Porting the UAL SXF Parser to a Windows Platform

BNL/SNS TECHNICAL NOTE

NO. 123

Yuri Malitsky

BNL, Upton, NY 11973, USA

August 22, 2003

COLLIDER-ACCELERATOR DEPARTMENT
BROOKHAVEN NATIONAL LABORATORY
UPTON, NEW YORK 11973

Porting the UAL SXF Parser to a Windows Platform

Yuri Malitsky

I. Rationale

The Unified Accelerator Libraries (UAL) is a modularized environment for "simulating a variety of properties of a variety of accelerators using a variety of simulation codes and methods [**1**]." Because of this wide spectrum of applications, support of multiple platforms is an essential requirement for the UAL software.

UAL was originally developed on the Sun platform with the native Sun WorkShop CC and GNU gcc. Working simultaneously on these two compilers proved to be an ideal combination, because while the WorkShop provided a commercial quality integrated development environment (IDE), the gcc improved the portability of the software. Recently, because of the sharp increase in demand for parallel applications and the rise of Linux popularity, UAL has been transferred from Sun to the Linux platform. Linux, however, does not have an IDE that can compete with the commercial toolkits available on Windows, such as Microsoft Visual Studio and Borland C++ Builder. Therefore it is tempting to port UAL to Windows and reestablish the previous development environment with two different compilers and a commercial IDE.

As a first step to this porting process, the Standard eXchange Format [**2**] parser has been chosen since it is a part of UAL but was also designed as an independent tool. The aim of this project was to establish a procedure and to explore any problems associated with the deployment process.

II. Infrastructure Issues

Despite the fact that the UAL software will be ported onto different platforms, it should remain as a single source code, working smoothly regardless of what platform it is installed on. It was therefore necessary that one environment at least vaguely resembled the other so that development and installation could be done as easily as possible.

Because UAL has already been established on Unix it was decided that the Windows version have the same *file structure* that UAL was used to (Figure 1). The tars folder would hold the various tarred tools, such as gperf, that would be opened and compiled at the start of the UAL compilation. The .exe and .lib files would be created int bin/msvc and lib/msvc rolders, respectively. The sxf directory contains the original SXF code, the only difference being the makefile used.

Another important aspect was *code management*, particularly, compiling and building applications. In Unix, this task is described in the makefiles. In Visual Studio, this process is also automated, but it is done with the help of Solution and Project files.

The Visual Studio Solutions function as containers that activate the compilation of Projects under their control. A Project includes a set of source files and related metadata such as component references and building instructions. Despite the differences in code management between Unix and Windows environments, just like with file structure, they had to be made similar. In order to do this, I used the "adaptor" Makefile Project that delegated commands from Visual Studio IDE to Windows nmake.

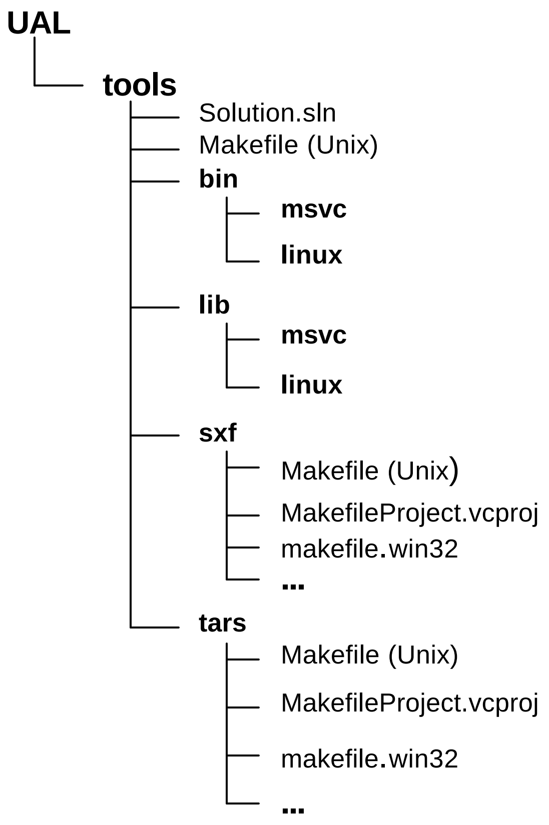

Figure 1. File Structure

III. SXF Parser

Standard eXchange Format (SXF) is a portable and fully-instantiated lattice description format used for capturing "snapshots" of actual lattice conditions, encountered during operations, and using them for offline simulation and "post mortem" analysis [**1**]. The SXF parser was chosen for this experiment for two major reasons. The first being that it is part of the UAL and yet designed as a generic tool, employed by various accelerator programs. And second, it was written and structured in the same manner as the rest of UAL. Besides, SXF, relies on the following external tools employed by other UAL modules:

- Flex [**3**] – a tool for generating programs that recognize lexical patterns in text. Flex reads in the given input files for a description of a scanner to generate. As output flex generates a C source file, 'lex.yy.c,' which defines a routine 'yylex().' This file is compiled and linked with the '-lfl' library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.
- Bison [**4**] – a tool for creating a parser that converts a grammar file into a C source file that parses the language described by the grammar. The job of the created parser is to group tokens into groupings according to the grammar rules, for example, to build identifiers and operators into expressions. The tokens themselves come from a function called the lexical analyzer that must be supplied in some fashion, in our case this is the yylex() routine created by flex.
- Gperf [**5**] – a perfect hash generator written in C++ that transforms an *n* element user-specified keyword set into a perfect hash function. The

function uniquely maps keywords specified onto the range 0 to k, when $k >= n$. If $k = n$ then the function is a minimal perfect hash function. Gperf generates a 0 to k element static lookup table and a pair of C functions. These functions determine whether a given character string occurs in specified keywords, using at most one probe into the lookup table.

Flex and bison are a part of GNU software and the Unix platform and therefore were not included with the UAL application. Unfortunately this was not the case for Windows and so the following options had to be considered (in order of preference):

1. Original GNU code of the tools that would be compiled in Visual Studio.
2. Modified source code adapted for the Visual Studio.
3. Executable programs that implement the functionality of the original tools.

A version of flex that could function as specified by the first option exists [**6**], however such a variant turned out to be non-available with bison. The GNU bison [**7**] is intertwined with C and C++ files that refused to work if not run in their particular language. A prepared executable file of bison was readily available [**8**], yet using it would mean committing to a highly importable and inflexible solution. Thus the second option was investigated [**9**] and incorporated into the SXF application. Yet because bison is dependent upon the output given by flex, Yusuf Motiwala's approach was replaced by a Windows modified flex, distributed with bison [**9**], to avoid any future compatibility problems.

The gperf tool is not a part of the Unix platform and so was distributed with the UAL software. With the use of a makefile the tool compiled without any major difficulties.

When all three of the tools were successfully compiled, it was still a concern if they would collaborate and work in unison under SXF. Fortunately, this proved not to be a problem since the SXF was compiled effortlessly and then run the sxf_tester executable provided with the SXF examples.

IV. Summary

The paper presents the procedure for porting the UAL software onto the Windows platform. It considers different aspects usually associated with the software development, such as file system organization, code management, selection and adjustment of compiler arguments. Its first application, SXF parser, encompasses the deployment of several GNU tools (such as bison, flex, gperf), and has been successfully compiled and tested against the UNIX version.

V. Acknowledgment

I would like to thank Jie Wei and Larry Hoff for their support and Nikolay Malitsky for making my last summer here an enjoyable one.

References:

1. N. Malitsky and R. Talman, UAL User Guide. BNL, December 20, 2002
2. H. Grote, J. Holt, N. Malitsky, F. Pilat, R. Talman, C.G. Trahern. "SXF (Standard exchange Format): definition, syntax, examples," RHIC/AP/155, August 10, 2003.
3. "GNU flex Manual," http://www.gnu.org/manual/flex-2.5.4/html_mono/flex.html
4. "GNU bison Manual," http://www.gnu.org/manual/bison-1.35/html_mono/bison.html
5. "GNU gperf Manual," http://www.gnu.org/manual/gperf-2.7/html_mono/gperf.html
6. Y. Motwala, "Building Flex for/with Microsoft Visual C++ 6.0 and .Net" http://www.geocities.com/ymotiwala/flex.html
7. GNU, "Index of /gnu/bison," http://ftp.gnu.org/gnu/bison
8. Unix Tools Community, "Tools Warehouse," http://www.interopsystems.com/tools/warehouse.htm
9. S. Seymour, "Setting Up and Using Flex (Lex for Windows with Visual C++ 6.0." http://people.bu.edu/kalathur/cs568_spring_03/FlexTutorial.html

Appendix A

The following is a list of discrepancies between the Unix and Windows that have been encountered during the course of this project.

A1. Makefile

- **Windows uses the backslash (\) instead of the Unix forward slash (/) to define file paths.**
  Change all of the forward slashes to backslashes. So while a Unix path would be defined as "./tools/sxf," in Windows it would be written as ".\tools\sxf."

- **Windows nmake is only able to decipher a few specific extensions by default (.exe, .obj, .asm, .c, .cpp, .cxx, .bas, .cbl, .for, .pas, .res and .rc).**
  All other extensions such as .cc and .yy need to be specified with the SUFFIXES directive (ex: .SUFFIXES: .cc .yy).

- **The Windows makefile has a different syntax for the include command.**
  In order to include files such as *.config files, they have to be specified with the "**!**include" command instead of the Unix's plain "include".

A2. C++ Compiler

A2.1 Compiling Source Files

Comparison of Window and Unix Compiler Arguments

Windows cl	*Purpose*	*Unix gcc*
/c	compiles without linking	**-c**
/D*name*	defines constants	**-D***name*
/EH sc	specifies the model of exception handling	***not used***
/Fo*pathname*	names the object file	**-o** *pathname*
/I*directory*	searches a directory for header files	**-I***directory*
to be defined	creates dynamic objects	**-fpic**
/TP	specifies C++ source files (/Tp specifies a single C++ file)	**g++**

Unix Example:

```
g++ -fpic –I$(SXF)/src –c example.cc –o $(SXF)/lib/linux/obj/example.obj
```

Windows Example:

```
cl /TP /EHsc /I$(SXF)\src /c example.cc /Fo$(SXF)\lib\msvc\obj\example.obj
```

A2.2 Building Libraries

Comparing Window and Unix
Linker Arguments for Building Libraries

Windows lib	*Purpose*	*Unix g++*
-out:*pathname*	renames the executable file	**-o** *pathname*
to be defined	create shared library	**-shared**

Unix Example:

```
g++ -shared –o  $(SXF)/lib/linux/libSXF.so  $(OBJS)
```

Windows Example:

```
lib –out:$(SXF)\lib\msvc\libSXF.lib  $(OBJS)
```

A3.3 Building Executables

Comparing Windows and Unix
Linker Arguments for Building Executables

Windows link	*Purpose*	*Unix g++*
-out:*pathname*	renames the executable file	**-o** *pathname*
/LIBPATH:*dir*	sets the library path	**-L***dir*

Unix Example:

```
g++ –o ./linux/sxf_tester.exe ./src/sxf_tester.obj –lEchoSXF –lSXF –L$(LDIR)
```

Windows Example:

```
link –out:\linux\sxf_tester.exe .\src\sxf_tester.obj libSXF.lib libEchoSXF.lib \
        /LIBPATH$(LDIR)
```