Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

A Foreign Function Interface Generator for occam-pi

Damian J. DIMMICH and Christian L. JACOBSEN

Computing Laboratory, University of Kent,
Canterbury, CT2 7NZ, England.

{djd20 , clj3} @kent.ac.uk

Abstract. occam-pi is a programming language based on the CSP process algebra and the pi-calculus, and has a powerful syntax for expressing concurrency. occam-pi does not however, come with interfaces to a broad range of standard libraries (such as those used for graphics or mathematics). Programmers wishing to use these must write their own wrappers using occam-pi's foreign function interface, which can be tedious and time consuming. SWIG offers automatic generation of wrappers for libraries written in C and C++, allowing access to these for the target languages supported by SWIG. This paper describes the occam-pi module for SWIG, which will allow automatic wrapper generation for occam-pi, and will ensure that occam-pi's library base can be grown in a quick and efficient manner. Access to database, graphics and hardware interfacing libraries can all be provided with relative ease when using SWIG to automate the bulk of the work.

Introduction

This paper presents a tool for rapid and automated wrapping of C libraries for occam-π [1], a small and concise language for concurrent programming. While occam-π already has a foreign function interface (FFI)[2], which provides the means for extensibility, creating and maintaining large scale foreign library support manually is time consuming. However, by automating the wrapping of foreign libraries, access can be provided to a large existing codebase from within occam-π, without a prohibitively large investment in time. Both language developers and users will benefit, as both will be able to easily add support for new libraries.

SWIG (Simple Wrapper and Interface Generator)[3] is a multi-language foreign function interface generator, providing the infrastructure needed for automatic library wrapping. Support for the generation of wrappers for individual languages is provided by self-contained modules, which are added to SWIG. This paper describes the occam-π module[1], which was created by the authors to enable automatic generation of occam-π wrappers for C code.

We start the paper by providing details on the background and motivation for this work and the tools used. In section 2 we give a brief overview of the occam-π foreign function interface, followed by implementation details of the occam-π module for the SWIG framework in section 3. Section 4 provides an overview of using SWIG, and section 5 has examples of using SWIG to wrap a simple fictitious C library as well as how occam-π can be integrated with the OpenGL library. Finally, section 6 provides concluding remarks and ideas for future work.

[1]The occam-π module is not at the time of writing part of the official SWIG distribution.

1. Background and Motivation

occam-π is a programming language designed explicitly for concurrent programming and is a concise expression of the CSP[4][5] process algebra. It also incorporates ideas from the pi-calculus[6]. While occam-π is a small language, it is nevertheless powerful enough to express applications ranging from control systems for small robots[7] to large applications modelling aspects of the real world[8]. It is not however, always feasible to program an entire solution exclusively in occam-π. Applications needing to deal with file I/O, graphical user interfaces, databases or operating system services, are not possible unless a very large existing code-base is rewritten in occam-π, or made available to occam-π through existing foreign libraries implementing such services.

occam-π's foreign function interface allows users to reuse existing C code or write certain portions of an application in C. It does however require a large amount of wrapper code to interface the library with occam-π. This is not a big problem when dealing with small amounts of code, but writing a wrapper for even a relatively modest library can quickly become time-consuming, tedious, and therefore error-prone. This becomes a further problem when one considers that the library being wrapped can evolve over time, and the wrappers must be updated to reflect changes in the library in order to be useful.

Without better access to existing libraries and code, it may be difficult to argue that occam-π is a better choice for architecting large, complex systems than other languages. It must be made simpler to leverage the large amount of existing work and infrastructure that is provided through operating system and other libraries. We believe that it is imperative for the future success of occam-π that it does not just evolve the mechanisms, which will be needed to express new and exciting concurrent ideas, but also that it is able to make use of the large amount of existing work, in the form of system, graphical and database libraries, which has gone before it, and which will add functionality to occam-π.

1.1. Interface Generators

The work presented in this paper is not the first to provide automatic wrapping of foreign library code for occam-π. The occam to C interface generator Ocinf[9] was the first widely available interface generator for occam. Ocinf can generate the glue code to wrap C functions, data structures and macros so that they could be used from occam2. Since the occam-π syntax is a superset of occam2's and they share the same FFI mechanisms, it would still be possible to use Ocinf to generate interfaces for occam-π. Ocinf however, has not been maintained since 1996 and relies on outdated versions of Lex and Yacc[10]. It has proven to be difficult to get Ocinf to work, since Lex and Yacc have evolved and no longer support much of the old syntax. Making the Ocinf code base work with the current versions of the tools would require rewriting significant portions of 7,000 lines of Lex and Yacc productions.

With the emergence of SWIG as the de facto standard in open source interface and wrapper generation, we chose not to pursue the Ocinf tool further. The SWIG framework is a general purpose code generator which has been in constant development since 1996. It currently allows more than 11 languages to interface with C and C++ code, including Java, Perl, Python and Tcl. SWIG is a modular framework written in C++, allowing the addition of a new language specific module providing the means for adding interface generation support for virtually any programming language with a C interface. Additionally, SWIG comes with good documentation and an extensive test suite which can help to ensure higher levels of reliability.

Uses of SWIG range from scientific[11] to business[12] to government. While other interface generators exist, they are generally language-specific and not designed to provide wrapper generation capabilities for other languages. SWIG was from the outset designed to be language-independent, and gives it a wide and active user base.

1.2. SWIG Module Internals

SWIG itself consists of two main parts, an advanced C/C++ parser, and language specific modules. The C/C++ parser reads header files or specialised SWIG interface files and generates a parse tree. The specialised interface files can provide the parser additional SWIG specific directives which allow the interface file author to rename, ignore or apply contracts to functions, use target language specific features or otherwise customise the generated wrapper.

The language specific module inherits a number of generic functions for dealing with specific syntax. Functions are overloaded by the modules and customised to generate wrapper code for a given language. The actual wrapper code is generated after the parse tree has undergone a series of transformations, some of which a specific module may take part in. Library functions are provided to allow easy manipulation and inspection of the parse tree. The SWIG documentation[13] provides more detailed insight into how SWIG functions and provides details on how one would go about writing a new language module.

2. An Overview of the occam-π FFI

The occam-π FFI requires that foreign C functions be wrapped up in code that informs the occam-π compiler KRoC [14] how to interface with a given foreign function. This is required as the calling conventions for occam-π and the C code differ. The wrapper essentially presents the arguments on the occam-π stack to the external C code in a manner that it can use. We will illustrate the wrapping process with the following C function:

```
int aCfunction(char *a, int b);
```

occam-π performs all FFI calls as a call to a function with only one argument, using C calling conventions. The argument is a pointer into the occam-π stack, in which actual arguments reside, placed there by virtue of the external function signature provided to the occam-π compiler. The arguments on the stack are all one word in length, and the pointer into the stack can therefore conveniently be accessed as an array of **int**s. In order to correctly access an argument, it must first be cast to the correct data-type, and possibly also dereferenced in cases where the argument on the stack is in fact only a pointer to the real data.

```
void _aCfunction(int w[]) {
  *(int) *(w[0]) = aCfunction((char *)w[1], (int)w[2]);
}
```

The code above defines an occam-π callable external C function with the name `_aCfunction` which takes an array of **int**s. This functions job is to call the real `aCfunction` with the provided arguments, which then performs the actual work.

The array passed to `_aCfunction` contains pointers to data or in some cases the data itself, which is to be passed to the wrapped function, as well as a pointer used to capture the return value of the called function. While a function in C may have a return value, external functions in occam-π are presented as **PROC**s, which may not return a value directly. Instead reference variables can be used for this task. In cases where a function has no return value one simply omits the use of a reference variable to hold the result of the called external function.

In essence, the wrapper function just expands the array **int** `w[]` to its sub components and typecasts them to the correct C types that the wrapped function expects. The occam-π components that completes the wrapping are defined as follows:

```
#PRAGMA EXTERNAL "PROC C.aCfunction(RESULT INT result,
                                    BYTE a, VAL INT b) = 0"
```

```
INLINE PROC aCfunction(RESULT INT result, BYTE a, VAL INT b)
  C.aCfunction(result, a, size)
:
```

The first component is a **#PRAGMA** compiler directive which informs the compiler of the name of foreign function, its type and parameters. The **#PRAGMA EXTERNAL** directive is similar to C's **extern** keyword. Function names are prefixed with one of "C.", "B.", or "BX." for a standard blocking[2] C call, a non-blocking C call and an interruptible non-blocking C call respectively. This prefix is used to determine the type of foreign function call, and is not used when determining the name of the external C function, which should in fact be prefixed with an underscore instead (regardless of its type): the **PROC** `C.aCfunction` will call the C function called `_aCfunction`.

The second **PROC** is optional and serves only to provide a more convenient name to the end user by presenting the wrapped function without the call type prefix. While this is not strictly necessary, it enables the wrapper to provide an interface which follows the wrapped library closer.

As demonstrated, it should be clear that manually producing wrapper code for a small number of functions is not a problem. However writing such code for larger bodies of functions is laborious and error prone. The OpenGL[15] library is prime example of a library where automation is a must, as the library consists of over five hundred functions.

More information on how to use KRoC's foreign function interface and various types of system calls can be found in D. C. Wood's paper[2]. Details of performing non-blocking system calls from KRoC can be found in Fred Barnes' paper[16]. Non-blocking foreign functions ("B." and "BX." prefixes) can not currently be generated automatically by SWIG, and is an area of future work.

3. Using SWIG to Generate Wrappers

The wrapper code generated by SWIG is much the same as one would generate by hand as demonstrated above. In this section we will provide more detail on how the occam-π SWIG module performs the mapping from the interface file to the generated wrapper.

3.1. Generating Valid occam-π PROC Names

In order to allow C names to be mapped to occam-π, all '_' characters must be replaced by '.' characters. This is done as the occam-π syntax does not allow underscore characters in identifier names. A function such as **int** `this_function(`**char** `a_variable)` would map to **PROC** `this.function(`**RESULT INT** `r, `**BYTE** `a.variable)`. The only real effect this has is on function names and **struct** naming since parameter names are not actually used by the programmer.

3.2. Autogenerating Missing Parameter Names

SWIG needs to to generate parameter names automatically for the occam-π wrappers should they be absent in function definitions. Consider a function prototype such as:

```
int somefn(int, int);
```

SWIG will automatically generate the missing parameter names for the **PROC**s which wrap such functions. This does not affect the user of the wrappers, as the parameter names are of

[2]by blocking, we mean blocking of the occam-π runtime kernel

no importance, other than possibly providing semantic information about their use. Parameter names are however necessary in order to make the occam-π wrapper code to compile.

The code listed above would map to a **PROC** header similar to:

```
PROC somefn(RESULT INT return.value, VAL INT a.name0, VAL INT a.name1)
```

The occam-π module for SWIG generates unique variable names for all autogenerated parameter names in **PROC** headers, ensuring that there are no parameter name collisions.

3.3. Data Type Mappings

The mapping of primitive C data types to occam-π are straightforward, as there is a direct association from one to the other. The mappings are based on the way parameters are presented on the occam-π stack during a foreign function call. For example an occam-π **INT** maps to a C **int** ∗ (that is, the value on the occam-π stack is a pointer to the pass by reference **INT** and dereferencing is needed to get to the actual value). The complete set of type mappings can be found in [2].

3.4. Structures

C's **struct**s can be mapped to occam-π's **PACKED RECORD**s. Ordinary occam-π **RECORD**s cannot be used, as the occam-π compiler is free to lay out the fields in this type of record as it sees fit. **PACKED RECORD**s on the other hand are laid out exactly as they are specified in the source code, leaving it up to the programmer (or in this case SWIG) to add padding where necessary. As an example, the following C **struct**:

```
struct example {
  char  a;
  short b;
};
```

would map to the following **PACKED RECORD** on a 32 bit machine:

```
DATA TYPE example
  PACKED RECORD
    BYTE  a:
    BYTE  padding:
    INT16 b:
:
```

The handling of **struct**s is somewhat volatile however as it relies on C **struct**s being laid out in a certain way. This may not necessarily be the case across different architectures or compilers and certainly not when wordsizes or endianness differ. This makes the use of **struct**s a potential hazard when it comes to the portability of the generated wrapper.

In cases where this would be a problem, it is possible to use the set of C accessor and mutator functions automatically generated by SWIG for the structure. These can be used by the occam-π program to access and mutate a given structure. It should even be possible for SWIG to produce code to automatically convert from a occam-π version of a structure to a C version (and vice versa), in order to provide more transparent **struct** access to the end user. This would of course be significantly slower than mapping the structures directly into occam-π **PACKED RECORD**s.

3.5. Unions

A C **union** allows several variables of different data types to be declared to occupy the same storage location. Since occam-π does not have a corresponding data type a workaround needs to be implemented. The occam-π **RETYPES** keyword allows the programmer to assign data of one type into a data structure of another. A **struct** that is a member of a **union** could then be retyped to an array of bytes. This is useful since the **PROC** wrapping a function that takes a **union** as one of it's arguments can take an array of **BYTE**'s which are then cast to the correct type in the C part of the wrapper. This means that any data structure which is a member of a **union** can be easily passed to C. Functions which return **union**s can return a **char** * which can then be retyped to the corresponding structure in occam-π. The remaining difficulty with this approach is that occam-π programmers need to make sure that they are retyping to and from the correct type of data, as it is easy to mistakenly assign the **BYTE** array to the wrong data structure and vice versa. The occam-π compiler, like the C compiler, is unable to check for the correctness of such an assignment.

3.6. Pointers and Arrays

It is not always possible to know if a pointer is just a pointer to a single value, or in fact a pointer to an array. Different wrapping functions would be needed in each case. The problem occurs as in C, an array can be specified as using square brackets, **int** `a[]`, or as a pointer, **int** `*a`.

By default pointers are treated as if they are not arrays, and mapped into a pass by reference occam-π parameter. If a parameter actually does refer to an array, it is possible to force SWIG to generate a correct wrapper for that function by prepending 'array_' to the parameter name. Examples of this are provided throughout the paper where needed.

3.7. Typeless pointers

The current default behaviour for type mapping **void** * to occam-π is to use an **INT** data type. Since **void** * can be used in a function which takes an arbitrary data type, this restricts its usage somewhat to only allow **INT**'s to be passed to the function. As an example of mapping a **void** *, OpenGL's glCallLists function is shown here:

```
void glCallLists(GLsizei n, GLenum type, const GLvoid *array_lists);
```

Note that the 'array_' string is prepended to the 'lists' variable name, to indicate that it receives an array. The `GLsizei n`[3] variable tells the function the size of the data type being passed to it, the `GLenum type` variable specifies the type being passed into the `GLvoid *array_lists` so that the function knows how to cast it and use it correctly. Since the default type mapping behaviour here is to type map a C **void** * to an occam-π **INT**, some of the ability to pass it data of an arbitrary type is lost. So, when calling glCallLists from occam-π one always has to specify that one is passing an integer to glCallLists, by passing the correct **enum** value. Here is an example of calling glCallLists from occam-π:

```
-- A simple PROC that takes in an arbitrarily length array
-- of BYTEs and prints them to screen.  The code needs to
-- cast the BYTEs to INTs so that they can be passed
-- to the wrapped glCallLists function.
PROC printStringi(VAL []BYTE s, VAL INT fontOffset,
                  VAL INT length)
  MOBILE []INT tmp:
  SEQ
    tmp := MOBILE [length]INT
```

```
    SEQ i = 0 FOR length
      tmp[i] := (INT (s[i]))
    glPushAttrib(GL.LIST.BIT)
    glListBase(fontOffset)
    -- SIZE s returns the number of elements in s.
    glCallLists(SIZE s, GL.UNSIGNED.INT, tmp)
    glPopAttrib()
:
```

It is possible, by manually writing some C helper functions, to allow the end users of a library to pass a greater range of data types to functions taking **void** * parameters. This can be done by writing proxy C functions for every type of data that the original function accepts, which then calls the typeless function with the appropriate parameters. In this way it would for example be possible to provide a **PROC** glCallLists.R32(**VAL INT** n, []**REAL32** lst) which accepts REAL32 data. Access to **PROC**s accepting other types can be provided in a similar manner. It may even be possible to let SWIG automate much of this work by using its macro system.

3.8. Dealing with Enumerations

C **enum**s allow a user to define a list of keywords which correspond to a growing integer value. These are wrapped as series of integer constants. So for an **enum** defined as:

```
enum example {
  ITEM1 = 1,
  ITEM4 = 4,
  ITEM5,
  LASTITEM = 10
};
```

the following occam-π code is generated:

```
-- Enum example wrapping Start
VAL INT ITEM1 IS 1:
VAL INT ITEM4 IS 4:
VAL INT ITEM5 IS 5:
VAL INT LASTITEM IS 10:
-- Enum example wrapping End
```

If several enumerations define the same named constant, a name clash occurs when the wrapper is generated. If this is a problem, it is possible to change the names of the **enum** members in the interface file (while not affecting the original definition). This will in turn affect the names of the generated constants in the wrapper, thus making it possible to avoid the name clash.

Should **enum** name clashes be a regular occurrence, it would be possible to implement an option of naming the **enum**s differently to ensure that the wrapped constants have unique names. For example, the wrapped constant names above could be generated as follows **VAL INT** example.ITEM1 **IS** 1:, using the **enum**'s name as a prefix to the constants name. The programmer using the wrapped code would have to be aware of the convention used in the names of **enum** constants.

[3] GLsizei, GLenum and GLvoid are simply C **typedef** declarations, mapping them to **int**, **enum** and **void** types respectively. These are used to enable more architecture independent code, should types work slightly differently on other platforms.

3.9. Preprocessor directives

C's **#define** preprocessor directives are treated similarly to **enum**s. Any value definitions are mapped to corresponding constants in occam-π. More complex macros are ignored. The following listings show how some **#define** statements map to occam-π:

```
/* C */
#define AN_INTEGER 42
#define NOT_AN_INTEGER 5.43
```

```
-- occam-pi
VAL INT AN.INTEGER  IS 42:
VAL REAL64 NOT.AN.INTEGER  IS 5.43:
```

3.10. Wrapping Global Variables

SWIG's default behaviour for wrapping global variables is to generate wrapper functions, which allow the client language to get and set their values. While this is not an ideal solution in a concurrent language, as one could be setting the global variable in parallel leading to race conditions and unpredictable behaviour, it is the simplest solution. occam-π itself does not normally allow global shared data other than constants.

There are plans to address this issue by adding functionality to the SWIG occam-π module, which will allow the usage of a locking mechanism, such as a semaphore, to make sure that global data in the C world does not get accessed in parallel. The wrapper generator could generate two wrapper **PROC**'s for getting and setting the global variable, as well as a third **PROC**, which would need to be called by the user to initialise the semaphores at startup. occam-π provides an easy to use, lightweight semaphore library, and it would therefore be easy to manage access to global data from occam-π.

If a library is not itself thread safe, the end user of the library currently needs to be aware of the dangers presented by global shared data, if the library contains it.

4. Using SWIG

Using SWIG is very simple. From the command line you can generate wrappers from a C header file by running the command:

```
$ swig -occampi -module myheader myheader.h
```

where myheader.h is the C header file that contains the function definitions of the functions that you would like to make use of from occam-π.

In many cases it is enough to simply point SWIG at a C header file and have it generate a wrapper for occam-π from that header file. However it is generally better to take a copy of the C header (.h) file, which describes the functions and data-structures to be wrapped and copy that into a SWIG interface (.i) file. So when wrapping the OpenGL library, gl.h, the OpenGL header file, would be copied to gl.i. SWIG specific directives can be defined at the head of the file, such as defining the name of the module, which will determine the names of the generated wrappers. At the top of the interface file we might add the following code:

```
%module gl
%{
#include <GL/gl.h>
%}
```

This names the SWIG generated wrappers "gl" and tells SWIG to include the code **#include** <GL/gl.h> in the generated wrappers, so that, when compiled they are able to reference the targets original header files. To have SWIG generate a wrapper for occam-π from the newly created .i interface file the following command would be run:

```
$ swig -occampi gl.i
```

Note, that the "-module" command line option is no longer needed, since the module name is specified by the %module gl directive above.

As the previous section already stated, the occam-π modules default behaviour is to typemap pointers to single length corresponding primitives in occam-π. The interface file must specify which pointers are pointers to arrays. This can be done by modifying the name of the variable that is to be typemapped by prefixing it with 'array_'. For example, in the OpenGL interface file, for the function call glCallLists, which takes an **int**, an **enum** and an array of **void** * one would modify the code from:

```
void glCallLists(GLsizei n, GLenum type, const GLvoid *lists);
```

to the following:

```
void glCallLists(GLsizei n, GLenum type, const GLvoid *array_lists);
```

When SWIG is run, it generates three files: modulename_wrap.c, modulename_wrap.h and occ_modulename.inc, where modulename is the name that the interface file specifies with the %module directive. The generated C files can then be compiled and linked into a shared library. On Linux one would run the commands:

```
$ gcc -I. -g -Wall -c -o modulename_wrap.o modulename_wrap.c
$ ld -r -o liboccmodulename.so modulename_wrap.o
```

The .inc file needs to then be included in the occam-π program with the following directive:

```
#INCLUDE "occ_modulename.inc"
```

This previously created shared library is then linked in to the KRoC binary along with the library that has just been wrapped. This command may need to be modified to include the correct library include and linking path.

```
$ kroc myprogram.occ -I. -L. -loccmodulename -lwrappedlibrary
```

SWIG has many other features which are not specific to the occam-π module, designed to aid the interface builder in creating more advanced interfaces between higher-level languages and C. These are fully documented in the SWIG documentation[13].

5. Examples

5.1. A Simple Math Library Demo

This example is written to illustrate how one would use SWIG to interface with C code. A basic knowledge of occam-π and C will help to understand the example but is not necessary. In order to build the listed code, KRoC, SWIG and gcc are required.

For this example we are using a fictitious floating point library called "calc.c" which contains a range of standard floating point arithmetic functions. The listing on the following page shows the header file for this "calc.c" library.

```
float add(float a, float b);
float subtract(float a, float b);
float multiply(float a, float b);
float divide(float a, float b);
float square(float a);
```

An interface file for SWIG was created from the calc.h header file. In order to create the interface file, the calc.h header file was copied to a new file called calc.i. This file was then modified to look like the listing below. The first line tells SWIG what the names of the generated wrapper files are to be called. The next three lines inform SWIG that it should embed the `#include "calc.h"` statement into the generated C header file. It is in the interface file that any additional information, such as whether pointers to data are arrays or single values must be included. In this case the only modifications to the interface file that are needed are the four lines of code that where added at the start of the file:

```
%module calc
%{
#include "calc.h"
%}
float add(float a, float b);
float subtract(float a, float b);
float multiply(float a, float b);
float divide(float a, float b);
float square(float a);
```

The occam-π program calculate.occ was written to demonstrate the use of the C functions:

```
#USE "course.lib"
#INCLUDE "occ_calc.inc"

PROC main (CHAN BYTE kyb, scr, err)
  INITIAL REAL32 a IS 4.25:
  INITIAL REAL32 b IS 42.01:
  REAL32 result:
  SEQ
    out.string("SWIG/Occam—pi example for CPA 2005*n*n", 0, scr)
    add(result, a, b)
    out.string("Result of addition: ", 0, scr)
    out.real32(result, 0, 3, scr)
    subtract(result, a, b)
    out.string("*nResult of subtraction: ", 0, scr)
    out.real32(result, 0, 3, scr)
    multiply(result, a, b)
    out.string("*nResult of multiplication: ", 0, scr)
    out.real32(result, 0, 3, scr)
    divide(result, a, b)
    out.string("*nResult of division: ", 0, scr)
    out.real32(result, 0, 3, scr)
    square(result, a)
    out.string("*nResult of squaring: ", 0, scr)
    out.real32(result, 0, 3, scr)
    out.string("*n*n",0,scr)
:
```

A build script was then written which incorporates SWIG in the build process:

```
#!/bin/bash

#Generate wrappers
swig -occampi calc.i

#compile C source file
gcc -c -o calc.o calc.c

#compile C wrapper file
gcc -c -o calc_wrap.o calc_wrap.c

#link source and wrapper into shared library
ld -r -o libcalc.so calc.o calc_wrap.o

#compile occam control code, linking in newly
#created library and the course library.
kroc calculate.occ -L. -lcalc -lcourse
```

That is all that is required for a simple set of functions to be wrapped.

5.2. Wrapping OpenGL

The initial development of the SWIG occam-π module was initiated by the need for a robust graphics library for occam-π. The OpenGL library was chosen as the target to be wrapped since it is an industry-standard which is supported on most modern platforms, often with hardware acceleration. The OpenGL standard itself contains no window management functionality or support for GUI events, so another library must provide the functionality needed to open a window which establishes an OpenGL rendering context as well as an input interface. For the window management, the SDL graphics and user interface library was chosen, due to its simplicity and high level of cross-platform compatibility. A subset of the SDL library was wrapped to allow the user to create and control windows as well as creating a rendering context for OpenGL.

In order to create the wrapper for OpenGL the header files gl.h and glu.h where copied to gl.i and glu.i respectively. The newly created .i files had the following code added to their headers (for gl.i and glu.i respectively):

```
%module gl
%{
#include <GL/gl.h>
%}
...
```

```
%module glu
%{
#include <GL/glu.h>
%}
...
```

A third file was then created called opengl.i which linked the previous two modules together into one:

```
%module opengl

%include gl.i
%include glu.i
```

Finally, the SWIG occam-π module was run to generate the wrappers:

```
$ swig -occampi opengl.i
```

This generated three files, opengl_wrap.c, opengl_wrap.h and occ_opengl.inc. To make use of the OpenGL library, the C wrappers where compiled into a shared library and the

occ_opengl.inc file was added to the program requested. The wrappers along with more detailed instructions on how to generate them can be found at `http://www.cs.kent.ac.uk/people/rpg/djd20/project.html`

An application using the OpenGL library is described in [17]. Figure 1 is an image of the running application, depicting a cellular automaton written in occam-π.

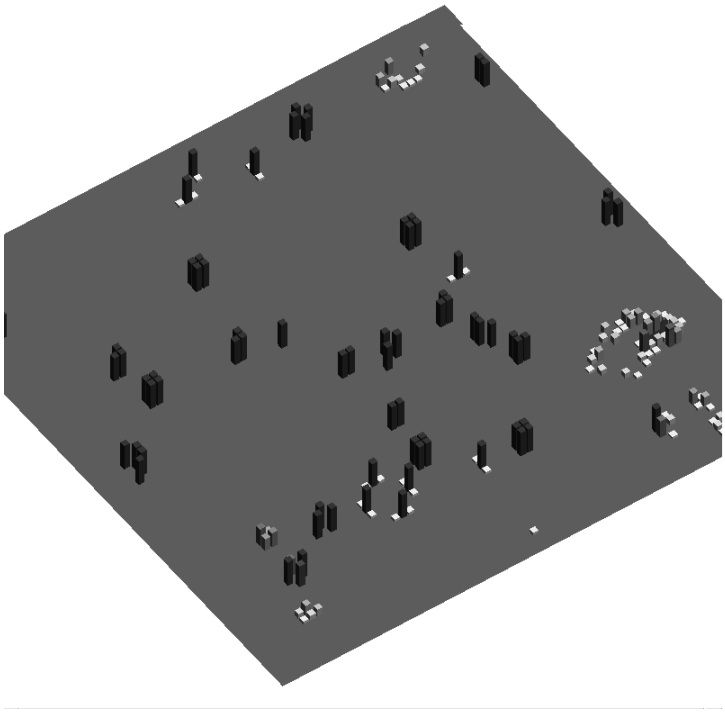

Figure 1. Lazy simulation of Conway's Game of Life.

6. Further Work

6.1. occam-π as a Control Language

A number of high level languages have been used as control or infrastructure languages for legacy C code, allowing the user to combine the ability to express more concisely things that would be difficult in C, as well as harnessing the speed and power of the existing C code. The existing C code may be in the form of libraries or legacy applications, as well as new code specifically written for an application. High level languages are able to provide features not often often found in lower level languages, such as pattern matching, higher order data structures, or in the case of occam-π, a powerful set of concurrency primitives.

Higher level languages are often considered to be simpler to maintain than lower level programming languages in terms of the infrastructure that one is able to create with them. The additional syntax-enforced structure that control languages provide allow for a cleaner implementation of the entire system. It has been noted in [11] that the overall quality of the code, including that of the faster, lower level code was improved through the use of a stricter, more structured control language as the legacy code is adapted to work better with the control infrastructure.

Further areas of exploration could involve experimenting with occam-π to help ease the parallelising of scientific code or exploring the use of occam-π as a control infrastructure for robotics or sensor networks. It would be interesting to work with the Player/Stage[18] project for example, which is used for modelling robotics and provides a comprehensive staging environment for these.

The authors feel that occam-π's CSP based model would be very suitable for parallelising existing code as well as making use of technologies for distributing CPU workload across different machines using upcoming technologies such as KRoC.net[19]. SWIG also allows for easy wrapping of libraries such as MPICH2[20] or LAMMPI[21], allowing occam-π to take advantage of industry-standard MPI communications mechanisms on clusters. An MPI library wrapper for occam-π will be available shortly.

While the goal of the occam-π module was to initially support the wrapping of C libraries, there is nothing preventing it from supporting the wrapping of existing C++ code. SWIG is capable of parsing and generating C code and target language wrappers for most C++ code. A future modification to the SWIG occam-π module would be to add support for wrapping C++ classes. SWIG creates C++ wrappers by generating C wrapper code which is much simpler to interface with by foreign function interfaces. Wrapping of C++ code is not yet fully supported by the occam-π module for SWIG.

Adding C++ support would allow a larger codebase to be used from occam-π. One could potentially wrap C++ classes as mobile processes and call member methods by communicating with them down channels, leaving them with a slightly object oriented feel.

6.2. Further Improvements to the SWIG Module

Throughout the text we have mentioned several areas where the occam-π SWIG module could be improved in order to generate code automatically more often, with less intervention by the user. These areas will be addressed as the occam-π module approaches maturity.

Non-blocking and interruptible non-blocking C calls, as mentioned in section 2 on page 237, are currently not supported. It should be possible, by using SWIG's 'feature' directive, to allow users to mark which functions they want wrapped as blocking ("C"), non-blocking ("B") and/or interruptible non-blocking ("BX"). Automatic wrapping of interruptible non-blocking systems calls would be especially desirable, they need to be further wrapped in a reasonable amount of template occam-π code in order to make them useful.

The issue of cross platform compatibility of generated wrappers when using C **`struct`**s needs to be investigated, and a good default behaviour for the module must be chosen. The effect of name classes for **`enum`**s likewise needs to be investigated, and a default behaviour needs to be decided upon. Finally it would be desirable for SWIG be able to automatically generate code for functions using typeless pointers, so that they can be passed a range of data types. Further investigation of the SWIG macro system, which implements a similar feature for `malloc` and `free`, would be needed.

6.3. occam-π on Other Platforms

With the development of the Transterpreter[22] which can be obtained from `http://www.transterpreter.org/` it is possible to run occam-π applications on practically any platform which has a C compiler. A Symbian port of the Transterpreter is close to completion which will allow it to run on Nokia Series 60 and similar class devices. The Transterpreter also runs on the LEGO Mindstorms, custom robotics hardware and standard desktop hardware. The recent release of a new OpenGL based standard for graphics specifically targeted at mobile devices, called OpenGL ES[23] would allow, with SWIG generated wrappers, one to write mobile phone applications or games on such devices.

Currently the Transterpreter runs as a little endian machine on all platforms, which is a problem when using the FFI on a big endian machine, as passed parameters have the wrong endianness. While it might be possible to instrument the occam-π SWIG module to generate code to byte swap arguments as they pass in and out of C functions, we are planning on eventually running the Transterpreter with the same endianness as the host architecture. While this would solve the problem, the changes needed in the compiler to mark up data contained

in the bytecode file, have not yet been implemented. Such metadata would enable endianness correction at load time.

Acknowledgements

Many thanks to Matthew Jadud for the valuable feedback he gave us whilst writing this paper, as well as regularly suggesting good ideas or new avenues to explore.

References

[1] F.R.M. Barnes and P.H. Welch. Communicating Mobile Processes. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, volume 62 of *WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575*, pages 201–218, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[2] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent, Canterbury, August 1998.

[3] David M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. 4'th annual Tcl/Tk workshop, 1996.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[5] W. A. Roscoe. *Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[6] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes – parts I and II. *Journal of Information and Computation*, 100:1–77, 1992. Available as technical report: ECS-LFCS-89-85/86, University of Edinburgh, UK.

[7] Mathew C. Jadud Christian L. Jacobsen. Towards concrete concurrency: occam-pi on the LEGO Mindstorms. St. Louis, Missouri, USA, February 2005. SIGCE'05.

[8] Theory underpinning nanotech assemblers, 2005. http://www.cs.york.ac.uk/nature/tuna/.

[9] C. S. Lewis. OCINF - The Occam-C Interface Generation Tool. Technical report, Computing Laboratory, University of Kent, Canterbury, 1996.

[10] Doug Brown John Levine, Tony Mason. *lex & yacc*. O'Reilly, 1992.

[11] David M. Beazley. Feeding a large-scale physics application to Python. 6th International Python Conference, San Jose, California, 1997.

[12] Greg Stein. Python at Google. Google at PyCon 2005, March 2005.

[13] David M. Beazley et al. SWIG-1.3 Documentation. Technical report, University of Chicago, 2005.

[14] P.H. Welch, J. Moores, F.R.M. Barnes, and D.C. Wood. The KRoC Home Page, 2000. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`.

[15] Tom Davis Mason Woo, Jackie Nieder and Dave Schriener. *OpenGL Programming Guide, Third Edition*. Addison Wesley, Reading, Massachusetts, third edition, 1999.

[16] F.R.M. Barnes. Blocking System Calls in KRoC/Linux. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures*, volume 58 of *Concurrent Systems Engineering*, pages 155–178, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[17] A.T. Sampson, P.H. Welch, and F.R.M. Barnes. Lazy cellular automata with communicating processes. In J. Broenink, H. Roebbers, J. Sunter, P. Welch, and D. Wood, editors, *Communicating Process Architectures 2005*, Amsterdam, The Netherlands, September 2005. IOS Press.

[18] The Player/Stage project, 2005. http://playerstage.sourceforge.net/.

[19] Mario Schweigler, Fred Barnes, and Peter Welch. Flexible, Transparent and Dynamic occam Networking with KRoC.net. In Jan F Broenink and Gerald H Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 199–224, Amsterdam, The Netherlands, September 2003. IOS Press.

[20] The MPICH2 project home page, 2005. http://www-unix.mcs.anl.gov/mpi/mpich2/index.htm.

[21] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[22] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Dr. Ian R. East, Prof David Duce, Dr Mark Green, Jeremy M. R. Martin, and Prof. Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 99–106. IOS Press, Amsterdam, September 2004.

[23] OpenGL ES, 2005. http://www.khronos.org/opengles/spec/.