wxErlang - a GUI library for Erlang

Mats-Ola Persson
Computing science dept.
University of Gothenburg
November 2, 2005
Abstract

This report describes the Erlang GUI library wxErlang, it explains both how a programmer can use the library to write GUI applications and how the library itself has been implemented.

WxErlang is a binding to the cross-platform, industrial-strength C++ library wxWidgets. It enables Erlang programmers to easily write cross-platform GUI applications that adopt the native look-and-feel of the underlying platform.

Erlang, for which wxErlang is implemented, is a concurrent functional programming language. WxWidgets on the other hand is implemented in C++ which is a mainstream object oriented language. There are many conceptual differences between these languages and this report describes how these differences have been resolved.
Abstract

WxErlang är ett bibliotek som används för att skriva program med grafiska gränssnitt i Erlang. WxErlang är en s.k. binding till C++ biblioteket wxWidgets, som man använder för att skriva applikationer med grafiska gränssnitt i C++. Bindningen gör det möjligt för Erlang programmeraren att komma åt den funktionalitet som wxWidgets tillhandahåller och kan därigenom skapa applikationer med grafiska gränssnitt.

Acknowledgements

First and foremost, I would like to thank my supervisors Professor John Hughes at Chalmers University of Technology and Dan Gudmundsen at Ericsson AB for their help and their guidance throughout this project.

The developers of wxWidgets should also be thanked for their effort put in to the wxWidgets library and Daan Leijen for his great paper on wxHaskell.

A special thanks to Therese Nordqvist, Anders Carlsson and Per Wismen for their helpful comments, and ideas about the report.
Contents

1 Introduction 4

2 Erlang 6
   2.1 Sequential Erlang programs ................................................. 6
   2.2 Concurrent Erlang Programs .................................................. 8
   2.3 Distributed Erlang Programs ................................................. 10
   2.4 Error Handling in Erlang ..................................................... 10
       2.4.1 Other mechanisms ....................................................... 12

3 Using wxErlang 13
   3.1 Event driven programming ................................................... 13
   3.2 Hello wxErlang! .................................................................. 13
   3.3 Etop ................................................................................. 14
       3.3.1 The code view window .................................................. 17
       3.3.2 Sizers ..................................................................... 19
       3.3.3 The process view window ............................................. 20
       3.3.4 The popup menu .......................................................... 21
       3.3.5 Events and event handling .......................................... 22
       3.3.6 Wrap up .................................................................. 24

4 Design 25
   4.1 Object orientation ............................................................... 25
   4.2 Safety .............................................................................. 26
   4.3 Memory management ........................................................... 27
   4.4 Functions ....................................................................... 27
       4.4.1 Optional arguments ..................................................... 27
       4.4.2 Multiple return values ............................................... 28
   4.5 Overlaps ........................................................................ 29
   4.6 Constants ....................................................................... 29
       4.6.1 Programming with multiple processes ................. 29

5 Implementation 31
   5.1 Bindings in Erlang ............................................................... 31
       5.1.1 Ports ..................................................................... 31
   5.2 The different components of wxErlang ............................. 32
   5.3 Getting Erlang to talk with C++ ........................................... 32
       5.3.1 Wrapper functions and interface functions ............. 33
       5.3.2 Events .................................................................. 36
   5.4 Code generation ................................................................. 36
       5.4.1 Simple example .......................................................... 37
       5.4.2 Annotated functions .................................................... 38
       5.4.3 Automatic code generation ...................................... 39
       5.4.4 Constants ................................................................ 40
       5.4.5 Other statements ....................................................... 40
Chapter 1

Introduction

An ideal graphical user interface (GUI) library combines ease of use with efficiency, portability and a lot of standard functionality. Unfortunately, not many such GUI libraries exist. Writing such a library is no real problem, but one should not underestimate the work required for maintaining such a library. Many promising libraries have had a great start but have then become unusable or outdated due to insufficient maintenance. To avoid this problem one can either make sure that the library is developed by a large active community, or one can build an abstraction upon an existing successful and well-maintained library.

WxErlang is a library built upon wxWidgets[30] which is a free industrial-strength cross-platform GUI framework written in C++. WxWidgets provides a common interface to native widgets (graphical components) on all major GUI platforms, including Windows, GTK+, and Mac OS X. Not only can one use wxWidgets on different platforms, the programs also adopt the native look-and-feel of the underlying platform, enabling programmers to write, compile and run programs on a number of different platforms with little or no changes to the source code.

Erlang[14], for which wxErlang is written, is a concurrent programming language designed for handling problems of massively concurrent nature like telecommunications systems, web servers, etc. Erlang addresses many problems that arise in this category of applications and include mechanisms for coping with: creation/destruction of processes, message-passing, fault-tolerance, distributed programming, real-time response times, hot code upgrades, etc.

Erlang was developed at Ericsson AB, and was originally more or less a dialect of Prolog that added constructs for concurrency. Over time it has drifted from the logic to the functional school of programming. It now includes features that can be found in most modern functional languages, such as higher order functions, list comprehensions, pattern matching and so on. However, a few legacies from Prolog remain, e.g. syntax and a dynamic type system.

WxErlang enables Erlang programmers to combine the powerful features of Erlang with the functionality provided by wxWidgets. This mixture adds up to an environment where the Erlang programmer can easily implement GUI applications with a native look and feel.

Building wxErlang would not have been a problem if wxWidgets itself would have been written in Erlang. Unfortunately, this is not the case as wxWidgets is written in C++. In order to expose the functionality of wxWidgets to Erlang a so called binding between C++ (wxWidgets) and Erlang has been created. A binding is a connection between two languages, enabling the languages to communicate with each other, or in this case, enabling Erlang to access C++ and wxWidgets.

How the communication itself works in practice is not really interesting. But what is interesting is how the different concepts and mechanisms in C++ are translated to Erlang. How these translations should be done is not entirely evident, since Erlang and C++ are two very different languages. Ideally
such a translation should smoothly incorporate wxWidgets into the Erlang environment, avoiding the introduction of previously foreign concepts to Erlang.

This report will present the wxErlang library, both from a programmers (user of the library) point of view and from a more detailed point of view, investigating the implementation of the library and describing the different parts of it.

The reader is not required to have any previous knowledge of Erlang or writing GUI programs. In later sections that concern implementational details, the reader is assumed to have some basic understanding of C and C++ programming.
Chapter 2

Erlang

This chapter will give a tour of the programming language Erlang and its features. It will give the readers a basic understanding of Erlang and enable them to understand and follow examples shown in later chapters.

No previous knowledge about Erlang is assumed nor required to read this chapter. Readers already familiar with Erlang may skip this chapter or just quickly skim through it.

Much of the text in this chapter has been shamelessly stolen from [2] and only minor changes has been made to it. A more exhaustive tutorial can be found in [3].

2.1 Sequential Erlang programs

An Erlang program consists of a number of modules, each consisting of a number of function declarations. Let us start with a little example that gives us an idea about what an Erlang program looks like. The following example is an implementation of the module math, which contains and exports the function fac.

\[-\text{module(math).}
\text{-export([fac/1]).}\]

\begin{verbatim}
fac(N) when N > 0 -> N * fac(N - 1);
fac(0) -> 1.
\end{verbatim}

This can be evaluated in the Erlang shell as follows:

\begin{verbatim}
> math:fac(25).
15511210043330985984000000
\end{verbatim}

The \texttt{fac} function is defined using two clauses, when the function is called Erlang will look at its argument and determine which of the clauses to use. If the argument is greater than zero the first clause will be evaluated. If the argument is equal to zero then the second clause is evaluated.

In general when a function is called, Erlang will try to find a clause that matches the input to the function and then evaluate its corresponding code\(^1\). This way of writing functions makes the code both very easy to read and write.

The annotation \texttt{-module(Name)} denotes the module name and in \texttt{-export(Funs)}, \texttt{Funs} is a list of functions which this module exports. In our case we only export one function: \texttt{fac/1} where \texttt{fac} is the name of the function and \texttt{/1} tells how many arguments the function accepts.

Functions that are not declared in the export list cannot be called from outside the module. In following examples we might omit the \texttt{-module} and \texttt{-export} directives when these are of no interest.

\(^{1}\)If no matching clause is found Erlang will generate an error.
A more interesting and complex example might be a function to search for a value in a binary tree. We represent the nodes of the tree with tuples of the form \( \{ \text{Key, Value, S, B} \} \). The \( S \) element of the tuple are trees smaller than \( \text{Key} \) and the \( B \) element of the tuple are trees with keys bigger or equal to \( \text{Key} \).

A leaf corresponds to the value \text{nil}.

\[
\text{lookup}(\text{Key}, \{ \text{Key, Val, S, B} \}) \rightarrow \\
\{ \text{ok, Val} \};
\]
\[
\text{lookup}(\text{Key}, \{ \text{Key1, Val, S, B} \}) \text{ when } \text{Key} < \text{Key1} \rightarrow \\
\text{lookup}(\text{Key}, \text{S});
\]
\[
\text{lookup}(\text{Key}, \{ \text{Key1, Val, S, B} \}) \rightarrow \% \text{Key} \geq \text{Key1} \\
\text{lookup}(\text{Key}, \text{B});
\]
\[
\text{lookup}(\text{Key}, \text{nil}) \rightarrow \\
\text{not-found}.
\]

This function, like the \text{fac} function, is defined in a number of clauses. For instance, the first clause is executed when the second argument is a tuple with four elements and the first element of the tuple is the same as the first argument given to the function, or in other words we have found a matching node. This clause simply returns the tuple \{ \text{ok, Value} \}.

The next two clauses compare the keys and navigate through the tree. The last clause is executed when we have found a leaf.

Typically, when an argument is not used, like \text{Key} in the last clause of \text{lookup}. Instead of writing \text{Key} one usually writes _ or _\text{Key} to denote that the variable is not in use.

Lists are written as in Prolog - here follows examples of the familiar functions \text{append} and \text{member}:

\[
\text{append}([H|T], \text{L}) \rightarrow [H|\text{append}(T,\text{L})];
\]
\[
\text{append}([], \text{L}) \rightarrow \text{L}.
\]

\([H|T]\) denotes a list with \( H \) as its head (the first element) and \( T \) its tail (the rest of the list). [] denotes the empty list.

\[
\text{member}(\text{H}, [\text{H}|\_]) \rightarrow \text{true};
\]
\[
\text{member}(\text{H}, [\_|\text{T}]) \rightarrow \text{member}(\text{H}, \text{T});
\]
\[
\text{member}(\_, [], \_)) \rightarrow \text{false}.
\]

The Erlang primitive data types are:

- **Atoms** - true, foo, bar, 'Hello Joe'
- **Integers** - 1232345212345, 242
- **Floats** - 3.141592, 4.2413
- **Pids** - process identifiers
- **Refs** - unique identifiers
- **Funs** - functions

Complex data objects are represented as:

- **Tuples** - for storing a fixed number of objects, thus: \{a, 1234\}
- **Lists** - for storing a variable number of objects, thus: [a, ffe, 1234, 3.14]

Functions are introduced with the syntax:

\[
\text{fun}(\text{Arg1}, \ldots, \text{ArgN}) \rightarrow \ldots \text{end}.
\]

So, for example, the following sequence of expressions:
K = 2,
F = fun(X) -> X * K end,

creates a functional object X * K which is bound to the variable F. In this expression X is a free variable and K is bound to the integer 2. To call F one simply states F(Arg). As with many functional programming languages functions are first class values, e.g. one can pass them to functions just as any other value.

To illustrate this, suppose we define the higher-order function adder as follows:

adder(C) -> fun(X) -> X + C end.

Evaluating, for example, adder(10) returns a function which adds 10 to its input. Another example, familiar to functional programmers is the map function, which we can define as follows:

map(F, [H|T]) -> [F(H)|map(T)];
map(F, []) -> [].

Then we can simply combine the adder" function with map:
map(adder(10), [1,2,3,4,5]).

this will yield [11,12,13,14,15] as result. If one does not have a functional programming background, one might not immediately how realize powerful higher-order functions are, but they are extremely useful and a fundamental concept in functional programming languages[12].

2.2 Concurrent Erlang Programs

Erlang is a concurrent programming language — parallel activities can be programmed in Erlang itself and do not make use of any concurrency mechanisms in the underlying operating system. New processes are created with the spawn primitive. Processes can communicate with each other by sending messages back and forth. To send a message, one uses the !-operator (pronounced bang) and receive is used to accept messages.

A call to spawn(M, F, [A1, .. An]) creates a new parallel process that executes M:F(A1,..An), where M is a module and F one of its exported functions. When F terminates the process dies.

Let us consider a rather simple example:

-module(echo).
-export([start/0, loop/0]).

start() -> spawn(echo, loop, []).

loop() ->
    receive
        {From, Message} ->
            From ! Message,
            loop()
    end.

When start is called, a new process will be created that enters the loop where it waits for messages of the form {From, Message}. When a message is received it sends back the Message part of the original message.

The module echo might be used as follows:

... 
Pid = echo:start(),
Pid ! {self(), hello},
...
This will create a process and send the message {self(), hello} to it (the function self() returns the process identifier of the current process). Note that receive is a pattern matching operation; the syntax

```
receive
  Message1 -> ... ;
  Message2 -> ... ;
  ....
end
```

means "try to receive a message described by one of the patterns Message1, Message2, ..." The process evaluating this primitive is suspended until a message which matches one of the patterns is received. If a match occurs the code after the '->' is evaluated.

Message sending is non-blocking with "send and pray" semantics; the sending process does not wait for the message to be received and there is no way to check whether it was delivered.

Let us consider a slightly bigger example that clarifies concurrency primitives in Erlang a bit more. We look at a typical example from introductory books on object oriented programming, namely how to implement a "bank",

```
start() ->
  register(bank_server,
    spawn(bank_server,
      server, [[]]).

server(Data) ->
  receive
    {From, {ask, Who}} ->
      reply(From, lookup(Who, Data)),
      server(Data);
    {From, {deposit, Who, Amount}} ->
      reply(From, ok),
      server(deposit(Who, Amount, Data));
    {From, {withdraw, Who, Amount}} ->
      case lookup(Who, Data) of
        undefined ->
          reply(From, no),
          server(Data);
        Balance
          when Balance >= Amount ->
            reply(From, ok),
            server(deposit(Who, -Amount, Data));
        _ ->
          reply(From, no),
          server(Data)
      end
  end.

reply(To, X) -> To ! {bank_server, X}.
```

% definitions of lookup and deposit omitted

Calling bank_server:start() creates a global process called bank_server², e.g. one can send messages

²This is done by the call to register.
to it using bank_server ! Message instead of Pid ! Message. The Data variable contains information about the bank’s customers and their balance etc.

2.3 Distributed Erlang Programs

Erlang applications are often designed to run on distributed networks of processors. We use the term node to mean a complete self-contained Erlang system. In a distributed network one or more Erlang nodes may run on a single computer.

Distributed programming has very much in common with ”normal” non-distributed concurrent programming, the only difference is that in a distributed environment processes may be running on different nodes. This fact Erlang has taken in to account, and all the primitives involving processes are network transparent. For instance, Pid = spawn(N, M, F, Args), creates a process on the erlang node N. All operations on the process works exactly as if it were a local process.

2.4 Error Handling in Erlang

Erlang has three mechanisms for trapping run-time errors, which can be used to:

- Monitor the evaluation of an expression.
- Monitor the behavior of a process.
- Raise an exception when an undefined function is called

Suppose we evaluate N/M where M is equal to zero. This will cause a run-time error and the process in which this evaluation is done will die.

We can detect this error in one of two ways. The first method makes use of the primitives catch and throw which are used for monitoring the evaluation of an expression. The expression X = catch N/M evaluates to the tuple {‘EXIT’, What} where What is an error descriptor, in this particular case: {badarith, {erl_eval, eval_op, [’/,’1,0]}}. The expression catch Expr evaluates to Expr if no error occurs. The catch primitive is more or less a place where one can execute ”unsafe” code, that might be erroneous.

The primitive throw(Expr) works as follows: first Expr is evaluated, then its value is caught by the nearest catch-clause. If there is no catch-clause, then a no-catch error will thrown.

Using catch and throw, programs can be written to monitor and correct local errors within a process.

The second error handling mechanism determines what happens when a process dies, i.e terminates or generates an error (that is not caught by the catch mechanism described above). In Erlang one can use the built-in function link to see why a process terminates. If the process Pid1 calls link(Pid2) then a link is established between these two processes. If one of these processes dies, it broadcasts a message containing information about why it died to all processes that are linked to it. Hence, if Pid1 dies, then Pid2 is sent a message, saying that Pid1 died and why. The link is symmetric, i.e if Pid2 dies then an error message is sent to Pid1.

Next we consider an example that shows how this mechanism works. Consider the function control(M, F, Args). When it is executed it starts two processes. One of these evaluates the expression M:F(Args) and the other monitors the process performing the evaluation. If the process performing the evaluation dies, a new process is created by the monitoring process.

-module(controllant).
-export([control/3, start_controllant/3]).

control(M, F, A) ->
  spawn(monitor, start_controllant, [M, F, A]).

start_controllant(Mod, Func, Args) ->
  process_flag(trap_exit, true),
  restart(Mod, Func, Args).
restart(Mod, Func, Args) ->
    Pid = spawn_link(Mod, Func, Args),
    receive
        {'EXIT', Pid, Why} ->
            restart(Mod, Func, Args)
    end.

By default, a process does not receive exit messages from other processes that it is linked to. To enable a process access to the exit messages one needs to call the function `process_flag(trap_exit, true)`. The function `spawn_link` creates a new parallel process and creates a link\(^3\).

As the matter of fact, if Erlang’s standard error handling mechanism does not suit your needs, you can customize the error handling mechanism. The following example shows a sample implementation of such a customization.

Suppose that one calls `M:F(Args)` and no code for module `M` has been loaded into the system, then we can tailor our system to call the function `undefined_function(M,F,Args)` in the module `error_handler`. A typical definition might be:

```
-module(error_handler).
-export([undefined_function/3]).

undefined_function(Mod, Func, Args) ->
    case code:is_loaded(Mod) of
        {file, File} ->
            exit({undef_fun, ....});
        false ->
            case code:load_file(Mod) of
                {module, _} ->
                    apply(Mod, Func, Args);
                {error, _} ->
                    exit({undef_mod, ...})
            end
    end.
```

This example is not that interesting by itself, but what is interesting is the possibility to customize the error handling behavior. One can even use this as a way to control how code is loaded into the system.

Since [2] was written a new mechanism for trapping errors has been introduced. Many felt that the `catch` and `throw` mechanism was not flexible enough, and a new one has been introduced, namely the `try-catch-end`. We won’t go into great detail why this mechanism is needed, the interested reader is referred to [5]. The `try-catch-end` works almost exactly as the corresponding `try-catch-finally` statement in Java and C++. A generic `try-catch-end` is on the form:

```
try
    Body
catch
    Err1 -> ...;
    Err2 -> ...;
    ...
    ...
    ErrN -> ...;
```

\(^3\)This is done atomically.
Its semantics are very much a functional interpretation of Java’s try-catch-finally statement: execute the Body, if an error occurs evaluate and return the value of the first clause that, by pattern matching, matches the error. If no error occurs during evaluation of Body then the value of it is returned.

2.4.1 Other mechanisms

In this section we will briefly mention a few other constructs in Erlang that have not previously been described and are used in wxErlang.

Macros

Erlang has a construct called macro. A macro is very similar to a function, but differs on a few points. A macro is like an alias for a piece of code and every occurrence of the the macro is replaced during compile time with its corresponding piece of code. Consider the following example which creates the macro TRUE and associates it with 1.

```
-define(TRUE, 1).
```

To use a macro one uses the syntax ?TRUE. One can also define macros that accept arguments, for instance, one can define a debug macro in the following way.

```
-define(DEBUG(Value), io:format("*** debug : ~p", [Value])).
```

To use it one states ?DEBUG(SomeValue).

Records

Another handy construct is records. A record is a data structure for storing a fixed number of elements, like tuples. But records and tuples differ on one point, records use named fields, whereas tuples do not. With tuples, one needs to know at what position in the tuple a value is, in a record one needs to know the field’s name to access its value. To define a record one uses the following syntax.

```
-record(person, { name, surname, age, occupation }).
```

This will define the record person that has the fields name, surname, age and occupation. The following code snippet will create a new instance of the record person and assign it to P.

```
P = #person { name = "Homer",
   surname = "Simpson",
   age = 38,
   occupation = "Nuclear Safety Inspector, sector 7G" }.
```

To extract a value from P, for instance name, one evaluates P#person.name.
Chapter 3

Using wxErlang

In this chapter a number of examples will be presented together with their code. The different concepts of wxErlang and GUI programming in general will be briefly introduced. No prior knowledge of wxErlang, wxWidgets nor GUI programming is required.

3.1 Event driven programming

Before we start looking at wxErlang, we will describe GUI programming in general and define the notion of event driven programming.

Programming GUI applications is often synonymous with writing programs in an event driven (signal driven) way. The nature of event driven programs differs from that of traditional ones, where the application itself defines in which order things can happen, i.e. the flow of the application is defined by the application itself.

The flow of an event driven program, however, is largely generated by external events. Typically an event driven program consists of a number of small programs a number of event handlers and a dispatcher. When an event occurs, the dispatcher receives the events and then executes the appropriate event handler.

This way of programming closely relates to programming in Erlang. A commonly used approach in Erlang is to have server processes which are waiting for different messages. When a message is received the process will perform some computation, much like the dispatcher triggers an event handler upon receiving an event.

3.2 Hello wxErlang!

Let’s start off by considering a simple example, just to get the feeling of what a program looks like. The following program shows a frame with a text label centered above a button, like the one shown in figure 3.1. Pressing the button will hide the window.

![Figure 3.1: A simple wxErlang application.](image)

```erlang
start() ->
  wx:start(),
  Frame = wx:frame(?NULL, ?wxID_ANY, "Example"),
```
Label = wx:static_text(Frame, ?wxID_ANY, "Hello wxErlang!"),
Ok = wx:button(Frame, ?wxID_ANY, [{label, "Ok"}]),
wx:connect(Ok, ?wxEVT_COMMAND_BUTTON_CLICKED,
    fun(_, _) ->
        wx:hide(Frame)
    end),
VSizer = wx:box_sizer(?wxVERTICAL),
wx:add(VSizer, Label, [{flag, ?wxEXPAND bor ?wxALIGN_CENTER}]),
wx:add_spacer(VSizer, 5),
wx:add(VSizer, Ok, [{flag, ?wxALIGN_CENTER}]),
wx:set_sizer(Frame, VSizer),
wx:set_size_hints(VSizer, Frame),
wx:show(Frame),
ok.

First we call wx:start; we do this to initialize the graphical subsystem and start an event loop. All programs using wxErlang will have to make (at least) one call to this function.

The call to wx:frame(?NULL, ?wxID_ANY, "Example") creates a top level window frame, with the text "Example" written in the title bar. Inside the frame we create a text label and a clickable button. This is done by the calls to wx:static_text(Frame, ?wxID_ANY, "Hello wxErlang!") and wx:button(Frame, ?wxID_ANY, [{label, "Ok"}]). The label will have the text "Hello wxErlang" and the button will have the text "Ok".

The call to wx:connect will register an event-handler for the event triggered by button clicks. The button is said to be listening for clicks. When this event occurs the main window frame will be hidden.

Finally, the layout of the window is specified. In wxErlang we use something called sizers to model how widgets should be placed on a display area. In this case, the label will be placed above the button and both the label and button will be placed in the center of the frame. We will take a closer look at sizers later.

3.3 Etop

The following example is perhaps a more realistic case of programming with wxErlang. We show how to (re-)implement the program etop using wxErlang. Etop is pretty much an Erlang version of the unix command top embedded inside a nice GUI. The command top shows the currently running processes of a unix system. Etop shows the currently running Erlang processes on an Erlang node.

We will first look at what the program looks like and how it works, then implement selected parts of the program. Figure 3.2 shows the main window of the application. From here we can easily get an overview of the system’s current state. The window’s title bar shows the node’s name, the boxes just below show some general information about the node, i.e. used memory and total number of processes, etc. The big table (grid) at the bottom shows information about each individual process, like its pid, memory usage, etc. Exactly what the information means is another story which we will leave untold. We are only interested in how etop presents the information and, it enables us to manipulate it and so on.

Like most GUI programs, etop listens to different events generated by the user, for example mouse clicks. If for instance a user right clicks on a process in the table, a menu, like the one shown in 3.3 will appear. This menu will present a number of actions that the user can perform on the selected process.

The user can choose from View code, View process information and Kill.
Figure 3.2: The main window of etop running on Mac OS X.
If the user would select View code then the program would open a new window. This window contains a text field that shows the source code of the currently running function, under the premise that the code could be located. If on the other hand, the code could not be located, this menu item would have been grayed out (disabled), rendering the menu item not selectable.

The View process information will open another window containing some additional process information, like what processes it is linked to, whether it traps exit signals, etc. Figure 3.4 shows what the window looks like.

The Kill menu item holds a slightly risky functionality. It enables the user to kill any currently running process, regardless of whether it is one crucial for the system or not. If the user selects Kill etop will prompt the user with a yes/no question, asking the user to confirm that he really wants to kill the selected process.

The program also has a menu bar, but it is not shown in the figures. This comes from the fact that under Mac OS X, under which the screen shots are taken, the menu bar is not presented in the window. It is presented at the top of the screen. However, if we were to run the same program under for instance Windows, the menu bar would have been shown in the frame just as for any other Windows application.

Some parts of the menu are shown in figure 3.5. We can from the menu choose to, among other things, set some basic properties, save (dump to file) a description of the current state of the system to a file, quit the program and so on.
**Figure 3.5: The main menu.**

Etop is a really simple program, but it gives us a flavor of what programs wxErlang enables us to implement (whether wxErlang will be usable in large applications remains to be seen). Currently, this is the largest application where wxErlang has been used and it has been part of the evaluation of the current (prototype) implementation of WxErlang.

As hinted earlier, there is another version of the etop program (see figure 3.6). This program is implemented using the GS library [1], which is the standard Erlang GUI library. GS is currently distributed with Erlang.

GS differs from wxErlang in many ways, but the main difference is that GS is at a much higher level and it is using many of the powerful features of Erlang. Unfortunately, the GS library does not supply even a fraction of the wxErlang library’s functionality.

Comparing the two implementations of etop, the GS version is shorter, about 350 lines (compared to 550 lines for wxErlang). This comparison is not entirely fair, since the functionality of the two implementations differs a bit. A wxErlang version of etop that copies the behavior of the GS version would perhaps end up around 450 lines or so.

Whether lines of code judges the usability of a library remains unsaid.

**Figure 3.6: The main window of the GS version of etop running on Mac OS X.**

Next, we will start looking at some of the code used to implement etop. Not all of the code will be presented since the full source code is about 550 lines. The first example will be a minor detail of the program, namely the code view window, that is shown when we select View code from the popup menu.

### 3.3.1 The code view window

In this section some important concepts of wxErlang (that were inherited directly from wxWidgets) will be introduced, and the code that creates etop’s code view window (see figure 3.3) will be explained. Let’s

---

1 The wxErlang contains some additional features.
create_code_frame(Parent) ->
    Frame = wx:frame(Parent, ?wxID_ANY, "View"),
    wx:connect(Frame, ?wx.EVT_CLOSE_WINDOW,
        fun(_,_) ->
            wx:hide(Frame)
        end),
    TxtCtrl = wx:text_ctrl(Frame, ?wxID_ANY, [{style,
        ?wxTE_READONLY bor
        ?wxTE_MULTILINE}]),
    {Frame, TxtCtrl}.

Figure 3.7: The code view window

start by having a look at the code in figure 3.3.1. What the function create_code_frame does is, in simple words, the following. First it creates a frame (window). The frame will contain a text field which is both multi-lined and non-editable. The frame will also be listening for close events.

Usually, when a user wants to close a window she/he will click the "x"-button on the window’s title bar. When the user does so she/he will trigger a close event which will be sent to the window. By default, when the window retrieves a close event, it will be destroyed\(^2\).

We have chosen to alter this behavior and instead of destroying the window when it receives close events, we only hide it. We do this by creating a new event handler using the function connect. Note that there is a fundamental difference between hiding and destroying widgets, a hidden widget can be shown again; a destroyed thing can never be used again.

The reason for altering the default closing behavior is, other than the pedagogic ones, that we want avoid to re-creating a new window every time the user wants to view code. When the user wants to view code we only update the window’s content and (re-)display it.

In order to make widgets react to different events, like mouse clicks and close events, etc., we create event handlers. This is done by using the connect function, to which we will return later.

If we go through the code step by step: we start by creating a new window frame, called Frame.

The window will have Parent as its parent. A parent is, in GUI library terminology, a widget that contains another widget. The contained widget is then called the child.

If we return to our example, Frame's parent will be etop's main window (shown in figure 3.2).

The macro ?wxID_ANY denotes the frame’s window id number. In wxWidgets, all widgets should have a unique window id number. If we specify ?wxID_ANY as id number, wxWidgets generates a unique one for us.

These numbers are, among others things, used in both the event handling system and by wxWidgets internally\(^3\).

Some functions take optional arguments, that is, some arguments are assigned a default value which is used unless anything else is said. Such a mechanism is handy when some arguments of a function are almost always called with the same values or the arguments are only needed in special cases.

This is a commonly used mechanism in wxErlang and it really helps the programmer in many ways. Functions that accept optional arguments, in general, accept a list of tuples of the form {argument_name, Value} as its last argument. For instance, if a function has a number of optional arguments, and one of these is border which defaults to 5 then to change border’s value, to say 10, we use the tuple {border, 10}. It should be stressed that different functions, in most cases, have different optional arguments.

When we create the text field we use this functionality.

\(^2\)In C++ terms: freed, deleted.

\(^3\)We will return to this further on.
To create a multi-lined and non-editable text field one has to alter the optional style argument of the function `wx:text_ctrl` to `?wxTE_READONLY bor ?wxTE_MULTILINE`.

It might not be perfectly evident what we are doing here, but we are in fact creating a so called bit-pattern that will alter the default behavior of a text field. A bit-pattern is a list of boolean values where each element of the list describes a particular true or false property, like whether a text field is multi-lined or not. For instance, to make a text field multi-lined, one would change the style argument to `?wxTE_MULTILINE`, to make a text field read only (non-editable) one sets the style argument to `?wxTE_READONLY`. To make the text field both read only and multi-lined one would use the `bor` (bitwise-or) operator, i.e `?wxTE_READONLY bor ?wxTE_MULTILINE`.

Normally one does not use bit-patterns in Erlang, since a much more natural way to describe these true/false properties is to use a list of atoms and a true property corresponds to that an atom is included in the list. The reason that wxErlang does not use these lists are due to the fact that the underlying library uses bit-patterns.

Lastly, we create an event handler to hide the window on close events. When the Frame "receives" the close event which is identified by the number `?wxEVT_WINDOW_CLOSE`, the so called callback function will be executed, which will, as stated earlier hide the window. We will talk more about events later.

### 3.3.2 Sizers

Before we continue implementing etop, we need to explore and clarify another concept of wxWidgets.

Sizers will probably become the method of choice when defining the layout of different widgets like text labels, buttons and so on, in wxErlang. Using sizers we can easily create visually appealing layouts. The concept of sizers is borrowed from similar mechanisms in other GUI frameworks, like Java’s AWT[25], GTK+[10] and Qt[22]. It is based on the idea that each individual child of a sizer reports its minimal size and its ability to be re-sized, etc. Using this information, the sizer then makes sure that the children of the sizer are laid out in an appropriate manner. As a consequence of this, the programmer does not need to set the sizes of individual widgets; this is automatically handled by the sizer.

When creating layouts with sizers one can create advanced layouts since sizers can not only contain widgets, they can also contain other sizers. Hence, we can compose different sizers into almost arbitrarily complex layouts. We can also to some extent control a few additional properties of a sizer’s children, like each individual child’s alignment, border and re-size behavior.

There are a number of sizers in wxErlang (and wxWidgets), that each represents a certain way to lay out children. Here follows a short listing of these with a brief description. The interested reader is referred to the wxWidgets documentation[30].

- `wx_box_sizer` - can layout its children either vertically or horizontally,
- `wx_grid_sizer` - is a two-dimension sizer. All children of this sizer are given the same size, which is the size of the biggest child,
- `wx_flex_grid_sizer` - is also a two-dimensional sizer, but it differs from `wx_grid_sizer`. Instead of giving all children the same size, it tries to minimize the size of each child. The width of each column and the height of each row is calculated individually according to the minimal requirements from the respectively biggest child.
- `wx_static_box_sizer` - works just like the `wx_box_sizer`. The difference between the two sizers is that a border will be painted around the children of the `wx_static_box_sizer` (in wxWidgets, it is called a static box). A static box commonly denotes some logical grouping of different widgets. We use these in etop’s main window (see upper part of figure 3.2) to group information concerning load and memory.

4That are also independent of the underlying platform
Even though we have not described how the different sizers work in detail, we can continue our exploration of etop, since one quickly gets the hang of them once seen "in action".

In our next example we will look at the Process view window which shows detailed information about a single process.

### 3.3.3 The process view window

In this section we will implement the process view window shown in figure 3.4. This window is a bit more complex than the code view window, since it contains more than one widget and an actual layout. The following code shows how the window is created.

```erlang
create_proc_frame(Parent) ->
    Frame = wx:frame(Parent, ?wxID_ANY, "Process"),
    wx:connect(Frame, ?wxEVT_CLOSE_WINDOW,
        fun(_, _) -> wx:hide(Frame) end),
    Box = wx:static_box_sizer(?wxHORIZONTAL, Frame, [{label, "Process info"}]),
    Grid = wx:flex_grid_sizer(2, [{hgap, 3}, {vgap, 3}]),
    % add the labels
    [Links, ExitSignals, ErrorHandler, Priority, GroupLeader, GarbageCollection] =
        map(fun(Title) ->
            TitleLabel = wx:static_text(Frame, ?wxID_ANY, Title ++ ": "),
            ValueLabel = wx:static_text(Frame, ?wxID_ANY, ""),
            wx:add(Grid, TitleLabel, [{flag, ?wxALIGN_RIGHT}]),
            wx:add(Grid, ValueLabel),
            ValueLabel
        end, ["Links", "Traps exit-signals", "Error handler", "Priority", "Group leader", "Garbage collection"]),
    % add Grid to Box
    wx:add(Box, Grid, [{border, 10}]),
    wx:set_sizer_and_fit(Frame, Box),

    #proc_frame { frame = Frame,
                  links = Links,
                  exit_signal = ExitSignals,
                  priority = Priority,
                  group_leader = GroupLeader,
                  garbage_collection = GarbageCollection,
                  error_handler = ErrorHandler
    }.
```

Let’s dissect the function `create_proc_frame` and see what it does. We start by creating a new window, called `Frame`, and make it listen for close events, like we did in the previous example.

Next, we create two sizers, `Box` and `Grid`. `Box` is a `wx_static_box_sizer`. The `Grid` sizer will contain all the text labels that present the information; `Box` will contain `Grid`.

The `Box` sizer is created when we call `wx:static_box_sizer(?wxHORIZONTAL, Frame, [{label, "Process info"}]). This sizer will grow horizontally as children are added to it. Unlike other sizers, which are merely a layout abstraction, `wx_static_box_sizers` are actually a composition of a widget and a sizer. More precisely,
a **wx_static_box** with a **wx_box_sizer** inside. The static box paints the border and the box sizer lays its children out.

Like most widgets a static box requires a parent widget, and in this case **Frame** will be used. We also specify the optional argument **label**, to change it from nothing to "Process info". This label will be shown above or on the static box's drawn border, depending on what platform you are using.

The **Grid** sizer is created from the call to **wx:flex_grid_sizer(2, [{hgap, 3}, {vgap, 3}])**. The first argument denotes how many columns each row should contain. The other two optional arguments specify that there should be a 3 pixel wide gap between the sizer's children. It should perhaps be stressed that **Grid** will grow (add more rows) as children are added to it.

Now, we add the text labels to the **Grid** sizer. Each row of **Grid** is composed of two text labels, the first one shows a title, for instance "Trap exit-signals :". The second one will, in this case, either show "true" or "false", depending on whether the selected process traps exits or not.

There are six such pairs and to avoid writing repetitive code, we use the function **map** to create and add the text labels to **Grid**.

For each title, we create two text labels, **TitleLabel** and **ValueLabel**. **TitleLabel** will, as the name suggests, show the title and **ValueLabel** will show the corresponding value. We add both labels to **Grid**. When we add **TitleLabel** to **Grid** we align it to the right, which is done by setting the optional argument **flag** to **{flag, ?wxALIGN_RIGHT}**. We return the **ValueLabel**, hence the call to **map** will create a list of all **ValueLabels**.

Finally, the sizer **Grid** is added to **Box**, with an invisible ten pixel border, and **Box** is made **Frame**'s sizer.

Remember when we created the **Grid** sizer we specified the gap between its children should be 3 pixels wide. There is a subtle difference between a **gap** and a **border**, which we used when we created the **Box**. The gap is the space between the widgets and the border makes a child allocate more space than needed.

The function **create_proc_frame** returns a record containing **Frame** and all "value labels". The reason for storing these widgets is that when a user selects **View process information** from the popup menu (see figure 3.3) we will update the content of the widgets and (re-)show the window.

### 3.3.4 The popup menu

When the big table at the bottom of the window (see figure 3.2) is right clicked, a so called **popup menu** or **context menu** appears (see figure 3.3). The menu contains the four **menu items**, namely, **View code**, **View process information**, a **separator** and **Kill**.

We will in the following section see how this menu is created.

We need to add a note about how the etop application works in order to entirely understand all the implementational details of the menu. When etop is started, it first creates all the windows, menus, event handlers and so on. After this is done, a server, called **etop_gui_server**, is started. This server will wait for different messages and when, for instance, the message **show_proc_info** is received, the server will update the process information window (see figure 3.4) and make it visible to the user.

Creating menus, in general, is very straight-forward; we first create an empty menu then append/prepend menu items to it. Each selectable menu item is associated with a **magic number**. This number will later be used, when creating the menu's event-handler(s). For convenience, one usually, define macros for these numbers. In our case the following macros will be used.

```c
#define(ID_KILL, 0).
#define(ID_VIEW, 1).
#define(ID_PROC, 2).
```
create_popup_menu() ->
    PopupMenu = wx:menu(),
    wx:append(PopupMenu, ?ID_VIEW, "View code"),
    wx:append(PopupMenu, ?ID_PROC, "View process information"),
    wx:append_separator(PopupMenu),
    wx:append(PopupMenu, ?ID_KILL, "Kill"),
    wx:connect(PopupMenu, ?ID_VIEW, ?wxEVT_COMMAND_MENU_SELECTED,
        fun(_, _) -> etop_gui_server ! view_code end),
    wx:connect(PopupMenu, ?ID_PROC, ?wxEVT_COMMAND_MENU_SELECTED,
        fun(_, _) -> etop_gui_server ! show_proc_info end),
    wx:connect(PopupMenu, ?ID_KILL, ?wxEVT_COMMAND_MENU_SELECTED,
        fun(_, _) -> etop_gui_server ! kill end),
    PopupMenu.

As stated earlier, it is quite straight-forward. We create an empty menu, then append the items (with their corresponding magic number) and the separator.

For instance, the call `wx:append(PopupMenu, ?ID_PROC, "View process information")` will append a new menu item to the menu `PopupMenu`, associate `?ID_PROC` with it and it will show the text "View process information".

When the user selects one of the menu items an event is generated and sent to the `PopupMenu`. In order to "catch" these events we, as always, create event handlers. This example differs from previous examples of events and event handling. A bit simplified\(^5\), the difference lies in the nature of the events. In previous examples we were listening for close events. These events only have one originator: the window that was closed.

In this example, we are listening for events that are generated when the user selects one of many menu items. To tell the different menu items apart we use the magic number described earlier. For instance, the event handler created by the call `wx:connect(PopupMenu, ?ID_KILL, ?wxEVT_COMMAND_MENU_SELECTED, ...)` will be triggered when the menu item corresponding to `?ID_KILL` is activated.

Whenever an event handler in our example is triggered, we sent a descriptive message to our server process `etop_gui_server`. The server will then take the appropriate action, i.e show the code view window.

### 3.3.5 Events and event handling

At a first glance, event handling in wxErlang might seem a bit awkward, but one quickly gets over it. We will in this section take a closer look and describe more accurately how the event handling in wxErlang works and the properties thereof.

When a widget, for example a button, is clicked, an event will be generated and "sent" to the button. The button can now choose if it wants to handle the event. If the button does not handle the event (i.e

\(^5\)And not entirely correct.
has no event handler for it), then it is propagated to the button’s parent. The parent may in its turn propagate the event to its parent

6

, and so on.

On the other hand, if the event is handled by an event handler, then the event propagation will by default stop

7

.

The propagation scheme enables us to, for instance, listen for button click events in a widget that has a button as one of its children, which might be a possible source of confusion. In some cases we might need to tell events generated by different widgets apart when, for instance, a window contains two buttons. Remember that each widget is created with a unique window id; this is where these numbers come in handy.

To get a better understanding of how things relate we show three examples, and briefly explain what will happen.

- **wx_frame:connect(Frame, ?wxEvt_Window_Close, F)** - will create an event handler that is triggered when: Frame receives the event identified by the number ?wxEvt_Window_Close and the event was generated by any widget,

- **wx_menu:connect(PopupMenu, ?ID_VIEW, ?wxEvt_Command_Menu_Selected, F)** - will create an event handler that is triggered when: PopupMenu receives the event identified by the number ?wxEvt_Command_Menu_Selected and the event was generated by the widget with window id ?ID_VIEW,

- **wx_menu:connect(Menu, ?wxEvt_Command_Menu_Selected, ?ID_KILL, ?ID_PROC, F)** - will create an event handler that is triggered when: Menu receives the event identified by the number ?wxExt_Command_Menu_Selected and the event was generated by a widget with window id ID such that ?ID_KILL <= ID <= ?ID_PROC.

In fact, the first and second version of connect is defined in terms of the last one and they relate in the following manner:

```erlang
connect(Widget, EventID, Fun) ->
    connect(Widget, ?wxID_ANY, EventID, Fun).
connect(Widget, WindowID, EventID, Fun) ->
    connect(Widget, WindowID, WindowID, EventID, Fun).
connect(Widget, FirstWindowID, LastWindowID, EventID, Fun) ->
    ...
```

We should also mention a brief word about the so called callbacks, i.e. the function that is called when a event handler is triggered. The function accepts two arguments, the first argument is an event object that contains information about the event. The second argument contains a list of tuples of the form {attribute, Value}. These are merely used as a nicer way to access an event’s data. Unfortunately, however, this functionality is currently very conservatively used and is, in most cases, the empty list.

The programmer should take notice when using functions that concern the current process in callback functions. This comes from the fact that the callbacks will be executed from another process and not from the one creating the event handler. A call to, for instance, self() might not return the pid that the programmer is expecting.

To summarize, in order to make a widget listen and react to an event, one needs to make sure that the event will eventually reach the widget and the correct event id and window id are used. If all these conditions are satisfied, the event handler’s “fun” (commonly referred to as callback) will be called.

For information on this please refer to the wxWidgets documentation and the future documentation of wxErlang.

---

6There are a few exceptions, the interested reader is referred to the wxWidgets documentation for more information.

7One can however change this behavior.
3.3.6 Wrap up

In the previous sections, we have seen how the etop application is implemented. The observant reader might have noticed that the presented examples do not make up the whole application. However, the examples presented are perhaps the most interesting excerpts of the application's source code. The remainder are either similar to earlier presented examples, or do not show any new mechanism, or even a combination of both.

Writing GUI applications is in general quite verbose and wxErlang is no exception. Writing the same program in C++ using wxWidgets directly, would probably have yielded approximately the same line count. In fact, the wxErlang interface differs very little from the C++ interface. Unfortunately, this is not because the C++ interface is exceptionally good — as one could have hoped — this is because most of the wxErlang interface is automatically generated from the C++ interface. We will return to this topic later.

Interface designers

The process used when writing GUI applications is typically of a trial-and-error nature, which is both slow and tedious. A more appealing and more efficient way to create the GUI of an application is to use a so called interface designer. An interface designer allows the programmer (or designer) to easily create the GUI of an application, typically using a WYSIWYG (What You See Is What You Get) editor.

There are currently a number of interface designers available for wxWidgets, among others XRCed[32], DialogBlocks[7], wxGlade[28] and wxDesigner[27].

In general an interface designer generates "dead" code, that is, buttons and such do not react to any events. The programmer will then have to fill in the code that makes the application "live".

Luckily, the interface designers mentioned earlier do not only have the ability to generate C++ code, they can also generate XRC. XRC is a specially designed XML[31] format used by wxWidgets to describe and store different visual components, like windows and buttons, etc. These files can later be loaded by an application at runtime and the visual components can be restored.

The following example shows how we load the frame FRAME1 from the XRC file test.xrc.

```erlang
start() ->
    wx:start(),
    XRC = wx:xml_resource_get(),
    wx:init_all_handlers(XRC),
    wx:load(XRC, "test.xrc"),
    Frame = wx:load_frame(XRC, ?NULL, "FRAME1"),
    wx:show(Frame).
```

First we initialize the wxErlang system as always. The call to `wx:xml_resource_get()` returns the resource controller and the call `wx:init_all_handlers(XRC)` initializes it. Next we load the XRC file, the call `wx:load(XRC, "test.xrc")` (re-)creates the frame FRAME1. Finally, we make the frame visible to the user.

For more information about the XRC the reader is referred to the wxWidgets documentation.

Using interface designers is to be preferred, but one should beware that the interface designers might also restrict the programmer. An interface designer might, for instance, not support changing color on a button and so on. One should also beware that these applications are originally designed to use with C++. The programmer (or designer) is required to be the familiar with concepts of object orientation and wxWidgets to some extent.
Chapter 4

Design

With a thorough wxErlang example (etop) behind us, we will now start looking at how wxErlang really works. We will explore and clarify different design decisions, underlying mechanisms and concepts and so on. Many of these issues come from the fact that unlike regular Erlang libraries, wxErlang is based on a library written in C++. These two languages are fundamentally different; Erlang is a concurrent functional programming language and C++ is an (imperative) object oriented language. Some paradigm-wise mismatches have been successfully resolved, others have not.

Throughout this chapter, we stay at a rather high level, avoiding discussing code in detail.

4.1 Object orientation

Many Erlang programmers frown upon object orientation (OO); whether this attitude is justified will remain unsaid. We will, however, need to investigate how different concepts in object oriented programming (OOP) work and define some terminology. We need to understand OOP to some extent in order to find a suitable encoding of them within Erlang.

OO programming focuses around, as the name suggests, something called objects. An object consists of both attributes (some data) and a set of methods (functions, operations). These methods manipulate or use the object’s attributes in some way. Different objects that have the same set of attributes and methods are said to belong to the same class.

The basic idea behind OO programming is to use objects to model the different real-life objects of a system. For instance, if we were to model a car in an OO language, the car could be composed of four wheel objects and an engine object.

As stated earlier all objects have a set of attributes and methods, these correspond to properties of an object and ways to manipulate the object. For instance, the engine object’s attributes could be its horse powers and perhaps how many cylinders it has. Its methods could be start/stop the engine, etc.

Consider the following real life objects: a car, a boat and an airplane. These three objects, seen from a particular point of view, share at least one common thing: they are used to transport people and goods around.

In OOP we can capture and model this relation with, so called, inheritance. Inheritance supplies the programmer with an easy way to extend and re-use existing code. Returning to our example; we could model the real life objects as classes that inherit from the class transport. The transport class will be used to describe common features of something that transports things. Using OO terminology, one says that the Car, Boat and Airplane are subclasses of Transport or that Car, Boat and Airplane inherit from Transport.

When A inherits from B, then A ”inherits” the methods and attributes defined in B. That is, A will have all operations and attributes from B. Hence the programmer can use A just as if it was a B.
The programmer can also modify and extend the behavior of an object by overloading/overriding inherited methods and by adding new methods and attributes. When overloading one changes the behavior of different inherited methods.

Inheritance is probably the single most important feature in OOP, since it allows the programmer to re-use existing code and easily extend the functionality of existing classes.

Since Erlang does not natively support OOP and wxWidgets is implemented in an OO style, we need to model a few of its concepts using the mechanisms present in Erlang. There are many ways to do this, we have chosen one that hopefully will be convenient for the programmer.

Some GUI libraries, like GTK+[10], are implemented in an OO style, although the language in which the library is implemented does not support OO natively. Instead, the library somehow emulates the different concepts of OOP.

WxErlang also does this to some extent, perhaps not in a way that is apparent to the programmer which is good since we want to avoid introducing a new paradigm in Erlang.

In OO languages one generally states `Obj.MethodName(...)`. The dot operator looks up and uses an object’s method. This has been translated to `wx:method_name(Obj, ...)` in wxErlang. This may lead to name clashes when two different objects define a method with the same name. Such functions are “merged” and will behave differently depending on the type of the arguments. Also, if we in wxErlang try to use a method that is not in the set of the object’s methods, an exception will be raised.

To relate to the concepts of OOP, the notion of classes remains untranslated and there is no way to extend a class. Luckily, this is not as bad as it could have been. Some OO libraries require the user to constantly overload and extend the functionality of classes in order to use them. WxWidgets does, in most cases, not require the programmer to extend classes and overload functions. There are however classes that are implemented in this way in wxWidgets, and they are currently impossible to use from wxErlang.

In the future versions of wxErlang we can hopefully overload and override functions dynamically. How such a mechanism would work and what consequences it would have will not be discussed in this paper.

Objects on the other hand, are easily modeled. The methods of an object are those functions in the `wx` module for which an object is defined. Usually these are the same as those of a corresponding C++ object. The attributes are also the same as those of a corresponding C++ object, with one exception: objects also know their own type.1

Whether the representation of OO concepts is sufficient and feasible remains to been seen.

4.2 Safety

WxErlang imposes a safety mechanism on the wxWidgets library. As functions are called the different arguments sent to them are evaluated to make sure that they are valid. The checks performed concern the types of the arguments and, in some cases we also perform some sanity checks.

Without this mechanism a function call with incorrect arguments could potentially cause crashes2. If a check fails, an exception will be raised.

---

1We will return to this later.

2Not of the ERTS though, we return to this later.
Each function in the wxErlang interface knows which types it accepts and when the function is called it will check the arguments sent to it.

Type checking primitive values is no problem since we can use functions like `is_integer`, `is_float` in Erlang. Type checking objects are a bit trickier; we need to know the type of the object, i.e. the corresponding C++ class of which the object is an instance, and how this type relates to other types in terms of inheritance.

If type checking primitive values is an `is-of-type` relation, then type checking objects is an `inherits-from` relation.

Consider the a call to the function `wx::show(Frame)`. The `Show` method, in C++, is defined in the class `wxWindow`. In wxWidgets, `wxWindow` serves as a base class to most widgets, hence all classes that inherit from it will also ”have” the method. To type check the call `wx::show(Frame)` we retrieve `Frame`’s type and check whether it corresponds to the C++ class `wxWindow` or any of its descendants.

To find out which classes inherit from which, the reader is referred to the wxWidgets documentation.

The checks performed are implemented in an efficient manner and should not noticeably affect the performance of wxErlang. It should also be stated that we can trick the safety mechanism quite easily, it is only a matter of manufacturing an erroneous value.

### 4.3 Memory management

C and C++ have no automatic memory management, which means that the memory allocated by a program needs to be manually freed by the programmer, otherwise the program will ”leak memory”. A program that leaks some memory is not really a big problem. A program that leaks a lot of memory could, however, slow down not only the application, but also the underlying operating system. To avoid this one should always try to write programs that do not leak memory, no matter how small the programs are.

Note that the memory used by a program will automatically be freed by the underlying operating system when the program terminates.

When we are programming in wxErlang we are actually indirectly programming in C++, and as we have seen before, some concepts and mechanisms shine through the Erlang ”facade”. Memory management is, unfortunately, no exception.

Like the C++ programmer, one has to free, for instance, widgets that are no longer used. To free a widget, or a object, one calls the function `wx::delete(Widget)`\(^3\). Once something is freed it should not be used again.

In some cases widgets take ownership of other widgets, that is, the widget will be automatically freed when the owner is freed. Other functions may indirectly free objects. The wxWidgets documentation\([30]\) explicitly states when this is the case.

As a rule of thumb: values that are not Erlang native, like integers, floats, lists, etc, should be freed. After they have been freed they should not be used again.

### 4.4 Functions

In the following section we will have a look at the differences of the function construct in Erlang and C++, then describe how these are translated between the two languages.

#### 4.4.1 Optional arguments

In C++ we can define functions that allow optional arguments, or more precisely, one can assign default values to a number of arguments in a function declaration. These default values will be used unless

\(^3\)In some cases `wx::destroy(Widget)` is preferred.
anything else is stated. This construct is very convenient when there are a few variables in a function that very rarely change. Instead of requiring the programmer to supply a rarely changing variable every time the function is called, he/she can ignore it in most cases and change it whenever it is needed.

In C++ a function declaration with optional arguments is of the form:

```cpp
int Function(Arg1, Arg2, ..., ArgM, OptArg1 = Default1, ..., OptArgN = DefaultN) {
    ...
}```

In Erlang we have no corresponding construct, as a matter of fact, we do not need one. We can easily emulate a similar functionality in Erlang using Erlang primitives only and this is in fact commonly used in many Erlang libraries. We use lists with, so called, tagged values, e.g. lists of the form `[{Key1, Value1}, {Key2, Value2}, ..., {KeyK, ValueK}]`. If we relate to our previous C++ example, we would translate that function declaration into:

```erlang
function(Arg1, Arg2, ..., ArgM, Opts) ->
    ...
```

If we want to call the function with a non-default value we simply call the function in the following manner: `function(Val1, Val2, ..., ValM, [{varname, Val}])`. The argument `varname` will, instead of its default value, have the value `Val`.

```erlang
function(Arg1, Arg2, ..., ArgM) ->
    function(Arg1, Arg2, ..., ArgM, []).
function(Arg1, Arg2, ..., ArgM, Optionals) ->
    ...
```

### 4.4.2 Multiple return values

A function in Erlang that returns multiple values commonly does so by using tuples. In C++, on the other hand, tuples are natively not supported. There are, however, still situations where functions are required to return multiple values. Functions that return multiple values in C++ use, so called, *out arguments*, e.g. arguments that are solely used to return values.

Consider the following C++ example. If we have the C++ function `void GetPoint(int *x, int *y)` a call to it takes the form `Obj.GetPoint(&x, &y)`, after the call the `x` and `y` variables will be set to appropriate values.

The previous example would in wxErlang be translated to the function `get_point(Obj)` which returns a tuple on the form `{X, Y}`. In general, if a C++ function returns `n` values, the corresponding Erlang will return a tuple with `n` values.

In some cases, a C++ function not only uses the out arguments as out arguments. Sometimes it also uses the in value. Suppose we have the C++ function `void AddOne(int *value)` and it is defined in such a way that it adds one to its argument, i.e. the following code snippet will output "11" on the screen.

```cpp
int x = 10;
AddOne(&x);
cout << x << endl;
```

The wxErlang function corresponding to the `AddOne` function would be `add_one(X)` and it will return the integer `X + 1`.

In/out arguments are rarely used in wxWidgets.
WxWidgets may also combine the optional arguments feature with out arguments. We have not been able to come up with a very accurate interpretation of this functionality and we handle them just as if they were out arguments. The problems in these situations are a bit subtle and we will not go into great detail here.

4.5 Overlaps

Some classes present in wxWidgets overlap some of Erlang’s built-in features. For instance, wxWidgets contains abstractions for concurrency and networking. Since Erlang already contains very nice mechanisms for handling such things, we in general do not want, nor need, to use wxWidgets’ abstractions. Instead of trying to incorporate such mechanisms into wxErlang, risking extra work, we simply excluded these.

In fact, almost all classes that do not have anything to do with GUI programming are excluded from wxErlang.

Some value abstraction classes in wxWidgets overlap Erlang’s native values. For instance, wxWidgets defines the `wxString` class, which is an abstraction for manipulating and handling strings. We do not want to use `wxString`, since Erlang programmers are used to Erlang strings.

To avoid this, wxErlang automatically translates Erlang strings into `wxString` and vice versa, efficiently hiding the fact that the programmer is actually using `wxStrings"under the hood"`.

Not only does wxErlang translate strings, it also defines a few other such translation schemes. It currently translates the class `wxPoint`, which is a point in the plane, to a 2-tuple on the form `{X,Y}`, the class `wxSize`, which is a dimension, to a 2-tuple `{Width, Height}` and the class `wxStringArray` into a list of strings.

The intention is to add more of these translations as the wxErlang project evolves.

4.6 Constants

When programming in C++ one commonly uses constants or macros. The constants are really aliases for integers, which one can use instead of the aliases. But by using aliases the programmer can create more readable and understandable code.

In fact, this functionality in C++ is used similarly to the way that Erlang programmers use atoms.

We do not translate C++ constants used in wxWidgets into atoms. Instead we translate them into Erlang macros. For instance, the constant `wxEVET_WINDOW_CLOSE` in wxWidgets is translated to the Erlang macro `?wxEVET_WINDOW_CLOSE`.

One might argue that translating constants to atoms is a more natural translation, e.g. instead of using `?wxEVET_CLOSE_WINDOW` we should use the atom `wxevent_close_window`. It is also possible to do this, but at a rather high cost, both efficiency wise and for the programmer. First, it would require us to have a mapping, from atoms to integers, present at runtime. Secondly, it would also be, quite counter intuitive for the programmer. Some functions return values that might correspond to a certain C++ constant. If using atoms instead, the programmer would be required to some how resolve what atom the returned value corresponds to.

4.6.1 Programming with multiple processes

A key feature of Erlang is its powerful built-in mechanisms for concurrent programming. A typical Erlang program may have hundreds of processes running concurrently. Unfortunately, programs written using multiple processes and wxErlang need cautionary remarks, more precisely, programs where different processes are using wxErlang, need special attention.
Suppose that we have started a number of different programs that are using wxErlang from the same Erlang shell. These programs will, in most cases, run in different computing contexts protected from each other. We want to protect the programs from each other if something goes wrong, for instance if one program crashes we do not want others to crash as well.

A computing context is really an instance of the wxErlang library and each computing context is separated from all other wxErlang instances. What happens in one computing context can not directly effect what happens in an other.

Remember that one has to call the function `wx:start()` before one can access the wxErlang library. This function creates a new computing context and sets up the process issuing the call to use the newly created computing context. That is, all wxErlang function calls issued by the process will be directed to this particular computing context.

On rare occasions we may want to use multiple computing contexts, hence one needs to switch between different computing contexts. This is done by using the function `set_current_context(Context)`, where `Context` is the value returned by `wx:start()`. The following example shows how one could use the function `wx:set_current_context(Context)`.

```
Context1 = wx:start(),
Context2 = wx:start(),
... % calls to wxErlang will be handled by the second wxErlang system
...
wx:set_current_context(Context1)
... % calls to wxErlang will be handled by the first wxErlang system.
...
```

Note: objects created in one computing context should not be used in another computing context since this will most certainly lead to unexpected behavior.

By default a process does not have a computing context set. One can set it (or change it) by using the functions `wx:start()` (which will create a new computing context) or by using the function `wx:set_current_context(Context)`. The latter of these functions can also be used when one wants multiple processes to access the same computing context. More precisely, if one wants a number of processes to use the same computing context `C` all of the involved processes have to call `wx:set_current_context(C)`.

If a process with no computing context would try to call wxErlang functions an error will be raised.

Another note about processes and wxErlang concerns memory management. If an object is freed in one process, the object is freed in all other processes and should not be used again.

---

4If we are creating a debugger or so.

5This is because the objects come from different address spaces (= computing contexts).
Chapter 5

Implementation

In previous chapters we have seen how to write programs using wxErlang and how wxWidgets have translated into Erlang’s environment. In the following chapter we will, quite thoroughly, describe how wxErlang has been implemented.

The first sections of this chapter will describe the different parts of the system. We will then move on to details about the implementation and some general technical aspects. Finally, we will look at how the wxErlang interface and the so called wrapper functions are created.

The text in this chapter is of a much more technical nature than the text in previous chapters.

It should be stressed that the current version of wxErlang is a prototype implementation. As a consequence of this things might, and probably will, change as new versions of wxErlang are released. Hence the text in the following section might be both outdated and incorrect.

5.1 Bindings in Erlang

A binding is, as stated earlier, a connection between two languages that enables one to use some functionality implemented in the other. In our case we create a binding between C++ and Erlang, enabling Erlang to access the wxWidgets library.

This section will give us an overview of how bindings in Erlang generally work. Then we have a closer look at how the wxErlang binding works.

5.1.1 Ports

The abstraction used when creating bindings in Erlang is called ports. Ports use message passing constructs to model the communication between Erlang and the outside world, creating the illusion of something very familiar to an Erlang programmer: a process.

There essentially are two implementations of this abstraction: a port program and a port driver (also called linked-in driver).

A port program is really a "help-interpreter" running in parallel with the Erlang runtime system (ERTS), supplying extra functionality. The data sent between Erlang and the port program is transmitted using the underlying operating system’s IPC (interprocess communication) mechanism. On Unix, pipes will be used.

Port drivers (linked-in drivers), on the other hand, run inside the ERTS. They are dynamically linked into the ERTS, enabling much faster communication since all code is running in the same operating system process.

Compared to a port program, linked-in drivers are trickier to implement. If the port-driver were to crash it would bring down the ERTS as well. Other errors, like leaking memory, deadlocks, etc., will propagate to the ERTS as well.
Whether to implement a binding as a port program or as a linked-in driver depends on the nature of the binding. If the binding requires high performance and a lot of information is sent between the port and the ERTS, then one should probably implement the binding as a linked-in driver. Otherwise one can just as well implement the binding as a port program since this involves fewer risks.

wxErlang is implemented as a port program. This seems to suffice, since other similar bindings are implemented in this way[1][23].

5.2 The different components of wxErlang

In this section we will describe the different components of wxErlang, briefly describe them and introduce some terminology.

The current implementation of wxErlang is logically divided into two components: an Erlang proxy process and a port program. The proxy process functions as a mediator between an Erlang program that uses wxErlang and the wx-interpreter. The port program, or the wx-interpreter, simply receives instructions and executes them. Figure 5.1 shows a snapshot of a currently running program that uses wxErlang. The arrows denote how the different parts communicate.

When the user program, shown in 5.1, calls a wxErlang function a message will be created and sent to the proxy process. The message contains both the arguments sent to the function and the function’s name.

The proxy process will add some extra information to the message and then forward it to the wx-interpreter, which will then decode the message and interpret it, and in this case, execute the corresponding C++ function. The function’s return values, if any, will then be sent back to the caller in a similar manner.

5.3 Getting Erlang to talk with C++

When creating a binding from one language to another we need to translate the values in one language to the other and vice versa. Some of these translations are quite natural, others are not as natural, some translations might lose accuracy or even be translated into something entirely different.

![Diagram of wxErlang components](image)
For instance, most languages support integers, but this does not necessarily mean that the conversion process is easily done. If one language represents their integers with 16 bits and the other language 32 bits, we might lose accuracy when converting from 32 bits to 16 bits.

We have a similar problem when converting values between Erlang and C++. 

C++ has a large number of different primitive types (like characters, integers, floats, doubles, etc.), while Erlang has a quite limited set of primitive types. There is no simple one-to-one translation between the primitive C++ types and Erlang’s primitive types. For instance, Erlang does not differentiate between unsigned (non-negative) and signed integers (both positive and negative) like C++. Erlang treats all integers as signed. Fortunately, we can still encode an unsigned integer as a non-negative signed integer, and due to the fact that Erlang’s integers can grow arbitrarily large, we can represent any type of C++ integer in Erlang. When converting from Erlang integers to C++ integers, however, we might get into trouble. C++ does not support arbitrarily large integers natively, hence we might need to truncate large Erlang integers before converting it to C++, hence we could lose some accuracy.

Our translation scheme of primitive values is very simple. We translate all C++ chars, shorts, ints and longs, unsigned or not, to Erlang’s integer representation, C++ floats, doubles to Erlang’s float representation and vice versa.

Other values, like strings, points, etc. are, as described in earlier sections, translated in a more complex manner. For instance, the class wxPoint in wxWidgets is translated into an Erlang tuple \{X, Y\}.

Then there are values that have no sensible translation in Erlang, like values of type wx_frame and other object instances. Such values are, in wxErlang, represented by the record #wx_object { ptr, type, wxe }. The field ptr contains the C++ pointer to the value. The field type contains type information about the value and the wxe field specifies which port program created the value.

Values of unrecognized type will by default be treated in this manner.

5.3.1 Wrapper functions and interface functions

In this section we will look in detail how the two different languages are glued together.

Consider the member function Hide(), defined in the class wxWindow. To include this function to the wxErlang interface we create two functions: an interface function (implemented in Erlang) and a wrapper function (implemented in C++).

The interface function hide(Self) enables the wxErlang programmer to use the C++ function Hide(). It more or less only initiates the sequence of actions required by the function call procedure described in section 5.2.

The wrapper functions on the other hand are doing most of the work; converting values back and forth between different representation and performing the actual call to the Hide() function.

We will in this section look closer at these functions and how they work, but before we do this it should be stressed that all of the code used to implement the interface functions as well as the wrapper functions are generated. This would both be very repetitive and time consuming to write by hand, on top of that the interface functions are very similar to each other this is also true for the wrapper functions. Code that is generated is presented within "[|" and "|]" and for instance [| unmarshal(arguments )|] is the generated code for unmarshaling (converting) the arguments values.

Interface functions

Now let’s have a look at the interface functions of wxErlang. As stated earlier these functions are those that are accessible to the wxErlang programmer from the wx module. When one of these functions are called they will first make sure that the arguments sent to it holds the properties put on them by wxErlang’s safety mechanism (see section 4.2). If so, the function call procedure will then be initiated. Other than this these functions do not hold any real functionality.

\[\text{No matter how many bits the C++ integer is using.}\]
Figure 5.2 shows what a general and a slightly simplified interface function look like. Code within "[| and "|]" is individual (and generated) for each interface function.

```
[| function(Arg1, Arg2, ... ArgN) |] ->
case [| check(Arg1) |] andalso [| check(Arg2) |] ... [| check(ArgN) |] of
  true ->
    wx_core:call([| function |], [| [Arg1, ... ArgN] |]);
  false ->
    erlang:error({bad_type, [| .... |]});
end,
end.
```

Figure 5.2: A wxErlang function written in pseudo code

Starting from the top: in the first line we declare the function, could for instance be `hide(Self)`. Next we generate the sanity and type checks of the arguments. If all these succeed the function `wx_core:call` will be called which starts the function call procedure.

On the other hand if some of the checks fail a `bad_type` error will be raised.

For further details on the implementation the reader is referred to appendix B.

Note that the types and sanity checks are performed on the "Erlang side", which enables us to write the C++ wrappers very aggressively since the values are correct, hence we can skip many checks that would perhaps normally be needed in the wrapper code.

### C++ wrapper functions

The wrapper functions play an important role, they do not only perform the call to the actual function they also provide a translation mechanism between the two languages. Among other things they translate both values between different representations and different constructs like optional arguments and multiple return values (see sections 4.4.2 and 4.4.1).

Figure 5.3 shows how a general (and simplified) wrapper is implemented, written in pseudo C++ code. Code between "[|" and "|]" is individual (and generated) for that function.

The figure shows how a wrapper function that wraps a function which returns one value and has no optional arguments, like the constructor `wxBoxSizer(int orient)` `(wx::box_sizer(Orient))`, is implemented.

```
boolean [| function_id |] (List arguments) {

  List unmarshaled_args;
  [| return_type_of_wrapped_function |] retval;

  unmarshaled_args = [| unmarshal(arguments) |];
  retval = [| Function(unmarshaled_args) |];
  return send([| marshal(retval) |]);
}
```

Figure 5.3: A general and simplified wrapper written in pseudo code
If we examine the example more closely we start by declaring the function denoted by [\texttt{function\_id}]\(^{1}\). Its name will be unique and is created from its name and the types of the arguments used by the function.

The function accepts a list of arguments that will be sent to the wrapped function. The arguments are encoded like Erlang values and will later be converted into C++.

Next, we declare two variables: \texttt{List unmarshaled\_args}, [\texttt{\text{return\_type\_of\_wrapped\_function}] \texttt{retval}}. The first of these variables will be used to hold the values of the converted arguments. The second will be used to hold the return value of the function, hence if the function returns an \texttt{int} the type of \texttt{retval} will be an \texttt{int} as well.

Then we convert the arguments from its Erlang representation to the C++ representation ([\texttt{unmarshal}\texttt{(arguments)}]) and call the wrapped function with the converted arguments as \texttt{input([\texttt{Function(unmarshaled\_args)}])}.

Finally we call \texttt{send([\texttt{marshal(retval)}])} which will convert the return value to Erlang representation and send it back the Erlang side. The send function returns true or false depending on whether the transmission was successful or not.

For further details on the implementation the reader is referred to appendix A.

To get a better understanding of how we can use wrapper functions to translate constructs from one language to the other we will have a closer look at how we translate C++’s notion of functions with optional arguments into Erlang’s notion of functions with optional arguments.

Suppose we have a C++ function definition like the following
\[
\texttt{void Foo(Type1 Arg1 = Def1, Type2 Arg2 = Def2),}
\]
that is a function with two optional arguments, also assume that this function is defined in the C++ class \texttt{Bar}.

If we are to call this function from C++ using the argument’s default value, we would write \texttt{Foo()}. If we would like to change the first argument’s default value, we would write \texttt{void Foo(SomeOtherValue)}. If we would like to change both the arguments’ default values, we would write \texttt{void Foo(SomeOtherValue1, SomeOtherValue2)}. If we on the other hand were to change the second argument’s default value to \texttt{SomeOtherValue}, we are then required to write \texttt{Foo(Def1, SomeOtherValue)}. In some sense we can say that the optional argument construct of C++ is based on the arguments’ position.

As described in section 4.4.1, the corresponding functions in wxErlang would be both \texttt{foo(Self)} and \texttt{foo(Self, Opts)}. The first definition does not allow us to change any of the default values, while the second does. The optional arguments used by \texttt{foo} are \texttt{arg1} and \texttt{arg2}, where \texttt{arg1} corresponds to \texttt{Arg1} and \texttt{arg2} corresponds \texttt{Arg2}.

To, for instance, change the default value of \texttt{arg2} we would write \texttt{foo(Obj, \{\texttt{arg2, SomeOtherValue}\})}\(^{2}\) (under the assumption that the values fulfill the requirements issued by wxErlang’s safety mechanisms, see section 4.2).

To change both default values of the function we would write \texttt{foo(Obj, \{\texttt{arg1, SomeOtherValue1}, \texttt{arg2, SomeOtherValue2}\})}).

Note that order does not matter, that is we could just as well have written \texttt{\{\texttt{arg2, SomeOtherValue2}, \texttt{arg1, SomeOtherValue1}\}}.

It might not be entirely evident how it is possible to translate C++’s position based notion of optional arguments into the \texttt{name-of-argument} based notion of optional arguments used in wxErlang.

The secret lies in the fact that when the wrapper function decodes the arguments sent to the function one can also extract the arguments position. That is, if we are changing the value of \texttt{arg2} to \texttt{SomeOtherValue}, then we are able to decode the value of \texttt{SomeOtherValue} as well as retrieve the argument’s position.

Later when the C++ wrapper function is going to call the wrapped function it will check whether the optional arguments have been changed or not. If an optional argument is changed then this value will be used, otherwise the default value will. Once again

\(^{2}\text{We are using a naming convention that translates "Arg2" into "arg2".}\)
relating to our Foo example, when the wrapper function will call Foo, the call to it will be
Foo(isArgSet(arguments, 1) ? Arg1 : Def1, isArgSet(arguments, 2) ? Arg2 : Def2). Where isArgSet
returns true whether a particular argument is set and false otherwise. The expression
isArgSet(arguments, N) ? ArgN : DefN evaluates to ArgN if isArgSet(arguments, N) is true and
otherwise to DefN (a C/C++ if-expression).

5.3.2 Events

In this section we will look at how events, event handlers and so on work in wxErlang. Unlike other
sections that have touched upon this topic, we will in this section look "under the hood".

As described in earlier sections one creates event handlers using the function connect, see section 3.3.5.
Such a function call is typically of the form connect(Widget, EventId, FirstId, LastId, Fun).
Such a call will, like a regular function, send a message to the proxy process. The proxy process will recognize
the message as a connect message and associate the callback function (Fun) with a unique id. The proxy
process then creates a new message containing Widget, FirstId, LastId and the unique id, and send it
to the wx-interpreter.

When the wx-interpreter receives the message it will also recognize the message as a connect message
and it will from it create a new event handler. When the event handler is triggered it will send a special
message to the proxy containing the unique id. When the proxy receives such a message it will extract
the id, retrieve the corresponding callback and execute it.

This might seem like a tedious procedure, but the alternative, to send the whole function object (the
callback) back and forth, is not feasible. Serialized function objects have a tendency to grow quite large.
Using the approach with callback id is much more efficient since no large amount of data needs to be
sent back and forth.

5.4 Code generation

WxWidgets is indeed a very large library, reaching a class count of about 500 classes and about 2800
methods[16]. Writing the so called wrapper code and interface code that glues Erlang and C++ together
by hand is not feasible. It would be a very time consuming, boring and error-prone project. Instead
much of the code in wxErlang is generated.

An interface specification language has been designed for this particular reason. We use the language
to describe how the different classes of wxWidgets relate to each other, what functions they define and
so on. Then, from the interface specified, the wxErlang interface and the C++ wrapper code will be
generated.

Many other wxWidgets bindings use some mechanism to generate their interface. WxErlang’s code
generation mechanism is especially influenced by the code generation mechanism of wxBasic[6]. WxBasic
is as the name suggests a wxWidgets binding to BASIC.

WxBasic generates from its interface specification its own version of a BASIC-interpreter, with a
special built-in interface to wxWidgets. Actually, wxBasic is really a dialect of BASIC that supports
structured programming and has been influenced by QBasic.

From wxErlang’s point of view this is not as far fetched as it might initially seem. Remember that
wxErlang is implemented as a port program, see section 5.1.1. A port program is really a help-interpreter,
that runs in parallel with Erlang (the ERTS), supplying extra functionality. As a matter of fact, most
of the help-interpreter’s code is generated using our interface specification language.

\[3\text{A modified truth.}\]
We will in the following sections, using a few examples, describe how the wxErlang interface (interface functions) and the C++ wrapper functions (wrapper code) are generated. We will, however, not describe how each individual interface function nor the wrapper functions work, since we have previously described this in section 5.3.

5.4.1 Simple example

To differentiate between C++ classes/functions and Erlang we use the following naming convention: C++ classes and functions contain capital letters like for instance, wxFrame, Car and VeryLongName. In Erlang we use underscores (_) instead, e.g. wx_frame, car and very_long_name.

We will have a look at a really simple example. The example consists of two classes: Car and Truck. The Car class defines two functions, a constructor and the function GetCurrentVelocity. The other class Truck inherits from Car and defines a constructor and the function GetWeightOfLoad.

```erlang
%class Car
Car(unsigned int NoCylinders)
int GetCurrentVelocity()

%class Truck Car
Truck(unsigned int NoCylinders, unsigned int NoTrailers = 0)
unsigned int GetWeightOfLoad()
```

From this code we will generate four C++ wrapper functions and five wxErlang interface functions. One C++ wrapper function for each function declared in the above example and five interface functions, one for each function plus one extra for the one with the optional argument (see section 4.4.1). They will be implemented as described in section 5.3.

Here follows a listing of each of the Erlang functions with a brief description.

- **car(NoCylinders)** - the constructor of the car class, when this function is called a new car object is created. The argument NoCylinders to this function must be a positive integer. This comes from the fact that the corresponding argument in the specification is unsigned,

- **get_current_velocity(Self)** - returns an integer. The Self argument must be an object that "inherits" from the class Car, e.g. in this the object must be of type car or truck,

- **truck(NoCylinders)**, **truck(NoCylinders, Opts)** - these two functions are the constructors of truck objects. There are two interface functions generated, since the function Truck, in the specification, accepts optional arguments.

In general, when a function from the specification has optional arguments, two interface functions will be generated: one that accepts the optional arguments and one that does not. The reason for generating two declarations is of convenience for the programmer. The reader is referred to section 4.4.1.

The type checking mechanism requires that the arguments used when calling truck is a positive integer. If the optional argument NoTrailers is used, it must also be a positive integer.

- **get_weight_of_load(Self)** - returns a positive integer. The Self argument must inherit from truck, e.g. be of type truck. Note that an object of type car is not accepted.

In general, to define a class that inherits from N classes and that defines M functions we write as follows:

```erlang
%class ClassName Parent1 Parent2 .. ParentN
Type1 FunctionName1(Arg11, Arg12, ..., Arg1O)
Type2 FunctionName2(Arg21, Arg22, ..., Arg2P)
```

4A constructor creates a new object from the class.
TypeM FunctionNameM(ArgM1, ArgM2, ArgMQ)

The function declarations are regular C++ function declarations with special annotation. We will return to the annotation in next section.

It should be stressed that we are not programming, in the sense that we are creating new code. Instead, one should think of it as if we are describing code, since we are only specifying how different C++ classes relate and what functions they define. We will then from the specification generate the corresponding wrapper and interface code. If the specification is erroneous we might end up with incorrect wrapper and interface code. Even worse, we might not be able to detect these errors.

5.4.2 Annotated functions

As described in an earlier section, the notion of functions in C++ differs in many ways from the notion of functions in Erlang. As if this is not enough, function declarations in C++ can be ambiguous, meaning that we can not mechanically look at a C++ function declaration and understand how different arguments are used and how they relate.

For instance, the function declaration

```c++
void GetPoint(int *x, int *y)
```

does not tell whether the pointers x and y are pointers to an int, if they are arrays or if they are used as out arguments.

In order to cope with this problem we annotate the declarations with something called tags.

Suppose we have the function `void GetPoint(int *x, int *y)` and we know that the arguments of this function are out arguments. To describe this in our language we declare the function in the following way.

```
%class SomeClass ...
...
void GetPoint([out]int *x, [out]int *y)
...
```

The tags tells the code generation mechanism to generate code that treats the arguments as out arguments. As described in section 4.4.2, the interface function `get_point(Self)` will be generated. This function returns a tuple `{X, Y}`, where the X value correspond to the first out argument and Y to the other.

As we also saw in section 4.4.2 there are other types of arguments as well. The specification language does not only support tags for these types of arguments, but also a few others. Here follows a listing of the tags and a short description of what each does.

- **[in]** - which denotes an in argument. Arguments with no tag defaults to [in],
- **[out]** - denotes an out argument,
- **[inout]** - denotes an *in-/out argument*, e.g. an argument that is both used as an in argument and as an out argument (see section 4.4.2),
- **[array]** - denotes that an argument is an array (this is currently not supported),
- **[nonnull]** - will make the safety mechanism, described in section 4.2, not allow values that corresponds to NULL.
The annotation currently lacks, at least, one wanted construct. Namely, a way to describe how different arguments relate.

In C/C++ one quite often comes across functions with arguments that describe other arguments of the function, typically the length of an array or so. That is, some function, say, `void DoSomething(int *array, int array_length)` uses the the `array_length` argument to describe the length of the array `array`.

Although, wxErlang does not support arrays currently, in the future, however, the plan is to model C++ arrays as Erlang lists. Then a call to the `void DoSomething(int *array, int array_length)` function would be translated `do_something(ListOfInts, length(ListOfInts))` in wxErlang.

It would be much nicer if the function `do_something` itself could resolve the length of the list, allowing the programmer to simply call `do_something(ListOfInts)`.

5.4.3 Automatic code generation

The reason for much of the wxErlang’s code being generated is due to wxWidgets’ size. Although, our interface specification language and its `class`-statements speed things up, it would still be a very time consuming project to write all these definitions by hand. One would also be required to check and rewrite these whenever a new version of wxWidgets is released. This is simply not feasible; we need a way to generate code directly from wxWidgets. We can do this by using the command `%importclass`.

Using the `%importclass`-statement, we parse and generate code from wxWidgets’ header files. The following example shows how we would add the class `wxFrame` to wxErlang. In this case `wxFrame` will “inherit” from `wxWindow`.

```
%importclass wxFrame wxWindow
```

We could just as well have defined the `wxFrame` using the `class`-statement and then list all its function declarations below.

In general, to generate a class with N parents using the `%importclass`-statement we write:

```
%importclass ClassName Parent1 Parent2 ... ParentN
```

Code generated with the `%importclass` statement assumes that all arguments of all functions are regular in arguments. In most cases this is true, but not always. To change a previously defined function we can just add another modified function declaration. This definition will override the previous one.

Say that the function `int SomeFunction(int *out_arg)` is defined in the class `SomeClass`. The function uses the argument `out_arg` as an out argument, contradicting the assumption used by the `%importclass` statement. The following code snippet will re-define `SomeClass`’s function `int SomeFunction(int *out_arg)` to behave as if `out_arg` is an out argument

```
%importclass SomeClass ...

int SomeFunction([out]int *out_arg) ...
```

It is not obvious how the `%importclass` mechanism works. One could suspect that there has been a C++ header file parser implemented but this is not the case. In fact, parsing C++ header files is very complicated, many different constructs exist with many variations. Implementing such a parser would require a considerable amount of work.

Instead a tool is used to translate the header files into XML documents[31]. These documents are then parsed, using `xmerl` (an Erlang XML Parser), and from the output the code will then be generated.

The tool used to convert the header files into XML documents is called Doxygen[8] and is really an tool for creating documentation from specially annotated header files. Doxygen is very similar to the more well-known tool Javadoc[13], but unlike Javadoc, Doxygen can also generate documentation from C/C++ header files.
5.4.4 Constants

As described in section 4.6, we translate C++ constants into Erlang macros. For instance, the C++ constant `wxEVT_WINDOW_CLOSE` is in wxErlang translated to the macro `?wxEVT_WINDOW_CLOSE`.

The interface specification language has constructs for handling constants in a convenient way. We can easily add constants and also rename them as needed. Say that want to include the C++ constant `wxEVT_WINDOW_CLOSE` in our wxErlang interface. The following example shows how this is done in the language.

```erlang%
%num wxEvt_WINDOW_CLOSE
```

There is also a variation of the `%num`-statement, that allows one to rename a constant. For instance, if we have a constant, say `COLOUR_RED` in C++ and we would like to add it to our interface. We would not, however, want to name it `COLOUR_RED` but `COLOR_RED`. We would then write as the following example.

```erlang%
%num COLOUR_RED COLOR_RED
```

This functionality is conservatively used.

The code generation mechanism will from all the `%num`-statements generate a Erlang header file that contains all macros. How this mechanism works differs a bit from how the wrapper and the interface code is generated (which will be discussed later). Using the `%num`-statements, a C++ program is created. When the program is executed it will write an Erlang header containing all Erlang macros.

This might seem like a complicated way of creating the Erlang header but, in fact, it is probably the simplest way, since the C++ program can access the values of the constants directly.

5.4.5 Other statements

The interface specification language also supports a few other statements. Some of these are statements of a trivial nature and will not be described, others deserve a mention.

```erlang%
%builtin
```

The `%builtin`-statement is used to add "regular" Erlang functions to the wxErlang interface. For instance, the different variants of `connect` (see section 3.3.5) are added to the interface in this way. The code to add the different version of `connect` is shown in the following example.

```erlang%
%builtin wx_core connect(Object, EventType, Fun)
%builtin wx_core connect(Object, EventId, EventType, Fun)
%builtin wx_core connect(Object, FirstId, LastId, EventType, Fun)
```

When a call is made to the function `wx:connect` we will, in this case, just forward the call to the corresponding `connect` function in the `wx_core` module.

```erlang%
%event and %eventid
```

We have not touched events nor the event handling system at all in this chapter. There is, however, a mechanism for declaring a class to be an event class and associating the event with event ids.

We declare a class to be an event class in the following way, and associate a number of event ids with it.

```erlang%
%event wxMouseEvent
%eventid wxEVT_LEFT_DOWN
%eventid wxEVT_LEFT_UP
...```
To put it simple and avoid many details; the associated event ids are used in the event handling system to be able to cast (change the type) the event objects in to the right type.

Blocks of code
In order to do some minor adjustments of the wxErlang interface, our language also supports injection of arbitrary C++ code. Using this functionality, one could, for instance, add an extra C++ class to the interface or just some missing constant. Code that is between "{%" and "%}" will be pasted into the wrapper code.

The following example shows how we could add a new class to the wxErlang interface.

```cpp
{%
/* arbitrary C++ code */
class SillyExample {
public:
    SillyExample() { counter = 0; }
    int Dump() { return counter++; }
private:
    int counter;
};
/* end C++ code */
%

%c{class SillyExample
SillyExample()
int Dump()
%

This would add SillyExample to our interface and we would be able to use just it as any other class. Unfortunately, we can not define such an class using the %importclass-statement.

5.4.6 The compiler
This section will present the interface specification compiler and how the code generation mechanism works. This topic closely relates to section 5.3.1 and we will refer to this section frequently, therefore the reader is assumed to be fairly acquainted with that section and the figures presented there.

The compiler accepts an interface specification file containing statements like those we have seen in previous examples and will from this file generate a number of both Erlang modules and headers and C++ source code files. Here follows a (incomplete) listing of the generated files.

- The C++ wrapper code - this file contains all the wrapper functions generated from the input file. The wrappers are, as described in section 5.3.1, used to convert the values between their different representations and to perform the actual function call,

- The wxErlang main module wx - this source file contains all interface functions and is the wx module that the wxErlang programmer uses. The interface functions are also described in section 5.3.1,

- A header generator - this file is actually a program that, when executed, creates an Erlang header. The Erlang header contains all constants (macros) defined in the input file with the %num statement translated into Erlang, see section 5.4.4.

Wrapper and interface generation
We will now dive into the details of how the wrappers and the interface functions are generated.

When we generate the wrapper- and interface functions we follow the templates described in section 5.3 and figures 5.3 and 5.3.1. In general all functions differ in a few ways, due to differences in the arguments’ type. These differences are in the figures denoted by the code inside "[|" and "]|".

41
To generate type dependent code something called *type handlers* have been introduced. A type handler contains information and functions that perform different operations on the types. For instance, a type handler contains functions for generating code that convert values from Erlang representation to C++ representation, functions for generating type- and sanity checking code, etc. All the type handlers are stored in a database that we can query for specific type handlers.

To clarify further, let us consider an example. In the example we generate an interface function. This process involves, among other things, generating a function name, generating a number of checks that ensure the validity of the arguments and finally the function call that (indirectly) calls the corresponding wrapper function. Interface functions are in detail described in section 5.3.1.

Suppose that we are to generate the interface function for the function

```cpp
bool PopupMenu(wxMenu *menu, int x, int y),
```

which is defined in the C++ class `wxWindow`. The corresponding interface function will be `popup_menu(Self, Menu, X, Y)`.

A call to this function is valid if:

- **Self** - is an object that inherits from `wx_window`,
- **Menu** - is an object that inherits from `wx_menu`,
- **X** and **Y** - are integers.

It is trivial to generate most of the interface functions and we will skip to the non-trivial parts of it, namely, the checks.

The type- and sanity check code for the **Self** argument is generated in the following manner. We query the type handler database for a type handler that takes care of objects of type `wx_window`. Using this type handler we will then generate a check that ensures that the **Self** argument inherits from `wx_window`.

The checks performed on the argument **Menu** is performed in more or less the same way, except that the check ensures that the **Menu** inherits from `wx_menu`.

The information used to generate these types of checks are taken directly from the interface specification file, and is created from the `%class` and `%importclass` statements.

Checking the last two arguments of the function `popup_menu(Self, Menu, X, Y)`, however, differs a bit from checking **Self** and **Menu**. We retrieve the type handlers, like we did in previous examples, but instead of checking for inheritance, we simply check the arguments are of type integer.

**Type handler database**

The type handler database is probably the single most important component in the code generation mechanism, since it is used whenever we generate code that is type dependent. It also enables one to easily extend wxErlang with new type specific behavior.

To make the component even more useful, our language supports constructs for associating different types with other types. When programming in C/C++ one commonly uses the `typedef` construct to give types more meaningful names. Actually, what it does is to generate an alias for a particular type.

Our interface specification language supports a similar construct and is used to unravel aliased types. For instance, the type `wxWindowID` is merely a `typedef` of a regular integer.

To make the code generation mechanism aware of this fact we state:

```erlang
%typedef wxWindowID int
```

This will make the code generation mechanism handle every `wxWindowID` as an `int`.

We will not look at how the rest of the code is generated, since it is indeed generated in a very similar manner; querying the database for type handlers and using them for generating code.
Implementation

This section briefly describes how the compiler is implemented. We will stay at a rather high level avoiding details.

The compiler is implemented in Erlang and is a *multiple pass compiler*, meaning that it will go through the code many times, collecting all needed information. During these passes we, among other things, build class trees from which the type checks are generated.

The parser, used by the compiler, is generated using *yacc*, which is a parser generator written in Erlang, and *lex*, which is a lexer generator written in Erlang. Both these tools are closely related to the more well-known tools *yacc*[15] and *lex*[17].

The compilation is pretty straightforward, first the program is translated into an abstract syntax tree (an internal representation of the specification). The abstract syntax tree is then sent to a number of Erlang functions that generate the C++ wrappers, the Erlang interface, etc.

5.4.7 Possible extensions

The interface specification language is by no means a complete language and neither is the compiler. There are many possible extensions left to be implemented. In this section we will talk about possible extensions to the language and the compiler.

First and foremost, one important missing feature is the ability to generate documentation. Currently, the wxErlang programmers are referred to wxWidgets' C++ documentation. It is not possible to document all the functions of the interface by hand, but it would be possible to document the most commonly used ones. It would also be possible to generate some standard documentation for all functions, containing type signature etc.

It would also be possible to introduce a statement that extracts all C++ constants from a header file, much like the `%importclass` statement extracts function definitions from classes. Such a statement could, probably, be implemented in the same way as the `%importclass` statement, by using Doxygen. See section 5.4.3

Another missing feature is constructs that allow modularization of specification files. This feature is really not needed, but it would be nice to have some way of dividing the interface specification into different files.

If a modularization mechanism were present, it would also be possible to add some constructs to the language, where we specify the behavior of the different type handlers, directly in the input files. See section 5.4.6. These are now ”hardcoded” into the code generation mechanism.

One could also add a mechanism that somehow validates the sanity of the statements, that alerts the user whenever a class inherits from a class not present in the interface and so on. Currently, there are very few such checks done, making it hard to find different errors in the interface specification file.

This list of missing and wanted features could be made very long, but whether one needs, or wants, to implement all these remains to be seen. It might even be worth considering implementing an Erlang back-end for SWIG. SWIG has constructs for most of the possible extensions described in this section and many more. We will get back to SWIG a bit later.
Chapter 6

Other issues and supplementary comments

During the development of wxErlang many different approaches in different situations have been considered. Some where bad, others not as bad. This chapter is dedicated to a few of the approaches that were considered, evaluated and finally rejected.

6.1 The interface

The interface and structure is a very important issue of a library. A library with a bad or a badly structured interface can be nearly useless — no matter what functionality it holds. When wxErlang’s interface was designed, the main concerns were to integrate the object oriented paradigm into Erlang as smoothly as possible, trying to avoid introducing new unwanted concepts. In the search of such an interface many approaches were considered, some of which are briefly described in this section.

Probably the biggest problem with wxWidgets is, as stated earlier, its size. It was quickly realized that it would not be feasible to do everything by hand. It would especially not be possible to document each function of the interface individually. A library with no documentation, however, is very difficult to use. To cope with this problem, it was decided that the interface should closely relate to the one defined by wxWidgets itself. This would enable reuse of the native wxWidgets documentation.

One of the earliest approaches when trying to incorporate wxWidgets into Erlang was to let Erlang modules correspond to wxWidgets classes. It feels like a natural translation of the object oriented paradigm into Erlang, but as it turned out, it was not as good as expected.

Such a translation would require many Erlang modules and the wxErlang programmer is also needed to be very familiar with the wxWidgets class hierarchy. In a sense, wxWidgets is "too" object oriented for this; it defines a lot of classes and also uses abstract classes, etc. This does not fit very well into Erlang and the approach was rather quickly rejected.

The next translation considered was perhaps a bit controversial. It made use of abstract modules. An abstract module is a module containing free variables, much like a lambda abstraction. To be able to use these modules one first needs to bind the free variables and makes an instance of the module.

In our case, we would bind these free variables with the object’s C++ pointer (an integer) and its type information.

This approach did not try to smoothly fit in to the Erlang environment and writing programs using this approach almost felt like writing programs in another language. A call to a method would be of the form:

\texttt{Object:method\_name(Arg1, Arg2, \ldots, ArgN)}

Where \texttt{Object} is an object, \texttt{method\_name} is the method’s name and \texttt{Arg1} is an argument.
Each module must also export all functions that the corresponding C++ class defines and inherits to make sense, otherwise this approach will be both tedious and counter intuitive to use. It would, like the previous approach, require the programmer to be very familiar with the wxWidgets class hierarchy.

The reason for rejecting this approach is due to the fact that abstract modules is a very undocumented and unused feature of Erlang, and that this approach does not "feel" like Erlang.

Finally, the current design of the interface was established.

6.2 Code generation

The other major part of the wxErlang implementation is the code generation mechanism. Many approaches were tried, evaluated and considered. In this section we will look at these and discuss them.

In the early stages of the project, some labor was put into researching currently existing bindings to wxWidgets from other languages. Special notice was taken of how the different bindings generated their interface. Some of our attempts at code generation have been inspired by a few of these, others have tried to "steal" the whole code generation mechanism.

One of the investigated bindings was wxHaskell, which is a binding from wxWidgets to Haskell[11]. WxHaskell generates its interface from wxEiffel’s hand-written(!) wrappers, the interested reader is referred to [16] for more details.

Some effort was, quite successfully, put into rewriting wxHaskell’s code generator making it generate Erlang code instead of Haskell code. It would probably have been possible to use this method to tackle the code generation problem. However, a number of unwanted dependencies would have been introduced. Not only would the code generation mechanism rely on wxHaskell and Haskell but also that the developers of wxEiffel would rewrite the wrappers whenever a new version of wxWidgets is released.

Another binding that was considered was wxPython[29], a Python[21] binding to wxWidgets. WxPython uses SWIG to generate its interface. We have briefly mentioned SWIG in earlier sections but we will now have a closer look at it.

6.2.1 SWIG

Some of the text presented in this section has been shamelessly stolen from the SWIG development documentation[24].

Introduction

SWIG (Simplified Wrapper and Interface Generator) is a software development tool that simplifies the task of interfacing different languages to C and C++ programs. In a nutshell, SWIG is a compiler that takes C declarations and creates the wrappers needed to access those declarations from other languages including Perl, Python, Tcl, Ruby, Guile, and Java. SWIG normally requires no modifications to existing code and can often be used to build a usable interface in only a few minutes. Possible applications of SWIG include:

- building interpreted interfaces to existing C programs,
- rapid prototyping and application development,
- interactive debugging,
- reengineering or re-factoring of legacy software into scripting language components,
- making a graphical user interface (using Tk for example),
- testing of C libraries and programs (using scripts),
- building high performance C modules for scripting languages.
SWIG was originally designed to make it extremely easy for scientists and engineers to build extensible scientific software without having to get a degree in software engineering. Because of this, the use of SWIG tends to be somewhat informal and ad hoc (i.e., SWIG does not require users to provide formal interface specifications as you would find in a dedicated IDL compiler. Instead, SWIG can parse the header files directly). Although this style of development isn’t appropriate for every project, it is particularly well suited for software development in the small; especially the research and development work that is commonly found in scientific and engineering projects.

A primary goal of the SWIG project is to make the language binding process extremely easy. SWIG supports most of C++. Some of the major features include:

- full C99 preprocessing,
- all ANSI C and C++ data-types,
- functions, variables, and constants,
- classes,
- single and multiple inheritance,
- overloaded functions and methods,
- overloaded operators,
- C++ templates (including member templates, specialization, and partial specialization),
- namespaces,
- variable length arguments,
- C++ smart pointers.

Currently, the only major C++ feature not supported is nested classes, a limitation that will be removed in a future release.

It is important to stress that SWIG is not a simplistic C++ lexing tool like several apparently similar wrapper generation tools. SWIG not only parses C++, it implements the full C++ type system and it is able to understand C++ semantics. SWIG generates its wrappers with full knowledge of this information. As a result, you will find SWIG to be just as capable of dealing with nasty corner cases as it is in wrapping simple C++ code. In fact, SWIG is able to handle C++ code that stresses the very limits of many C++ compilers.

When used as intended, SWIG requires minimal (if any) modification to existing C or C++ code. This makes SWIG extremely easy to use with existing packages and promotes software reuse and modularity. By making the C/C++ code independent of the high level interface, you can change the interface and reuse the code in other applications. It is also possible to support different types of interfaces depending on the application.

SWIG is designed to produce working code that needs no hand-modification (in fact, if you look at the output, you probably won’t want to modify it). You should think of your target language interface being defined entirely by the input to SWIG, not the resulting output file. While this approach may limit flexibility for hard-core hackers, it allows others to forget about the low-level implementation details.

SWIG and Erlang

The fact that SWIG lacks an Erlang back-end is very sad. Such a back-end would probably have made the development process of wxErlang a lot shorter; instead of writing a custom code generation mechanism one could have used the one provided by SWIG directly.
The code generation mechanism in wxErlang is heavily dependent on the wxWidgets library and is specially designed for wxWidgets. It only supports the subset of C++ that wxWidgets uses (e.g. no exceptions, no templates, no namespaces, etc). As a consequence of this, it is not in general possible to use the code generation mechanism used in wxErlang to generate other bindings. It would, however, be possible to extend the code generator to support more of C/C++ and make it generate non-wxWidgets dependent code. But a perhaps more attractive way is to implement an Erlang back-end for SWIG.

It should be stressed that implementing an Erlang back-end for SWIG is a very big project. One is required to translate all mechanisms and functionality provided in C and C++ into the Erlang environment, much like the wxErlang does. If the translations used in wxErlang is good in the general case remains unsaid.

If an Erlang back-end for SWIG were implemented, it would enable Erlang programmers to easily and effortlessly create bindings to different C/C++ libraries.

### 6.2.2 Comments on wxWidgets

Integrating a library that is written in one paradigm into another is, as we have seen, not entirely trivial. The way wxWidgets is implemented is really “only” for C++, meaning that it frequently uses the OO mechanisms of C++, making it hard to create bindings to. In this section we will stress a few issues of using wxWidgets as a library to create bindings to.

**Runtime type information (RTTI)**

Keeping type information available at run time means that we can check the types of different objects while the program is running. WxWidgets provides its own implementation of such a system, but unfortunately it is not used consistently; some objects do not use it.

If all objects kept track of their type at runtime, we would probably have been able to implement the safety mechanism (see section 4.2) a more easily.

**Object orientation**

As mentioned earlier wxWidgets also frequently uses the features of OO programming. This also complicates the binding creation to the library since they might not in fit as smoothly into a non-OO language.

For instance, the GTK+ library which is also an OO library for writing GUI applications uses the OO features much more conservatively. From this certain point of view GTK+ might be a more suitable library to create bindings to from a non-OO programming language. There are, however, other reasons and issues to why one would choose wxWidgets instead of GTK+.

**Differences between back-ends**

One also would hope that wxWidgets behaves in the same way on all platforms. Unfortunately, this is not the case. For instance, most classes and functions are defined by all different back-ends, but not all are. This requires that the programmer of cross-platform applications makes sure that the functions and classes used in the application are defined in all back-ends.

There are also other differences performance-wise; tests show that wxErlang using the GTK+ back-end is far faster than the Motif [19] back-end. Why this is so is unclear, but using the Motif back-end in applications that performs many operations over a short period of time is exceptionally slow.
Chapter 7

Related work

Although wxErlang provides the functionality to write GUI applications, the way we are writing them is not using the powerful mechanisms of Erlang very well. One could probably quite easily build abstractions upon wxErlang that are suitable for Erlang. The problem is finding these abstractions. Some attempts with varying degrees of success have been made to incorporate GUI libraries into Erlang.

For instance, the library ex11[9] uses processes to model different widgets and message-passing for getting and setting the different widget properties. Syntactically the interface provided by ex11 is both concise and elegant. Unfortunately, the library is very low-level and implements a limited set of widgets.

X11[18] (X Window System, version 11), which ex11 uses, is an interesting piece of software. The earliest versions of it were developed in the mid 1980's to provide a network transparent graphical user interface. The result was a strict client/server based architecture, where the X server displays the graphics and the X client is responsible for executing the program and continuously sending information to the server about what to display. The information is sent over a TCP/IP socket.

Ex11 exploits this structure and mimics the functionality of an X client.

Another such library is gs[1], which is a high-level API for writing GUI applications. It is designed to be simple, portable and easy to use. The current implementation of gs uses Tcl/Tk[26]. The intention is, like wxErlang and wxWidgets, to support more back-ends (Win32, Motif, etc).

Like in Erlang and ex11 processes play a central role. An Erlang application is often modeled as a hierarchy of processes, gs also tries to use this idiom. Different widgets belong to different processes, making the processes utterly responsible for the widgets. For instance, when a widget triggers an event a message will be sent to the process that owns the widget and a widget will be destroyed when the owning process terminates.

Both these libraries have both their pros and cons; gs is not suitable in all situations since it is too high-level[20] and in others ex11 might be too low-level. A more appropriate approach might be to provide both a high-level implementation and a low-level implementation, like wxHaskell’s two libraries WX and WXCore. The WX library provides an implementation that shows Haskell from its better side. The WXCore library, like wxErlang, is a low-level translation of wxWidgets and fills the “holes” of WX.
Chapter 8

Conclusion

WxErlang is a GUI library written in Erlang that enables Erlang programmers to write cross-platform GUI applications with a native look-and-feel of the underlying platform. WxErlang is built upon the C++ library wxWidgets and is merely a translation of wxWidgets from C++ to Erlang.

By building the library directly upon wxWidgets the time needed to develop wxErlang has been relatively short. Instead of implementing routines to show windows and create buttons, etc., one only needs to implement a translation mechanism that enables Erlang to talk to and use C++ and wxWidgets.

Since wxWidgets is such a large library with more than 2000 methods and 500 classes, writing wxErlang by hand is not possible, instead most of wxErlang is generated. A special interface specification language has been designed for this single purpose. Using the language one describes what classes, methods, constants, etc., that the wxWidgets interface defines. From these descriptions the wxErlang interface and the translation mechanism is generated.

Comparing wxErlang to the gs GUI library, which is the GUI library currently distributed with Erlang, the two libraries are implemented in a very different manner. The gs library is high-level and tries to use many of the features of Erlang to provide a nice way of writing GUI applications. The approach taken in wxErlang is rather low-level and does not use many of the features provided by Erlang. As a consequence of this, applications written using wxErlang are probably in general more verbose than the corresponding gs code. The wxErlang library will, however, provide more functionality as the wxErlang project evolves. Applications implemented using wxErlang also get a much more appealing interface.

WxErlang is a successful implementation of a prototype binding to the wxWidgets GUI library and serves as a proof of concept. The current version of the prototype implements roughly 10% of wxWidgets’ functionality. WxErlang has also been used to implement as small utility application which proves the programming model to be usable. Whether, wxErlang will be suitable in large scale projects remains unsaid, but no indications of the contrary have been found. The combination of features provided by Erlang and the functionality of wxWidgets creates an environment that enables developers to implement GUI applications easily.

Hopefully not only developers will benefit from wxErlang, but also researchers will find wxErlang a useful research platform in pursuit of a more Erlang-aware model of programming GUI applications.
Appendix A

Wrapper function

The following code snippet shows how the function `Hide()`, which is defined in `wxWindowBase`, looks like. The `Hide()` function operates on a window and when invoked it will hide the window and return a boolean corresponding to whether the operation succeeded or not.

The code will not be discussed in detail.

```c
bool wxWindowBase_Hide_ (wxeRequest *input, wxSocketBase *socket) {

    wxArg *arg[1];
    bool retval = false;
    int arity = -1;
    wxReply *reply = wxReply :: NewReplyTuple (input, 1);

    bool out0;

    if (!wxeDecodeArgHeader(input, &arity, 1, 1)){
        wxePrintError ("In wxWindowBase_Hide_: could not decode argument headers\r\n");
        return FALSE;
    }

    memset(arg, 0, 1 * sizeof(wxArg));
    for (int i = 0; i < arity; i++) {
        int arg_no;
        if(!wxeDecodeArg(input, &arg_no)) {
            wxePrintError("In wxWindowBase_Hide_: could not decode argument\r\n");
            return FALSE;
        }
        switch (arg_no) {
            case 0 :
                arg[0] = (wxArg *) malloc (sizeof(wxArg));
                input->DecodeLong(&(arg[0]->value));
                break;
        }
    }

    out0 = ((wxWindow*)(arg[0]->value))->Hide();

    retval = reply->EncodeTupleHeader(2) && reply->EncodeAtom("ok");
    {
        reply->EncodeAtom((const char *)((out0) ? "true" : "false"));
    }

```
retval = retval && reply->SendReply (socket);
if(arg[0]) { free(arg[0]);}

return retval;
}
Appendix B

Interface function

This code snippet shows what a wxErlang interface function looks like. This particular one is the interface function of `hide`, which hides a widget and returns a boolean value corresponding to whether the operation on the widget succeeded or not. The code will not be discussed in detail.

```
hide(Arg0) ->
    case is_record(Arg0, wx_object) andalso
        (lists:member(Arg0#wx_object.type, [wx_window, <all classes that inherit from wx_window> ])
            andalso Arg0#wx_object.ptr /= 0)) of
        true ->
            case wx_core:call('wxWindowBase_Hide_',
                [{0, fun(#wx_object {ptr = ___P___}) -> ___P___ end(Arg0)}], []) of
                {ok, __V__0} ->
                    __V__0;
                Error -> erlang:error({hide, Error})
            end;
        false -> erlang:error({bad_arg,hide})
    end.
```
Bibliography


