present case is easily extendable and applicable to application which use databases such as Mnesia. The quality and time efficiency of applying this methodology depends significantly on the level of documentation of the system, the application's complexity and the clarity of the application's implementation. The approach produces several positive outcomes, namely the updated ER model of the database, and a schema representation of some of the tables. The most important outcome however, is the set of specifications and formalized constraints that is available once this method has been applied. Such a test framework can be used (and continuously updated) later to ensure that the data constraints are always ensured when new functionality and components are being developed.

As mentioned above, there are two main steps in the methodology (performed iteratively), namely *identification of the constraints*, and *development of the test specifications*. There are several factors that influence the outcome of the test procedure using the above described method.

5.1 Available Documented Constraints

First of all, the availability of documented constraints would significantly facilitate the testing process. However, the fact that there is a set of documented constraints does not imply that one need not search for additional constraints. Considering that documentation can be outdated or incomplete, the constraints identification process should precede the constraints testing stage. Nevertheless, explicitly formulating the business constraints along the development of the system would greatly facilitate their later testing.

5.2 System Documentation

Availability of system documentation is also important when defining the data constraints and developing the test specifications. Semantic information, extracted out of the system documentation can add details to the ER model of the database in case it is developed through reverse engineering, or increase its understanding, in case it is readily available.

Furthermore, system implementation can help obtain the *domain knowledge* necessary for an effective detection of the data constraints. The experience of this project has shown, that a combination of absent documentation and insufficient domain knowledge can lead to a situation when 33% (8 out of 24) of the identified constraints will be unusable.

At the same time, absence of documentation is a fact of live and documentation easily gets outdated. The formalized constraints together with a QuickCheck framework are helpful in keeping the documentation alive, since changes in the program may make test cases fail.

5.3 Choice of Interfaces

A limitation encountered during the study was that the chosen XMLRPC interface did not provide access to the entire functionality of the Kred application. In the process of daily usage, the data within the application is modified through other existing system interfaces as well, for example the GUI. However, the effort required for the extra set up for the GUI testing was disproportionally high compared with the overall scope of the project. The inability to fully mimic all peculiarities of data handling during the tests has thus prevented us from a more thorough examination of the business logic. We can assume, that in a database application with multiple data access interfaces, a complete testing of the business logic also requires simultaneous testing of all available application interfaces.

In case that all, or most of the above conditions are fulfilled, the testing process can be focused on developing the test specifications. However, the current project has followed a different path and the following steps have been taken:

- Reverse engineer the database to obtain the database schema and ER model.
- Analyze the ER diagram to determine initial data constraints.
- Analyze the source code to identify other business logic constraints.
- Verify the obtained constraints with the developers who posses the domain knowledge about the system under test.
- Determine the interface functions that will be called in the testing process.
- Design test cases to test the data constraints.
- Implement the test case specifications using QuickCheck
- Run the tests and analyze the results.

6. Conclusions

In this case study we wanted to evaluate the methodology of Castro and Arts [4] for testing data consistency of data-intensive applications by examining a database application which uses an unnormalized non relational database. We have adopted a customized approach for extracting the data constraints by reverse engineering the database and expressing the constraints in a database-specific query language. Further, we have tested several interface functions with the QuickCheck testing tool and revealed a constraint violation.

There were several notable points in the process of constraint testing according to the adopted methodology. First, reverse engineering of the database structure is a crucial stage for the identification of data constraints. We have examined an unnormalized non relational database, and reverse engineered it according to a simplified version of the method described by Premerlany and Blaha [29] to obtain an ER diagram of the database. We have seen that important elements like multiplicity cannot be inferred without semantic information and can therefore affect the elicitation of database constraints.

The obtained ER model was used to extract and define data constraints that were present in the application. We have determined two types of constraints, namely referential constrains and business rules. Referential constraints can be identified by examining the ER model of the database and represent constraints based on foreign key relations between tables. We have seen that, despite our expectations and their relative discoverability referential integrity constraints can contain implementation faults, since we have found a violation of a referential constraint during the testing process.

On the other hand business rules cannot, in most of the cases, be identified through the examination of the database ER model, and therefore require the semantic knowledge of domain experts. We did not discover any violations of the business rules that have been tested. One of the reasons for this is the relatively small number of business rule constraints that were identified and tested. Another possible reason is the choice of system interface to be tested.

We have observed that the ratio of invalid or else incorrect constraints out of the total number of identified constraints was significantly higher in the first stages of the project, after the first iteration of database reverse engineering. Later on, the number of valid constraints has grown together with the understanding of the internals of the database application. Based on this, we can state that there might be a connection between the understanding of the application's implementation and the efficiency of the constraints elicitation process. On the other hand, improving and refining the process of constraints elicitation –both referential constraints and business rules – would be a topic for further research.

As mentioned above, in the current study we have examined a selection of data constraints and tested them with a limited number of interface functions. However, a complete elicitation of the data constraints present in the Kred application would require a revision and completion of the ER database model, further analysis of the relations between the entities in the model and interviews with domain experts.

We have limited ourselves and did not explore the effects of distribution and concurrency on the data constraints

within the application, despite both of them being important properties of Mnesia. Studying the effect of these two aspects can also be the topic of future research.

Taking into account the findings of the project, we can state that the adopted methodology could be applied to database applications which use non relational database management systems (particularly Mnesia), and unnormalized databases. We also contributed by applying the approach of Premerlany and Blaha to non relational databases and thus touching upon the topic of reverse engineering Mnesia databases.

Acknowledgments

The authors would like to thank everyone who has contributed to this paper with corrections, feedback and valuable input. Special thanks to the operational and software development teams at Kreditor for their help and support.

References

[1] M. Andersson, "Extracting an entity relationship schema from a relational database through reverse engineering," in *ER '94: Proceedings of the13th International Conference on the Entity-Relationship Approach*, (London, UK), pp. 403–419, Springer-Verlag, 1994.

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

[3] J. Arlat, A. Costes, Y. Crouzet, J. C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Trans. Comput.*, vol. 42, no. 8, pp. 913–923, 1993.

[4] T. Arts and L. M. Castro, "Testing data consistency of data-intensive applications using quickcheck," Technical report ITU, to be published, 2009.

[5] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with Quviq QuickCheck," in *ERLANG '06: Proc. of the 2006 ACM SIGPLAN workshop on Erlang*, pp. 2–10, ACM, 2006.

[6] N. Bassiliades and I. P. Vlahavas, "Modelling constraints with exceptions in object-oriented databases," in *ER '94: Proc. of the13th Int. Conf. on the Entity-Relationship Approach*, (London, UK), pp. 189–204, Springer-Verlag, 1994.

[7] J. Blom and B. Jonsson, "Automated test generation for industrial erlang applications," in *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, (New York, NY, USA), pp. 8–14, ACM, 2003.

[8] M.-C. Boudreau, D. Gefen, and D. W. Straub, "Validation in information systems research: A state-of-the-art assessment," *MIS Quarterly*, vol. 25, no. 1, pp. 1–16, 2001.

[9] C. Calero, M. Piattini, and M. Genero, "Empirical validation of referential integrity metrics," *Information and Software Technology*, vol. 43, no. 15, pp. 949 – 957, 2001.

[10] M. Y. Chan and S. C. Cheung, "Testing database applications with sql semantics," in *In Proc. of the 2nd Int. Symp. on Cooperative Database Systems for Advanced Applications*, pp. 363–374, Springer, 1999.

[11] W. K. Chan, S. C. Cheung, and T. H. Tse, "Fault-based testing of database application programs with conceptual data model," in *QSIC '05: Proc. of the Fifth Int. Conf. on Quality Software*, (Washington, DC, USA), pp. 187–196, IEEE Computer Society, 2005.

[12] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "An agenda for testing relational database applications: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, no. 1, pp. 17–44, 2004.

[13] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, 1976.

[14] K. H. Davis and A. K. Arora, "Converting a relational database model into an entity-relationship model," in *Proc. of the Sixth Int. Conf. on Entity-Relationship Approach*, pp. 271–285, 1988.

[15] Y. Deng, P. Frankl, and D. Chays, "Testing database transactions with agenda," in *ICSE '05: Proceedings of the 27th int. conf. on Software engineering*, (New York, NY, USA), pp. 78–87, ACM, 2005.

[16] J. Dietrich and A. Paschke, "On the test-driven development and validation of business rules," in *Information Systems Technology and its Applications, 4th Int. Conf., 23-25 May, 2005, Palmerston North, New Zealand, volume 63 of LNI*, pp. 31–48, GI, 2005.

[17] A. Eisenberg and J. Melton, "Background sql:1999, formerly known as sql3," *Commun. ACM*, 2008.

[18] M. Grochtmann and D. benz Ag, "Test case design using classification trees," 1994.

[19] J.-L. Hainaut, "Database reverse engineering: Models, techniques and strategies," in *Proc. Of the 10 th Int. Conf. on Entity-Relationship Approach.*

[20] S. Helke, T. Neustupny, and T. Santen, "Automating test case generation from z specifications with isabelle," in *ZUM '97: Proc. of the 10th Int. Conf. of Z Users on The Z Formal Specification Notation*, (London, UK), pp. 52–71, Springer-Verlag, 1997.

[21] H. Huang, W.-T. Tsai, S. Bhattacharya, X. Chen, Y. Wang, and J. Sun, "Business rule extraction techniques for cobol programs," vol. 10, (New York, NY, USA), pp. 3–35, John Wiley & Sons, Inc., 1998.

[22] W. Kent, "A simple guide to five normal forms in relational database theory," *Commun. ACM*, vol. 26, no. 2, pp. 120–125, 1983.

[23] F. N. Kerlinger, *Foundations of Behavioral Research*. Harcourt Brace Jovanovich, 1986.

[24] M. Ketabchi, S. Mathur, T. Risch, and J. Chen, "Comparative analysis of rdbms and oodbms: a case study," in *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Comp. Soc. Int. Conf.*, (New York, NY, USA), pp. 528–537, IEEE, 1990.

[25] V. V. Kuliamin, "Model based testing of large-scale software: How can simple models help to test complex system," in *In Proc. of 1-st Int. Symp. on Leveraging Applications of Formal Methods*, pp. 311–316, 2004.

[26] H. Mattsson, H. Nilsson, and C. Wikstrom, "Mnesia - a distributed robust dbms for telecommunications applications," in *PADL '99: Proc. of the First Int. Workshop on Practical Aspects of Declarative Languages*, (London, UK), pp. 152–163, Springer-Verlag, 1998.

[27] J. Nielsen and V. L. Phillips, "Estimating the relative usability of two interfaces: heuristic, formal, and empirical methods compared," in *CHI '93: Proc. of the INTERACT '93 and CHI '93 conf. on Human factors in computing systems*, pp. 214–221, ACM, 1993.

[28] M. Phalippou and R. Castanet, "Relations d'implantation et hypotheses de test sur des automates a entrees et sorties = implementation relations and test hypotheses on input-output automata," in *Travaux Universitaires - These nouveau doctorat*, (Universite de Bordeaux 1, Talence, FRANCE), 1994.

[29] W. J. Premerlani and M. R. Blaha, "An approach for reverse engineering of relational databases," *Commun. ACM*, vol. 37, no. 5, pp. 42–ff., 1994.

[30] G. J. Tretmans and A. F. E. Belinfante, "Automatic testing with formal methods," Technical Report TR-CTIT-99-17, Enschede, December 1999.

[31] J. Tretmans, P. Kars, and E. Brinksma, "Protocol conformance testing: A formal perspective on iso is-9646," in *Proc. of the IFIP TC6/WG6.1 Fourth Int. Workshop on Protocol Test Systems IV*, (Amsterdam, The Netherlands), pp. 131–142, 1992.

[32] H. Singh, M. Conrad, and S. Sadeghipour, "Test case design based on z and the classification-tree method," in *ICFEM '97: Proc. of the 1st Int. Conf. on Formal Engineering Methods*, (Washington, DC, USA), p. 81, IEEE Computer Society, 1997.

[33] S. De Souza, D. R. S. Maldonado, J. C. Fabbri, S. Camargo, P. Ferraz, D. Souza, and W. Lopes, "Mutation testing applied to estelle specifications," *Software Quality Control*, vol. 8, no. 4, pp. 285–301, 1999.

[34] I. O. for Standardization, "Information technology, open systems interconnection, conformance testing methodology and framework. international standard is-9646," (Geneve, CH), p.290-294, ISO, 1991.

[35] G. project, "http://projects.gnome.org/dia/," 2009.

[36] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: an empirical study," *J. Syst. Softw.*, vol. 31, no. 3, pp. 185–196, 1995.

Automatic Testing of TCP/IP Implementations Using Quickcheck

Javier Paris
University of A Coruna. Spain
javierparis@udc.es

Thomas Arts
IT University of Gothenburg and Quviq AB. Sweden
thomas.arts@ituniv.se

Abstract

We describe how to use model based testing for testing a network stack. We present a framework that together with the property based testing tool QuickCheck can be used to test the TCP layer of the Internet protocol stack. TCP is a rather difficult protocol to test, since it hides a lot of operations for the user that communicates to the stack via a socket interface. Internally, a lot happens and by only controlling the interface, full testing is not possible. This is typical for more complex protocols and we therefore claim that the presented method can easily be extended to other cases.

We present an automatic test case generator for TCP using Quickcheck. This tester generates packet flows to test specific features of a TCP stack. It then controls the stack under test to run the test by using the interface provided by it (for example, the socket interface), and by sending replies to the packets created by the stack under test. We validated the test framework on the standard Linux TCP/IP implementation.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing Tools

General Terms Reliability

1. Introduction

One of the problems when developing a TCP/IP stack is testing it. Test cases are not easy to create because of the nature of the environment in which the stack operates: it requires the ability to create a flow of packets that check particular features of TCP/IP, which often involve timing, where there are two actors involved (the local and the remote stack, with complex interactions).

Compliance is usually tested using another stack included in a commercial OS (such as Linux) and a packet sniffer to see the actual packet exchange and manually check if there is any deviation from the standard. There are several problems in this approach:

- Checking a packet dump by hand takes time and it is easy to overlook errors.
- It is hard to check special features which only arise under specific conditions because it is difficult to get the connection into that state. It is even more difficult to test behaviour under failure conditions.
- Because of the dependency on timing and concurrency, stack operations may seem to behave non-deterministically, making it harder to find error cases, but also difficult to reproduce error cases and re-check the stack when the error is fixed.

We introduce a tool for automatically testing TCP implementations through the use of the Quickcheck automatic test case generator. Using Quickcheck we can generate a large number of different test cases with complex behaviour. Specifically, we will focus on two scenarios that we can automatically generate as test cases for the stack. One of them is a simple connection establishment example, and will provide a good introduction. The other example is a complex situation of a simultaneous close which would be difficult to test a manually executed test case.

The rest of the paper is organized as follows. In Sect. 2 we introduce TCP/IP and a couple of test cases we want to test. In Sect. 3 we describe the environment in which the tests are performed. In Sect. 4 we provide a short introduction to Quickcheck, and provide an small example of a test using it. In Sect. 5 we describe in detail how a scenario of the test cases described in Sect. 2 would be. In Sect. 6 we describe some of the problems observed during the design of the tool. Finally, in section 7 we provide our conclusions and explain the future work.

2. TCP in a nutshell

The Internet protocol stack is the defacto standard for Internet communication. It specifies several protocols in five different layers. Each of the lower layers provides services to the upper layer:

- The *physical* layer specifies the electric and physical components of the network: connectors, cables, frequencies (for radio transmission), etc.
- The *link* layer uses the physical network to send packets between directly connected computers. Two well known protocols in this layer are Ethernet or PPP.
- The *network* layer, which uses the *link* layer to send packets between indirectly connected computers, that is, between computers which may be in different but interconnected physical networks. The main protocol in this layer is IP, but there are several others, like ICMP and IGMP.
- The *transport* layer, which uses the *network* layer to control the packet interchange between the two ends of the communication, providing multiplexing, rate control, and packet loss recovery. Typical examples are TCP (Transmission Control Protocol) and UDP, where TCP is the most complex of the two, since it guarantees lossless ordered packet delivery.
- The *application* layer that uses TCP or UDP for transport of the application specific data. Examples are HTTP or IMAP.

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.

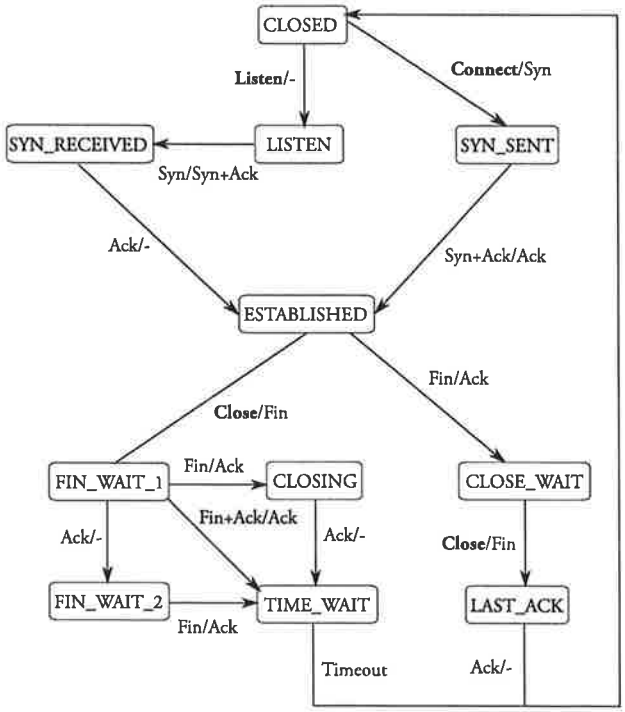

Figure 1. TCP Connection States

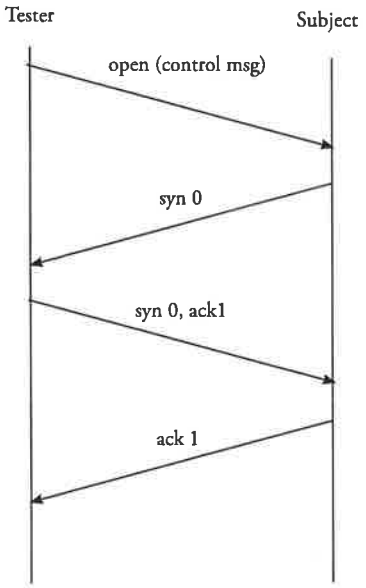

Figure 2. Typical TCP Connection Establishment

The TCP protocol ensures reliable connections on top of unreliable layers. When sending a webpage from one computer to another, TCP divides the page in packages of a certain size and ensures that they are transported to a TCP layer on the other side, which puts the packages together. TCP takes care of resending and assembling, such that the application layer need not to care about that. This requires TCP to have state in order to 'remember' which packets have been sent and together with the high dynamic nature of the protocol, it makes it the most complex one of the four lower layers. TCP connections require synchronization between the peers. All TCP packets have a *s*equence number that identifies the order of the data that is transmitted. This number is used by the peer to order the data as it is received, and to identify lost packets. It is important to note that because a TCP connection is symmetric there is a different sequence number for each of the peers.

TCP packets also include an *acknowledge* number, which reports the last sequence number that the sender of the packet has seen. This is used by the receiver to control the transmission rate and to identify lost packets.

On a high abstraction level, TCP can be modelled by a simple state machine that keeps track of a connection [RFC793 1981]. Of course, many of these connections are concurrently active at the same time. Transitions from one state to another are triggered by timers, the reception of packets or user commands.

At the start of a TCP connection both peers exchange their initial sequence numbers. This process is the connection establishment, and can be seen at the top of Figure 1. A more detailed description of the TCP state machine can be found at [RFC793 1981],[Zaghal and Khan].

When both peers have finished the connection establishment both should be in the ESTABLISHED state, where the actual data transmission takes place.

After the data exchange is over, the connection is closed. This allows both peers to free the resources used to keep the state of the connection. A connection which is closing will transit through the states at the lower part of Figure 1.

Note that the event description in Figure 1 is of the form *Receive/Sent*, that is, when the peer in a certain state receives the *Receive* event, it will send a *Sent* to the other peer and transit to the next state. Also note that the standard interface to TCP, i.e., the socket interface, only gives access to stimulate the events: *Listen*, *Connect*, *Send*, *Receive*, and *Close*. Our approach aims to also control all other events in order to perform proper testing.

Testing TCP

In this paper we describe how we can test TCP by using the above presented state machine as a model for our test cases. We assume all lower layers of the protocol to be correct and check at the end points of communication.

For example, we want to test the transitions caused by connection establishment and closing, including obscure cases such as simultaneous closing[1]. In other words, we want to generate sequences of events that make the TCP implementation move through all possible states in the state machine in all possible ways. We need to create different sequences to get to state ESTABLISHED and different sequences to get to CLOSED again. A typical connection establishment would follow Figure 2:

1. Both Peers start with the connection in the `CLOSED` state.
2. A program in Peer B listens for incoming connections and opens a socket. This puts the connection in the `LISTEN` state.
3. Peer A wants to start a connection and sends a packet with its initial sequence number (0 in our example), and moves to the `SYN_SENT` state.
4. Peer B receives the packet sent by A, and replies with its own initial sequence number (also 0), and acknowledges the *syn* sent by peer A saying that the next sequence number it expects is 1. Peer B moves to the `SYN_RECEIVED` state.

[1] This happens when both peers start a connection close at the same time, instead of doing it sequentially.

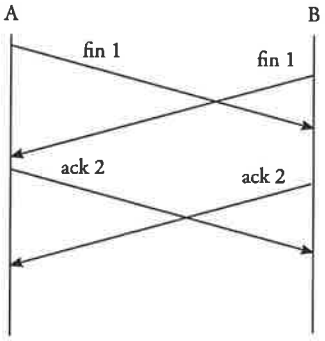

Figure 3. Simultaneous Connection Close

5. Peer A receives the packet sent by B in the previous state, and acknowledges it by sending an *Ack* packet. After this Peer A moves to the `ESTABLISHED` state.
6. Peer B receives the *Ack* and moves to the `ESTABLISHED` state.

This is a simple example, but there are several things that can be checked when this sequence is traversed. The tested stack must generate the correct sequence of packets (in both cases, as starter and as receiver); it must create correct packets (for example, the checksum check should always pass); it also has to keep the state correctly (the sequence and acknowledge numbers of the sequence of packets must be the same).

A rarer scenario is the simultaneous closing of a connection, which happens when both peers try to close the connection at the same time. Figure 3 shows the packet exchange that would happen.

As it can be seen, both peers send a *Fin* packet to start the closing procedure. In this case the connection is simply closed by acknowledging the *Fin* sent by the other peer. While the case is fairly simple to understand, it is difficult to reproduce in a real environment, because it is necessary to perfectly synchronize the connection close in both peers. This requires the tester to have full control over both Peers.

3. Test Setup

TCP connections are usually controlled by the application layer protocols, the most common being the socket interface. These interfaces abstract most of the complex behaviour of TCP, which is good for normal connections, but makes it difficult to analyze if the behaviour of the stack is correct. The tests should not only test the behaviour seen at the interface level, but go deeper.

We also want the test to be as general as possible, that is, not geared towards testing a particular TCP/IP stack. It should be easy to adapt the test specification to check any TCP/IP stack. If no particular structure can be assumed in the tested stack the only way to test it is by doing black box tests.

We have decided to test the behaviour by looking at the actual packets sent over the network. This should provide a reasonable way of doing an in-depth test and providing a test specification that may be used to test very different implementations of TCP. This approach has its limitations. For example, there may be errors that do not show in the packet trace, or are difficult to spot. As an example, an incorrect computation of timers would not be seen in the trace unless it was very serious.

The test setup can be seen in Figure 4, and includes:

1. A computer with the stack under test (from now on **Subject**). This stack needs to be controlled and we use a controller program for doing so. This controller translates the high level commands to execute during the test, such as opening and closing connections, and sending data, to the specific API of Subject. The controller has no state, it just translates the commands coming from QuickCheck [2].
2. A computer running QuickCheck (from now on **Tester**) simulating TCP by means of a QuickCheck specification module and a special IP stack. The IP stack is used to send the TCP packets generated by QuickCheck and to receive incoming packets from the subject. We use a special IP stack, written in Erlang since we want our TCP model to communicate with an IP layer. This would require RAW sockets, but the implementation of them is different for each IP implementation. Thus, using our own IP stack on the Tester side, provides a portable solution for testing. It also allows us in future testing to send incorrect IP packets, which would be hard when using an off-the-shelf IP implementation.
3. A network connecting both computers. Initially both are supposed to be directly connected by an Ethernet network, but this test scenario will be expanded when we add test for cases that require routing. For simplicity we assume that there are no problems in the Ethernet network, that is, no packet loss or delay. These are very rare anyway and we want to control them, not having them spontaneously appear.

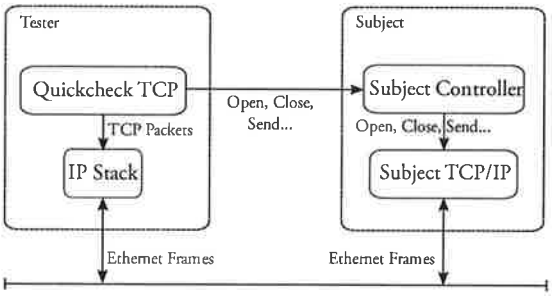

Figure 4. Test Setup

The IP stack is based on the TCP/IP stack described in [Paris et al. 2005]. This processes packets from the network, and creates IP packets to send the TCP packets created by QuickCheck. Because it is implemented in userspace, it uses a packet sniffer to capture packets from the network, and to inject packets into it. This sniffer has been developed using the `libpcap` library[Jacobson et al.] and it supports several operating systems including Linux, Windows, and several BSD variants (such as FreeBSD or Mac OS X).

4. QuickCheck

We base our approach on property-based testing using QuickCheck [Claessen and Hughes 2000], in a commercial version for Erlang developed by Quviq AB [Hughes 2007, Arts et al. 2006]. Compared to the original version of QuickCheck, the commercial version of it has many supporting libraries for protocols, among which a library for finite state machine models that we in particular use.

QuickCheck tests universally quantified *properties*, instead of single test cases. QuickCheck generates random test cases from each property, tests whether the property is true in that case, and reports cases for which the property fails. QuickCheck "shrink" failing test cases automatically, by searching for similar, but smaller

[2] This is a potential weakness of the testing model because Erlang uses TCP for its distribution protocol and we use Erlang to connect to the controller.

test cases that also fail. The result of shrinking is a "minimal"[3] failing case, which often makes the root cause of the problem very easy to find.

This approach has several advantages. Properties are easier to understand than normal test cases, and a property can be used to generate thousands of different test cases. Each test case can also be more complex than a manually written test case.

As an example of Quickcheck we will present a test for IP checksum implementations.

4.1 Testing IP Checksum Implementations with Quickcheck

The IP checksum is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. Ones complement is a way of representing negative numbers.

The IP checksum uses base 16, that is 2 bytes. In 16 bits you can represent the numbers 0 to 65535. The idea with ones' complement is to use half the numbers in this interval for representing negative numbers. Thus, 0 up to 32767 are the positive numbers and 65535 is -0, or an alternative representation of zero. The number 65534 is -1 etc. Until 32768 which is -32767. Hence the interval -32767 up to 32767 can be represented.

4.1.1 Ones Complement

Given an implementation of the IP checksum, a first function we want to test is the ones' complement of a word, say that this is implemented as Erlang function `ip_checksum:ones_complement/1` and that it takes a binary with one 16 bit word as input.

The QuickCheck property that we use to test this function should **not be a re-implementation** of the function under test. This holds for testing in general and also for QuickCheck properties. We do not want to implement a function in order to test it. A way of testing without writing a new version of the ones' complement is to follow the definition and say that a number and its ones' complement sum to -0. Since we want to be general, we specify the property for any base and check it with argument `Base` substituted by 16.

```
prop_ones_complement(Base) ->
 ?FORALL(I,choose(0,(1 bsl Base)-1),
      begin
        CI = ip_checksum:ones_complement(I),
        (1 bsl Base)-1 == I+CI
      end).
```

The QuickCheck macro `?FORALL` is the logic quantifier with 3 arguments, the first one, the variable `I` is bound to a random value generated by the second argument `choose(0,(1 bsl Base)-1)`. This generator randomly chooses a number between zero and the number obtained by shifting a bit `Base` positions to the left and subtracting one, thus between 0 and $2^{Base} - 1$.

The third argument of the `?FORALL` macro is the actual logic expression implemented as an Erlang expression. First the function is applied to the randomly chosen value `I`, which results in `CI`. The sum of `I` and `CI` should be -0 or $2^{Base} - 1$.

This property is checked by QuickCheck by generating random values specified by the generator and evaluating the Erlang expression. If the result is `false`, then an error is detected and smaller values are tested until a minimum example is found for which the property can be falsified.

4.1.2 Padding

It is not clear from the specification presented above, but if you need to compute the checksum of a list of bytes in base 16, then there should be an even number of bytes. Likewise, if we would like to do ones' complement in 32 bits base, we would need to extend a sequence of bytes such that it is divisible by 4.

Extending a bit string such that it is divisible by the base is called padding. We assume there is a function that performs the padding called `ip_checksum:padd/1` taking a bit string as argument and returning a new bit string which is an extended version with as many zero bits as needed.

We like to present a general example that works for padding bitstrings with any number of bits. We assume a generator for bitstrings given in QuickCheck, but for older versions of QuickCheck, one can define a simplistic version oneself for bitstrings up to 200 bits:

```
bitstring() ->
   ?LET(NrBits,choose(0,200),bitstring(NrBits)).

bitstring(NrBits) ->
   ?LET(Bits,vector(NrBits,choose(0,1)),
        to_bitstring(Bits)).

to_bitstring([]) ->
   <<>>;
to_bitstrong([Bit|Bits]) ->
   Rest = to_bitstring(Bits),
   << Bit:1, Rest/bitstring>>.
```

After being able to generate arbitrary bitstrings, we can specify the property for padding arbitrary strings for an arbitrary number of bytes. In the specific case of IP checksum, we would use 16 as base.

```
prop_padding(PadSize) ->
  ?FORALL(BitString,bitstring(),
     begin
       Bits = bit_size(BitString),
       <<B:Bits/bits, Padded/bits>> =
         ip_checksum:padd(BitString),
            Zeros = bit_size(Padded),
       ((Bits+Zeros) rem PadSize) == 0 andalso
              B == BitString andalso
         <<0:Zeros>> == Padded
      end).
```

In this property, the function under test is called with an arbitrary bitstring; it should pad this bitstring for `PadSize` bits. We match the result with a bitlist in which the prefix contains exactly the number of bits that the generated bitstring has and the rest is the padded suffix. We check that the number of bits in the result is divisible by the `PadSize`, that the original bits are unchanged and that the padded bits are all zeros.

4.1.3 Ones Complement Sum

The ones' complement sum is computed by adding a number of words in ones complement. We assume it is implemented by the function `ip_checksum:ones_sum/2` which takes a bitstring as argument. We assume that padding is done outside the `ones_sum` function and only test that the function works for bitstrings of which the length is divisible by the given base.

We do not assume that the `ones_sum/2` function returns the ones' complement of the sum, it just returns the sum. We can compute the sum, add its ones' complement to the exiting list, compute the sum once more and expect to get zero (or actually -0) out of the result. We present a general property and can call it with base 16 in the specific case.

```
prop_ones_sum(Base) ->
   ?FORALL(I,choose(0,1024),
      ?FORALL(Bin,bitstring(I*Base),
         begin
```

[3] In the sense that it cannot shrink to a failing test with the shrinking algorithm used.

```
        Sum = ip_checksum:sum(Bin),
        CSum = ip_checksum:ones_complement(Sum),
    ip_checksum:sum(<<CSum/bits, Bin/bits>>) ==
       <<((1 bsl Base)-1):Base>>
    end)).
```

4.1.4 Checksum

After computing ones' complement sum, one has to take the ones' complement of the result to compute the checksum, as we already did in the above property. For checking the checksum function, we use the same trick as we used for checking the sum function, knowing that if we extend the binary with the checksum and we compute the ones' complement sum we get -0. Alternatively, we could also compute the checksum and do a logical and with the ones' complement sum or so.

```
prop_checksum(Base) ->
   ?FORALL(Bin,bitstring(),
      begin
        CheckSum = ip_checksum:checksum(Bin),
        ip_checksum:sum(<<CheckSum/bits, Bin/bits>>) ==
            <<((1 bsl Base)-1):Base>>
      end).
```

In this way, simple side-effect free functions can be checked by QuickCheck. One uses a, possibly user defined, data generator to create random values in a certain domain. These values are used in a test that reflects a property of the function more than a concrete case.

However, in most Erlang Software, side-effects play an important role and we want to be able to test functions that have side-effects as well. Quviq QuickCheck supports this by libraries that help to specify properties for such software. One of them is the library for finite state machines.

4.2 Quickcheck Finite State Machine Specifications

The TCP protocol is a so called stateful protocol, i.e., specific events may have different effects depending in which state the TCP connection is. If we would perform traditional testing, we would create sequences of events to get TCP into a certain state and consecutive sequences to test certain specifics. With QuickCheck we follow a similar approach, but then quantifying over all possible sequences of events.

Similar to the above described side-effect free case we define a property which in this case states that for all sequences of events the TCP protocol respects the postconditions of these events. The property is rather simple, defining the postconditions and the generator that generates all possible valid sequences is where the work is.

We use the finite state machine library named `eqc_fsm` to specify the state machine given in Sect. 2. That state machine is then used to generate test cases from. Specifying such a state machine is a bit like writing a state machine in `gen_fsm`, but the transitions are non-deterministic and chosen by QuickCheck when generating possible sequences.

In order to specify a state machine, one has to specify for each state which next states it has and which actions or events should be generated to get from one state to the next. Like in `gen_fsm` the state transitions are given as callback functions. One starts by specifying the initial state and initial state data and from there, functions are provided for each state. In addition to this more operational part from the state machine, one also specifies preconditions for events; a precondition is a logic expression specifying whether or not a certain event can take place in a certain state with certain state data. Likewise, one specifies postconditions. A postcondition is a logic expression that specifies whether the return value of a given action or event in a certain state could occur.

In the next section we describe our state machine model in detail.

5. State Machine Model

Our approach is to test the behaviour of a TCP connection with a state machine model. This state machine is modelled using the `eqc_fsm` module of Quickcheck described above.

A generated test simulates the behaviour of a real TCP connection. The current state of the finite state machine models the state in which the connection in the subject should be.

Given the state machine model that we describe in this section, QuickCheck generates a sequence of commands. Either the controller forwards these commands to Subject, which in its turn can result in a TCP packet being sent from the Subject to Tester; or the command provokes the Tester to send a TCP packet to Subject. The result are packages that arrive on the QuickCheck side and the postconditions in the model check that the expected packets have arrived. Replies are generated in advance, based on the expected behaviour.

Note that we test connections, we randomly choose a port, but do all operations on that port during one test. Concurrent connections should be tested differently with a more elaborated state machine model in which we have free and used ports and in which preconditions are used to select free ports when starting a connection.

5.1 Checking Connection Establishment

As an example, we will see how the **tester** checks that the subject correctly establishes a connection (see Figure 2). There are three relevant TCP states:

5.1.1 CLOSED

`CLOSED` is the state in which the connection should be before it is started. This state corresponds to an Erlang function `closed` in the state machine model and from this state, two other states can be reached, the `syn_sent` state and the `syn_rcvd` state.

```
closed(S) ->
[{syn_sent,
   {call, ?MODULE, open,
     [S#state.ip, S#state.port, {var, listener},
      {var, sut}]}},
 {syn_rcvd,
   {call, ?MODULE, listen,
     [S#state.sut_ip, S#state.sut_port, S#state.ip,
      S#state.port, {var, listener}, {var, sut}]}}
].
```

This describes the possible transitions from the `CLOSED` state. The state machine may either proceed to the syn_sent state (by calling the function `open`), or to the syn_receive state (by calling `listen`). Both calls take several parameters from the state of the test (`S`), such as the IP addresses and ports. It also uses two parameter of the test (`listener` and `sut`), which are the symbolic representation of the Erlang process identifiers of the IP stack and the subject controller respectively.

Note that there is no specific state for the `LISTEN` state. One can view the transition between `CLOSED` to `LISTEN` as a silent transition, or a τ step in process algebraic terms. The transition does not transmit any packet on the network.

The `LISTEN` state provides a way of receiving multiple incoming connections to a local port, it is only a way of telling apart ports on which a process is waiting for incoming connections, and ports where no one is listening. A connection is created by either accepting an incoming message on a listen call to a `CLOSED` port or by starting a connection from a `CLOSED` port. Our QuickCheck model

controls the Subject to either do the one or the other in a random way.

In our example, where the subject is starting the connection, the state would proceed to syn_sent. Quickcheck would call the open function, which would:

1. Tell the controller to open a connection.
2. Ask the IP stack to receive a TCP packet from the subject.

This should move the subject state to SYN_SENT. Quickcheck now calls the `postcondition` function to check if the received TCP packet is correct (a timeout in the listener will raise an exception if no packet is received at all).

```
postcondition(closed, syn_sent, S,
{call, ?MODULE, open, [Ip, Port, _, _]}, Syn) ->
   check_flags(Syn, [syn]) and
     (Syn#tcp.dst_ip==Ip) and
     (Syn#tcp.dst_port==Port) and
     (Syn#tcp.data == <<>>);
```

The postcondition is valid for a transition from CLOSED to SYN_SENT, caused by a call to `open(Ip, Port, Listener, Sut)`, if the message was a Syn message, i.e., the corresponding flag in the header is set (see Fig. 1). It is also checked that the IP address and Port match and that the data part of the message is empty.

When the postcondition is valid, Quickcheck will now update the state. In addition, the state machine model supports state data, i.e., a data parameter that carries from one state to the other. We use the state data to store the last message that we have sent, in order to be able to read sequence numbers and other data in consecutive states.

```
next_state_data(closed, syn_sent, S, Syn,
                {call,_,_,_}) ->
   S#state{last_msg = Syn};
```

Now the peer Subject is in the SYN_SENT state and the peer Tester is in the SYN_RECEIVED state where a *Syn* message has been received.

5.1.2 SYN_SENT

SYN_SENT is the state in which the subject will be after sending its *Syn* packet (Figure 1). In this state the subject will be awaiting a Syn+Ack packet from the tester. The possible transitions from this state are represented as follows in our state machine model:

```
syn_sent(S) ->
 [{established,
    {call, ?MODULE, syn_ack,
       [{var, listener}, S#state.last_msg]}}
 ].
```

That is, from this state the Subject stack only has one transition to ESTABLISHED by calling `syn_ack`:

```
syn_ack(Listener, Syn) ->
   Syn_ack = #tcp{dst_port = Syn#tcp.src_port,
                  src_port = Syn#tcp.dst_port,
                  dst_ip   = Syn#tcp.src_ip,
                  src_ip   = Syn#tcp.dst_ip,
                  seq      = 0,
                  ack      = seq:add(Syn#tcp.seq, 1),
                  is_ack   = 1,
                  is_syn   = 1,
                  window   = ?DEFAULT_WINDOW
                  },
   Bin_Packet = packet:tcp_to_binary_chksum(Syn_ack),
   tcp_listener:send_packet(Listener,
                            Bin_Packet,
                            Syn#tcp.src_ip),
   get_parsed_tcp_packet(Listener).
```

This function shows how the tester generates replies by using the values from the previous packet received (Syn). The newly created packet has the `ack` and `syn` flags set, and will acknowledge the `syn` by adding 1 to its sequence number.

The Erlang record representation of a packet is then converted into a binary and this packet is sent over the network. After the peer receives the packet we expect it to acknowledge it, so the last step is to get a new inbound TCP packet. Here we benefit from our refined model. The TCP interface would just allow us to listen on one site and open on the other side and we would have got a return after that both sides are in established. Now we create and inspect internal states and can check responses. We could inject faults, cause a timeout or whatever at this point, but we cannot communicate to the open connection of Subject via the API before getting it in the state ESTABLISHED.

The postcondition for this transition validates that the received message is indeed an *Ack*.

```
postcondition(syn_sent, established, S,
              {call, ?MODULE, syn_ack,[_, Syn]},Ack) ->
   check_flags(Ack, [ack]) and
   (Ack#tcp.seq==nxt_seq(Syn)) and
   (Ack#tcp.ack==1) and
   (Ack#tcp.data==<<>>);
```

In this case we validate that the only flag set is the `ack` flag. This is just an `ack` for the *Syn* sent by the tester, that the acknowledge sequence number is is increased by one (we sent zero), and that the sequence number should be one more than the `syn` packet it previously sent, to account the *Syn* flag of that packet. Additionally the data part should be empty.

The state data is once more updated by just storing the last message received, i.e., the *Ack*.

```
next_state_data(syn_sent, established, S, Ack,
                {call, _, _, _}) ->
   S#state{last_msg = Ack};
```

Now the Subject stack is in the state ESTABLISHED and the tester is in the ESTABLISHED state, since it has received an *Ack*. Therewith we have modeled the connection establishment of Figure 2 from one side. The opening where we instruct the Subject to go to the LISTEN state and where the tester sends the first *Syn* message is added to the same model in a similar way.

5.2 Testing Simultaneous Connection Closing

Now we add to the model the possibilities for connection closing and we show how one of the generated test cases corresponds to the simultaneous closing of two connections as shown in Figure 3. This is an interesting test case because it is difficult to simulate in a real network, but it is easy to generate from our model.

Usually, testing would be done by using an operating system stack as peer. Doing a simultaneous closing for testing would require calling the close system call at almost the same time in both peers of the connections. As there are some time issues that fall out of control of the socket interface (for example, the exact time at which the `fin` packet is generated by the kernel cannot be exactly controlled), it is very difficult to reliably generate a situation like this.

However, we will see how the state machine generates a test case that checks this behaviour in a similar way to the connection establishment.

In a simultaneous closing, both stacks would pass through the FIN_WAIT_1 state, they proceed to the CLOSING state, and finally to the TIME_WAIT state (see Figure 1). The CLOSING state is specific to the simultaneous closing, no other close situation passes through it.

We start with a connection for which both peers are in the ESTABLISHED state.

5.2.1 ESTABLISHED

There are two possible transitions from ESTABLISHED, one initialized via a close in the TCP interface and one initialized by the other peer:

```
established(S) ->
   [{fin_wait_1,
       {call, ?MODULE, active_close,
          [{var, listener}, {var, sut}]}},
    {close_wait,
       {call, ?MODULE, passive_close,
          [{var, listener},S#state.last_msg]}}].
```

A peer may go to the state FIN_WAIT_1 by an active close (the TCP stack got a close via the API), or to the CLOSE_WAIT state by doing a passive close (the stack receives a *Fin* from the other peer). It is important to recall that the current state reflects the state in which Subject should be, so a simultaneous close arises when the subject stack starts the close before receiving a *Fin* packet from the tester.

In a test that checks the simultaneous closing, Quickcheck would generate an active close, i.e. have the controller send a close to Subject, which corresponds to the first alternative in the state transitions above.

```
active_close(Listener, Sut) ->
   simple_sender:close(Sut),
   get_parsed_tcp_packet(Listener).
```

After the Subject is told to close the connection, a *Fin* packet is expected to be received in the IP stack of the tester. First, the postcondition for this transition is validated:

```
postcondition(established, fin_wait_1, S,
              {call,?MODULE, active_close, [_, _]},
              Fin) ->
   Ack = S#state.last_msg,
   check_flags(Fin, [ack, fin]) and
   (Fin#tcp.ack==1) and  %% Acks our Syn
   (Fin#tcp.seq==nxt_seq(Ack)) and
   (Fin#tcp.data== <<>>);
```

The check is fairly similar to the previous ones. The `fin` flag should be set as well as the `ack` flag[4]. Sequence numbers should increase and the data part is empty. The state data is updated as in the previous cases, storing the received message:

```
next_state_data(established, fin_wait_1, S, Fin,
                {call, _, _, _}) ->
   S#state{last_msg = Fin};
```

Now the Subject is in the state FIN_WAIT_1, but the Tester peer is still in the state ESTABLISHED. We can choose which action we want the tester to perform. Either it acknowledges the received *Fin*, or it also sends a *Fin*, because we close the connection (the simultaneous close case).

5.2.2 FIN_WAIT_1

Thus, we expect two possible state transitions from the state FIN_WAIT_1. However, as can be seen in Figure 1, there are three possible transitions from this state. The third one is the tester sending a *Fin+Ack* to Subject. Here the specification of Figure 1 is hard to understand and additional knowledge is necessary. From additional text in the specification we learn that it is possible that the peer Tester in state ESTABLISHED receives *Fin* and then responds by sending a combined acknowledge on that *Fin* with its own *Fin*. This is then resulting in a state transition from Tester to LAST_ACK, thereby taking away the need to send an explicit *Close* from the interface. In Figure 1 it is shown that normally one would wait from a close from the interface before sending the last *Fin* in order to completely close the connection.

```
fin_wait_1(S) ->
 [{time_wait,
     {call, ?MODULE, fin_ack,
        [{var, listener}, S#state.last_msg]}},
  {fin_wait_2, {call, ?MODULE, ack_received_fin,
        [{var, listener}, S#state.last_msg]}},
  {closing, {call, ?MODULE, simultaneous_close,
        [{var, listener}, S#state.last_msg]}}
  ].
```

The test generated to check simultaneous closing would cause Subject to perform a state transition to CLOSING by calling the `simultaneous_close` function:

```
simultaneous_close(Listener, Fin) ->
   Local_Fin = #tcp{dst_port = Fin#tcp.src_port,
                    src_port = Fin#tcp.dst_port,
                    dst_ip = Fin#tcp.src_ip,
                    src_ip = Fin#tcp.dst_ip,
                    seq = 1,
                    ack = Fin#tcp.seq,
                    is_ack = 1,
                    is_fin = 1,
                    window= ?DEFAULT_WINDOW
                   },
   Bin_Packet = packet:tcp_to_binary_chksum(Local_Fin),
   tcp_listener:send_packet(Listener,
                            Bin_Packet,
                            Fin#tcp.src_ip),
   get_parsed_tcp_packet(Listener).
```

In the *Fin* packet we are creating the acknowledge number is taken directly from the sequence number of the subject's *Fin*, instead of increasing it by 1 to acknowledge the packet. This is correct, since we actually simulate to have not seen that message yet. After receiving the constructed packet the subject stack will 'see' a simultaneous close.

The postcondition that needs to be validated is that Tester received an *Ack*:

```
postcondition(fin_wait_1, closing,S,{call, ?MODULE,
                                     simultaneous_close,
                                     [_, _]}, Ack) ->
   Fin = S#state.last_msg,
   check_flags(Ack, [ack]) and
     (Ack#tcp.ack==2) and
     (Ack#tcp.seq==nxt_seq(Fin)) and
     (Ack#tcp.data== <<>>);
```

The next state data is similar to previous cases, we only save the last received message:

```
next_state_data(fin_wait_1, closing, S, Ack,
                            {call, _, _, _}) ->
   S#state{last_msg = Ack};
```

Now Subject has moved to state CLOSING whereas Tester is in the state FIN_WAIT_2, since it has already received the *Ack* from Subject.

5.2.3 CLOSING

In the state CLOSING Subject waits to receive an *Ack* packet that acknowledges the *Fin* it sent before.

[4] Apart from the first message, where the sequence number is unknown, all messages have the ack flag set and contain the sequence number of the previous message.

```
closing(S) ->
 [{time_wait,
    {call, ?MODULE, send_ack,
        [{var, listener}, S#state.last_msg]}}].
```

Thus in this state, QuickCheck generates an acknowledge to the *Fin* we ignored in the previous state:

```
send_ack(Listener, Fin) ->
   Ack = #tcp{dst_port = Fin#tcp.src_port,
              src_port = Fin#tcp.dst_port,
              dst_ip   = Fin#tcp.src_ip,
              src_ip   = Fin#tcp.dst_ip,
              seq      = 2,
              ack      = nxt_seq(Fin),
              is_ack   = 1,
              window   = ?DEFAULT_WINDOW
             },
   Bin_Packet = packet:tcp_to_binary_chksum(Ack),
   tcp_listener:send_packet(Listener,
                            Bin_Packet,
                            Fin#tcp.src_ip).
```

Notice that now the ack is computed using `nxt_fin`, which increases the sequence number by 1. After this packet is received by the subject, the connection would move to `TIME_WAIT`, which is similar to a closed connection. There is nothing to check in this state:

```
postcondition(closing, time_wait, _S,
              {call, ?MODULE, send_ack, [_, _Ack]},
              _) ->
   true;
```

There are a few states left, that need to be modeled in the same way. After that, a large number of random tests can be generated that open a connection in one of the two ways and closes them in any of the possible ways. We can create tests that open and close the same port in different ways quickly after each other by making silent transitions from `TIME_WAIT` and `LAST_ACK` to `CLOSED`.

We have in the examples always started the sequence numbers by zero, but one could actually pick a random number and increase from there as yet one more thing to test.

However, we only test one connection at a time and no concurrent connections, which may not trigger certain errors in the subject stack. We also have concentrated on positive testing only, that is, it is not checked that the subject reacts correctly to malformed or unexpected packets.

6. Testing a reference stack

We developed our specification by testing against the Linux kernel TCP stack, assuming that that one is fairly well tested already. Thus, if problems arise, they are likely a fault in our specification. This assumption proved to be correct as all the problems found were in the specification. Two tricky issues one needs to be aware of.

Occupied Ports

We tested one connection at a time and read all TCP communication on the network. Thus, when the listener reads a *Fin* message, it assumed it to be part of the test. However, for a specific *Fin* message the sequence number and Port were wrong, as the postcondition revealed. This could have been an error in the Subject TCP stack, but was not. Out generators did not require the tests to end with the connection closed, so at times a connection was left in some intermediate state. After some time, the subject TCP/IP stack tried to close the connection and sent packets to the tester which were confused as packets related to the test run at that time.

An easy solution to this problem would be to make the listener specification ignore any packet with a wrong destination port. However, we did not want to do that because the tester would not be able to detect if Subject is sending incorrect port numbers. We needed to make the packet listener more intelligent and context aware. Of course, we would have serious problems when we extend the specification to test concurrent connections because this filtering will be unavoidable. However, if we have thoroughly tested the case of one connection and port numbers are always correct, we may probably assume that we can use the port as connection identifier for the concurrent case. This is not that tricky, since if the concurrent case would contain an error causing a changed port number in the return message, then something unexpected would happen and a test will fail.

Reusing Ports

Checking the behaviour in the `TIME_WAIT` state is also difficult. When the stack arrives at this state, it must wait for a time in case the last ack packet has been lost. This time is fairly long (several minutes). While the stack is in this state, trying to open a new connection using the same port number will result in a reset. This is the correct behaviour according to the standard, but it also makes tests much longer. We have added a configurable time for this state, with the idea that the developer of a TCP/IP stack has control over the time waited, and it can be changed to run the tests. However, this has to be considered when testing an out of the box stack, like we did with Linux.

Testing other Stacks

After we corrected several errors in the specification and the Linux stack passed the tests, we tried with a less well tested stack in order to try to actually find errors in the stack. We selected the stack described in [Paris et al. 2005] because we are very familiar with it, it is likely to have some bugs not yet discovered, and being written in Erlang it was very easy to adapt for the tests.

After running several tests we actually found an error in the `LAST_ACK` state, which caused sequential connections using the same port number to fail at times. The bug was due to a lag in cleaning up the list of currently opened connections, which caused the stack to react to the incoming `syn` for the new connections as if it belonged to the old one.

This did not always happen, so the ability of Quickcheck to generate hundreds of test cases was very useful to detect the problem.

7. Conclusions

We have developed a QuickCheck specification for automatic checking of TCP implementations. By generating sequences of TCP packets using quickcheck we are able to generate test cases which are not easy to reproduce in a real environment, like simultaneously closing the connnection from both sides.

This QuickCheck specification differs from earlier QuickCheck specifications for protocols in the use of two interfaces to communicate with the subject under test. Previously we have seen QuickCheck specifications that triggered the subject under test by communicating via the protocol API (e.g. [Arts et al. 2006]). For TCP this is insufficient, we also need to interact via the other end of the protocol, the IP level of the stack.

The use of a very simple controller for the subject makes it easy to adapt the tester to any stack. Ideally, if the stack uses the socket interface it should be able to use the provided controller without changes. Our goal is that the tester can easily be used by anyone developing a TCP/IP stack.

As future work we consider the extension of the model to cover operation of a connection in the `ESTABLISHED` state, such

as sending and receiving data, reordering of data, flow control, retransmissions, *etc.*

Another interesting extension is to add negative testing, which is fairly difficult to do in a real environment because it requires an incorrect implementation of TCP. Negative testing is useful to check the robustness of the subject stack. The idea is that the tester can generate illegal packets as part of a legal sequence and see whether the subject correctly handles these illegal packets.

Negative testing is especially interesting for a TCP/IP stack whose normal environment is the Internet. Any error detected is a potential security problem in a TCP/IP stack. It is also interesting because negative testing cannot be done by using a normal stack as a peer.

We also want to test the behaviour of TCP with many concurrent connections. This is interesting because network stacks are by design highly concurrent, and concurrency is usually a source of errors, so testing their behaviour under those conditions is very desirable.

Acknowledgments

Partially supported by Xunta de Galicia PGIDIT07TIC005105PR.

References

Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing Telecoms Software with Quviq QuickCheck. In *ERLANG'06:Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, New York, NY, USA, 2006. ACM Press.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

John Hughes. Quickcheck Testing for Fun and Profit. In Michael Hanus, editor, *9th International Symposium on Practical Aspects of Declarative Languages*. Springer, 2007.

Information Sciences Institute. RFC 793: Transmission control protocol, 1981. URL `http://rfc.sunsite.dk/rfc/rfc793.html`. Edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc793.html.

Van Jacobson, Craig Leres, and Steven McCanne. libpcap: Packet capture library. URL `http://www.tcpdump.org`.

Javier Paris, Victor Gulias, and Alberto Valderruten. A high performance erlang TCP/IP stack. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 52–61, New York, NY, USA, 2005. ACM. ISBN 1-59593-066-3. doi: http://doi.acm.org/10.1145/1088361.1088372.

Raid Zaghal and Javed Khan. EFSM/SDL modeling of the original TCP standard (RFC793) and the Congestion Control Mechanism of TCP Reno. URL `http://www.medianet.kent.edu/technicalreports.html`.

Recent Improvements to the McErlang Model Checker *

Clara Benac Earle and Lars-Åke Fredlund
Grupo Babel, Facultad de Informática, Universidad Politécnica de Madrid
{cbenac,lfredlund}@fi.upm.es

Abstract

In this paper we describe a number of recent improvements to the McErlang model checker, including a new source to source translation to enable more Erlang programs to work under McErlang, a methodology for writing properties that can be verified by McErlang, and a combination of simulation and model checking. The latter two features are illustrated by means of the messenger example found in the documentation of the Erlang/OTP distribution.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Reliability

1. Introduction

With the emergence of multicore technology, virtually all software will become concurrent. However, concurrent programming is difficult: parallel algorithms are often poorly understood, resulting in implementations with subtle bugs and high development costs. To combat this complexity, the software industry is starting to adopt rigorous development methods (hitherto mostly used in safety-critical systems and hardware design). One of such methods is model checking, a complementary technique to testing for verifying concurrent systems. Model checking provides the possibility to, in theory, fully verify a system by exploring all its possible executions.

McErlang [2, 10, 13], our model checker for Erlang programs, provides support for virtually the full, rather complex, programming language. The model checker has full Erlang data type support, support for general process communication, node semantics (inter-process communication behaves in a subtly different way from intra-process communication), fault detection and fault tolerance, and crucially can verify programs written using the high-level OTP Erlang component library used by most Erlang programs.

Figure 1 illustrates the differences between a normal Erlang workflow (left) and one using the McErlang model checker (right). The model is the Erlang program to be analyzed, which undergoes a source-to-source translation to prepare the program for running under the model checker. Then the normal Erlang compiler translates the program to either Beam byte code (an Erlang byte code language) or directly to native machine code. Finally the program is run under the McErlang run time system, under the control of a verification algorithm, by the normal Erlang bytecode interpreter. The pure computation part of the code, i.e, code with no side effects, including garbage collection, is executed by the normal Erlang run time system. However, the side effect part is executed under the McErlang run time system which is a complete rewrite in Erlang of the basic process creating, scheduling, communication and fault-handling machinery of Erlang, comprising a significant portion of the code of the model checker.

Naturally, the new run time system offers easy check pointing (capturing the state of all nodes and processes, of the message queues of all processes, and all messages in transit between processes), of the whole program state as a feature (impossible to achieve in the normal Erlang run time system due to the physical distribution of processes).

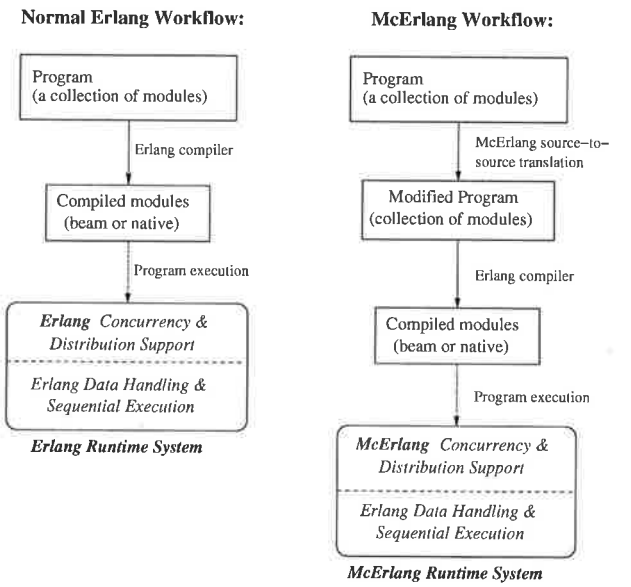

Figure 1. Usage of McErlang

McErlang has been successfully applied in a number of case studies, for example: a resource manager, a Video–on–demand server [12], leader election protocols [13], and RoboCup teams [6].

In this paper we describe a number of new features of McErlang version 1.0, not previously documented in any article. McErlang 1.0 is the first open source version of McErlang; source code licensed under a BSD license can be downloaded from `http://babel.ls.fi.upm.es/trac/McErlang/`.

* This work has been partially supported by the FP7-ICT-2007-1 Objective 1.2. IST number 215868 ProTest project from the European Commission, the DESAFIOS (TIN2006-15660-C02-02) project from the Spanish Ministerio de Educación y Ciencia, and the S-0505/TIC/0407 PROMESAS project from Comunidad Autónoma de Madrid.

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.

The new features include: a new source to source translation needed to enable more Erlang programs to be run under the control of McErlang, a methodology for writing properties that can be verified by McErlang, and a combination of simulation and model checking. The latter two features are illustrated by means of the messenger example found in the Erlang documentation. The use of this case-study is motivated by several reasons: the code is rather familiar for Erlang programmers, it is an example of distributed code, and the example is not overly complex.

The rest of the paper is organized as follows. Related work is presented in Sect. 2. In Sect. 3 we informally describe the source-to-source translation needed for running Erlang code inside McErlang. The messenger case-study is explained in Sect. 4, and Sect. 5 describes the methodology used to write properties and model check them against the case-study. The combination of simulation and model checking is discussed in Sect. 6. Finally Sect. 7 draws some conclusions.

2. Related Work

Software model checking is a very active research field. Thus, we will here only mention a few of the most important related works, and the ones which have provided inspiration for McErlang.

For Erlang the *etomcrl* toolset [4] already provides a model checking capability. In comparison, however, the *etomcrl* toolset supports only a smaller subset of Erlang, for instance lacking the concept of direct process communication, distribution and fault tolerance (i.e. nodes, processes, links, monitors, ...). Other verification tools for Erlang include Huch's abstract interpretation model checker [17] which uses abstract interpretations to reduce the size of the state space. Another work addressing the verification of Erlang programs is the "*Verification of Erlang Programs*" project [11] which uses theorem proving technology. Furthermore there is the QuickCheck tool for Erlang [18, 5], which however primarily focuses on testing software rather than formal verification.

Much of the inspiration for McErlang naturally comes from the work on the SPIN tool [15] and the CADP toolset [9], as they both constitute very capable language based platforms for the verification of software, and for testing new verification algorithms.

The VeriSoft tool by [14] is one of the earlier examples of providing a verification functionality to a real, complex, programming language (such as C or C++) instead of a simpler specification language. Another successful example of such a verification project is the Modex tool [16] which is closely connected to SPIN. A recent work on the verification of complex concurrent program code is the work on model checking file system implementations by [21]. Another recent work is the Zing model checker by [3] which aims at checking concurrent systems. Other interesting tools are Bogor [8] and Java Pathfinder [20]. A key difference between these tools and McErlang is the programming language they analyse, e.g. Java and Erlang. By using Erlang we have all the advantages (and disadvantages) of using a functional programing language for specification and programming (access to higher-order functions, no shared variables, etc.).

In contrast to e.g. the Java Pathfinder tool we do not implement a new byte code interpreter (for Beam, the Erlang byte code format). The chief reason for this is that the Erlang byte code format is much less standardised than Java's; indeed there is no documentation available apart from the source code. In McErlang, rather, programs are run by the normal Erlang byte code interpreter, but have been modified to return control to the model checker after a side effect has occurred.

3. Compiling Erlang Code Using McErlang

To execute Erlang code under the McErlang model checker a translation step is necessary, whereby Erlang code is translated into modified Erlang code suitable for running under the control of the McErlang model checker (which has its proper runtime system for managing processes, communication and nodes). Thereafter, the normal Erlang compiler translates the modified source code into beam (or native) object files.

Recently we have developed a new translator, which in contrast to the earlier translator handles the full Erlang language.[1]

3.1 Translation Phase

The new transformation is implemented on the HiPE Core Erlang format [7]. We let the standard Erlang compiler generate Core Erlang code, which is subjected to a number of transformations, and finally the standard Erlang compiler is again used to generate beam (or native) code from the resulting transformed HiPE Core Erlang code. The input to the McErlang compiler is a list of all the modules that comprise the source code for the project to model check (including, in principle, the source code for OTP behaviours such as `genserver`, `supervisor`, ...).

Below we describe, in order, the different transformations and analyses implemented by the transformation tool:

- *Erlang translated to HiPE Core Erlang*: the Erlang modules that comprise the project are translated into the HiPE Core Erlang format using the standard Erlang compiler.
- *Normalize HiPE Core Erlang code*:

  In McErlang most transformations work on a reduced HiPE Core Erlang language, which is more regular than standard HiPE Core Erlang. As a first step the McErlang translator transforms an arbitrary HiPE Core Erlang module into a normalized HiPE Core Erlang module.

  Simplified, the normalized HiPE Core Expressions e are shown in Fig. 2, where *vars* denotes a sequence of variables, *patterns* a sequence of patterns, i a natural number, and a an atom.

  In essence, the normalized format stratifies expression construction and removes ambiguity. For example, the mixing of let expressions (the variable binding construct) and other expressions is forbidden (so let expressions grow only in the body part). Moreover arguments to function calls may not contain complex expressions.
- *Guard expressions are changed*:

  Since the representation of process identifiers, node identifiers and references differ from the standard representation in Erlang we must replace calls to some standard built-in functions (e.g., `is_pid`) with calls to new functions (e.g. `mcerlang:is_pid`). Unfortunately it is not possible to call these new functions in guards.

  Instead, as an example, in a guard we replace a call to `is_pid` with pattern matching against the new process identifier format {**pid**,nodename::**atom**(),number::**integer**()}. That is, a call `is_pid(Pid)` is translated into the guard fragment:

```
is_tuple(Pid),
size(Pid)=:=3,
element(1,Pid)=:=pid,
is_integer(element(3,Pid))
```

[1] The earlier translator for instance lacked support for the **try** ... **of** ... **catch** ... **end** construct, it could not handle bit strings nor binaries, nor could it translate correctly most higher-order or anonymous functions.

```
e       ::= e_n | let vars =e_n in e

e_n     ::= try e of vars -> e
            catch vars -> e
          | case e_s of clause ... clause
          | receive clause ... clause
            after e_s -> e
          | call e_s:e_s(e_s, ..., e_s)
          | primop a(e_s, ..., e_s)
          | apply e_s(e_s, ..., e_s)
          | fun (vars) -> e
          | letrec
              a/i = fun (vars) -> e,
              ...
              a/i = fun (vars) -> e
            in e

clause  ::= patterns guard -> e

e_s     ::= containing data constructors,
            variables and constants
```

Figure 2. Normalized Core Format

Calls to the functions erlang: **self** /0, erlang: **node**/0, erlang: is_port /1, erlang: is_reference /1, erlang: **node**/1 in guards are translated in a similar fashion.

- *Function calls are remapped*:

  Programs running under the McErlang model checker need to invoke the McErlang application programming interface (e.g., the module mcerlang) instead of the normal Erlang API (e.g., the module erlang). Thus the transformation transforms the code so that e.g. instead of calling the function erlang:send/2 the modified program calls mcerlang:send/2.

  The mapping of function calls, and information regarding which functions in binary modules (for which no source code is available) have side effects, is defined in a configuration file (which may be modified for each McErlang project).

  We will describe the semantics of the transformation configuration file by explaining the excerpt in Fig. 3.

```
[
 {gen_server,
   [{translated_to, mce_erl_gen_server}]},
 {supervisor,
   [{translated_to, mce_erl_supervisor}]},
 {gen_fsm,
   [{translated_to, mce_erl_gen_fsm}]},
 {erlang,
   [{rcv, false}]},
 {{erlang, spawn, 4},
   [rcv, {translated_to, {mcerlang, spawn}}]},
 {{erlang, open_port, 2},
   [blacklisted]},
 ...
]
```

Figure 3. Remapping function calls

The configuration information is represented as a normal Erlang term, and contains a number of commands, on two basic formats:

```
{module, [attribute1, ..., attributeN]}
```

or

```
{{module, functionname, arity},
  [attribute1, ..., attributeN]}
```

As an example, consider the first line in the above specification:

```
{gen_server,
  [{translated_to, mce_erl_gen_server}]}
```

This command maps any call to a function in the gen_server module to a corresponding call in the module mce_erl_gen_server. The command {erlang, [{rcv, false}]} expresses that by default no function in the erlang module will ever execute a receive statement. In the next line we override this default by specifying that indeed erlang:**spawn**/4 (which spawns a function on a specified node) can actually cause a receive statement to be executed, and secondly that any call to it should be mapped to a call to mcerlang:**spawn**/4 instead.

Finally the erlang: open_port/2 function is blacklisted, i.e., any occurrence of that function in compiled code will cause the translator to emit an error message. The reason for blacklisting a function might be that we want to apply a model checking algorithm to the translated program[2]. Since the order of actions in the model checked programs may not be preserved (the model checker interleaves the execution of many alternative execution threads), API calls that cause effects external to the McErlang runtime system (e.g., file I/O) that cannot be easily undone, should not be allowed.

Depending on the particular application requirements, it may be a good idea to use the OTP version of a particular module, or use a (simplified) module that we provide. Ideally, of course, one would in most circumstances like to reuse the real Erlang/OTP module that the program uses.

However there are cases when this is not desirable. An Erlang/OTP module may for instance have a more deterministic behaviour than its documentation permits. Since the module code may later change, we might want to test the code under the more "liberal" behaviour allowed by its documentation (i.e., use an alternative less deterministic module implementation).

Another reason to rewrite a module is to reduce memory usage or improve execution speed for model checking runs. Since one of the factors limiting the efficacy of model checking is memory usage, it may be a good idea to use a module requiring less memory (potentially at the cost of worse performance) when applying a model checking algorithm to a program. For instance, using the orddict module instead of the dict module.

- *Dynamic calls to `apply` are made safe*:

  In practice it is not possible to, at translation time, find all call sites where a function call should be remapped, if the module and function name arguments to HiPE Core Erlang call expressions are computed at runtime. Thus McErlang inserts code to dynamically remap such function calls at runtime.

- *Global analysis to detect side effects*:

  To translate receive expressions, and all the call sites of functions that contain receive expression, a global analysis is performed over all the modules in a project to discover which

[2] McErlang is also capable of simply executing the code, but of course with substantially different process scheduling behaviour compared to the normal Erlang runtime system.

functions directly contain a receive statement (or may possibly invoke another function that cause a receive statement to be executed, recursively).

- *Receive statements are replaced by special return values*:

  Receive statements cannot be executed directly in McErlang, as internally in McErlang communication between (simulated) processes does not use message passing. In fact, McErlang normally runs in a single process, regardless of how many processes are spawned by the simulated code. During translation receive statements are transformed into an expression which returns a special value (a kind of continuation), signaling the intention to execute a receive statement. To ensure that such special return values are not captured by the environment in which they reside, the transformation must also modify the environment. An example:

```
echo() ->
  receive
    {msg, Pid, Msg} -> ok
  end,
  Pid!{echo, Msg}.
```

  If we just transform the receive statement we would lose its special return value:

```
echo() ->
  [[receive
    {msg, Pid, Msg} -> ok
  end]],
  Pid!{echo, Msg}.
```

  (where [[...]] performs the translation of a receive statement). Instead we (conceptually) embed the translated receive statement in a new let construct:

```
echo() ->
  let X =
    [[receive
      {msg, Pid, Msg} -> ok
    end]]
  in Pid!{echo, Msg}.
```

  with the semantics that the argument part of the let construct is executed immediately, and if that part returns a special value, then a special value with the body part as continuation is returned. If the argument part returns a normal expression, then the body part is executed directly.

- *HiPE Core Erlang modules translated to Beam files*: the translation produces one beam file for each Erlang module translated, by applying the standard Erlang compiler to the resulting HiPE Core Erlang code.

As an conclusion of the translation effort, using the HiPE Core Erlang language as the basis for the translation has proven useful, but not without problems. A significant advantage is the fact that the intermediate language has much fewer constructs compared to normal Erlang. Moreover the handling of variables is far cleaner in HiPE Core Erlang: variable binding and variable scope is explicit.

On the negative side the Erlang compiler (up to at least R13B) still contains a few bugs with respect to the compiling of HiPE Core Erlang code. In addition the restriction to use only normal Erlang guard functions also in HiPE Core Erlang is limiting; it would be nice to have support for more expressive guard expressions.

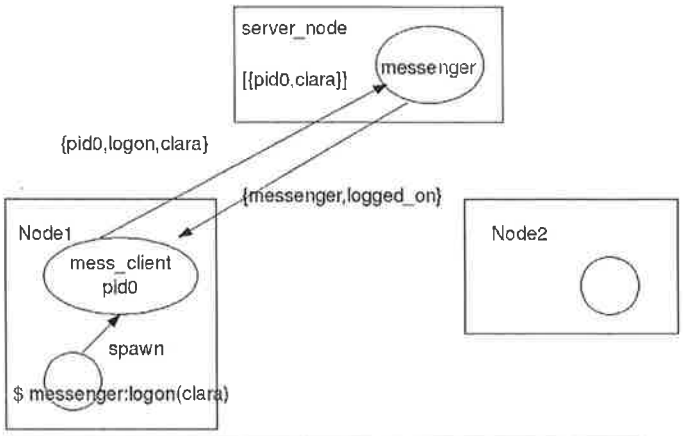

Figure 4. User clara sends a logon message to the messenger server

4. The Messenger Case Study

To illustrate the use of McErlang we consider the messenger example from the "Getting started with Erlang" document in the Erlang/OTP R12B documentation[3].

The messenger is a program which allows users to log in on different nodes and send simple messages to each other. Clients connect to a central server, specifying their identities and locations that are then stored in the state of the server. Thus a user won't need to know the name of the Erlang node where a user is located to send a message to that user, only his identity.

Let us consider the example with three nodes depicted in Fig. 4, the node server_node where the messenger server is already running and two other nodes, Node1 and Node2. If a process in Node1 calls the messenger:logon(clara) command the result of this call is the spawning of a process registered as mess_client that will send a message to the messenger server containing the name of the user and the pid of mess_client. The server will then look into its state, concretely into the list where the identities and locations of users that are already registered are stored, and if the user clara does not occur in this list it will be added to it and a confirmation message will be sent to the mess_client.

Lets now assume that both a user clara and a user fred in Fig. 5 are logged on and clara wants to send a message to fred. Again the message will be sent from the mess_client running in Node1 to the messenger server which, after checking that both clara and fred are logged on, will forward the message to the mess_client running in Node1.

Upon a logoff, the mess_client will send a message to the messenger server that will produce the deletion of the user that wanted to log off from the list of users maintained by the messenger server.

4.1 Executing the Messenger Example Inside McErlang

To run the messenger example inside McErlang we need to implement the start up of the system, and simulate the actions of the (simulated) users of the messenger service. This test-case for using the messenger service is what we call a scenario. There are different ways of generating scenarios. Random scenarios can for instance be generated by using the QuickCheck tool [18, 5].

[3] Section 3.5 in [1].

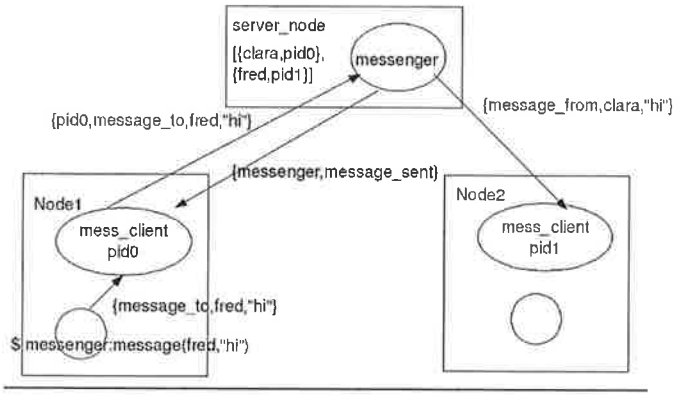

Figure 5. User clara sends a message to user fred

5. Model Checking the Case Study

McErlang is a model checker for programs written in Erlang. The idea is to replace the part of the standard Erlang runtime system that concerns distribution, concurrency and communication with a new runtime system which simulates processes inside the model checker, and which offers easy access to the program state. The labeled transition system generated by the model checker comprises two elements:

- system states which record the state of all nodes, all processes, all message queues, etc of the program to model check. Such states are *stable* in the sense that all processes in a state are either waiting in a receive statement to receive a message, or have just been spawned.
- transitions or computation steps between a source state and a destination state. A transition is labeled by a sequence of actions that represent selecting one process in the source state which is ready to run, and letting it execute until it is again waiting in a receive statement (or has terminated). The actions that label the transition are the side effects caused by the execution of the process (i.e., sending a message to another process, linking to another process, ...).

Thus the model checker uses an interleaving semantics to execute Erlang programs.

McErlang provides a number of different ways to formulate correctness properties for checking on a given program. The alternative we will consider here is to express the desired property in Lineal Temporal Logic (LTL), and to use the LTL2Buchi tool [19] developed by Hans Svensson and distributed with McErlang to automatically translate an LTL formula into a Büchi automaton. Given a program and such an automaton, McErlang will run them in lockstep letting the automaton investigate each new program state generated. If the property does not hold, a counterexample (an execution trace) is generated.

We have model checked several interesting safety and liveness properties of the simple messenger case-study. One such property can be informally expressed as follows: if a user who is logged on sends a message to another user who is also logged on then the recipient of the message will eventually receive the message. As mentioned before, McErlang allows access to the program states and the actions between states. We explain in the following sections how this information can be used to write properties.

5.1 Using Probe Actions

After several attempts, one possible and more precise description of the aforesaid property is the following:

if user1 does not send a message m to user2 until user2 is logged on, then if user1 does send a message m to user2 then eventually user2 receives the message m.

The formalization in LTL of the above formula has the following shape:

$$\neg p \ \mathit{Until}\ q \ \Rightarrow \ (\mathit{Eventually}\ p \ \Rightarrow \ \mathit{Eventually}\ r) \quad (1)$$

where the predicates p, q and r are the following:

- p: user1 sends message m to user2
- q: user2 is logged on
- r: user2 receives the message m from user1

Linear Temporal Logic is defined over program runs: p *Until* q holds for a program run if at every state of the run the p predicate holds, until a state in the program run is encountered where the q predicate holds (and q must hold for some state on the run). *Eventually* r holds for a program run if the predicate r holds at some program state in the run. Normal logical implication is denoted by the "⇒" symbol.

For simplicity and modularity the property and the predicates present in the property are considered separately. To write the predicates or basic facts in the formula (user1 sends a message m to user2, etc.) McErlang allows access to the program states and the sequence of actions labelling transitions between states. These predicates can be written directly in Erlang, using all the expressiveness of the language. For example, a predicate stating that a user is logged on can be implemented as a function "logon" that returns true when an action corresponding to the user being logged on is found labelling a transition. The process of searching for the desired action is simplified if we annotate the program with what we call "probe actions", which serve to make the internal state of a program visible to the model checker in a simple fashion.

In the messenger case-study we have annotated the program code with probe actions that are referred to in the predicates. For example, the following probe action has been added to the client function for expressing that a user is logged on:

```
client(Server_Node, Name) ->
  {messenger, Server_Node}!{self(), logon, Name},
  await_result(),
  mce_erl:probe(logon, Name),
  client(Server_Node).
```

From the example we can see that a probe action, as created using the mce_erl:probe function, has two arguments, corresponding to a label naming the particular probe, and an arbitrary Erlang term as probe argument.

Below we finally define the "logon" predicate which provided a user name as argument, defines an anonymous function that returns true if its second argument is a sequence of actions containing a logon probe action corresponding to a logon by the named user:

```
logon(Name) ->
  fun (_, Actions, _) ->
    lists:any
    (fun (Action) ->
       try
         logon =
           mce_erl_actions:
             get_probe_label(Action),
         Name =
           mce_erl_actions:
             get_probe_term(Action),
         true
       catch _:_ -> false end
```

```
    end, Actions)
  end.
```

Similarly, probe actions and predicates have been written for the other predicates appearing in property (1).

The first property was checked against a several scenarios, for example, a scenario consisting of a user clara that first sends a logon message to the messenger server, then sends the message "hi" to the user fred and finally sends a logoff message, and a user fred that sends a logon message end a logoff message. McErlang immediately reported that a counterexample had been found. Examining the error trace showed that the property did not hold because fred could logoff before receiving the message.

One option to address the problem found is to generate only test cases where fred never logs out. However, we prefer to instead rewrite the property to handle the situation when fred logs out.

Thus we modify the property (1) as follows:

if user1 does not send a message m to user2 until user2 is logged on, then if user1 does send a message m to user2 then eventually user2 receives the message m from user1, or user2 is logged off.

The resulting LTL formula is the following:

$$\neg p \; \mathit{Until} \; q \;\Rightarrow\; (\mathit{Eventually}\; p \;\Rightarrow\; \mathit{Eventually}\; (r \vee s)) \quad (2)$$

where s represents the predicate "fred is logged off".

The property (2) has been checked against several scenarios, returning always a positive result.

5.2 Using Probe States

Working with probe actions in LTL formulas can sometimes be difficult, as we have manually "remember" the occurrence of important actions in the formula. In formulas (1) and (2) above, this was accomplished using the until formula.

Instead of using probe actions we can use so called "probe states". In contrast to probe actions, which are enabled in a single transition step only, such probes are persistent from the point in the execution of the program when they are enabled, until they are explicitly deleted.

As an example we instrument the login code of the server to record, using the function mce_erl: probe_state , the fact that a user has logged in:

```
%%% Server adds a new user to the user list
server_logon(From, Name, User_List) ->
  %% check if logged on anywhere else
  case lists:keymember(Name, 2, User_List) of
    true ->
      From!{messenger,stop,
            user_exists_at_other_node},
      User_List;
    false ->
      From!{messenger, logged_on},
      mce_erl:probe_state({logged_on,Name}),
      [{From, Name}|User_List]
  end.
```

Similarly we delete the probe state using the function mce_erl: del_probe_state when a user logs out:

```
%%% Server deletes a user from the user list
server_logoff(From,User_List) ->
  case lists:keysearch(From,1,User_List) of
    {value,{From,Name}} ->
      mce_erl:del_probe_state
        ({logged_on,Name});
    false ->
      ok
  end,
  lists:keydelete(From,1,User_List).
```

We can test for the existence of a probe state using the function mce_erl: has_probe_state as exemplified in the function logged_on below which checks if a user is logged on:

```
logged_on(Name) ->
  fun (State,_,_) ->
    mce_erl:has_probe_state
      ({logged_on,Name},State)
  end.
```

This predicate will be abbreviated as "t" below.

We can now reformulate the second property above, removing the until operator:

$$\mathit{Always}\; ((p \wedge t) \;\Rightarrow\; \mathit{Eventually}\; (r \vee \neg t)) \quad (3)$$

Note that since t is a state predicate we can safely negate it to compute its logical negation ("the user is not logged on") whereas the negation of the action predicate $\neg q$ in properties (1) and (2) only expresses that the "logon action is not present in the current transition" (but it may have occurred earlier in the execution of the program).

5.2.1 A Comparison of Using Probe Actions Versus Probe States

In a comparison between formulas (2) and (3), clearly formula (3) is easier to understand. In (2) we are forced to *search* for a desirable situation "$\neg p \; \mathit{Until} \; q$" (a message was sent to a logged-in recipient) which is expressed directly in (3): "$p \wedge t$". Moreover the state predicate t can be reused in negated form in formula (3), whereas it cannot in (2) where we instead have to formulate the additional predicate "the recipient logged off" (s). So, it would seem that we gain in clarity by having such probe states.

However, using probe states in a program may incur a cost. That is, the size of the program state graph may increase, since probe states are actually components of the program state (and so the introduction of probe states may cause fewer states to compare equal, leading to larger state spaces).

On the other hand, if we are able to write more compact formulas (which correspond to Büchi automatons with fewer states) by utilising probe states, then the state space explored using model checking may become smaller, since such an "exploration state" is the combination of a program state and a Büchi automaton state.

To provide initial data for a comparison, we have compared the explored state spaces for formulas (2) and (3), on an example with 3 users, and annotating the program with probe states when checking property (3).

The explored state space for formula (2) was 103288 states, whereas for formula (3) it was 179422 states. Clearly a substantial difference; however it can be argued that the formulas express different correctness properties. Supposing we prefix (2) with an "always" operator obtaining:

$$\mathit{Always}\; (\; \neg p \; \mathit{Until} \; q \;\Rightarrow\; (\mathit{Eventually}\; p \;\Rightarrow\; \mathit{Eventually}\; (r \vee s))\;) \quad (4)$$

The resulting formula (4) has 184120 states, i.e., a slight increase in the number of states compared to formula (3).

5.3 Verification methodology

A schema of the verification methodology used in this example is shown in Fig. 6. To summarize, the steps we have followed to check that a program verifies a property are the following:

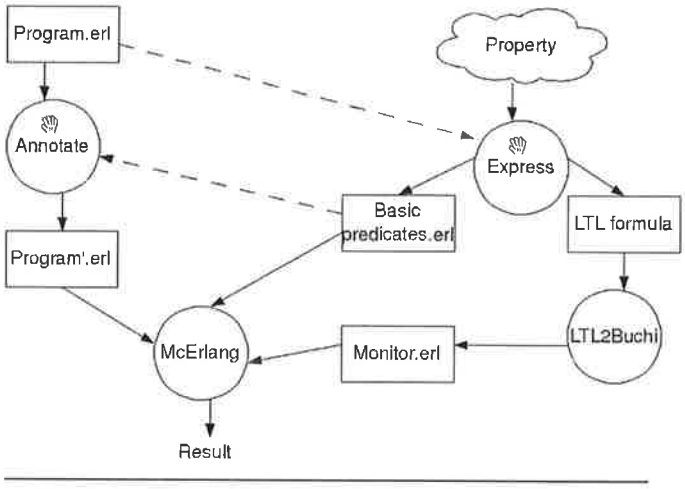

Figure 6. verification schema

- Express the property in a suitable formalism, in this case, an LTL formula with some predicates. This is done by hand.
- Use the LTL2Buchi tool to translate the LTL property to a Büchi automaton implemented as a monitor
- Annotate, by hand, the program with probe actions and/or probe states[4]
- Invoke the McErlang model checker with the annotated program, the predicates, and the monitor.

The result provided by the model checker will be either a positive result if the property holds or a counterexample (an execution trace) if the property does not hold.

6. Combining Model Checking and Simulation

McErlang gives the possibility of combining simulation and model checking algorithms during a verification. The typical usage is to begin by simulating (following only a single program run) the system under test, and then, when an interesting state is seen, seamlessly switching to model checking (considering all program runs).

One reason for combining simulation and model checking is to avoid a state space blow-up during model checking when more complex OTP behaviours are used. An example is the `supervisor` pattern, which is frequently used for describing the hierarchic process structure of a system. Clearly we wish to be able to verify programs making use of the supervisor behaviour, however, in practice the use of the behaviour can greatly increase the state space necessary to traverse during model checking.

One possible solution to the problem is to use a combination of simulation and model checking to shrink the state space: we simulate the system, forbidding (using a custom scheduler) any actions not corresponding to the booting of the system (setting up the hierarchical process structure), until no more booting steps can be taken, at which point we switch to model checking mode. Thus we commence model checking only in a stable system state after the booting-up phase has terminated, potentially greatly reducing the resulting state space, while still allowing the supervisor design pattern to be used.

Although the simple messenger example does not use the supervisor behaviour, we can still benefit from combining simulation and model checking. Concretely we simulate the messenger program up to the point where the only enabled program action is for a user to send a command to a messenger server (thus we are not model checking the start-up of the server_node and the messenger server), and switch to model checking from then on. The resulting reduction in state space is shown in the table below.

We have conducted two experiments, one with two users and one with three users. The property checked is (3), e.g.:

$$Always\ ((p \wedge t) \ \Rightarrow\ Eventually\ (r \vee \neg t)) \quad (3)$$

The column "states (mc)" gives the number of states in the experiment where only the model checking algorithm was used, whereas the column "states (sim+mc)" gives the number of states in the experiment combining simulation and model checking.

number of users	states (mc)	states (sim+mc)
2	3324	912
3	179422	56938

As can be seen in the table, the savings in the size of the state graph obtained from combining simulation and model checking, are quite significant.

7. Conclusions

In this paper we have introduced three new features of McErlang: (a) a source to source translation by means of several transformations on the HiPE Core Erlang format, necessary for an Erlang program to be executed inside McErlang, (b) a methodology for writing properties that can be verified using McErlang, and (c) a combination of simulation and model checking.

Previously we had implemented the source-to-source translation directly on Erlang abstract syntax trees (using the `erl_syntax` library). The experience with using HiPE Core Erlang for the transformation is much more positive: the format is much more regular and compact, and in particular the problem of variable bindings occurring deep in a subexpression has been addressed. Unfortunately the transformation we have developed is not fool-proof; the fundamental problem is that the transformation must potentially modify all the source code of an Erlang system. Since source code is not available for all modules, either because some functions have been implemented in C (e.g., the lists module), or because the module is a part of a commercial software for which no source code is available (e.g., QuickCheck), this presents a problem. Currently we are re-working the compilation scheme of modules, to be able to identify such problems at model checking time (i.e., at runtime).

In this paper we have also described a methodology for writing properties that can be input to the McErlang model checker. A property can be expressed as an LTL formula where the predicates are basic facts occurring in the program itself. Writing predicates in Erlang is aided by annotating the program to analyze. Besides this, the fact that McErlang gives full access to the program states and the actions that occur between states, is used when writing properties. This verification methodology is illustrated by means of the well-know messenger example, where several properties have been checked.

Finally, we have mentioned the possibility of combining simulation and model checking, and presented some preliminary figures from the messenger case study. The figures indicate that for this example, the size of the state graph when using pure model checking is significantly larger than the state graph corresponding to using an approach combining simulation and model checking. It will be

[4] Strictly speaking it is not necessary to annotate a program using probe actions or probe states. McErlang can in principle compute the information directly from inspecting the system state. However, this is often not easy as the value of a variable cannot be conveniently computed (variable names are not preserved by the compilation process).

interesting to compare the state space reduction achieved using this slightly "ad-hoc" method with the state space reductions that can be achieved using a more general partial-order based state generation algorithm.

Acknowledgment

Thanks are due to Hans Svensson for many valuable discussions on matters related to model checking and Erlang semantics in general, and for writing and integrating the Ltl2Buchi tool in McErlang. We also would like to thank the anonymous referees for their valuable comments on a preliminary version of this article.

References

[1] http://erlang.org/doc/getting_started/part_frame.html.

[2] https://babel.ls.fi.upm.es/trac/McErlang/.

[3] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Lecture Notes in Computer Science*, volume Vol. 3114, pages 484 – 487, Jan 2004.

[4] T. Arts, C. Benac Earle, and J. Sánchez Penas. Translating Erlang to mucrl. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD2004)*. IEEE Computer Society Press, June 2004.

[5] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *ACM Sigplan International Erlang Workshop*. ACM Press, 2006.

[6] C. Benac Earle, L. Fredlund, J. Iglesias, and A. Ledezma. Verifying robocup teams. *Electronic Notes in Theoretical Computer Science*, 5348/2009:34–48, 2008.

[7] R. Carlsson. An introduction to Core Erlang. In *Proceedings of the 2001 ACM SIGPLAN Erlang Workshop*, 2001.

[8] M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 148–152. Springer, 2005.

[9] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer, 1996.

[10] L. Fredlund and C. Benac Earle. Model checking Erlang programs: The functional approach. In *ACM Sigplan International Erlang Workshop*, Portland, USA, 2006.

[11] L. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):405 – 420, Aug 2003.

[12] L. Fredlund and J. Sánchez Penas. Model checking a VoD server using McErlang. In *In proceedings of the 2007 Eurocast conference*, Feb 2007.

[13] 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.

[14] P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *Computer Aided Verification*, pages 476–479, 1997.

[15] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[16] G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. Softw. Eng.*, 28(4):364–377, 2002.

[17] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, 1999.

[18] J. Hughes. Quickcheck testing for fun and profit. In *9th International Symposium on Practical Aspects of Declarative Languages*. Springer, 2007.

[19] H. Svensson. Implementing an LTL-to-Büchi translator in Erlang. In *Proceedings of the 2009 ACM SIGPLAN Erlang Workshop*, 2009.

[20] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a Java model checker, 2000.

[21] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Sixth Symposium on Operating Systems Design and Implementation*, pages 273–288. USENIX, 2004.

Author Index