An evaluation of Erlang and C++

A THESIS

By

Jaap Kreijkamp,

* * * * *

Department of Computer and Information Science

Linköpings Universitet

1996

THESIS ABSTRACT

Software development is the largest part of the design costs for Ericsson and other telecommunication oriented organizations. Projects resulting in several millions lines of code are not uncommon.

Reducing the cost of software design, improving quality and cutting time to market are thus of paramount importance and have been the major goals in Computer Science research in Ericsson for many years.

To achieve this goal, Ericsson has developed a new language, Erlang, which promises to overcome most of the major problems of the languages currently used.

The goal of this thesis is to compare Erlang to C++, a language currently often used for large software projects at Ericsson, but not without problems. The comparison is done through the writing of a benchmark application in both languages.

Furthermore a short discussion is given about the limitations and possible improvements on Erlang.

Acknowledgements

I would like to thank the many individuals who have dragged me through the process of creating this work. Without them I would not have gotten this far.

First of all, I wish to pay my gratitude to my supervisor, Bengt Lennartsson, who helped me getting started with the project, and helped me make the thesis to what you see now.

I wish to thank the people of my corridor to accept this strange dutchie, and the Pelab group for their discussions and lunches.

Of course, I want to thank my friends in the Netherlands, for keeping in touch over the electronic highway. Especially Chris and Sam for our endless conversations.

Also many thanks to Sanne and Louise, for getting me away from the computer now and then, you should have come earlier!

And last but not least, many thanks to my family for their support during my stay far far away from them, sorry for the phone bill, mom.

TABLE OF CONTENTS

List of Figures

CHAPTER I

Introduction

1.1 Topic of this thesis

Software development is by far the largest part of the design costs of Ericsson's products[Eri]. For the AXE-10 telephony switching system, about 62 million lines of source code have been written. A typical delivered AXE-10 telephone exchange contains between five and ten million lines of source code.

For a company like Ericsson, reducing cost and development time of software development is of great importance, and is one of the major research areas within the company. This research has resulted in the language *Erlang*.

Part of the key to success for a new language is determined by how well it performs against the regular languages. This has led to the goal of this thesis:

Benchmarking different implementations of typical telecom applications.

Because this goal is open to different interpretations, a more strict one is given below:

Benchmarking of typical telecom applications written in different programming languages, including Erlang.

We had chosen for two other languages, C++ and Orca[Bal91] but because of time constraints the Orca language was dropped. Because of the size of typical telecom applications, the benchmark application is an implementation of a simplified telephone system.

1.2 Background

1.2.1 Erlang

Erlang is a concurrent programming language designed for the prototyping and implementation of reliable soft real-time systems. It was developed at the Computer Science Laboratory, which is part of the research organization of Ericsson.

The concurrency is supported by lightweight processes which can send messages to other processes. The language is declarative, thus it does not have destructive assignment, has pattern matching, garbage collection, tail recursion optimization, and exception handling.

Erlang has been available since 1990.

1.2.2 C++

C++ is a general purpose imperative object oriented language based on C[1].

The key concept in C++ is class which can be considered as a user-defined type. Classes provide data hiding, inheritance, which gives the possibility to extend existing classes, user defined type conversion, and operator overloading.

Pointers play an important role in C++, giving it its power and its problems. Memory management has to be handled by the user.

C++ is developed mainly between 1983 and 1991 for a large part by Bjarne Stroustrup.

1.3 Structure of this thesis

- Chapter II gives a short introduction to telecommunication applications.

[1] Which in turn is derived from B, which in turn is derived from BCPL [Rit93].

- Chapter III gives an introduction to the Erlang language.
- Chapter IV explains the structure of the application designed to use as the benchmark.
- Chapter V describes a small meta-language for writing NFA[2]'s.
- Chapter VI gives some aspects of the Erlang implementation of the benchmark and discusses some of the limitations of the language.
- Chapter VII gives some aspects of the C++ implementation of the benchmark and discusses some of the limitations of the language.
- Chapter VIII gives a comparison of the speed of the Erlang and the C++ implementation of the benchmark.
- Chapter IX gives a final discussion and conclusions.

[2]Nondeterministic Finite Automata

CHAPTER II

Introduction to Telecommunication Applications

The major type of application in the telecommunication world is concerned about making connections between different entities. The most obvious example is connecting two phone-lines to each other by setting the appropriate switches, but you can also think of applications as setting up a connection between the phone-line and an audio-channel on a voice-board for a voice response system running on an ordinary personal computer.

Those kinds of connections are set up in response to receiving certain signals, for example an *off-hook* signal or *dial-3* signal. Which kind of connections are set up depends on what signal is received, where the signal came from, some status information (what signals did we receive already), and specific data (what services are allowed to this user, which switch to set to connect to user with number 3759).

Like with many services providing systems, it could be disastrous if the system went off-line, because of a bug or a change to the software or database even if it needs to be off-line only for a few minutes.

Because telecommunication systems tend to have a (at least for software) very long life span, a large number of users, and requirements which will continue to grow over this time span, the systems have to be scalable.

To summarize the main properties of this kind of applications:

1. large amounts of data, which must be available fast, both for lookup and update

2. many external signals (generated by special hardware)
3. provided services can change (at runtime)
4. robust, a bug in a subcomponent should not crash the whole system under any circumstances
5. scalable, number of users/nodes must be expandable

2.1 Data

It seems that the public telecommunications network is evolving towards a database-driven architecture in which centralized databases are consulted as part of normal call-processing to provide a variety of services. This centralizing of the databases has several advantages over keeping separate databases on several levels, for example it makes it easier to add new services and to make emphnodes mobile [GH88] [NR88].

The database is thus getting a critical component the application is build around. The database tends to be very large and access times must be very short. Data can be changed by services themselves or through a network managing tool, which also should be able change the structure. Also it is very important that the system is fault tolerant and of course consistent [NTW95].

Of course it would not be possible to use a database for the benchmark with a comparable size and functionality as in real applications.

2.2 Signals

The main function of most telecommunication applications is to make a connection between two or more devices. These devices can vary from telephones and computers to voice boards and tone generators.

In the data sent from a device to the system you can make distinction between user data and control signals. User data is the data the user wants to present to the other party, for example voice, email, video or raw computer data. Control signals are used for setting up and closing down connections and also for administrative purposes like billing information.

2.3 Services

The services describe how the application should react on a even given a certain internal state. To make it easier to maintain and extend these services often more abstract (and sometimes graphical) representations are used. The representations are often closely related to finite automata, describing states, and connecting those by arrows representing an event (and thus a state change).

2.4 Fault tolerance

Because the applications often are for providing certain services to clients continuously, down-time of the application could be a serious thread. Most applications must be able to handle most of the cases of hardware failure and failure of parts of the application.

2.5 Scalability

Because telecommunication networks still tend to grow and the applications to control them often have a lifetime for several decades, the software must be able to handle continuous growth in nodes and users, and should also easily be adapted to new hardware technology.

CHAPTER III

An introduction to Erlang

This chapter will only give an introduction to Erlang, many topics will not be explained here. A good introduction to the Erlang language is [JAW93]. In order to not to break a tradition, the first example will be a program that computes the factorial of an integer:

```
-module(mymath).
-export([factorial/1]).

factorial(0) ->
    1;

factorial(N) ->
    N * factorial(N - 1).
```

The `-module(mymath)` is used to give your collection of function name. Such a collection is called a *module* and all Erlang functions belong to some particular *module*. The `-export([factorial/1])` tells which functions are *exported*, which means are visible outside the module. In this case only the function `factorial` is exported and this function has only one argument (as indicated with the `/1`).

After compiling[1] the module the new function can be used from within another module or from the interactive *shell*. The shell prompts for an expression an evaluates any expression the user enters, and gives back the result:

[1] How the code is compiled is beyond the scope of this thesis.

```
> mymath:factorial(7).
5040
>
```

The evaluation of functions is based on pattern matching (like in Prolog, but Erlang does not have backtracking). When evaluating `mymath:factorial(7)` all the defining clauses of factorial are tried in the order of declaration, until the arguments matches. If it does not match any of the clauses a runtime error is generated. In the case above first `factorial(0)` is tried, but rejected because $7 \neq 0$. The next clause does match, because `N` is an unbound variable, no value has been assigned to it yet.

When a match has been found the right-hand side of the '`->`' is evaluated and any variables occurring in the the function definition are substituted in the right-hand side of the clause before evaluation. In the above example the variable '`N`' will get the value '7' and after substitution `7 * factorial(7 - 1)` is evaluated.

Some common operations, or operations impossible to make yourself are provided by Erlang itself. These functions are called *build-in functions* or *BIF*s. For example `length([12, 6, abc]` gives `3`.

3.1 Data Types

The Erlang types can be divided in two main groups:

simple data types[2] these types cannot be split into more primitive subtypes:

- **Number** which can be *integer* (`23`, `-5`, ...) or *float* (`3.1415`, `-1.1e11`, ...).
- **Atom** which is nothing more than a label or constant name, for example: `foo`, `dogs`, `'An atom'`.
- **Process Identifier** is a special type for identifying processes. More about this in section 3.2.
- **Reference** is a special type (and outside the scope of this thesis).
- **Binary** is a special type (and outside the scope of this thesis).

compound data types which can be used to group together other data types:

- **Tuple** for example: `{age, 25}`, `{}`, `{1, {3, 4}}`. Tuples are used for storing a *fixed* number of items.
- **List** for example: `[]`, `[1, 2, a]`. Lists are used for storing a *variable* number of items. Tuples are used for storing a fixed number of items.
- **Record** which is a special kind of a tuple.

The values of Erlang data types can be stored in *variables*. A variable in Erlang always starts with an upper-case letter, for example: `A`, `DataBase`. An example of some assignments would be:

```
User = 'Jaap Kreykamp',
FirstMonth = jan,
Salary = getsalary(User),
Account = {User, [{FirstMonth, [100*2, -50, 70, Salary]},
                  {feb, [Salary]},
                  {mar, [333-tax(Salary), Salary]}]}.
```

3.2 Concurrency

One of the main features of Erlang is its *concurrency*. This means that activities which run in parallel to each other can be programmed in Erlang without the need of special libraries.

A parallel activity or *process* can be seen as a program running on its own virtual computer, having its own private memory and its own flow of execution. Because it is often necessary to communicate to those processes, information can be exchanged by sending *messages* to processes.

A process is created with the BIF spawn and the process identifier (*Pid*) of the new process is returned. Messages can be send to a process with `Pid ! Msg`, where `Pid` is the process identifier of the destination process, and `Msg` is the data to be send to the process and can be of any type. Example:

```
get_pid('left_wheel') ! {self(), {speed, 75}}
```

3.3 A large example

In this section a larger example is given, a simple bank accounting system. The user can start the server, and when it is running, ask the current amount on an account, deposit and withdraw money.

```
-module(bank). % module name

-export([start/0, ask/1, deposit/2, withdraw/2, server/1]). % visible functions

% to start the server
start() ->
    register(bank, spawn(bank, server, [[]])).

% asks current amount on account of User
ask(User) ->
     request({ask, User}).

% adds X to amount on account of User (create if not exist)
deposit(User, X) ->
     request({deposit, User, X}).

% tries to withdraw X, can not become negative
withdraw(User, X) ->
     request({get, User, X}).

% does the request to the server, and waits for the reply
request(Msg) ->
    {bank, node()} ! {self(), Msg},
    receive
        {bank, Reply} ->
            Reply
    end.


server(Data) -> % Data is a list of User, Amount pairs
    receive
        {From, {deposit, User, X}} ->
            case lists:keysearch(User, 1, Data) of
                false ->
```

```
                To ! {bank, ok}, % send acknowledge
                server([{User, X} | Data]);
            {_, Y} ->
                To ! {bank, ok}, % send acknowledge
                % remove old value
                Data1 = lists:keydelete(User, 1, Data),
                % call server again with updated user added to list
                server([{User, Y+X} | Data])
        end;

    {From, {ask, User}} ->
        case lists:keysearch(User, 1, Data) of
            false ->
                To ! {bank, not_found}; % send failure
            {_, X} ->
                To ! {bank, X}, % send amount
        end,
        server(Data); % data hasn't change

    {From, {withdraw, User, X}} ->
        case lists:keysearch(User, 1, Data) of
            false ->
                To ! {bank, not_found}, % send failure
                server(Data); % data hasn't change
            {_, Y} when X =< Y ->
                To ! {bank, ok}, % send acknowledge
                % remove old value
                Data1 = lists:keydelete(User, 1, Data),
                % call server again with updated user added to list
                server([{User, Y-X} | Data]);
            _ ->
                To ! {bank, no}, % not enough on account
                server(Data) % data hasn't changed
        end
end.
```

CHAPTER IV

Structure of the benchmark application

In this chapter the design of the benchmark application is explained, including a short description of the first design and the problems with it.

4.1 The type of application

The first part of the project was to decide which kind of application would be the most suitable for the benchmark. Different scenarios were possible, for example an abstract benchmark[1] which tries to handle events and calculations in about the same proportions as a real telecommunication application.

The problem with those kinds of benchmarks is that it is impossible to come up with a good, language independent, estimation of the event and program-flow, even when you have a good profiler. Also such a benchmark would be nothing more than an abstract set of functions, which is not very interesting, outside maybe the results.

Another possible application would be an application from an arbitrary domain, but with comparable properties as seen in applications in the telecommunication world. For example maybe an agent based application would fit this model. But the question then arises is why not choosing an application from the telecommunications world itself.

This led to the choice of writing a simplified telephone service which should have at least most of the properties a real telephone service has.

[1] A benchmark which measures some basic operations, like sending messages

Because the design was mainly focused around Erlang implementation the description below uses the Erlang terminology, and some details slightly differ in the C++ implementation (for example processes are implemented as threads[Sun95b], and paths are represented as strings).

4.2 The old design

The first model was though lack of experience with Erlang not easy to implement in Erlang because of some properties of messages in the language. Only a partial implementation in Erlang exists. Because some of the ideas in this model are taken to the new model, a short explanation of this old model is given in this section.

Even through the benchmark is only a simplified telephone control system, this model is quite complex and has been split in several components, which in turn often consists of several sub-components. The components were:

Device Driver The connection between the application and the emulated switching hardware.

Phone DB The database containing the mapping between telephone numbers and telephone hardware addresses used by the device driver. The database process is also responsible for creating and deleting phone processes.

User DB The database containing user information, including the phone number. The database process is also responsible for creating and deleting user processes.

Phone A process representing an active telephone (ringing, off hook, ...). There can be several instances of these phone processes and they are created and deleted dynamically.

User A process representing an active user (calling, being called, ...). There can be several instances of these user processes and they are created and deleted dynamically.

Analyze The process which is used to analyze a phone number and finding, if the number is correct and complete, the associated entry in the user database.

Connection A connection process represents the connection between two (or more in the case of conference calling) users. It sets up and removes the physical connection through the device driver. Like the user and the phone processes, these connection processes are created and removed dynamically.

Figure 1 is an attempt to visualize the structure.

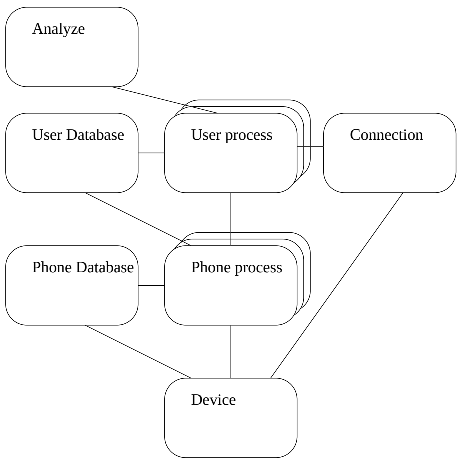


Figure 1: The structure of the original model

The main problem with this model was that messages were send to processes directly, using their process id. Because the receiving process could terminate before getting the message this model could lead to lost messages. The problem of losing messages is described in more detail in 6.3.5.

4.3 The new model

In the new model there is an abstract layer, from now on called the base layer, where addressing of processes, sending and receiving of messages, and many other basic functions are handled. This way the application components themselves do not have to worry about complex tasks as resending lost messages, and keeping track of processes. The main properties of the new model are:

- Hierarchical layout of processes and name space.
- Addressing of processes independent of process identifiers.
- Automatic acknowledgment of messages.
- Recovery facilities.

The hierarchical structure of the name space is quite similar to the Unix file structure. A path is stored as a list of names, with two special names, '/' and '..' representing the root process and the parent process respectively.

By using this hierarchical name-space for addressing processes the underlying process id's is hidden and a process does not have to be aware if a process is already running when sending a message to it or that the process will be activated by it.

By making the layout hierarchical every process has a parent (except the root process) recovery of a crashed process can be made easier by making the parent process responsible for it. All children of an exited (or crashed) process are killed automatically so there will be no orphans[2].

Messages are acknowledged. The system is designed to guarantee delivery of messages with two exceptions:

- An incorrect path is given, an error message is then given back.
- A process crash results in the sender process going down (by propagating downwards the crash, or the sender is a (grand) child of the crashed process).

4.4 Layout

As stated above the new model has a hierarchical layout. A visualization of the structure is given in figure 2.

[2]Processes without a parent process.

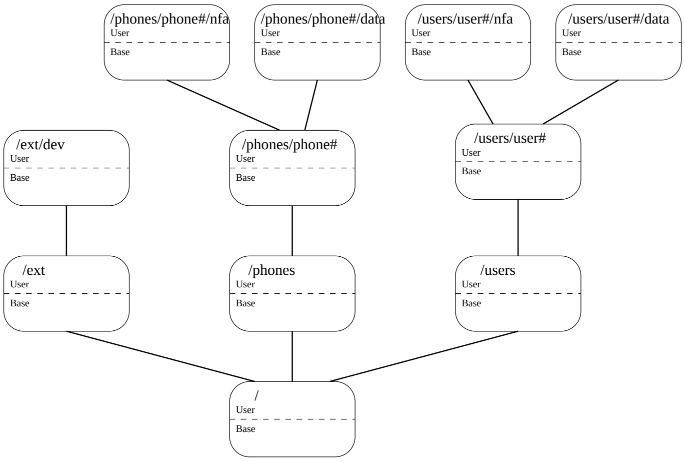


Figure 2: The structure of the new model

4.4.1 The root process

The only function of the root process is to start up its children and do recovery if necessary.

4.4.2 The users process

The users process is the process which maintains the user database and makes user processes available when they are needed. The users database contains phone number user data pairs. The users process does not have knowledge about the contents of the user data. It is up to the user process to process it.

4.4.3 The user processes

The user processes are spawned by the users process and given as arguments its phone number and the user data. The user process starts a small database process (as child) to process queries on the user data. A user process also spawns a NFA process which is a state-machine for processing the events.

4.4.4 The phones process

The phones process is comparable to the users process and its function is to start up the phone processes. In this model a user can have multiple phones and phones can be shared by multiple users.

4.4.5 The phone processes

The phone processes are spawned by the phones process and are given the hardware address of the phone and the phone-data. A phone process also starts up a small database process and a NFA process.

4.4.6 The external process

The external process manages the device process and the connections process.

4.4.7 The device process

The device process is the connection between the hardware and the application. In the benchmark case it does not connect to real hardware but can be used to simulate events like *phone on address 0x23 off-hook*.

4.4.8 The connections process

The connections process is the process which maintains all the connections between users. It does this through a connection process for every connection.

4.4.9 The connection processes

The connection process sets up the physical connection through the device process and disconnects after one of the user processes quits. By creating a process for every connection there is no need for managing tables of connections and thus keeping the code simpler.

4.4.10 The user database

The user database is the most complex database in the application. This because it has a special query to check to which phones a call from a certain user should be forwarded, making it possible to connect certain clients to a special phone without the need for a private number. The main reason for this complexity was to add more database complexity to the application. The attributes which can determine the list of phones a call can be forwarded to are *time*, *day*, and *phone-number*.

4.4.11 The phone database

The phone database is much simpler than the user database. At the moment it only contains the phone number of the phone.

4.4.12 The user and phone NFA's

The different internal states a telephone call can be in is best described by a finite state machine. To meet the requirement of abstraction of services and making it easier to map such a final state machine to actual Erlang or C++ code a simple language was designed which is closely related to NFA's. This language will be described in chapter V.

CHAPTER V

The NFA language

The purpose of the NFA, Nondeterministic Finite Automata, is to provide a higher level abstraction of NFA's than Erlang or C++. The language is declarative, partly to make it easier implement it in Erlang and also because it does fit well in the language. The language does not have global variables, and unlike Erlang the variables are explicitly typed.

5.1 Flow of control

The language does not have loop constructs, but there is an *if* statement to give the language more power and to be able to make state transitions depending on internal state.

The transition between two states is done by using the `goto` statement. There is also a `gosub` and `return` statement which can be used to implement NFA's within a NFA and to exit from a final state in the NFA.

5.2 Types

The major difference between the NFA language and Erlang is that the NFA language is typed. This gives the compiler the possibly to do type-checking at compile time, and in the authors opinion, makes the code more readable.

The events send and received by the NFA are also typed. For every event name, there can be only one definition, thus there is no polymorphisme. This makes it possible to determine the types of the *out* variables. The same holds for rules and functions.

The types supported are:

boolean which can only have the values `true` and `false`.

number which theoretically can contain any integer number but is in the C++ version limited to `long`.

string which can contain an arbitrary string.

address which is used for the *send* statement to give the destination address.

unknown on which no operations are allowed and can only be used to pass data through.

list can contain an arbitrary number of elements of the same type. The type of the elements must be given by definition.

record which can contain a fixed number of elements of different types. The types of the elements must be given by definition.

5.3 Events

The communication between the NFA and the rest of the application is done by *events*[1]. All the events a NFA can send and receive must be defined before the rules.

Events are send by the *send* statement, which requires a destination address and an event.

Events are received by using the *when* statement, which is followed by a list of accepted events. An event is accepted if the name matches and all arguments, except

[1]It can be done also by using external functions, but this should be mainly used to get static information like the address of a user process given its phone number.

the out variables matches. When an event matches all out variables are bound and the accompanying block is executed.

A special event, `*`, can be used to match any event.

CHAPTER VI

The Erlang implementation of the benchmark

This chapter describes the global structure of the Erlang implementation, and discusses some of the limitations of the current Erlang implementation.

6.1 Global structure

Because the design described in chapter IV was made with Erlang in mind, the Erlang implementation follows it quite accurately. The benchmark can be split into three parts:

- The NFA compiler, which is written as a separate program which generates Erlang modules which then can be compiled by the Erlang compiler and loaded in the same way as ordinary Erlang code.
- The base layer, which handles process management and message passing.
- The processes, which implement the functionality (together with the NFA's) of the application.

6.2 The base layer

Every conceptual process in the model exists as two Erlang processes. The first process is the *base process*, which handles all the message sending and receiving, the addressing of processes and recovery. The other process is the *user process*, implementing the functionality of the component.

6.2.1 The sending mechanism

The sending mechanism of messages is the most complex part of the Erlang implementation, and works as follows (see also figure 3):

- The user calls send(Path, Msg)*(a)* and gets back a message id*(b)*.
- If the Path is relative; it is converted to an absolute path.
- A packet is formed containing the new destination path, the absolute source path, a unique identifier, the Pid of the source base process and the message data, and is stored in the send queue.
- When all previous messages are delivered, the message is forwarded to the first node on the path*(c)*, but not removed from the queue, to make resending possible if something goes wrong. The node to forward to is found by checking the destination path against the path table, taking the path from it which matches best[1].
- When the message is delivered to the destination node*(d)*, the message is given to the user process of the node*(e)*, and an acknowledge is directly sent to the original sender*(f)*. The source path and id are added to the path table, to improve performance.
- The acknowledge is given back to the original sender process*(g)*, and the accompanying path and pid are stored in the path table. The message is now removed from the send queue.
- When a message can not be delivered because of a non-existing node, the parent of this node is asked to create it. If it succeeds to create the node the forwarding is continued, otherwise a failure is sent back to the original sender.

Some details have been left out in the description above, for example all forwarded messages are acknowledged to the forwarding process, and exiting nodes are detected and possibly restarted if they are on the destination path.

[1] `"/foo/bar"` is a better match against `"/foo/bar/cow"` than `"/foo"`, and `"/foo/brick"` does not match at all.

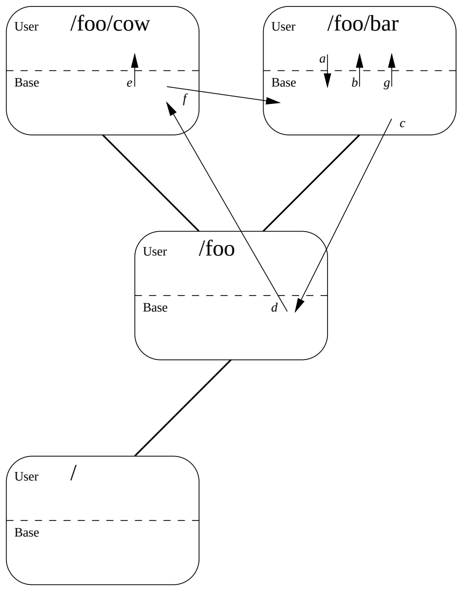


Figure 3: Sending a message

The main reason for this complexity is to guarantee messages are not lost when a process quits during the delivering. Also the model hides the process id's (which can change because of quitting and restarting) from the user.

6.3 Limitations of Erlang

In this section some limitations of the current version of Erlang are discussed.

6.3.1 Types

One of the most obvious limitations of Erlang especially by large projects is the lack of explicit typing. This means that variables and arguments are not declared with the type of the value it may contain. The main problem this gives is that the compiler cannot check at compile-time if the right type of values or arguments are given. This often results in that a typing error would not be discovered until runtime and then it can sometimes be quite hard to locate the original source of error.

The plus side of having no types is that writing reasonably small programs is faster because the programmer does not have to bother with defining types and declaring variables. This makes Erlang ideal for prototyping.

To solve the problem of typing at least partly, a type checker is currently being build at Glasgow University[MW].

6.3.2 Side effects

Although the Erlang book[JAW93] lets the reader believe that *"Erlang has a declarative syntax and is largely free from side-effects"*, the use of processes, which have their own state is a main cause of side effects: many other processes can send messages to a process and thereby changing the state of this process and the replies send by it.

This mechanism is a bit comparable to objects in an object oriented language. The process only lets the user change its state through a given interface. By being able to control what users are allowed to change to the state of an object, enough abstraction is possible to keep the design clear.

Thus this is not necessarily a bad attribute, on the contrary, it is what gives Erlang its power but it is not very different from operations on objects where the Pid is the pointer to the object and registered processes can be compared to global (pointers to) objects.

6.3.3 Object references

One of the advantages of Erlang is having no pointers and therefore does not have the source of errors associated with them. This can sometimes be a disadvantage too. In certain cases a data-structure needs to have multiple references to the same element. Without pointers this is not easy to implement and results often in very time expensive code to access the needed element. For example building a fast hash-list with more than one possible key often leads to keeping several duplicates, one for every key, which leads to very expensive updates (every copy has to be updated).

As described in the section 6.3.2 pointers can be emulated through processes of course at the cost of having a process per object pointed to which, even through processes are lightweight in Erlang, this is far more costly than a real pointer in C++.

6.3.4 Memory leakage

Although the programmer does not have to take care of memory management, there is a problem which can lead to memory leakage, which eventually can cause the whole system to crash (the runtime system in Erlang v. 4.3 terminates when it is not able to allocate enough memory). This is when messages are send to a process which are never removed from its mailbox (when there is no matching clause in the receive body). This is a serious problem which does exist in real world applications written in Erlang[Ben].

6.3.5 Losing messages

When processes are dynamically created and deleted, it is possible to loose messages. This can lead to deadlocks or inconsistent internal states of processes. The problem arises when a process exits after a message is send to it, but before it is actually received. An example of such a case is shown in figure 4

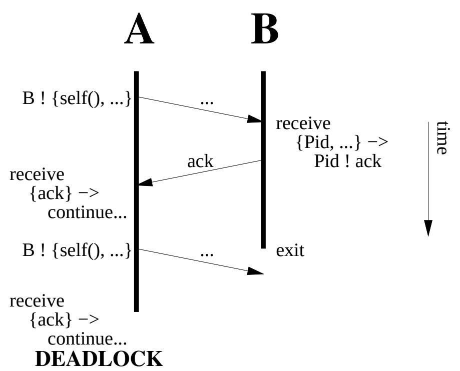


Figure 4: A lost acknowledgment message

A solution to this would be that when a process terminates the mailbox is given back to all processes linked to the terminating process as an argument in the EXIT message. The supervising process then can re-send the messages in the mailbox after restarting the exited process. This method could accumulate the problem described in section 6.3.4.

6.3.6 Control flow

Erlang has not the possibility to return from a function before the end of the function has reached. This together with the lack of a normal *if*-statement can lead sometimes less readable code by getting several levels of indentation. This can sometimes be solved by adding guards in the function header but in many cases this is not possible. It is good practise to keep functions as short as possible any way thus in practise this is not much of a problem.

CHAPTER VII

The C++ implementation of the benchmark

Because of the many possible sources of errors when writing in C++, it can be useful to adapt the programming environment to the problem by making some basic classes and functions. Also when writing multi-threaded code, it is good practise to keep the thread control and synchronization localized in as few classes as possible[1].

The following sections describe the global structure of the application. Also some of the fundamental problems of C++ and an approach for solving them is given.

7.1 Global structure

Following the model as described in chapter IV, and like the Erlang implementation, the application can be split into three parts:

- The NFA compiler, which is written as a separate program which generates shared libraries which can be loaded at runtime in the benchmark (making it possible to update the NFA's without having to stop the application).
- The base layer, which handle process management, message passing and memory management.
- The processes, which implement the functionality (together with the NFA's) of the application.

[1]Especially when the debugger is not thread aware

7.2 Class structure

The class structure can be split also in three categories, basic types, low-level classes and user classes. This is described in the following sections.

7.2.1 Basic types

The characteristic of a basic type is that it is a type or class which is general enough to be used outside this (type of) application.

At the moment the only basic type aside of the build in types such as int and float provided in this implementation is the string class. That this is indeed a basic type can be shown by that it is also used in the nfa-compiler, which does not use the low-level and user classes (mainly because most of it has been written before the class structure was designed).

Lists and other container types are not considered as basic types, because this would require templates which are not supported completely by all C++ compilers, and often leads to a significant growth of executable size.

7.2.2 Low-level classes

Low-level classes are the classes needed to implement the user classes and handle garbage collection, process management etc.

7.2.3 User classes

The classes which implement the application are the user classes. They have some kind of build in garbage collection (see 7.4) and type operations (see 7.3).

7.3 Dynamic types

One of the features missing in C++ is being able to lookup and compare types of objects. Often a pointer to an object of base class A is also used to point to objects of classes derived from class A. One of the major sources of errors is introduced by casting such a pointer to the wrong derived class and then using the new pointer to access members which does not exist in the object.

To make the ability to compare types really useful not only the user should be able to see if two types are equal, but also if and object is a subclass or a superclass.

7.3.1 Properties of types

The inheritance of classes can be described as a DAG (Directed Acyclic Graph)[2], where class `A` is greater thar class `B` when `A` is a subclass of `B`. The ordering of the classes in an inheritance graph is transitive but non-reflexive. This means that when $a < b$ and $b < c$ then also $a < c$, but $\neg(a < b)$ is not equal to $(a \geq b)$.

7.3.2 Operation on types

The only operation on types are compare operations. It is defined the following way:

- `Type1 < Type2` $\Leftrightarrow$ `Type1` is a superclass of `Type2`
- `Type1 > Type2` $\Leftrightarrow$ `Type1` is a subclass of `Type2`
- `Type1 == Type2` $\Leftrightarrow$ `Type1` is the same type (class) as `Type2`
- `Type1 != Type2` $\Leftrightarrow$ `Type1` is not the same type (class) as `Type2`

Example:

[2]It is really a set of DAG's but in this implementation all classes share the same base class, which provides the type functions.

```
Class A
Class B : A
Class C : A
Class D : A
Class E : B, C
Class F : D

A < E   A is a superclass of E
F > A   F is a subclass of A
C >= A  C is a subclass or the same class as A
B == B  B is the same class as B
A != B  A is not the same class as B
```

7.3.3 Types as first class data-type

There are many reasons to avoid the use of templates in C++. The main reason that not all C++ implementations implement them completely and if they do often not very efficiently. Also the size of the binary grows enormously when using templates because most compilers generates for every new template instance with a different type, the code for this template with the type filled in.

The need for templates diminishes when types can be first class data-types[3]. In the implementation described below this is possible with one limitation, that the variable is not copied. What this implies is described in "Implementation".

An example of a generic list:

```
// list.h

class list : public object // object is superclass of all objects
{
public:

    // This defines a function
    // "virtual const types& type() const"
    // which gives back (a reference to) the type of this class.
```

[3]Thus that you can assign it to a variable.

```
    // It also defines a static variable "Type" which contains
    // the type of this class.
    DEFINE_TYPE();

    list(const types& t); // create a list with elements of type t
    int add(object* elt); // add element

    ...
private:

    const types& type_; // type of elements (must be reference)
};

// list.cc

// Initiate (static) var list::Type
// Because you can not define the same macro with different number
// of arguments, the macro has the form:
// IMPLEMENT_TYPE<n>(class-name, super1, ..., supern)
IMPLEMENT_TYPE1(list, object); // to initiate (static) var list::Type

list::list(const types& t) : type_(t)
{
    ...
}

int list::add(object* elt)
{
    if ((elt -> type()) >= type_) // true -> right type
    {
        // ok to add
        ...
        return 1; // ok
    }
    else
    {
        // error wrong type!
        cerr << ... << (elt -> type()) << ...
        ...
        return 0; // error
    }
}
```

...

The major limitation is that it limits the amount of compile-time type-checking which can be done (and which you will have using real templates). This is a major limitation because the error will not be discovered until the type-checking code is executed.

7.3.4 Implementation

The type model is implemented through the *types* class and some macros to make it easier to use the types system in new classes. For performance and memory usage reasons the type information is stored as a static class variable which is defined in every class using the type system.

Every class using the type-system also has a virtual function to ask a reference to the type of this class. A reference is given to prevent copying as a type is constant any way.

An instance of the types class stores the name of the class it represents and pointers to all the types of the direct super-classes of this class.

To compare if two types are the same or not is done by just checking if the type objects are the same object (by looking at the addresses of the objects), this is also the reason why you can not make copies of the type-objects.

To check if `Type1 > Type2` (`Type1` is a subclass of `Type2`) is done through a recursive algorithm:

If there is a direct superclass `S` for `Type1` such that `S.Type >= Type2` then `Type1 > Type2` is `true` else `Type1 > Type2` is `false`.

with `Type1 >= Type2` defined as `(Type1 == Type2) || (Type1 > Type2)`. Here use is made of the order of evaluation as defined in C++, first `Type1 == Type2` is evaluated, and only if this is not `true`, `Type1 > Type2` is evaluated.

`Type1 < Type2` is defined as `Type2 > Type1` and `Type1 <= Type2` as `(Type1 == Type2) || (Type2 > Type1)`.

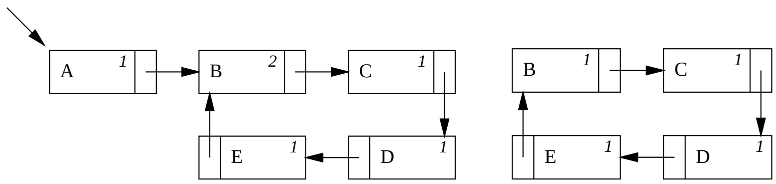
7.4 Garbage collection

Because of the limitation of C++ that you can not store a derived class in a variable of its base class[4] you often are forced to use pointers. This can lead to problems when different pointers point to the same object for example whose responsibility it is to delete the object.

In C++ there is no build-in facility for garbage collection (like in Erlang) so it is the responsibility of the programmer to clean up dereffered objects.

The solution used by the author is by counting the number of references to the object[Lin92]. The major drawbacks of the solution are that extra care should be taken by circular structures, returns and the programmer has to explicitly refer or De-refer pointers.

7.4.1 Circular structures



(a) Before unref A (b) After unref A

Figure 5: Circular reference

[4]when `class B` is derived from `class A`, it is not possible to do `A x = B();`, but it is possible to do `A* x = new B();`.

As shown in figure 5 it is possible to get dangling objects if there are cycles in the pointer graph (thus following pointers you can get back to the object you started from).

The solution is to encapsulate those cyclic structures in a container class, a class used to store a collection of objects in (for example a list). These classes can be aware of the cyclic structure. For example the nodes, which contains the pointers to the objects and pointers to other nodes, can be build as a simple struct, not using references (and types). Because the nodes are private to the container object, it is not very difficult to keep track of them.

7.4.2 Returning objects

Consider the following code:

```
...

mything* foo()
{
    mything* p = REF(new mything(...)); // create new object

    ... // do something with object

    return DEREF(p); // end of p's scope, but refcount becomes 0!
}

int main()
{
    mything* o = REF(foo());

    ...
}
```

The problem here is that the object first get dereferred before the new reference is made. In the current implementation this is solved by not deleting the object immediately when the reference counter becomes 0, but to put the object in a to be deleted list. This list is then periodically checked and all objects which reference counter still is 0 is then deleted.

Checking the to-be-deleted list is done by calling the update function in the environment class. This is done in the base layer whenever a user process uses base-layer functions (to send or receive for example).

7.4.3 Explicit usage

For the problem that the programmer has to explicitly reference or dereference pointers is no solution, except keeping the model as simple as possible. This is done by by providing two macros[5] REF(o) and DEREF(o), with as argument the object which should be referred to or dereferred.

[5] Actually three macros but the third one, NOREF(o), is not really necessary and could be written as DEREF(REF(o))

CHAPTER VIII

Benchmark results

In this chapter some benchmarking results are presented and usability of them are discussed. Because of lack of time, only the standard Erlang System is used, version 4.3, much better results can be obtained by using the JAM or BEAM version.

8.1 Measurements

Because of some persistent bugs in the C++ version only a very limited test has been done. the test was: calling 100 times the same telephone number, and as soon the phone on the other side starts ringing, replacing the hand set. This is done by replacing the device driver by a test driver which generates the events in reaction to the received events.

The testing has been done on a Sun Ultra Sparc 1.

The Erlang version handled the 100 calls in 3898 ms, the C++ version did take 5080 ms for the 100 calls.

8.2 Discussion

The results were quite surprising, because the normal version of Erlang is used, an optimized version would probably give even better results. But before making the conclusion Erlang is faster than C++ some points should be considered first.

It is the author's opinion that benchmarks have a lower value than sales people often wants us to believe. Often very different results can be obtained by changing small things in the test set and test environment. This often leads to results the designer, or the one he is employed by, wants it to show.

In the benchmark described here some other problems play a major role. First, the two languages are very different, which makes it hard to write a benchmark for it which does not give one an advantage over the other.

Also, to get more fair results both programs should be developed separately, preferably by people with experience programming in the particular language. The author had no experience programming in Erlang, but he had in C++.

Because of time-constraints only one general design was made, more with Erlang in mind than C++, making it questionable if it is the best design for it. Also because of the size[1] of the benchmark programs not much effort has been taken to optimize the code, which could lead to major speed improvements.

[1] To give an idea of the size of the applications: the C version is more that 13 KLOC, the Erlang version more than 5 KLOC, and the NFA code, which is shared by both, about 700 LOC.

CHAPTER IX

Conclusions

As stated in the previous chapter, the results of the benchmark are questionable. But even without practical benchmark results some conclusions can be made about using Erlang instead of C++.

As described in the previous chapters it is possible to set up a framework to give C++ some of the properties Erlang has, this does not make coding much easier, it is easy to make errors, and without a multi-threaded debugger it can be sometimes hard to find them. Only unless speed is important or operations or very complex data-structures (with circularities) Erlang will be the better choice for writing Telecommunication Applications.

9.1 Personal experiences

It was quite surprising to find out the Erlang version was faster than the C++ version. Both implementations were written in about the same time, but the C++ version is about three times larger than the Erlang version. Also important to note is that the C++ version still contains a memory leak.

Maybe with a few more days of coding the C++ version could be faster than the Erlang one, but this will probably not without great effort. Also the memory leak should be fixed to be able to do more elaborate benchmarking.

The conclusion made by the author is indeed that Erlang seems to be more suitable for this kind of application than C++. Some reasons why Erlang would be a better than C++ in telecommunication applications is given in the following section.

9.2 Why to choose Erlang over C++

The major strengths of Erlang is its ease of use, its robustness, its concurrency, its pattern matching, and its high-level data-types (tuples and lists). These qualities make it a good language for prototyping (because of its easiness of use and compactness), control tasks and fault tolerant applications.

In large applications, however it is the authors opinion that an optimal design combines Erlang code with C++ code, using C++ for areas where Erlang would be awkward or too slow, like database engines, low-level drivers (where often bit-field and shift operations are needed) and maybe to add specialized modules to Erlang.

So instead of trying to 'prove' Erlang is better than C++ it would be better to look for new and better ways to let them cooperate. Because Erlang is too small to become a mainstream language on its own and could use some of the power and support C++ has, and although C++ maybe a flexible language it misses many of the nice properties and ease of use of Erlang.

9.3 How to get better integration between Erlang and C++

Thus a major development for Erlang could be in making it easier to integrate C++ and Erlang code. It could be done for example by adding the capability to Erlang to dynamically link of C++ libraries at runtime[1].

This would give the user the capability to add new low-level functions (even build-in functions) without having to recompile the Erlang runtime system. To be able to do this, a function interface should be defined on how the library can add new functions and the internal representation of the Erlang data structures should be well defined and documented.

The model described above would make it possible to connect C++ to Erlang without the use of slow and limited mechanisms as sockets.

[1] This means loading new C++ libraries into Erlang while running[Sun95a].

9.4 Further work

The application code contains several loose ends and parts which can be much improved. The author's opinion however, is that time can be spend better on improving the interface between Erlang and C++, for example as discussed in 9.3.

For a more representative benchmark it could be worthwhile looking at [San96], porting the application described here to C++.

APPENDIX A

Grammar Summary of NFA

```
program
    : PROGRAM ID ';' export typedefs imports eventdefs rules
;

export
    : EXPORT ID ';'
;

typedefs
    : /* empty */
    | typedef typedefs
;

typedef
    : TYPEDEF type ID ';'
;

type
    : TYPE_NUMBER
    | TYPE_BOOLEAN
    | TYPE_STRING
    | TYPE_ADDRESS
    | TYPE_PROCESS
    | TYPE_UNKNOWN
    | LISTOF type
    | '{' types '}'
    | ID
;
```

```
imports
    :
    | IMPORT ID '{' fundefs '}' imports
;

fundefs
    :
    | fundef fundefs
;

fundef
    : FUNCTION ID '(' types ')' ';'
    | FUNCTION ID '(' types ')' ASSIGN type ';'
;

eventdefs
    :
    | eventdef eventdefs
;

eventdef
    : EVENT ID '(' types ')' ';'
;

types
    :
    | type types2
;

types2
    :
    | ',' type types2
;


rules
    :
    | rule rules
;

rule
    : RULE ID '(' args ')' body
```

```
;

args
    :
    | arg args2
;

args2
    :
    | ',' arg args2
;

arg
    : type ID
;

body
    : '{' typedefs eventdefs actions events_or_jump '}'
;

actions
    :
    | action actions
;

action
    : expr '!' send_event ';'
    | expr ASSIGN ID ';'
    | expr ';'
    | IF expr body
;

events_or_jump
    : WHEN '{' events '}'
    | jump
;

events
    :
    | event events
    | any
```

```
;

event
    : ID '(' items ')' body
;

any
    : '*' body
;

items
    :
    | item items2
;

items2
    :
    | ',' item items2
;

item
    : expr
    | ASSIGN ID
;

jump
    : RETURN ';'
    | GOTO call ';'
    | GOSUB call ';' jump
;

call
    : ID '(' exprs ')'
;

send_event
    : ID '(' exprs ')'
;

exprs
    :
    | expr exprs2
```

```
;

exprs2
    :
    | ',' expr exprs2
;

expr
    : or_expr
;

or_expr
    : and_expr
    | or_expr OR and_expr
;

and_expr
    : equality_expr
    | and_expr AND equality_expr
;

equality_expr
    : relational_expr
    | equality_expr '=' relational_expr
    | equality_expr NE relational_expr
;

relational_expr
    : additive_expr
    | relational_expr '<' additive_expr
    | relational_expr LE additive_expr
    | relational_expr '>' additive_expr
    | relational_expr GE additive_expr
;

additive_expr
    : multiplicative_expr
    | additive_expr '+' multiplicative_expr
    | additive_expr '-' multiplicative_expr
;

multiplicative_expr
```

```
    : unary_expr
    | multiplicative_expr '*' unary_expr
    | multiplicative_expr '/' unary_expr
;

unary_expr
    : postfix_expr
    | '+' unary_expr
    | '-' unary_expr
    | '?' unary_expr
    | NOT unary_expr
;

postfix_expr
    : primary_expr
    | postfix_expr '[' expr ']'
    | postfix_expr '.' NUMBER
;

primary_expr
    : ID '(' exprs ')'
    | ID
    | NUMBER
    | STRING
    | BOOLEAN
    | LISTOF type '[' exprs ']'
    | '{' exprs '}'
    | '(' expr ')'
;
```

Bibliography

[Bal91] Henri E. Bal. *Programming Distributed Systems.* Prentice Hall International, 1991. ISBN 0-13-722083-9.

[Ben] *The Use of Functional Languages in Industry, The Erlang Experience.*

[Eri] *Erlang Overview.*

[GH88] G. Gopal and G. Herman. Toward a database-driven network. In *IEEE Global Telecommunications Conference & Exhibition* [IEE88].

[IEE88] IEEE Communications Siciety. *Communications for the Information Age*, volume 1, November 1988.

[JAW93] Robert Virding Joe Armstrong and Mike Williams. *Concurrent Programming in ERLANG.* Prentice Hall International (UK) Ltd., 1993. ISBN 0-13-285792-8.

[Lin92] Mark A. Linton. *InterViews Reference Manual*, 1992.

[MW] Simon Marlow and Philip Wadler. Erltc: A type checker for erlang.

[NR88] T. E. Newman and R. B. Robrock. Customer control of intelligent network services. In *IEEE Global Telecommunications Conference & Exhibition* [IEE88].

[NTW95] H. Nilsson, T. Törnquist, and C. Wikström. *Amnesia, a distributed telecommunications DBMS*, 1995.

[Rit93] Dennis M. Ritchie. The Development of the C Language. *Second History of Programming Languages conference*, 1993.

[San96] Martin Sandberg. An integrated approach to system implementation. Master's thesis, Linköping University, 1996.

[Sun95a] Sun Microsystems. *Linker and Libraries Guide*, 1995. Solaris 2.5 Software Developer AnswerBook.

[Sun95b] Sun Microsystems. *Multithreaded Programming Guide*, 1995. Solaris 2.5 Software Developer AnswerBook.