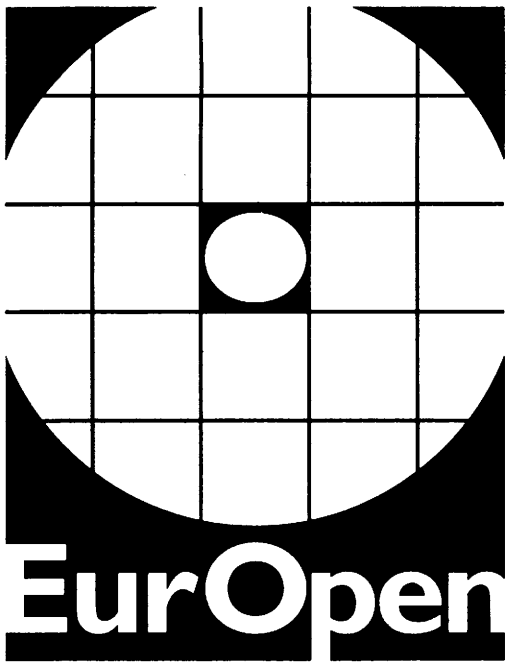
EurOpen & USENIX

Spring 1992 Workshop / Conference

Jersey, Channel Islands

6th – 9th April, Hotel de France



Proceedings of the Spring 1992 EurOpen/USENIX Workshop

April 6–9, 1992
Jersey, Channel Islands

ISBN 1 873611 03 X

Further copies of the proceedings may be obtained from:

EurOpen Secretariat
Owles Hall
Buntingford
Herts
SG9 9PL
United Kingdom

These proceedings were typeset in Times Roman and Courier on a PostScript printer driven by a sleepy dwarf. PostScript was generated using **refer**, **tt**, **pic**, **psfig**, **tbl**, **sed**, **eqn**, **troff**, **pm** and **psdit**.

ACKNOWLEDGEMENTS

The organisation of the first joint EurOpen USENIX workshop is now completed: the success of the event will depend on the participants! The workshop addresses a central theme to Open System users (designers, developers, managers, administrators and end-users): Portability. The programme covers a broad range of topics: from standards, patents and paradigms to practical solutions.

The programme committee has done its work well: Patrick Burbaud, Barry Shein and Ralph Treitz invested considerable time and effort not only in reading and rating submissions, but also in putting together the programme itself. The programme is clearly the result of their efforts – we are very grateful.

Jointly organising a workshop with USENIX has been to our advantage: input from both sides of the ocean is effective. Let's hope more events of this type will be organised in the future. We are sincerely greatful to

- The EurOpen Secretariat, whom once again have shown that a distributed organisation can work.
- Neil Todd, the EurOpen tutorial executive, has managed to align tutorial topics to the conference theme,
- The USENIX office and liason Ed Gould, for their full cooperation and support,
- Stuart McRobert and Jan-Simon Pendry have again done a wonderful job of the proceedings: their patience is amazing.

Frances Brazier
Programme chair

David Tilbrook
Programme co-chair

These proceedings were specially produced for EurOpen at the Department of Computing, Imperial College, London, using resources generously provided by the Computing Support Group. Danny Turner and Philip Male of Computer Newspaper Services Ltd kindly produced the bromides.

– sm, jsp.

Table of Contents

Author Index

UNIX Conferences in Europe 1977–1992

UKUUG/NLUUG meetings

1977 May	Glasgow University
1977 September	University of Salford
1978 January	Heriot Watt University, Edinburgh
1978 September	Essex University
1978 November	Dutch Meeting at Vrije University, Amsterdam
1979 March	University of Kent, Canterbury
1979 October	University of Newcastle
1980 March 24th	Vrije University, Amsterdam
1980 March 31st	Heriot Watt University, Edinburgh
1980 September	University College, London

EUUG/EurOpen Meetings

1981 April	CWI, Amsterdam, The Netherlands
1981 September	Nottingham University, UK
1982 April	CNAM, Paris, France
1982 September	University of Leeds, UK
1983 April	Wissenschaft Zentrum, Bonn, Germany
1983 September	Trinity College, Dublin, Eire
1984 April	University of Nijmegen, The Netherlands
1984 September	University of Cambridge, UK
1985 April	Palais des Congres, Paris, France
1985 September	Bella Center, Copenhagen, Denmark
1986 April	Centro Affari/Centro Congressi, Florence, Italy
1986 September	UMIST, Manchester, UK
1987 May	Helsinki/Stockholm, Finland/Sweden
1987 September	Trinity College, Dublin, Ireland
1988 April	Queen Elizabeth II Conference Centre, London, UK
1988 October	Hotel Estoril-Sol, Cascais, Portugal
1989 April	Palais des Congres, Brussels, Belgium
1989 September	Wirtschaftsuniversität, Vienna, Austria
1990 April	Sheraton Hotel, Munich, West Germany
1990 October	Nice Acropolis, Nice, France
1991 May	Kulturhuset, Tromsø, Norway
1991 September	Budapest, Hungary
1992 April	Jersey, Channel Islands

Software Pre-Porting

Marc Poinot

SEXTANT Avionique
Vélizy-Villacoublay
France

Abstract

This paper describes the three levels we have identified in a pre-porting process. The **Palas-X** SDE† has been developed, ported and installed taking into account these levels: the source code portability, the development / porting platform portability and then the installation / execution platform portability. Our idea is that there always are software services for which we have no or partial information, so we have to identify these parts, encapsulate them and keep in mind that there are potential problems in there.

1. Introduction

The **Palas-X** SDE [Ber91a] is developed by *SEXTANT Avionique* for software projects which have a lifetime of about twenty years. We want to reduce as much as possible the porting cost, so we prepare the software by taking into account the *Open Systems* standardization and our experience in porting. This preparation affects many steps in the development, the installation and the execution environment. We call this process **pre-porting** the software.

The pre-porting idea is to identify future problems in all parts involved of the software product. Most of the time, developers use to have one or two system oriented modules, which are dedicated to all system calls. But experiences show that this is not enough, porting problems may happen in another location that in the operating system calls.

Unfortunately, problems appear in all the development platform specific parts, that can be compiler, editor, debugger, etc. including the shell which often is the first interface to many tools. Moreover, you can lost many hours to modify your porting environment itself, and then once the software is ported it may not execute.

The software pre-porting has to impact the development, the porting, the installation and the execution platforms. Our approach is to be as close as possible to existing or oncoming standards, i.e. POSIX [ISO90a] and PCTE [Bou88a] services. Then the standard delta, i.e. differences comparing to the standard, are identified and gathered in specific modules, we expect that future systems will made this delta decrease to zero.

† Software Development Environment.

In section 2 we describe the constraints and the goals we have to reach. Section 3, 4 and 5 cover, respectively, the software code, the development platform and the installation and execution environment portabilities.

2. Constraints

2.1. Users' Requirements

Our SDE will progressively be used in all *SEXTANT Avionique* projects, including tools such as **Palas-X** itself, of course. Each staff has got its own sub-networks, with an heterogeneous set of machines, operating systems and environment. They have habits, local tools, most of them do not want to be aware of the UNIX platform, all they need is consistent and self-sufficient software. We have to port on these platforms but we must also try to bring the standards in such development teams. Therefore, we cannot force them to modify their working environment, we have to take into account all the existence constraints in the company. The present porting targets are UNIX workstations, but there may be needs for VAX / VMS or PC / XENIX platforms.

2.2. Development Requirements

Our SDE must be of a great maintenability as well as a high quality in the whole development cycle (which can reach itself up to twenty years into the maintenance status). This software will be executed on platforms which have not yet been conceived. We cannot avoid this assumption but we presume that such machines will run standardized models† within 5 years. This is our SDE's maintenability insurance. Rules have been set in order to reach a high quality level, these affect all steps of the development cycle. This was a great opportunity for us to put our pre-porting rules in the bag by modifying existing development rules with normalization and portability concepts, tips and tools.

Our team, 4 / 6 people for two years, knows the UNIX development. They are familiar with tools like C++ compilers [Lip90a, Str86a] *GNU Emacs* editor, *make* and all what can be done with shell scripts. Development has been done on Sun4 platforms with C++ compilers and debuggers, however these workstations are not our only future users' one. The whole software and the internal tools are managed inside *Adele 2* from the LGI [Est90a] at the IMAG (Grenoble, France), this provides us the software database system.

2.3. Normalization

The *Open Systems* models and standards take place in the *SEXTANT Avionique* information system policy. This involves either lower levels of software, such as operating system or network, and upper levels such as user interface. This task will spend years, and we think that first of all *Open Systems* must be understood and initiated by tool developers like us rather than being forced by market. We tried to follow all *OSF* recommendations, but RPC‡ used for our client / server based system. This is an important and interesting part because RPC

† A model is "something" like a set of software parts for a global information management system.

‡ Remote Procedure Call (inter-process communication through the network, based on TCP/IP)

were mandatory for one external service software. This is a case where one part of the software cannot be normalized, but anyway we are ready to use any other service instead of the previous (such as DCE† for example).

3. Software Development

3.1. Code Quality

The software has been coded with an object oriented language and this leads to natural quality [Mas89a] if some rules are set. We have two levels of rules: the first one is the source code, comments location and formats, syntaxes and naming rules have been defined, this allows developers to read another code without being lost. We always try to be as close as possible to the *ISO C*[ISO90b] libraries and syntax subset of C++. We cannot refer here to the development rules such as these which can be found in the *GNU* distribution, *X Window System* or *Kerberos* source code, but we pick many ideas from these softwares. The second level is development methods, we are developing with the *Adele 2* program data base which allows versioning and configuration managing.

One of the most important object oriented programming keyword is known to be encapsulation [Mey90a] this leads to privacy and modularity. We define classes with wide semantic ranges, each of these were involving specific parts of the software, including the external ones. The external services are the software parts which cannot be modified by the development staff, whatever kind of part these are. They can be generated or bought source code, libraries or execution time services. The main service is the operating system, we also have the *RPC* [Sun90a] *Motif* [OSF91a] the *curses* lib, the *shell* etc. A service offers entry points into another software, or into another source code. One entry point is centralized in one of our modules and we prohibit any access to this entry elsewhere than from this module. Furthermore, if an entry point has two semantics then we duplicate this entry.

3.2. External Services Encapsulation

We explained to our developers that the `getwd(2)` POSIX system call is not the same that the `getcwd(2)` they usually use (which doesn't allocate the returned string). They replied that it is really easy to modify now with some `sed` filter. But

- What is the modification cost now?
- What was the adaptation cost before?
- What about this new divergent version of the code?
- What about any new machine like some *NHX80302* coming straight from the Moon with a new almost-POSIX system?

The developer must be aware of the portability problem. There is no real difference between a call to `getwd` and one to `PX_getwd` which is our macro for the call to `getwd` because most of the time this is rearranged by the pre-processor. But when you have got 200 modules with somewhere in there many calls to many "not-invented-here" func-

† Distributed Computing Environment (included in the *OSF* model)

tions we cannot insure a good portability level or a correct maintenability of such a code. In fact, what we are doing is:

1. Redefine all external calls
2. Constraint language syntax
3. Set rules for semantics

Of course, these rules are often common sense but it is a great effort to make it done by a staff of developers who have at least 5 years experience of C projects on UNIX.

For example, when we are encapsulating the well known `strcmp` C library function. It may be called to test equality or to find out differences. There are two entries in our external "strings" services module, `ST_strcmp` which returns `true` or `false`, and `ST_streq` which returns differences as an integer value. Developers are forced to use the `ST_strcmp` and `ST_streq` functions instead of the usual calls. Then developers do not try to use an external specific behavior, i.e. a side effect or any manufacturer's special extension. The service interface (or header) is exactly the one which is expected by the developer. The way to obtain this service from an external mechanism is in the implementation itself.

3.3. Generated Code

Part of our software code is generated. This makes no doubt that code generation will go increasing and we prepared to that. We use UNIX code generator such as *lex* and *yacc* (lexical analyzer generator and compiler generator [Sch85a]). We also have a user interface generator which is really not portable, it is completely dedicated to the *X Window System* [Sch88a] through the *Motif* library. The generated code may be encapsulated and seen as an external service, but there are some cases, such as the *Motif* user interface generators, where our code has to be inserted into generated code. Two policies may be followed:

1. Encapsulate the generator definition language
2. Encapsulate the generated code

In some cases we can store two kinds of information: the definition language and one version of the generated code. The compilation / porting platform has got the two versions, but recompilation of the previously generated code will be avoided when possible. The *Motif* user interface generator itself is not portable, it cannot be found on all target platforms.

We have got two problems. The first one is to port the generated code. The second one is to insert our code, which is supposed to be portable, inside some parts of the generated code.

The first point is to insure portability before the code generation. Such an encapsulation is not an easy task for tools which have got a private definition language. Moreover, some tools are managing their own files, sometimes in their own database and it is often difficult to modify them. Then portability is focused on generated code encapsulation.

Tools which have a public definition language are encapsulated. Their use and the input files format are modified to be portable when this is possible (like with *yacc* which as well defined rules).

For example, the

```
yyerror(yacc)
```

function is used by the generated code of the *yacc* tool. The standard function prototype is

```
int yyerror(char*)
```

and this function has been re-declared and encapsulated in our source code. We wrote the body of the function and we put it into a well defined part of the "external code". On the *Domain / OS* system the generated function prototype is

```
void yyerror(char*)
```

and perhaps it is not an up-to-date version but it is the version we **actually** got. This leads to a source modification in a deep way as we have to modify the service interface (a `void` function instead of an `int` return) and modify the code itself (no `return` C keyword for the `void` function).

The generated code is as portable as C code can be from a system to another. In other words that implies all basic portability problems. The problem occurs when the generated code must be modified. The first reflex of a portable developer (sic) is then to write a modifier which is not generator dependent, or at least a parameterizable modifier.

For example, we have some `checkuser(3z)` function which is generated by a system tool. This performs a lot of verifications on an `uid` get from the `getuid(2)` system call. Just assume that we need to make our checks on the effective user id, we need now to use the `geteuid(2)` system call instead of the previous one. This is a generated code modification. There is certainly no answer to such a problem but we think that if this modification occurs it must be done by another tool, as portable as possible that is mastered by the development and the maintenance staff. Our modification processes are done with shell scripts.

We have noticed that new *X Window System* based user interface generators, which are coming up now, are more aware of such problems. Generated widgets or files are now coming more and more modular and portable. This is a very important point for us, certainly more important than any new blinking gadgets.

3.4. Parallel Version vs Pre-Processing

Some services modules clearly cannot be the same on every systems, these modules have to be modified depending on the system needs. There are two ways for modifying a service implementation code: the pre-processing switches and the parallel version.

The parallel version idea is to develop as many different versions of body code as needed. Each system may have its code for one service module. That is classical parallel version management but in many cases the differences are so close that maintenance may be a hard job.

The pre-processing way to modify a source code is well known in the UNIX development. This is a best way to maintain services modules, but a worse one if you had to trace versions, or if you have to increase large pieces of different source. Our approach was to mix both methods, and then to gather POSIX-like systems in one sole source with pre-processing selection.

For example, all UNIX resources are gathered in the `PX` module. The difference between the interface and the implementation is emphasized with the `exec` call of our `PX_Proc` C++ class. The developer needs

1	`int PX_Proc::exec(char* f)`		

2	`if (PX_fork() == GOOD)`
	`{`
	`PX_exec(f);`
	`}`
	`PX_wait();`
		`PX_call(f);`	

	A	B	C
3	`#define PX_POSIX`	`#define PX_FUNIX`	`#define PX_PCTE`

4	`#ifdef PX_POSIX`
	`#define PX_fork() (fork()==0?GOOD:BAD)`
	`#endif`
	`#ifdef PX_FUNIX`
	`#define PX_fork() (fork()==PX_CHILD?GOOD:BAD)`
	`#endif`
	`#ifdef PX_PCTE`
	`#define PX_call(f) call(f,PX_ARG,PX_ENV,PX_DYN,PX_STAT)`
	`#endif`		

Table 1: *Parallel versions and pre-processing*

an `PX_Proc::exec`† method which is doing something like "make a sub-process, execute this file and wait for its end". The `PX_Proc::exec` service has a little algorithm with some classical calls such as fork / exec / wait as in the Table 1 (the first column of the line 2). This class method is a super-service provided to the developer. Therefore the `fork`, `exec` and `wait` (including `wait` structures) were encapsulated by `pxdefine.h` header file (Table 1 line 4). On the POSIX platform the correct flag is set (A) `pxdefine.h` macros uses a simple pre-processing rewriting. On the almost-POSIX platform (B) the use of pre-processing may be enough. On the PCTE platform (C) the whole service has to be rewritten, and the `PX_Proc` module itself gets a divergent version, as in column 2 line 2. Given one `PX` body, a POSIX-like platform needs a *cpp* tuning and a non-POSIX platform needs a new version. We use both mechanisms but POSIX calls are first coded. Then other systems delta are pre-processed or rewritten in parallel versions.

3.5. Some Tips

We avoid `#if` switches inside the source code body. The pre-processing divergent versions are managed into the headers. This is one rule that there always is one interface (i.e. header) for divergent bodies.

The well known header key [Dar88a] has been extended, we are asking for a whole service inside the header itself. Only a part of the header is then pre-processed, Table 2 describes such a double header key.

For example, the `PX_STDIO` service must be explicitly mentioned, if intended to be used by a developer. Services are not system declaration rewriting. They can be, but most of the time they are general (such

† The C++ language prefixes its methods (functions) with the class they belong to. In that case, the scope of the `exec` method is the `PX_Proc` class.

as `PX_MALLOC` for all memory stuff, or `PX_RPC` for all *RPC* related headers and rewritings).

All what can be used to make the software parameterizable must be done. The `sysconf(2)` POSIX system call is used but it is not enough. This is a run-time parameterization and this only involves the system (which is obvious because that is POSIX.1). We need a static way to identify the platform, a compile-time either as a run-time set of tools and functions which allow us to tune our developments. Then all tricks will be trashed out and we will be able to cooperate between (Open) systems.

3.6. Maintenance Approach

Once the external services are encapsulated, we are expecting from the developers to know what they are coding. This is not trivial. Most of the time they include a lot of headers, with an uncomplete knowledge of the real services of the latters. We have to identify each needed system header, and gather them in global services (see *Some Tips* above). Once the latter is encapsulated, the service is available for the developers.

Development rules must not be lost during the maintenance, and external services modules may be at least modified, at last rewritten and sometimes by people who were not in the initial development staff. The software must be set to allow such a modification. It is an evidence that maintenance work must not alter the portability capabilities of the software. Any new service extension must be done into that service module. An interface modification, i.e. a service definition, must be avoided.

3.7. Conformance to POSIX.1

Error cases are identified using POSIX codes. However, *PCTE* error codes [CEC89a] cover a larger range and these have been taken into account for further porting.

We had problems with C++ headers which were colliding. The *OSF* and target system headers have non-empty intersection and functions syntaxes such as `pause(3)` and `signal(3)` were defined twice with non-compatible definitions. We choose to temporarily avoid such includes and redefine these prototypes ourselves.

4. Compilation / Test Platform

4.1. Hacker vs Developer Porting

Some tools are known for the ability they have to be installed on any platform, we have worldwide examples such as *GNU Emacs* or the *X Window System* which are freeware. But one cannot say these are portable, these are stars and every hacker has put its stone in such software. When you are in a ready-to-compile distribution, you have got the source code. All is done inside this distribution in order to make you compile it with two commands such as `vi config.h` and `make` or some derivations from this. These kind of softwares have **already** been ported. You are not really porting this software, you are switching on a platform which is already defined. Most of the time, there is no way to port such a software if you are not a hacker yourself. Who is

```
#ifndef PX_DEFINE
#define PX_DEFINE
 ...
#ifdef  PX_STDIO
 ...
#endif
 ...
#endif
```

Table 2: *Specific service inclusion*

really able to rewrite an `alloca.c` (clever memory allocation) in assembly language for a new machine? We are not.

Our approach was to split services into encapsulated slices which belong to "easy-to-port" and "not-easy-to-port" classes. Easy to port services can be ported by our developers. The not easy to port services will be ported by specialized companies. The service is clearly defined and the company only has to furnish one implementation of this encapsulated service.

4.2. Configuration Extraction

The configuration selects one implementation for each needed service interface. Needed interfaces are well defined, it depends on the porting target. We are there selecting services for one target. Bodies may depend on the host platform, some may not exist, other may be different depending on the local environment. We are there selecting one, possible, implementation.

For example, the user interface is a generic service. When the host has *X Window System* and *Motif* libraries we are able to choose the *X11* user interface. In another case the configuration takes the alpha-numerical interface. That is two different bodies for one service which is the (generic) user interface. Once the selection is done, we have got the source code distribution.

4.3. Porting Platform

There are many approaches for the porting platform. The first one is to have one platform for one machine, may be starting from one generic version which is modified by a developer. Another one is to have all platforms gathered in a sole one and a set of switches to tune it. You can also generate it, with tools such as *imake* or *makedepend.* These ways can be seen as another level of the parallel versus pre-processing way we already explained previously.

In fact, the problem is to lay on a portable platform and to tune it for the target. We think that the tuning can be done when the delta's cost to the standards are less than one day. Else the platform may need to be deeply modified. Our porting platform is a POSIX-like one (that is UNIX like) and we only have to manage little deltas such as *Makefile* syntax, compilers options, and identification of headers and libraries. There are many portable systems which have got very clever *Makefile* files. But most of the time a clever *Makefile* means a not portable one, or a very hard to maintain one if the latter is, actually, portable. The *imake* and *makedepend* tools are attempts to build platforms from a portable one. Unfortunately such tools are rare and porting them

begins to make the porting platform more and more heavy and then hard to maintain.

Thus, we have a set of generic *Makefiles* and a set of associated scripts. Scripts are making values substitutions, this makes the *Makefiles* more clear and this increase portability.

4.4. Development Tools

Most of the time the compilation / test platform is built with a set of manufacturers tools such as editor / compiler and debugger. These tools often are forgotten but they are critical in the porting process. There is no way to compile a software without a compiler, and there is no way to find a bug without an editor (and it is easier with a debugger). The compilers problems we may have are problems such as:

- Compiler specific (mandatory) option
- Compiler language version (syntactic or semantic constraints)
- Compiler (or related tools) generated files
- Compiler (or related tools) tables size

Our approach was to reduce the use of the tools to the most little usable set. Then we have to disallow all options we are not really known on other compilers, linkers.

For example, we cannot call our files `foo.sxt_c++` and compile them with that name, we may have size problems with `switch` structures. The C or C++ compiler may be more or less tolerant, some compilers force an error when they reach up more than 200 warnings which where ignored by another compiler. The development rules are then applied to the development file structure itself: file names, sizes and directory structure. Problems happen with hidden file names, such as a pre-processor which will generate intermediate file name `foo.i` when `foo.c` is the name of the the body file. This is not very funny when a set of related data files is named `foo.i` and the pre-processor overwrites it without any warning! Our naming convention uses the `.c` and `.h` extension but sometimes more than these two "no-problem" cases are needed.

The editor problem is not a real one because there always some editor on a "used" platform. The debugger is efficiently replaced by the editor / compiler set and the useful `printf(3)` function.

Our porting platform is now complete.

4.5. Conformance to Standards

We haven't found a way to answer to the question: "Is that system POSIX-like?". There is no way to find out if a part of a system has POSIX requirements, sometimes we meet a file called e.g. `posix.readme`, `libxpg2.a` or `osfcn.h` which are not really intended to be used (they are not clearly indicated). Header files are not documented enough. Nowadays porting is a hacker job not a developer's one. Headers and their contents, that is services, should be documented and clearly located in every POSIX-like system. Most of the documentation is arranged to make the developer use manufacturer dependent software services or tools. We try to avoid it as long as we are aware of it and responsible for such choices.

The shell and its utilities are not yet standardized, at this time, but the *IEEE P1003.2 D11* [IEE90a] seems to be clear enough to be taken into account. The C subset of C++ compiler is very close to an *ISO* C one,

and we have three C++ compilers which often react differently. Our way to use such a compiler is very classical. The *lex* and *yacc* code generators are quite well defined in most of the systems. Either for the grammar definition than for the way to use them. The *make* utility has been used with as less gadgets as possible.

5. Installation and Execution Environment

5.1. Identification

The ready-to-install distributions are software products sold on every available platforms. The approach is then to furnish a lot of binary and dedicated script files and then a specific license, or only a specific binary and a script on a host. If we lay on the side of our paper the binary code compatibility, we can note that their scripts, their execution environment, are dedicated to a specific platform. Does it mean that we cannot have a product for a POSIX-like platform whatever it is?

The installation / execution part is a delicate part where (in all cases) we have to modify our system in order to make it run on an unknown environment. The idea is to find out what kind of environment we have got, at this time the only way to do that is with a script which looks for specific services and asks questions to the user in charge of the installation. Such a script can be found in the *Kerberos* source code. This is, one more time, a hacker trick. Who is going to maintain such a script? When there is a delta between our platform and the host, who is responsible for filling in the hole?

Our answer is (1) the developers have to furnish standardized interfaces and have to ask for standardized services to the host, (2) the hosts have to furnish standardized services and (3) the installator staffs must be able to find out services and tune them. Of course, form a marketing point of view, we must provide an installation script with its manual which reduces the "Guess where it is?" part of the installation. We have a POSIX-like target installation platform.

5.2. Script Portability

A shell script contains (1) control structures, (2) builtin commands and (3) external commands. The first two points can be reduced to the *Bourne* shell [Bou82a] subset of syntax and builtin commands. The third part is encapsulated into another shell script which insure portability, using shell variable to switch from one to another version.

For example, the way to call the *sort* utility has been modified in POSIX.2 version. Portable shell scripts are calling our `sh.sort` script which encapsulate the various ways to call *sort* depending on the target system. Switching to the correct version is done using a shell variable which is set during the system identification.

There is a problem of "hardware" portability. That is, there are four user interfaces versions: the alpha-numerical one, the *X Window System* one, the batch one and an API library. We insure that whatever is the hardware, or the way the user wants to access to the software, we are able to port from one of these generic interfaces.

5.3. Maintenance Problems

Maintenance efficiency is focused on version upgrading and this needs a compatibility between versions. Execution problems are shell, system or tools ones. We must have portable shell scripts but many parts are platform dependent such as tools location, machine identification. The system problems often are access rights issues, badly mounted file systems or anarchic *RPC* setups. Tool problems we have come up when users want special features such as security, multi window systems or all-automatic environments.

We have decided to write a simple "problem analysis" manual, but a automatic analysis script has been rejected. Our portability work stops here, when the whole information system management is to be adapted. The maintenance has now to give the same software product from delivery to delivery. The system operator has to modify our execution environment, this is easily done because all binary files are called through documented and parameterizable scripts. The installation platform must take into account all the environment tuning. This is not only a marketing action, this is a portability one.

5.4. Conformance to Standards

All in all, the practical effect of the POSIX.2 standard to our software is to drive the option choice and the integration into the "UNIX way to do things...". We have taken into account the "utility argument syntax" for option syntax. The tool has been designed in order to work with pipes.

The *X Window System* user interface has been built on the *Motif* toolkit with respect to the *Motif* style guide. We furnish a set of *X* resources which can be loaded or not by the user.

6. Porting Experiences

6.1. Methodology

Most porting methodologies are home made, manufacturers are not concerned with such problems and software editors probably keep their methods secret. Only UNIX users group such as the French *AFUU* edit portability guides [AFU91a] which certainly are useful for software pre-porting. The porting methodology we use is:

1. Make our source code portable
2. Make our development platform portable
3. Make our installation / execution platform portable
4. From (1), (2) and (3) identify all external services
5. From (4) and using (2) build the porting platform
6. Identify the target platform
7. Extract distribution using (6) results
8. Tune platform (3) and (5) using (6)
9. Compile / test / install

The effort is into (4) and (6). The idea is to never trust the documentation except the *ISO/IEC* one which (in neither case) doesn't refer to any specific system. Even if this is not the most detailed, the most

Machine	OS	(1)	(2)	(3)	(3bis)	(4)	Total
HP400	Domain/OS 10.3	2.00	1.00	1.00	0.30	3.15	7.45
Sun3	SunOS4.0.3	0.30	0.15	0.00	0.00	0.00	0.45
Sun4	SunOS4.1.1	0.30	0.15	0.00	0.00	0.00	0.45
HP700	HP/UX	0.30	0.30	0.00	0.30	0.00	1.30

Table 3: *Porting time repartition*

explained with examples (but it is in fact...) this always is the point we start from.

6.2. Effective Porting

The total porting time has been collected from several porting experiences. So far we have ported on three systems, included our development platform. This last point is one of the most important, it emphasizes all habits which could have been problems for further porting platforms. The time has been added for each part we are focusing on, that is the main steps of the porting process. This process is:

1. The "discovery" of the platform, tools, headers, libs and all specific parts of the target platform
2. The tuning of the *Makefile* depending on the previous point
3. The compile time problems: language syntax and development rules and (3bis) the external services call
4. The installation and execution time problems

Note that the porting experience goes from the distribution tape extraction to the run of the execution script (including installation). Table 3 shows relatives times spend for each porting, time is in hours and minutes. The *hp700* target has **not** been ported yet, it is an evaluation. Porting target are sorted from the first ported to the last (future) one.

This table shows two important points. The first one is that the porting effort is located into the platform itself, i.e. into the tools and service identification and their calls in the *Makefile*. The second point is that further porting only need tuning into the system and code services. This emphasizes the fact that once the services are identified, we only need to find them out in the new porting platform. So, if the service is standardized then the effective cost of this part is (close to) zero.

Conclusion

All in all, even if we don't port immediately the whole software and its environment on a lot of platforms, this was a nice example of services identification and this have improved the **Palas-X** robustness. The effective cost for porting **Palas-C**† from Apollo / DomainOS to Sun3 / SunOS4.xx was two man months. The same task has been done with several source lines modified in the system module and the *Makefiles*, this has been done in one day.

There is no doubt that portability is often synonym of readable and understandable, but this is also antonym of efficiency and perhaps C hacking? From our point of view, portability is strongly bound to (1)

† This is the last previous version of **Palas** (born 1982)

maintainability and (2) reusability. In fact, portability is another way to have a great code quality and this leads to maintenance and reuse. Maintenance is 75% of the software lifetime [Ram84a] and everyone tries to reduce its cost. During the maintenance phase, the life cycle which goes from development (bug correction, enhancements...) to the end user may be very short and very used. A software which has got a high portability level, on all the steps we have described in this paper, has reduced the cost of this cycle when an event such as a system upgrade or a machine exchange or addition arrives. Reusability is also an immediate application of such a modularity, encapsulation and service identification. In fact we have redefined the nowadays system lacks: unambiguous interfaces of well-defined services.

Our idea is that a key to a portable software is a software where every service which isn't yours is identified and encapsulated. We need standards, this fixes the services interfaces and then our portability.

Acknowledgements

I would like to thank Pierre Bernas for his encouragements, Laurent Daniel and Clément Martin from *Université de Montréal* for their early and bright advice.

References

[AFU91a] Groupe portabilité AFUU, *UNIX et systèmes ouverts: de la portabilité au portage,* AFNOR Technique (1991).

[Ber91a] P. Bernas, "Intégration d'outils dans un AGL," *4th International Conference on Software Engineering and its Applications*, Toulouse, pp. 513-526 (December 1991).

[Bou88a] G. Boudier, F. Gallo, R. Minot, and I. Thomas, "An overview of PCTE and PCTE+," *Proceedings of the ACM SIGSOFT* **13**(5), pp. 248-257 (November 1988).

[Bou82a] S. Bourne, *The UNIX system,* Addison Wesley (1982).

[CEC89a] CEC, *PCTE: A basis for portable common tools environment. Functional Specification V1.5,* Commission of European Communities (June 1989).

[Dar88a] P. A. Darnell and P. E. Margolis, *C a software engineering approach,* Springer Verlag (1988).

[Est90a] J. Estublier and N. Belkhatir, "Adele2: un outil pour la gestion des logiciels," *Génie logiciel et systèmes experts* **21** (December 1990).

[IEE90a] IEEE, *p1003.2 D11 POSIX part 2, Shell and utilities,* IEEE (March 1990).

[ISO90a] ISO/IEC, *ISO/IEC 9945-1 POSIX part 1, System Application Programming Interface (C language),* ISO/IEC (December 1990).

[ISO90b] ISO/IEC, *ISO/IEC 9899 Programming Languages – C,* ISO/IEC (December 1990).

[Lip90a] S. B. Lippman, *C++ primer,* Addison Wesley (1990).

[Mas89a] G. Masini, A. Napoli, D. Colnet, D. Leonard, and K. Tombre, *Les languages à objets,* Intéreditions (1989).

[Mey90a] B. Meyer, *Conception et programmation par objets,* Interéditions (1990).

[OSF91a] OSF, *OSF/Motif programmer's reference,* Prentice Hall (1991).

[Ram84a] C. V. Ramamoorthy et al, *Software Engineering, problem and perspective,* COMPUTER (IEEE) (October 1984).

[Sch88a] Scheifler, Robert, Gettys, and Newman, *X Window System,* DEC Press (1988).

[Sch85a] A. T. Schreiner and H. G. Friedman, *Introduction to compiler construction with UNIX,* Prentice Hall (1985).

[Str86a] B. Stroustrup, *The C++ programming language,* Addison Wesley (1986).

[Sun90a] Sun, *Network Programming Guide (Part Number 800-3850-10),* Sun Microsystems (1990).

Porting Under UNIX – Problem Areas and a Proposed Strategy

Kenneth J. Chan David Jackson

University of Liverpool
Liverpool, England
{ kjc | dave }@compsci.liverpool.ac.uk

Abstract

Much has been written on software portability and on guidelines for writing highly portable programs. In a large number of UNIX environments – educational establishments being good examples – the members of software support teams know only too well that much of the software that they have to port either does not follow such guidelines, and, even when good programming practices have been adhered to, packages often (by necessity) contain a substantial amount of system-specific code. In such circumstances, the accumulated wisdom derived from the experience of porting many items of software over many years is drawn upon to provide the insight required to deal with a particular package. To the casual observer, the porting process can appear largely unstructured and *ad-hoc*, and a generalised methodology may seem almost impossible to define.

An initial aim of this paper, then, is to impose some sort of structure on the list of problem areas that porting specialists encounter in their activities. This is achieved by examining the typical difficulties that arise due to the particular "flavour" of UNIX that is being used, and also the typical problems that are found during each of the stages of compilation, linking and execution when installing a package. This broad classification is then used as a springboard for developing a strategy for porting software on UNIX systems. Throughout this paper, we present various examples of packages that have caused problems when ported to our particular departmental computer system, but other institutions will undoubtedly have suffered analogous tribulations.

1. Introduction

On occasion, the installation of a new piece of software will go without a hitch; that is, it will compile and link first time, and execute exactly as the specification says it should. Software porting and maintenance staff are kept in their jobs by the fact that things rarely run as smoothly as this, particularly when an application is developed on a different

machine and under a different operating system to that of the target environment. Porting from (say) VMS to UNIX can pose considerable problems if the software makes heavy use of system calls that have no direct UNIX equivalents. Even when performing UNIX-to-UNIX ports, however, there may be differences between the two operating systems that constitute significant obstacles to porting.

In certain instances, these variations in the source and target environments may entail a lot more than simple modifications of the source code (assuming that it is available). If certain resources required by the software are lacking, the effort involved in supplying them must be weighed carefully. These resources may take the form of completely separate packages which may themselves have to be ported. For example, the installation of the GNU C++ compiler (`g++`) necessitates the prior installation of the GNU C compiler (`gcc`); similarly, the *Cornell Program Synthesizer* makes use of BSD's `as`-style directives (such as those produced by the GNU `gas` assembler). Other resources may be on a smaller scale, but equally as essential: inter-process communication in the application may be implemented using shared memory, which the target operating system may not support; the application may require a certain granularity of timer resolution (set with `setitimer`) which again may be unsupported; and so on. The emulation (if this is at all possible) of such resources can involve substantial programming, and may be judged to be not worth the effort.

Where modification of source code becomes necessary, this task is greatly aided if the original programmer has followed established guidelines for writing portable code [Lap87a, Can90a, Dol90a]. However, it is not usually possible for the developer to test out the code on anything more than a small number of machines, and it is therefore unsurprising that any non-trivial systems software package tends to exhibit some machine-dependent characteristics. In specific circumstances, documentation relating to the porting of a package, or from one machine to another [Hew91a], can be of great help; more generally, the knowledge and expertise of the porting specialist is indispensable in these situations.

To an observer, the speed and accuracy with which an experienced member of a porting team can locate and define a malfunction caused by system-dependency, and then to make the appropriate amendments to get it to execute on the target machine, can be impressive to say the least. When asked, such specialists find it difficult to describe the precise thought processes that are involved when tackling the porting of a software package, and so a rationalisation of the vast folklore that forms the knowledge base of the UNIX porting gurus appears to be an insurmountable problem.

In the Computer Science Department of Liverpool University, the computing facilities take the form of a network of Hewlett-Packard workstations, connected via Ethernet and running HP's version of UNIX System V (HP-UX). This is administered by a team of technical support staff, for whom much of the daily activity is concerned with porting software. Importantly, a closer scrutiny of their activities reveals that certain categories of porting-related errors occur frequently in certain contexts, and a prime aim of this paper is to identify these and hence impose some sort of structure on at least some of the porting specialist's expertise. More specifically, it would seem that certain problem areas are very much related to differences between the source and target UNIX systems (especially between SysV and BSD), while others are often associated with a particular phase of the compilation-

link-execute cycle of installation (although it may still be the case that these errors are caused by variations in operating systems), and we categorise these accordingly. This is illustrated with examples of problems that have been encountered by our own technical staff, but which are almost certainly not peculiar to our own particular environment.

Investigation also suggests that, despite the unstructured and *ad-hoc* appearance of the porting process, support staff with extensive experience in these matters actually take a reasonably systematic approach, precisely because they are aware of the aforementioned categories of errors that can occur at each stage of the process. Gaining this knowledge is usually less well-structured, however, and so in a subsequent section of the paper, we attempt to establish a recommended strategy for porting software under UNIX systems.

2. Porting Problems Due to UNIX Differences

Liverpool University's Computer Science (LUCS) department is a prime example of the sort of institution that is affected greatly by variations in UNIX systems. To understand this, it will be helpful to be familiar with some of its background.

In 1988, LUCS was one of the first UK educational sites to purchase HP UNIX systems. This procurement was typical of other educational procurements of the time: emphasis was placed on acquiring as much equipment as possible at the expense (unfortunately) of software purchases. The department's position regarding software was that public-domain software would supplement the base-level software provided by the vendor that was successful in the tender. HP were chosen because they could deliver the required number of seats within budget. This presented a problem: there was, at the time, a distinct lack of software ported to the HP platform, since the vast majority of software had been developed on, and written for, SUN and VAX computers running BSD 4.2/4.3, reflecting the wider employment of these systems within the UNIX community.

An impressive list of standards compliance is claimed for HP-UX, including System V Interface Definition Issue II (SVID2), X/Open Portability Guide Issue III (XPG3), POSIX 1003.1, and so on. This implies that

- HP-UX conforms fully to AT&T System V (with BSD extensions), and to the end-user this does indeed appear to be the case. Examination of its anatomy suggests, however, that HP-UX is in fact built on a BSD kernel made System V compliant. This gives rise to certain peculiarities of HP-UX with regard to porting, discussed further in Section 3.2.
- HP-UX has passed some measure of portability which means that applications should be *easily* portable to it. Experience has shown that this is not always the case.

Since most public domain packages are written on BSD systems, porting difficulties that arise are usually associated with a transferral from BSD to SysV, rather than the reverse. In an attempt to circumvent these problems, some vendors offer systems that have a foot in both UNIX camps, in the form of either a "hybrid" system (e.g. HP-UX) or a BSD/SysV switchable system (such as Dynix). In the following subsections, we examine some of the typical problems encountered in porting across systems.

2.1. Include Files

Most packages will specify the names of files that contain header information that is necessary for the building of the application. These files are indicated with the `#include` pre-processor directive. A common problem is that the name of the file that is given in the directive may be different on another system, or may reside in a different directory from the one expected. For example, on BSD systems there exists a system include file called `strings.h`, which contains such things as the declarations of string handling and comparison functions. On System V, the equivalent file is named `string.h` (i.e. singular instead of plural). Similarly, include files that are conventionally found in the `/usr/include/sys` directory on BSD systems may reside in `/usr/include` on SysV.

There are essentially two approaches to dealing with these problems. The first is to modify the source code by changing include file names to the equivalent names on the target system. This is usually the preferred approach, but for some packages this may mean the modification of dozens of source code files. This brings us on to the second approach, which is to modify the system filestore to reflect the requirements of the application, possibly using soft-links to create pointers to equivalent files. It goes without saying that this is a course of action that should not be taken lightly: any changes to the directory structure do nothing to alter the inherent portability of the current package, and may adversely affect the porting of other packages. Additionally, it does not help other sites to install the package unless they are informed of the changes that have to be made to the filestore. HP-UX, with its BSD influence, offers a partial (albeit unofficial) solution along the lines of this latter approach: it provides a directory called `/etc/conf/h` which is intended to be used in the re-building of the kernel, but which also contains many of the files that are missing from `/usr/include`.

2.2. Terminal Handling

Terminal handling is extensively used in, for example, editors, to set up such things as single character input. The mechanisms for performing this on BSD systems (the `tty` interface) are very different from those on System V (the `termio` interface), and can cause major headaches during porting. This can be true even without varying machines, a case in point being SUN, who used to use `tty` and who now use `termio`. To illustrate the difference, consider the setting of even parity on a terminal. Using `tty`, this is achieved by masking an object called `sgttybsgflags` with a flag called `EVENP`; with `termio`, you have to mask a field of a structure `termio.c_iflag` with a constant called `INPCK`, and then mask `termio.c_cflag` with two other constants. There is clearly little similarity between the two mechanisms.

2.3. Command Output

The commands that are bundled with UNIX systems may generate different output on different machines. Usually, these differences are only slight, affecting (for example) the way the output is formatted. If the command is used in isolation, this has no significant impact on the end-user, but when the command is called by another application that expects the output to adhere to a strict convention, the effects can be catastrophic.

The *Cornell Program Synthesizer* generates syntax-directed editors, according to a definition of a language grammar supplied by its user. As part of the process of building an editor, *Cornell* creates an intermediate file containing assembler directives (pseudo-operations) in BSD format. The version of the `as` assembler supplied with HP-UX (and quite probably other systems) does not recognize the manner in which these directives are specified. As an example, the declaration of a global label in BSD-style assemblers is introduced using the directive `.globl`, whereas the HP assembler uses `global`.

This proved to be more than a minor irritant – it was sufficient to prevent further progress on the port without a major re-think. One possibility would have been to modify the source code to produce HP-style directives, but the sheer complexity of the package and the increased non-portability that would have been introduced made this undesirable. The solution adopted was to port an assembler using BSD-style directives – the GNU `gas` assembler. For similar reasons, the GNU version of `yacc` (`bison`) also required porting.

For *Cornell*, the easiest option was to port public domain packages, but the installation of these is not always straightforward. The curious way in which *g++* (the GNU version of *C++*) performs part of its initialisation forms a good example. During the building of *g++*, a number of object libraries are created which contain, amongst a whole host of other things, a number of functions with names of the form "`initxxx`", where `xxx` is the rest of the identifier. An initialisation routine is then generated by scanning these libraries for the names of functions beginning with the string "`init`", packaging all these names up to form a source code routine which calls the functions, and compiling the result. Now, the way in which the name scanning is done is by running `nm` on the libraries, which outputs the names of all the objects contained therein. Unfortunately, the way in which the output from `nm` is formatted can vary across machines, and this can cause the scanning of the routine names to fail.

It should be clear that the employment of such convoluted methods for building packages can give rise to its own problems; this is further illustrated by the *TeX* text-processing package. To save on execution time, some of the data structures that *TeX* constructs in its initialisation phase can be pre-fabricated by running the package until a certain point is reached and then typing the appropriate command to make it dump the contents of memory into a `core` file. A program is provided which then converts this core file to an executable `a.out` file which is essentially the *TeX* program with all its initial data structures already built. The problem here, of course, is that different UNIX systems rely on differing formats of both core files and `a.out` files, so that unless you have the conversion program that is tailored to your platform, you have to write your own (the solution adopted at LUCS, incidentally).

2.4. Semantics of System Calls

Like UNIX commands, the semantics of system calls can vary across systems. One of the best-known examples is the C library routine `sprintf`, which on BSD systems returns a pointer to the start of the array containing the output string, while under System V it returns an integer indicating the number of items written out. Other examples abound and, unless one is aware of them, they can be very hard to identify as the cause of a malfunctioning application.

The above examples illustrate that porting between different versions of UNIX is problematic enough. It is our experience at LUCS that even porting applications under what is said to be the *same* operating system is often frustratingly non-trivial. HP market two distinct hardware ranges – the Motorola 68000 (CISC) based series 300 and 400 workstations and servers, and the HP Precision Architecture (HP-PA RISC) based series 600, 700 and 800 workstations, servers and multi-user machines. It is claimed that the CISC and RISC versions of HP-UX became functionally identical as of release 7.0. That release has since been superseded, but differences still exist between the two versions that are severe enough to complicate what should be straightforward ports. New releases of HP-UX also create new problems, with applications that used to work now no longer functioning. It would be unfair of the authors, however, to assume that the systems in use at LUCS are unique in this respect.

3. Problems During the Porting Process

Most public domain packages are supplied with one or more `make` files that take on much (if not all) of the burden of building the application. Even with this automated assistance, difficulties may still arise as the software is taken through each of the phases of compilation, linking and execution. For each of these phases, we once again identify some typical problem areas.

3.1. Compile-Time Problems

Even where good coding practices are followed, the differences between compilers can greatly affect the ease of porting. More and more programs are now being written in Ansi C or even C++, and support programmers are having to become aware of how to understand and write code in each of these styles. Compilers often impose restrictions on programming constructs which they may or may not flag at compile-time. Classic examples of unimplemented features include `enums` and the passing of structures as parameters to functions; these are irritants that can be tiresome to re-code. When porting `xview`, our compiler objected to the use of an `enum` in the expressions tested in a `switch` statement, and these had to be cast to integers. Other restrictions such as those on the sizes of data segments are not always so easy to overcome.

The C compiler pre-processor `cpp` is a common source of complaint, since some packages use this in isolation and expect its output to adhere to a particular format. Using the *Cornell Synthesizer* as an example yet again, this system generates intermediate text files as part of the process of building editors, which it then runs through `cpp`. The output from `cpp` is then used as input to a separate program. As part of its output, `cpp` may generate `#line` directives; in our version of the pre-processor, a single space was output between the "#" character and the "`line`" string, which was not expected by the *Cornell* system and caused it to fail. The remedy was to use another `cpp`. The manner in which `cpp` processes directives also affected the porting of `xview`: the HP-supplied pre-processor did not support the `#elif` statement, and so all of these had to be converted to nested `#if` directives.

3.2. Link-Time Problems

It is quite common during porting, especially from BSD to System V, to be presented with missing routine/function error messages. For compatibility, HP-UX provides libraries of BSD functions (called `libBSD.a` and `libPW.a`), and it is often simplest just to link to these. In other cases, System V may provide equivalent functions with different names; the problem here is being aware of the equivalences. Even where function names are identical in the two systems, arguments to the functions may differ. Consider the input/output control function, called `ioctl` under both BSD and HP-UX. One of the arguments to this routine is a command, but the constant names used to represent these commands are completely different on each system. It is not obvious, for example, that HP's `FIOGSAIOOWN` command is equivalent to BSD's `FIOGETOWN`, or that `FIOSSAIOSTAT` is the same as `FIOASYNC`.

It has already been mentioned that HP-UX is, in fact, a hybrid of System V and BSD UNIX. Internally, the kernel makes extensive use of BSD routines which are not made accessible to the user. An include file called `/usr/include/sys/syscall.h` lists all the known Chapter 2 system calls, each name being associated with an index number, so that `exit` is assigned the index 1, `fork` has index 2, and so forth. Routine names that would normally be available under BSD, but which are not available under System V, are commented out of the file. The number given in the file for each system call is an index into a table maintained by the kernel and called `sysent`. Each entry in the table contains, amongst other information, the address of the function code in the kernel; the entries for unsupported system calls contain the address of an error routine. This means that if, during porting, it is found that a package makes a BSD system call that System V does not support, it is possible under HP-UX to use a debugger such as `adb` to find the address of the function and patch it into the address table, so restoring access to it. Again, this is not something that should be done lightly, particularly in a clustered environment in which there may be many instances of the kernel.

New releases of operating systems bring with them new run-time libraries, and it sometimes happens that programs that ran perfectly well with the old libraries do not behave well with the new ones. At LUCS, the Ada compiler translates Ada source to C source, which is then run through the standard C compiler. A recent upgrade to HP-UX introduced shared libraries, and it was discovered that any Ada program that employed parallel processing (tasking) would no longer work with these libraries. A means had to be found of forcing the C compiler to link with the archive libraries rather than the shared libraries – not as easy as it might sound, due to the fact that the C compilation and linking is directed from within the Ada translator.

3.3. Run-Time Problems

This is the usually the most difficult of the three phases for the programmer doing the porting, since compile-time and link-time problems often manifest themselves as error messages that afford some degree of assistance. Many run-time problems arise as a result of differences in the semantics of system calls, discussed earlier. Others are due to variations in file formats; e.g. a debugger will expect `a.out` files to follow a strict format, and any programs that access device files or the kernel memory (`/dev/kmem`) will expect certain conventions. Still another

area of system dependencies to be aware of concerns the reliance of some packages on particular word sizes, byte orderings, alignments, etc.

Interesting things can happen with variable argument functions (the `varargs` problem). For efficiency, some applications provide their own input and output routines such as `printf`; examples of programs which do this include device-independent `troff` (ditroff) and the `ded` text editor. In writing a routine like `printf` one doesn't know how many arguments to expect in advance. To work round this, the format string forming the first argument is used to determine the other arguments that are passed in the call, and these are picked off the run-time stack as they are needed. The problem with this is that a decision has to be made concerning the direction in which the stack grows: does it grow towards the high end of memory or the low end? For any machine that the program is ported to that follows the opposite convention to the one implemented, the application will fail.

4. An Approach to Porting Under UNIX

The types of frequently-occurring problems identified in the previous sections suggest that it is possible to establish and recommend a general set of guidelines that act as a strategy for equipping oneself with the necessary preparedness for porting a wide range of applications. These are stated as follows:

1. First of all, establish a good background. Make sure you know all the ins and outs of programming in C, including Ansi C and preferably C++. Be aware of what the majority of the C library routines do, what commands are available, what system calls there are, and so on. Learn the major differences between System V and BSD UNIX, and any peculiarities of your own host UNIX. Make sure your shell programming is up to scratch, and that you understand how things are done under UNIX (for example, how do you set up `sockets`?).

2. Gain familiarity with the available tools. Most applications come with `make` files, so understand how `make` works and how to amend the files. A number of applications make use of the compiler generation tools `Lex` and `Yacc`, so it is useful to gain experience with these. C program checkers like `lint` are good for finding non-portable constructs. Learn how to use both a source-level debugger and an assembly-level debugger such as `adb`; remember that you will not always have access to the source code.

3. For specific packages, be prepared to consult anyone and anything. Before you do anything else, read the README files – they can save you a lot of trouble during the port. Read any other documentation that is provided, in order to gain some knowledge of what the application is supposed to do; otherwise, it is difficult to know whether the port has been done properly. If necessary, consult other experts – they may have ported a similar package before. Don't be afraid to broadcast your queries on USENET – it's a quick and painless way of eliciting expert assistance.

4. Decide whether or not to wait – a problem may no longer be a problem in the next release of the operating system since new releases often introduce new functionality. HP-UX 6.5 did not

provide `SIGIO`, `syslogd` or `wait3`, but these were all present in HP-UX 7.0. Conversely, waiting may make things worse: an example of this is the Ada problem described in Section 3.2; other examples include changes to addresses within the kernel, which affected a debugger, and changes to NFS, which caused the `ded` editor to crash.

5. Make high-level decisions about the particular porting tactics to be used. In the case of header files, for instance, is it easiest and best to modify all the source code programs to refer to files available under the target UNIX, or should the source code be left well alone and the changes made to the file structure instead?
6. The hard bit! Do the port itself, remaining aware of the problem areas described in this paper.
7. Test the executables. Some packages, such as the `Icon` programming language and the `Gag` compiler-compiler, provide suites of test data and expected outputs – if these are available, they should be used.
8. Finally, prepare a list of all the modifications that have been made to install the application – it may well come in useful in the future.

5. Conclusions

In this paper, we have attempted to construct a framework for categorising the *types* of problems that are frequently encountered during the porting of applications. Numerous examples have been described to illustrate this structure, but these are in no way intended to constitute a comprehensive list – another establishment picked at random and installing the same packages will no doubt have experienced a completely different set of problems. Although most of the examples given in this paper are based on our own experience with HP-UX, the strategy for porting given in Section 4 is intended to form fairly general advice, and it is hoped that this is of use to others who, whether they wish it or not, will spend much of their time porting applications under UNIX.

References

[Can90a] L. W. Cannon et al, *Recommended C Style and Coding Standards,* USENET public domain document (June 1990).

[Dol90a] A. Dolenc, A. Lemmke, D. Keppel, and G. V. Reilly, *Notes on Writing Portable Programs in C,* USENET public domain document (November 1990).

[Hew91a] Hewlett-Packard, *HP-UX Portability Guide*, January 1991.

[Lap87a] J. E. Lapin, *Portable C and Unix System Programming,* Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632 (1987).

Certifying Binary Applications

Donald A. Lewine

Data General Corporation
Westboro, USA
lewine@cheshirecat.webo.dg.com

Abstract

The goal of creating a binary operating system interface standard is to define a common configuration which enables the executable binaries of application programs to be moved between computer systems. This paper gives an overview of the 88open standards which allow a POSIX or SVID conforming program to be compiled into a vendor independent binary. The bulk of the paper is dedicated to certifying binary compatibility. How does one know that an application meets the standards?

This paper describes two packages for certifying binary compatibility: ITS/88 and ACT/88. ITS/88 verifies that the Operating System supports all of the features of the standard. ITS/88 uses conventional Software Quality Assurance techniques. ACT/88 verifies that the application strictly conforms to the standards. ACT/88 uses a static, lint-like, analysis to look for errors in the code and dynamic run-time analysis to look for non-portable behavior. I believe that ACT/88 is unique technology and different from any known quality assurance tools.

1. Types of Portability

Before we dig into the details of certifying binary compatibility, it is instructive to look at the types of things people want to transport from system to system and what it takes to make those things portable.

1.1. Source Portability

People want to be able to move software at the source level. Programs are written to well known Application Program Interfaces and recompiled on each target. The POSIX standard [IEE90a], and the SVID [USL89a], provide for source level portability. In theory, any POSIX application can be compiled and executed on any POSIX platform.

While source portability is excellent for the free exchange of software, very few software vendors are willing to have their source code widely distributed.

1.2. Object portability

Object portability allows one to move compile a module on machine A and link it with a module compiled on machine B. Object portability lets software vendors distribute libraries to be linked with their customers code.

Object portability is one of the most desirable forms of portability and also one of the most difficult to achieve because all of the following need to be exactly specified:

- Machine instructions
- Data representation
- Register usage
- Subroutine calling conventions
- Interlanguage operability
- Object file format
- Symbol table format
- Link editor
- Libraries
- Networking
- Execution environment
- Archive format
- System commands
- Signal handling
- Header file values
- Installation

Obviously, object portability is only possible within a given computer architecture. Even if the architecture is constant, object compatibility requires noticeable effort. In order to enable object compatibility for the Motorola 88000, the 88open Consortium published the Object Compatibility Standard [88o90a]. The 88open OCS specifies all of the information listed above.

1.3. Binary Portability

Binary portability lets one move compiled and fully-linked programs from machine to machine. This is the form of portability that we see in the personal computer area where shrink wrapped executables are provided to the end-user.

Binary portability is somewhat easier to achieve than object compatibility because there are so many fewer interfaces to specify. The 88open Consortium published a standard to enable binary compatibility [88o90b]. The goal of this standard was to completely specify what a shrink wrapped application needs to do. We did not want to have a specification by example, as in, "IBM PC or 100% compatible."

2. Requirements for Portability

Of course, merely publishing the standards does not produce object or binary compatibility. One needs conforming systems and applications. Every system that claims to support conforming applications must provide all of the required interfaces with exactly the required semantics.

Testing for conformance in fairly straightforward. Every conforming application must use only the defined interfaces and it must use them correctly.

2.1. Extensions

To make matters more complicated, conforming systems may provide extension above and beyond the standard. Conforming applications may take advantage of these extensions, as long as they work correctly when the extensions are not present.

For example, a system may provide an accelerator for X-window requests. A conforming application may use the accelerator on those systems, however, it must work correctly on systems without the accelerator.

This capability means that finding an undefined system call in an application does not mean that the application is not conforming.

3. What Information Must an Application Binary Interface Specify?

Let us take a moment and look at the information that is required to enable binary portability.

We need to specify all of the legal data types and how those types are stored in memory. For example, consider the structure

```
struct demo
{
char    a;
long    b;
char    c;
double  d;
} x;
```

Every compiler must align the structure exactly the same way. The compatibility standards specify the method used to assign offsets to structure members.

The format of object files must be completely specified.

The application's view of memory must be consistent on all platforms. Different platforms may have different layouts for memory. For example, the stack may be in a different location. Applications must tolerate these differences.

The initial conditions and actions required by start-up code must be specified exactly.

All of the signal handling interface must be the same from platform to platform.

The effect of including header files must be the same from system to system. On the other hand, the text of header files may be different. Vendor extensions may be present under the control of a feature test macro.

System call interfaces must be identical.

One area that people often overlook is installation. A portable binary is distributed on some media. All of the steps required to read that media into the target filesystem must be specified.

If the installation procedure has a directory tree layout that it can rely upon, that layout must be documented. The installation procedure cannot use any directories that are not part of the standard.

All of the other details of the execution environment must be well specified.

4. Portability "Gremlins"

There are a few "features" that many applications seem to trip over. One problem is inadvertently using a vendor extension. Our portability verification tools must be able to detect this case.

Another problem is using behavior which is defined one way on one system and another way on a different system. For example, some systems support NFS and others do not. Applications must work correctly in either case.

There is also unspecified behavior. This is similar to Implementation-defined, except an application may not rely on a given implementation doing a well defined thing. For example, the initial value of the stack pointer can change from software release to software release.

All systems provide commands (and command options) that are above and beyond what is stated in the standard.

Even in cases where everything seems to be well specified, one programmer will read the spec one way and another will read it a different way. Portability tests must find this type of problem. There is also an on-going standards committee to resolve questions of interpretation.

5. Certifying the Platform

Before we can hope to support portable applications we must verify that the target system meets all of the required standards. The software quality assurance technology to do this verification is well known.

The 88open has developed a suite of assertion based tests to verify that all of the platform components meet the standards.

All of the system calls are tested to make sure that they accept valid input and generate the correct error code for each possible error condition.

The system libraries are tested to make sure that all of the required library functions are included and that they work correctly.

The basic network system calls are tested for systems that support the networking option.

The X11 library is tested to make sure that all of the required functions exist and function correctly. Since X11 is very large and poorly documented, these tests are not as exhaustive as some the tests for system calls, libraries and commands.

The required commands are tested to make sure that they function correctly.

There are tests to verify that the directory tree is in the form expected by applications.

Tests are done to verify that all of the development tools work as expected. These tools include:

- Compilers
- Linker
- Assembler
- Archiver
- cpio
- tar
- Object file format

6. Constraints Placed on Testing Process by Software Producers

The goal of the 88open compatibility certification process is a large number of applications which are portable across many platforms. We do not want to make the certification process complex or to require a great deal of work on the part of a software vendor. We must accept software in the format that a vendor wants to supply it. We must live with a number of constraints imposed on the testing process by the software producers.

6.1. Format of Packages

Software many be in any format. The software must be distributed on a QIC-150 format tape in either cpio or tar format. Any installation procedure acceptable to the marketplace is acceptable for testing.

6.2. Executing Commands

Applications my execute commands either as a shell script, via the `system()`, or `popen()` function.

The verification tools check all of the commands in installation shell scripts or arguments to `system()` and `popen()` to make sure that only valid (portable) commands are used.

6.3. No Access to Source

Software vendors like to keep their source code private so the 88open verification tools use only the distributed binaries to do their work.

7. Application Conformance Testing

The interesting part of the verification process is checking to make sure that applications follow all of the rules. There are tools to check source code for standards compliance, however, I know of no other binary verifiers.

The 88open Application Conformance Test (ACT/88) contains two major tools: the Static Binary Verifier (`sbv`) and the Dynamic Binary Verifier (`dbv`).

7.1. Static Binary Verifier

The Static Binary Verifier works on either object files or linked executables. It is designed to perform a cursory test on all of the code in the product.

7.1.1. Flow Analysis

The Static Binary Verifier steps through the application program instruction by instruction. Each time it encounters a branch instruction it follows both the branch taken path and the branch not taken path.

As `sbv` looks at each instruction it marks it as processed. When `sbv` follows a transfer of control that leads to a processed instruction, the analysis for that path can stop. In this way, `sbv` will examine every possible instruction in the program.

Of course `sbv` can be fooled by self-modifying code. Given modern coding practices, self-modifying code is very rare.

7.1.2. Correct Instruction Use

The Static Binary Verifier makes sure that all instructions are used properly. Any illegal or reserved instructions are flagged.

7.1.3. Correct Calling Sequence

The Static Binary Verifier check every call for the proper use of the 88open calling standards. For libraries or modules which are going to be linked against standard libraries, the proper calling sequence must be used. For completely linked programs, the results of this test can be ignored.

7.1.4. "May" Information

Because of the nature of static analysis, it is not possible to tell if a given branch will be taken. The Static Binary Verifier detect problems which may (or may not) show up when the program is executed. In these cases, `sbv` produces an advisory message that the programmer can choose to ignore.

7.2. Dynamic Binary Verifier

The Dynamic Binary Verifier operates on complete programs. It tests the code under actual operating conditions. It is able to be very accurate in reporting errors, however, it requires some external test harness to make sure that the program under test is completely exercised.

The Dynamic Binary Verifier works on unstripped executables. The verifier installs a small "verification stub" at the entry point of each system call. The program then executes transparently.

The Dynamic Verifier verifies all system calls and optionally all standard library calls.

`dbv` generates coverage metrics for the program under test. Three percentages are reported:

- The percent of all basic blocks executed;
- The percent of all of the system calls which get executed;
- The percent of all of the user procedures which get called.

```
Start DBV validation of appl at 4/7/92 14:30
End DBV validation of appl at 4/7/92 16:30

STANDARD FEATURES USED: BCS, OCS

                                    CALLS  ERRORS
     Standard Library Procedures:  295790       0
     System calls:                   3900       0
     Syslocal calls:                    0       0

     Basic block coverage factor:      85.0%
     System procedure coverage factor: 92.2%
     User procedure coverage factor:   95.0%

     Unexecuted user procedures:
       err       getpathpart
```

Figure 1: *Typical dbv output*

Each system or library call is checked for valid and portable arguments. Any non-portable call is reported to the user.

The **dbv** output gives the exact cause of the error and the symbolic location of the function call. In most cases, it is very easy to correct to violation.

Figure 1 shows some typical **dbv** output. The coverage metrics allow the person running the tests to improve the test harness and get the tests coverage to approach 100 percent.

Figure 2 shows some typical **dbv** error messages. The first problem was caused by

```
fd = open("test_file", O_CREAT);
```

The **mode** argument is missing. In the real world, missing arguments are one of the most frequent problems we encounter during application testing.

7.2.1. Command Testing

The **cmdtest** program is used to verify shell scripts and the arguments to **system()** and **popen()**.

The **cmdtest** program is run in a clean environment where there are no added commands or command extensions. It checks each command for proper syntax and then allows it to be executed. The combination of the syntax check and the pure environment insure that the command is portable.

```
open(path = 0x401FF0, oflag = 0x100, mode = 0xEFFFFBE0)
     invalid argument to open (Undefined mode bit)
call to open at main+0x34

socket(af = 0x1, type = 0x0, protocol = 0x0)
     socket type (0x0) is not in the list of valid values
call to socket at main+0x50
```

Figure 2: *Typical error messages*

8. Summary

Applications compatibility testing takes some time above and beyond ordinary Software Quality Assurance. The dynamic verification tests require a good test harness to completely exercise the product under test. In most cases, these test suites are already produced as a part of the normal SQA process. The amount of additional time is small and the improvement in product quality is well worth the effort.

References

[88o90a] 88open, *88open Object Compatibility Standard Release 1.1,* 88open Consortium Ltd. (1990).

[88o90b] 88open, *88open Binary Compatibility Standard Release 1.1,* 88open Consortium Ltd. (1990).

[IEE90a] IEEE, *Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface(API)[C Language],* International Organization for Standards, Geneve (1990).

[USL89a] USL, *System V Interface Definition,* Unix System Laboratories (1989).

Applications POSIX.1

Conformance Testing

Derek Jones

Knowledge Software Ltd,
Farnborough, UK
derek@knosof.co.uk

Abstract

The Standards for POSIX and C were designed to enable the portability of applications across platforms. A lot of work has gone into checking compilers and environments for conformance to these Standards, but almost nothing has been done to check applications conformance. The incorrect assumption being made that the development compiler will warn about any construct that needs looking at. This paper discusses a tool that checks applications software for conformance to these Standards at compile, link and runtime as well as the library interface. Any application that can pass through this checker without producing any warnings is a conforming POSIX program and a strictly conforming C program.

1. Introduction

POSIX was designed as a standard environment to enable the portability of applications software and to some extent people. This portability of applications software is achieved through the specification of a set of services that every POSIX conforming application can expect to exist on a conforming platform.

For a Standard to be of practical benefit there has to be a method of measuring adherence to its requirements. Work on test suites to check environments for conformance to POSIX are well advanced. There are at least three such suites commercially available. However, the checking of an applications' conformance to POSIX has not received nearly as much attention.

Since the basic goal of POSIX was to enable applications portability through a Portable Operating System Interface it is about time that this imbalance was redressed. This article is about a tool set that was designed to check applications software for conformance to POSIX.1 (using the C bindings). What is described here could equally well apply to the X/Open Portability Guide (XPG) or AT&T's SVID.

The tools used in the POSIX conformance checker were all derived from the Model Implementation C Checker. This Model Implementation was designed to check C programs for strict conformance to the C

standard. Model Implementations have been produced for Pascal and Ada. In March 1989 the British Standards Institution signed an agreement with Knowledge Software to produce one for C. The Model Implementation was formally validated by BSI in August 1990 (it was the joint World first validated C compiler) and is currently being used by the validation suite producers to check their own software.

Before describing the tools themselves this paper first discusses the need to check applications software and what the relevant standards have to say about what constitutes a conforming (and therefore potentially portable) program.

2. Why Check?

POSIX offers the software developer the opportunity for a significant reduction in cost and effort when porting applications to different platforms. Given the benefits what stands in the way of creating POSIX conforming applications? There are two main reasons why applications fail to conform to the requirements of POSIX. The immediate problem is one of know how and old habits. Once these are overcome problems are caused by human oversight and error.

Because of the broad range of services offered it can take some time for developers to think POSIX. Old, UNIX, programmer habits and know-how are easily transfered to a POSIX development environment. Programmers cannot be expected to be familiar with all the intricacies of POSIX and how it differs from what they are familiar with. Speaking UNIX with a POSIX accent will not solve portability problems, particularly to proprietary platforms that support POSIX. It is necessary to speak POSIX as a native language and if using UNIX perhaps with a UNIX accent. Training can go someway towards ensuring a smoother transition to a POSIX only environment.

Experience over 40 years of software development has shown that it is impossible to produce any significant applications that do not contain bugs. The same principle holds true for writing POSIX conforming applications. Mistakes will be made.

So some means of independently and accurately checking conformance could uncover the majority of these problems and save a considerable amount of time and money later. Studies have shown that the later a problem is discovered the more expensive it is to fix. Thus the obvious time to find these non conforming constructs is prior to the release of the software.

From the marketing perspective Open Systems are being demanded by users. Use of an independent verification tool to check conformance will add weight to any claims of conformance to Open Systems Standards by vendors. From the users perspective demanding such verification is a useful means of ensuring vendor compliance with any Open Systems agreements that they may have.

For those developers considering a move to POSIX information provided by a checking tool can be used to provide an estimate of porting costs for existing applications. By providing hard information on likely problems time/cost estimates for porting an application are likely to be much more accurate than uniformed estimates.

2.1. Don't Compilers Check?

The development compiler is only likely to check for constraint and syntax errors, since it is these constructs that a conforming implementation is required to detect (and must detect in order for a compiler to validate).

One of the principles behind the drafting of the C standard was that existing code should not be broken by wording in the standard. This meant that in many cases the behaviour was left undefined or implementation defined. By not specifying what had to be done, compiler implementors were free to make their own decisions. Thus preserving the correctness of existing, old code. So in general compilers are silent on those constructs whose behaviour may vary across implementations. This freedom means that C programs can behave differently with different ISO validated C compilers, even on the same machine. There are no requirements on compilers to flag occurrences of these non constraint/syntax errors.

The C Standard committee also recognised that compiler vendors would have to rely on existing tools to link separately compiled units together. Since existing linkers were unlikely to check for cross module inconsistencies in external variables and functions it was felt that the C Standard should not mandate such checks.

Runtime checking is not considered to be in the spirit of C programming. Thus compilers do not generate code to check that pointers are within bounds, that the correct number of parameters are passed or check that any of the runtime conditions are violated.

3. What to Check

Having shown the benefits of conforming to POSIX and that the best way of achieving this is to use some form of checking tool we now have to investigate what constructs ought to be flagged and why. There are two main sources of information on constructs that ought to be checked to achieve applications portability:

- The text of Standards documents. Here we are interested in applications written in the C language. So the relevant standards are the C language standard (ISO 9899) and the C language bindings provided by the POSIX.1 (ISO 9945-1) standard.
- Practical experience. The sources for this information tend to be first hand experiences and conversations with developers on problems that they have encountered. Books on software portability are starting to appear. But on the whole these tend to give general guidelines rather than specific cases. On problem with specific cases is that they go out of date. As compilers and O/S's evolve problems disappear and new ones appear.

The core of the POSIX checker is driven by the requirements given in the C and POSIX.1 standards. Messages are categorised in exactly the same manner as the standards documents. Also any construct that falls into any category (except conforming code) is flagged. Provided with these core checking abilities the user can then provide configuration information (done via source and target profiles, discussed later) to switch off any messages that are not of interest.

Thus no justification, other than appearing in a standards document, is given for flagging these core constructs. Those developers familiar

with the standards process will know that the contents of standards are sometimes driven by immediate political needs rather than technical merit. Attempting to weed out the political from the technical issues was not considered to be worthwhile. Matters are greatly simplified (from our point of view) by simply handling all constructs.

The necessity for checks based on practical experience occurs because we live in an imperfect world. Operating systems and compilers do not fully conform to standards and contain bugs. In some cases these bugs are actually features, they are there for compatibility with previous versions of the software. The justification for flagging these constructs goes along the lines "this construct is not supported/behaves differently on the xyz platform". From this observation we draw the conclusion that truly portable applications have to be written using a subset of the facilities and services described in standards documents.

3.1. Standards Conformance

The POSIX and C standards define two types of conformance, (1) implementation conformance and (2) application (or source code) conformance. In this paper we are interested in the latter.

Application conformance is broken down into various categories. The classification of these categories varies slightly between the two standards.

3.2. POSIX Specifics

POSIX itself is not specific to the C language. However, it does have a C binding (ISO 9945-1). This binding specifies an interface to the environment, but surprisingly there are no requirements in POSIX.1 for the C source code to conform to the C Standard. However, from the portability perspective any software that conforms to the C Standard should be portable across C compilers running in a POSIX environment. So here we will be considering the POSIX and C Standards as one.

A strictly conforming POSIX.1 application does not rely on any construct whose behaviour is not fully defined, thus it has the greatest portability. A conforming POSIX.1 application may only use facilities described in the standard. However, since the behaviour of some of those facilities may vary across implementations such an application may need to be modified to run on different platforms.

The POSIX.1 standard also defines <National Body> conforming applications and conforming applications using extensions. It is expected that applications conforming to these standards will have weaker portability criteria and are not considered further here.

At this moment in time there are constructs for which it is uncertain (at least to the author) what category of behaviour they cause.

3.3. C Specifics

The C standard defines terms for a strictly conforming and conforming applications. The C standard categories the behaviour of constructs as follows:

- Constraint/Syntax errors
- Undefined behaviour
- Implementation defined behaviour

- Unspecified behaviour
- Exceeding minimum limits

To be strictly conforming a program should not contain an instance of any of these constructs. To be conforming a program should not contain any constraint or syntax errors.

The C standard is all encompassing in that all constructs can be categorised. Over the last few years there has been a considerable debate concerning the status of various C constructs. This has resulted a feeling that any remaining poorly defined constructs are likely to be obscure. There is an active program of documenting C answers to interpretation questions raised by users of the C standard.

3.4. C/POSIX.1 Differences

The major difference between the POSIX.1 and C standards occurs at runtime. POSIX specifies a much larger set of support functions. Basically it provides an interface to the host operating system, whereas the C standard provides library functions independent of the host OS.

POSIX.1 specifies a set of services that must be provided at runtime. The rules and regulations governing the creation of a runable program are specified to be those given in the relevant language binding. In the case of the C language binding some of the minimum limits given in the C standard are increased, i.e. number of characters considered significant in an external identifier.

4. When to Check

There are various stages in the application software creation process where checks against Standards can be performed.

During development. As mentioned earlier the sooner problems are found the cheaper it is to fix them. So a checking tool is of use to developers.

During quality assurance. Developers will only usually run tests that relate to their own area of interest. The Q/A department will be looking at the application from an integrated point of view. A checking tool is thus of use in ensuring that all of the software conforms to the relevant company standards.

Prior to source code purchase. When OEM's are investigating the possibility of buying in software it can be difficult to assess vendors claims of portability and standards conformance. Some form of conformance measuring tool would thus be of use in verifying claims of conformance.

5. What is Checked (by this tool set)

Ideally it ought to be possible to check conformance by looking at the source code. However, there are theoretical as well as practical limitations to this approach. The solution adopted in the checking tools described here is to mirror the compile/link/execute method of creating an application, but with a different emphasis. We are primarily interested in checking as part of quality assurance testing, not running the application in a commercial environment.

The POSIX checker was designed to detect and flag all undefined, implementation defined and unspecified constructs as well as all constraint errors, syntax errors and exceeding minimum limits; both at compile, link and runtime.

The phases of conformance checking:

- Check that the source code conforms to the ISO C Standard. This tool is also capable of generating symbolic information for cross module checking and intermediate code that can be interpreted.
- The next phase is the build process (linking the separately compiled translation units together). Again the ISO C Standard specifies behaviour that should be followed, while POSIX goes one stage further and gives extra functionality that must be supported. This tool can also merge the intermediate code files, generated by the previous phase, into a form suitable for execution by the next phase.
- Finally the software is executed. This is where the POSIX interface checking is performed. The POSIX applications conformance tool checks that all calls to POSIX functions are within the bounds specified by the 1003.1 Standard. It also ensures that no library functions outside of POSIX and the C Standard are called and performs pointer checking.

The constructs checked varies between the phases of the checking process.

The first two phases (static analysis) basically check that the source code conforms to the ISO C Standard. There are also a few extra features that POSIX mandates at these stages, such as restrictions on what can be assigned to `errno` and there are additional header files.

The compile time checks are likely to flag significantly more constructs than the development compiler. Since compile time checking is the easiest to do (from the users point of view) and are the easiest to relate to every attempt has been made to flag constructs in this phase, if possible.

The linker in the POSIX checker was specifically written for handling C. When separately compiled modules are joined together (the link stage) checks are made to ensure that the types of external objects and functions agree. This checking is something that very few linkers perform.

The third phase (dynamic analysis) does runtime checking. This consists of, amongst other things:

- The parameters of calls to POSIX libraries are within the specified bounds.
- Pointers don't reference storage outside of objects.
- Accesses to pointed to objects are refering to initialised values (use of uninitialised local objects is flagged at compile time).
- Casts are within range.
- No minimum runtime limits are exceeded.

This checking is performed by actually executing the applications program.

6. Background of the Tools

All of the tools for manipulating C source code produced by Knowledge Software are based on the same source code. This source started life six years ago and has been through several rewrites since then. Existing products include a C compiler front end, C to other language translators, a C quality assurance tool kit and most recently the POSIX applications checker.

6.1. General Design Aims

It was recognised at an early stage that most existing C programs are a long way from being strictly conforming. The user interface to the POSIX checker was designed to smooth the transition from common usage C to conforming Standard C. Not only is it possible to tailor the severity of every error message but implementation defined features are user selectable.

This tailoring enables users to convert their code in an incremental fashion. Thus the work load can be spread over a period of time. It is also possible to achieve results quickly, rather than having to wait until all of the work is complete.

Although its function is to check applications this was not seen as an excuse to execute slowly. Developers do not like to use tools that are slow and cumbersome. Therefore every attempt was made to ensure that the POSIX checker ran at a reasonable rate.

6.2. Support for Multiple Architectures

As a provider of services POSIX does not concern itself with the underlying computer architecture. On the other hand the C Standard recognises that at their lowest level computers do vary in their implementation. Because it has to execute user programs the POSIX checker has to be able to handle different computer architectures. For this reason the overall design of the tools was not tied to any computer architecture. They can be configured to emulate various architectures. The user can configure the compiler and runtime system to match:

- The development compiler
- A variety of hosts
- The intersection of various host processors

All that is required is information on the target platform to be fed into the platform profiles used by the checker. These platform profiles are subdivided into cpu, compiler and OS profiles. Profiles also exist for individual standards. Information on the most common processors is supplied with the package.

6.3. The Source Code Checker

This is a "traditional" compiler front end. It differs from most front ends in that many of its settings are soft. They are read from configuration files at compile time. A significant amount of effort has gone into showing the correctness of this tool. This correctness has involved showing that all of the requirements of the ISO C Standard are implemented and also that the code generated is correct.

Profiling work done on this tool has been used by BSI and NIST to measure coverage of their respective validation suites to the C Standard.

6.4. The Interface Checker

This checker is essentially a linker that was tailor written for handling C programs. Most linkers perform very little interface checking across translation units. They are usually restricted to complaining about missing symbols. The POSIX checker linker performs full type checking across C translation units, i.e. it checks that the same identifier is declared with compatible type in every file in which it is used. It also merges its input into a form suitable for interpretation by the runtime checker.

It was recognised that developers often require the services of libraries not provided as part of POSIX, i.e. X windows. The POSIX checker was thus designed to be user extensible. It is possible to refer to non POSIX library functions, have the interface checked and call them at runtime. There is also a method of specifying what runtime interface checking needs to be performed. It is the interface checkers job to build a runtime system capable of executing the users program, including the required interfaces.

6.5. The Library

The POSIX checker library provides the functionality required by the POSIX.1 and C standard. The majority of the POSIX functionality is achieved through linking in the POSIX library on the host computer. For the C library functions there is the option of using the host libraries or internally written functions. Following the design aims of the other tools it also gives warnings on the use of any features that may not be supported in other libraries and checks that the parameters to functions are within bounds. This checking is independent of whether the library is implemented internally or through an interface to the host.

6.6. The Runtime Checker

This interpretes the intermediate code generated by the source code checker. This tool acts as a runtime interface between the users application and the POSIX environment. As well as checking the runtime requirements given in the C Standard it also checks the calls to POSIX services.

This interpreter has benefited from the considerable experience gained in tuning and porting interpreters for other products. The requirement that the software should execute quickly has influenced the design of the C abstract machine that exists at runtime.

6.7. Checking the Checker

A significant amount of work went into the checking and verification of the Model Implementation C Checker, on which the POSIX checker is based. This included producing tests that caused 99.6% of all basic blocks in the code to be executed, cross referencing the source code to the C standard and passing both the BSI and NIST C validation suites.

The latest version of the software is over 100,000 lines of C. It has been ported to Sparc, MC68000 and 80386 (DOS and UNIX) platforms. Source code licensees have also ported to i860, MIPS, RS/6000 and VAX.

7. Practical Experiences

Developers tend to have a narrow view of standards. It is coloured by what they know and the tools they use. Unless faced will "real life" situations developers are often loath to modify code or their work practices. Flagging constructs based on requirements in standards documents and giving references to those requirements would appear to satisfy developers needs for justification.

A portability tool that simply flagged constructs because they occured in standards documents would only be doing part of the job. It is necessary to flag perfectly conforming constructs simply because some implementations will process them incorrectly. Although these cases do apply to real world situations, some developers believe that these problems will be fixed eventually and need not affect them.

It is rare to find a developer willing to follow the strict letter of any standard. So the ability to switch off some messages is essential.

7.1. So What Does the Checker Achieve?

Any program that can go through all stages of the checking process without any errors or warnings being flagged is a strictly conforming POSIX.1 application and a strictly conforming C program. Thus it should be portable with regard to these Standards. Any porting problem is likely to be as a result of problems in the host environment rather than the application.

7.2. Static Checking

The C Standard was written to cater for a wide spectrum of platforms, from Coffee machines to Super computers and from 30 year old machines to the latest RISC technology. Experience has shown that constructs considered to be an area of concern to one group of users are of little interest to those working on other platforms. A tool that flags all constructs that lie outside of strictly conforming Standard C is seen as being very verbose. So verbose, in fact, that at times users ignore its output completely.

In order to reduce the number of "uninteresting" message generated, the concept of source and target platform was introduced. By telling the tools which platform should act as a reference environment and knowing the target platform it is possible to filter out those features that are common to both platforms (the idea being that if a program containing such a construct worked on the source platform then it will work on the target). This platform profile contains information on cpu characteristics, the OS and C compiler behaviour. Profiles for the "unknown", C abstract machine and POSIX.1 platform are available for those users who want the create maximally portable applications.

7.3. Dynamic Checking

The two main types of warnings generated at runtime relate to pointer problems and the POSIX library interface. Problems with pointers are usually seen as program bugs rather than as a portability problem. In order to fit in with this view of the world and speed up other checks it is possible to switch off the pointer checking.

To date little experience has been gained with dynamic checking. Apart from what every developer already suspects, that pointers don't always point where they are supposed, little hard information is available. One issue that has been highlighted however, is that although pointers may have well defined values some of the objects that they point at might themselves be uninitialised.

8. Conclusion

Applications conforming to the C and POSIX.1 standards offer a reduction in porting costs. The only reliable method of verifying that applications software conforms to the POSIX specification is to use some form of verification tool at all stages of the development and testing of the program. The benefits of such verification include confidence that the software is conforming and will port to other environments and marketing advantages in being able to backup claims of Open Systems conformance.

X is the Worst Window System – Except For All the Others

Berry Kercheval

Protocol Engines, Inc.
Mountain View, California, USA
berry@pei.com

Abstract

The current state of the art in window systems is reviewed. Some survey of the field is mixed with the author's personal experiences.

1. Introduction

Today's workstations are getting more and more powerful. The old-fashioned glass-teletype interface to computer systems has become inadequate to harness that power to the service of the user, particularly when multitasking is involved. You don't want to have to wait for a compilation to finish before reading your mail, yet if you put the compilation in background you cannot easily check on its progress.

"Window systems" have evolved over the last few years to fill this need [Meh88a].

What exactly *is* a window system anyway? I use the term to mean a software package for interfacing multiple processes to a user on a bit-mapped graphics display. Regions of the display can be used as "virtual displays" for individual processes.

Thus, window systems serve as a multiplexor to allow users to easily switch from task to task. Most of them allow graphics as well as text, so that some impressive (and not-so-impressive) user interfaces can be built. In fact, one could argue that the entire field of "scientific visualization" was only made possible by the combination of increasing system performance and the multiplexing and user interface capabilities of window systems.

Quite a few window systems have evolved over the last few years. Most either remain laboratory curiosities or fall by the wayside. In the battle for acceptance, X seems to have won, like it or not. Sadly, quite a few very interesting systems have been passed over in a kind of graphical Gresham's Law.† This paper will investigate some of the factors that influenced this "victory".

† "Bad money drives out good."

2. A Bit of History

Window systems can be traced back to the Xerox Alto and Star [Lip82a, Joh89a] which were greatly influenced by Sutherland's seminal Sketchpad system [Sut63a].

Smalltalk [Gol83a] also came out of Xerox, and is an interesting example of a unified window system and programming language. Smalltalk was also one of the earliest object oriented languages.

Macintosh, a direct descendent of the Star has done a great deal to make people "window aware", and has demonstrated the importance of a uniform user interface.

MIT's Project Athena has given birth to the X Window System [Sch86a], and Carnegie-Mellon University's Andrew Project [Ros86a] produced an interesting tiled window system.†

3. A Quick Survey of Window systems for UNIX Workstations

This section concentrates on window systems that are currently available on UNIX or UNIX-like workstations.

3.1. SunView

SunView was the original window system for Suns. Each process that wished to "do windows" accessed the frame buffer directly, through the pixrect interface and `/dev/win*`. There were three layers to the SunView interface

1. The `pixrect` layer supplied basic raster and bit-blit operations.
2. The `sunwindow` layer implemented windows, clipping of overlapping windows and user input multiplexing.
3. The `suntools` layer provided user interface objects such as tool frames, scrollbars and menus, the standard event loop and selection management.

Every process that needed code to do menus, scrollbars, buttons and the like had to link a copy of it in, leading to bloated binaries in the days before shared libraries.

In fact, for a while all the standard SunView tools (`shelltool`, `cmdtool`, `mailtool`) were in fact a *single* binary with multiple links. The `main` routine checked the name it was invoked under (`argv[0]`) and called the appropriate sub-main. The code that actually implemented the individual tools was so much smaller than the user interface code that this technique‡ produced a net gain in disk space (and with shared text images, a gain in swap space as well).

It gained wide and early popularity because it came with the machine and was essentially "free". Quite a lot of commercial software was written for it, and given enough memory it worked quite well. However, it ran strictly on a single machine and was completely unaware of the network.

† But the Andrew tools have since migrated to X.

‡ Dare we say "Hack"?

3.2. The X Window System

X came out of MIT's Project Athena and had a distinctly different philosophy. A "server" process was the sole owner of the framebuffer and input devices, and fielded requests from "client" processes. The client/server communication protocol was deliberately made simple, as it was felt that the window server should not enforce user interface policy on clients (and users). The developers consciously chose not to implement features that were either of dubious benefit or about whose implementation there was controversy. These aspects of X led to some good things and some bad things: On the good side, the servers and clients are largely machine independent and portable; clients and servers need not reside on the same machine. On the bad side, multiple copies of code for menus, buttons and so on must be linked into client programs; some styles of interaction (such as "rubber-band" drawing) are network intensive; and programming is difficult.

3.3. NeWS

NeWS is a PostScript-based window system originally called SunDew, and developed by Jim Gosling and David Rosenthal at Sun, and also follows the client/server model. The idea that the window server was a PostScript interpreter led to the immediate idea that you could download PostScript code to it to handle interaction [Gos88a].

With the addition of lightweight processes to PostScript (which allowed a much simpler programming model than the X event loop), NeWS became an elegant, streamlined system that reduced client size dramatically.

Both Sun and Silicon Graphics shipped systems with NeWS, but only SGI chose it as the standard window system. (Interestingly, as X became popular and more customers demanded it, SGI grafted X onto the side of NeWS. The latest release of SGI system software, however, has Display Postscript grafted onto the side of X.)

3.4. MGR

MGR (short for window ManaGeR) is a "lean, mean" window system written by Steve Uhler at Bellcore. It's fast, portable and free. It's also almost unknown.

Client processes communicate with the MGR server via pseudo-terminals over any reliable byte stream. MGR provides graphics primitives, bitmap and font manipulation, window manipulation, menus and message passing. Clients can register interest in events such as mouse motion or button presses, and then receive messages from the server as an ASCII string *specified by the client.*

MGR was originally written for Sun Workstations, but has been ported to the 3B1, Atari ST, IBM-PC (under Xenix or Minix), Macintosh and the DECstation 3100.

3.5. MEX

MEX, short for "Multiple EXposure", was a window system developed at Silicon Graphics for their IRIS workstations [Rho85a]. Another client/server system, it used shared memory for rapid communication between clients and the server. This meant that network transparency was not practical.

An interesting aspect of MEX derives from the IRIS graphics architecture. Unlike raster-graphics systems, the IRIS hardware allows operations in object-space coordinates and renders them with proprietary pipelined "Geometry Engines". The pipeline makes synchronization between graphics requests from asynchronous processes difficult.

3.6. 8.5

8.5 [Pik91a] is Rob Pike's window system for Plan 9. It draws on Pike's experience with a concurrent window system [Pik89a] based on the idea that it need not be complicated. The system is very compact and supplies most of what users need, apart from color. 8½ relies on Plan 9 filesystem semantics, especially the use of `mount` to multiplex the keyboard, mouse and screen. Unfortunately, while this multiplexing technique gives 8½ much of its elegance, it also renders it unportable to UNIX without serious kernel modifications.

4. What Made X Succeed?

Why did X "succeed" and the others "fail"? Several factors affect the choice of a window system.

- Cost
- Ease of Installation
- Support
- Portability
- Marketing

4.1. Cost

X was "free", but so was SunView and MGR. NeWS cost money, but a binary license could be had for a token sum. The cost for any of them was not more than a few hundred dollars for tapes, or zero if you could ftp the sources from an archive site (or get a friend to do so).

In some environments, however, getting even a trivial amount of money for a new software product is unbelievably complicated, involving letters of justification, multiple signatures and purchase orders, and generally a lot more hassle than it's worth. One could just FTP X and install it without any approvals, but getting the NeWS distribution could be near impossible.

So cost seems not to have been a major factor in X's success.

4.2. Installation

Sunview was automatically installed with SunOS, and was thus the default choice of many users.

X is fairly painless for a big package – `imake` helps a lot and a lot of effort went into portability. For most supported configurations, installation is a simple matter of going to the top of the source tree and typing `make`.

Other window systems vary in the amount of work it takes to install them, but most are pretty simple if your configuration is supported.

So, since installing most window systems is not a lot harder than any other large software package, ease of installation was not a factor in X's success.

4.3. Support

X is unsupported – by MIT. But some versions, DecWindows for instance, have vendor support, and there is the `xperts` mailing list for direct access to the designers. There is also a growing community of consultants and contract programmers skilled in X

By now the X Toolkits are stable and allow application programmers to build X applications quickly and easily, at least compared to coding with just Xlib, which makes the "Hello World" program remarkably difficult [Ros88a].

NeWS and Sunview Were supported by Sun, and MGR was basically "Here's `mgr.tar.Z`, love Steve". MEX was supported by SGI, but dropped in favor of NeWS later.

Support seems to favor NeWS, or SunView, with vendor support, but actually getting fixes out of Sun is not always easy. Many X problems can be solved quickly – a note to `xperts` brings a flood of helpful suggestions, and often source patches.

4.4. Speed

MGR is very fast – the V6 of window systems. 8½ is also quick, because of it's spare, elegant design. NeWS seems fast because the interaction can be local to the server. X's speed is highly dependent on network throughput; but if you can spin around the interaction/network loop in less than a monitor refresh time it's probably fast enough. Network delays or machine load can slow this down, though.

So speed seems not to be the reason X is ubiquitous.

4.5. Portability

The portability of SunView was largely irrelevant. The code was unavailable anyway,† and you couldn't run it on anything but Suns anyway. It relied on special features in the SunOs kernel (`/dev/win*`, `/dev/mouse`, etc.) which reduced portability.

NeWS is pretty portable. It has run on Suns, SGI Iris's and Macintosh (under A/UX). It's not trivial to do though. Implementing a PostScript interpreter *and* the lightweight processes requires a good team or a blend of graphics and operating systems skill.

X is quite portable (but again non-trivial). In the server, the parts most likely to need work, such as the OS-dependent and graphics parts, are isolated into modules. Treated this way, to port the server to a UNIX system with a "reasonable" frame buffer is quite straightforward. Chances are the OS part has been done already (since these days the choices are either BSD or System V) and to get graphics running one need only examine the "spans" routines and write some for the new frame buffer. A high performance server is more work though, and some quite interesting approaches have been taken [McC91a].

Porting X to a non-UNIX environment is much more difficult. One of the hardest parts of dealing with the MIT code is the search-and-destroy mission to find and expunge all places where it is assumed that `sizeof(int)==sizeof(long)`. When one function pushes six 2-byte ints on the stack and another pops off six 4-byte ints, the results are not pretty.

† At least not without paying a large fee for a Sun source license, which also required an AT&T source license

Lint is not as helpful as you might think, since many of these places are in the arguments of functions only called through function pointers. Fully prototyping *all* the structures and routines in the server helps a lot here.†

4.6. Marketing

Here's where the real meat lies. I feel that the "success" of X can be laid to a combination of marketing and NIH.

- SunView and NeWS were Sun products, and DEC and IBM, for instance didn't want to appear to be kow-towing to that upstart youngster.
- SGI's MEX ran only on Iris machines, and customers demanded an "industry standard" window system, so SGI switched to NeWS and then X.
- MGR was never aggressively pushed at all; a small fraternity of discriminating "wizards" have adopted it, but wide use is rare.
- X on the other hand, was not a product of any one company but of MIT, even though DEC and IBM (and others) contributed to its development.

Thus, for all its technical warts, X was a *politically* acceptable choice for many companies casting about for a standard, portable window system.

References

[Gol83a] Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation,* Addison Wesley (May 1983).

[Gos88a] James Gosling, Tony Hoeber, and David Rosenthal, "Programming with NeWS," *SunTechnology*, pp. 54-59 (Winter 1988).

[Joh89a] Jeff Johnson, Teresa Roberts, William Verplank, David C Smith, Charles H Irby, Marian Beard, and Kevin Mackey, "The Xerox Star: A Retrospective," *Computer* **22**(9), pp. 11-30 (September 1989).

[Lip82a] Daniel E. Lipkie, Steven R. Evans, Jown K. Newlin, and Robert L. Weissman, "Star Graphics: An Object Oriented Implementation," *Computer Graphics* **16**(3) (July 1982).

[McC91a] Joel McCormack, "Writing Fast X Servers for Dumb Color Frame Buffers," *DEC Western Research Laboratory Research Report 91/1* (February 1991).

[Meh88a] Sunil Mehta, "A Clear Need for Windows," *UNIX World* **V**(3), pp. 58-66 (March 1988).

[Pik89a] Rob Pike, "A Concurrent Window System," *Computing Systems* **2**(2), pp. 133-153 (Spring 1989).

[Pik91a] Rob Pike, "8½: the Plan 9 Window System," *Usenix 1991 Summer Conference Proceedings* (1991).

† I almost called this paper "int considered harmful". If everyone just used `short` and `long` and *never ever* used `int` at all this wouldn't happen. But I digress.

[Rho85a] Rocky Rhodes, Paul Haeberli, and Kipp Hickman, "Mex – A Window Manager for the IRIS," *Usenix 1985 Summer Conference Proceedings*, pp. 381-92 (1985).

[Ros86a] David Rosenthal and James Gosling, "A Window Manager for Bitmapped Displays and Unix," in *Methodology of Window Managers*, Springer-Verlag, New York (1986).

[Ros88a] David Rosenthal, "A Simple X11 Client Program -or- How hard can it really be to write "Hello, World"?," *Usenix 1988 Winter Conference Proceedings*, pp. 229-242 (1988).

[Sch86a] Robert W. Scheifler and James Gettys, "The X Window System," *ACM Transactions on Graphics* 5(2), pp. 79-109 (April 1986).

[Sut63a] Ivan E. Sutherland, "Sketchpad: A Man-Machine Graphical Communication System," *Conference Proceedings, Spring Joint Computer Conference* (1963).

Against User Interface Copyright

(October 20, 1991)

The League for Programming Freedom
league@prep.ai.mit.edu

Abstract

This paper describes how the copyrighting of software interfaces threatens programmers' freedom to write software. It also shows how interface copyright obstructs progress even as it denies users the benefit of competition. In addition, interface copyright benefits mainly those who have already been so successful as to set a de-facto standard – primarily large companies. Conclusion: interface copyright is unsound public policy.

Introduction

In June 1990, Lotus won a copyright infringement suit against Paperback Software, a small company that implemented a spreadsheet that obeys the same keystroke commands used in Lotus 1-2-3. Paperback was not accused of copying code from 1-2-3 – only of supporting compatible user commands. Such imitation was common practice until unexpected court decisions in recent years extended the scope of copyright law.

Within a week, Lotus went on to sue Borland over Quattro, a spreadsheet whose usual command language has only a few similarities to 1-2-3. Lotus claims that these similarities in keystroke sequences and/or the ability to customize the interface to emulate 1-2-3 are enough to infringe.

More ominously, Apple Computer has sued Microsoft and Hewlett Packard for implementing a window system whose displays partially resemble those of the Macintosh system. Subsequently Xerox sued Apple for implementing the Macintosh system, which derives some general concepts from the earlier Xerox Star system. These suits try to broaden the Lotus decision and establish copyright on a large class of user interfaces. The Xerox lawsuit was dismissed because of a technicality; but if it had succeeded, it would probably have created an even broader monopoly than the Apple lawsuit may.

And Ashton-Tate has sued Fox Software for implementing a database program that accepts the same programming language used in dBase. This particular lawsuit was dropped by Borland, which bought Ashton-Tate in 1991, but the possibility of copyrighted programming languages remains. Adobe claims that the Postscript language is copyrighted, though it has not sued those who reject this claim. Wolfram

Reasearch claims that the language of Mathematica is copyrighted and has threatened to sue the University of California. If a programming language becomes copyrighted, the impact on users who have spent years writing programs in the language would be devastating.

While this paper addresses primarily the issue of copyright on specific user interfaces, most of the arguments apply with added force to any broader monopoly.

What Is a User Interface?

A user interface is what you have to learn to operate a machine; in other words, it is the language you use to communicate with the machine. The user interface of a typewriter is the layout of the keys. The user interface of a car includes a steering wheel for turning, pedals to speed up and slow down, a lever to signal turns, etc.

When the machine is a computer program, the interface includes that of the computer – its keyboard, screen and mouse – plus those aspects specific to the program. These typically include the commands, menus, programming languages, and the way data is presented on the screen.

A copyright on a user interface means a government-imposed monopoly on its use. In the example of the typewriter, this would mean that each manufacturer would be forced to arrange the keys in a different layout.

The Purpose of Copyright

In the United States, the Constitution says that the purpose of copyright is to "promote the progress of science and the useful arts." Conspicuously absent is any hint of intention to enrich copyright holders to the detriment of the users of copyrighted works.

The Supreme Court made the reason for this absence explicit, stating in *Fox Film vs. Doyal* that "The sole interest of the United States and the primary object in conferring the [copyright] monopoly lie in the general benefits derived by the public from the labors of authors."

In other words, since copyright is a government-imposed monopoly, which interferes with the freedom of the public in a significant way, it is justified only if the benefit to the public exceeds the cost to the public.

The spirit of individual freedom must, if anything, incline us against monopoly. Following either the Supreme Court or the principle of freedom, the fundamental question is: what value does user interface copyright offer the public – and what price would we have to pay for it?

Reason #1: More Incentive Is Not Needed

The developers of the Star, the Macintosh system, 1-2-3 and dBase claim that without interface copyright there would be insufficient incentive to develop such products. This is disproved by their own actions.

Until 1986, user interface copyright was unheard of. The computer industry developed under a system where imitating a user interface was both standard practice and lawful. Under this system, today's plaintiffs

made their decisions to develop their products. When faced with the choice in actuality, they decided that they did, indeed, have "enough incentive".

Even though competitors were free to imitate these interfaces, this did not prevent most of the original products from being successful and producing a large return on the investment. In fact, they were so successful that they became *de facto* standards. (The Xerox Star was a failure due to poor marketing even though nothing similar existed.)

Even if interface copyright would increase the existing incentive, additional improvements in user interfaces would not necessarily result. Once you suck a bottle dry, more suction won't get more out of it. The existing incentive is so great that it may well suffice to motivate everyone who has an idea worth developing. Extra incentive, at the public's expense, will only increase the price of these developments.

Reason #2: "Look and Feel" Will Not Protect Small Companies

The proponents of user interface copyright claim that it would protect small companies from being wiped out by large competitors. Yet look around: today's interface copyright plaintiffs are large, established companies. User interface copyright is crushing when the interface is an effective standard. However, a small company is vulnerable when its product is little used, and its interface is little known. In this situation, user interface copyright won't help the small company much.

Imagine a small company with 10,000 customers: a large company may believe there is a potential market of a million users, not reached by the small company, for a similar product. The large company will try to use its marketing might to reach them before the small company can.

User interface copyright won't change this outcome. Forcing the large company to develop an incompatible interface will have little effect on the majority of potential customers – those who have not learned the other interface. They will buy from the large company anyway.

What's more, interface copyright will work against the small company if the large company's product becomes an effective standard. Then new customers will have an additional reason to prefer the large company. To survive, the small company will need to offer compatibility with this standard – but, due to user interface copyright, it will not be allowed to do so.

Instead of relying upon monopolistic measures, small companies are most successful when they rely on their own inherent advantages: agility, low overhead, and willingness to take risks.

Reason #3: Diversity in Interfaces Is Not Desirable

The copyright system was designed to encourage diversity; its details work toward this end. Diversity is the primary goal when it comes to novels, songs, and the other traditional domains of copyright. Readers want to read novels they have not yet read.

But diversity is not the goal of interface design. Users of any kind of machinery want consistency in interfaces because this promotes ease of use. Thus, by standardizing symbols on automobile dashboards, we have made it possible for any licensed driver to operate any car without additional instruction. Incompatibility in interfaces is a price to be paid when worthwhile, not a benefit.

Significantly better interfaces may be hard to think of, but it is easy to invent interfaces which are merely different. Interface copyright will surely succeed in encouraging this sort of "interface development". The result will be gratuitous incompatibility.

Reason #4: Meaningful Competition Is Reduced

Under the regime of interface copyright, there will be no compatible competition for established products. For a user to switch to a different brand will require retraining.

But users don't like to retrain, not even for a significant improvement. For example, the Dvorak keyboard layout, invented several decades ago, enables a typist to type faster and more accurately than is possible with the standard "QWERTY" layout. Nonetheless, few people use it. Even new typists don't learn Dvorak, because they want to learn the layout used on most typewriters.

Alternative products that require such an effort by the consumer are not effective competition. The monopoly on the established interface will yield in practice a monopoly on the functionality accessed by it. This will cause higher prices and less technological advancement – a windfall for lucky businesses, but bad for the public at large.

Reason #5: Incompatibility Does Not Go Away

If there had been a 50-year interface copyright for the steering wheel, it would have expired not long ago. During the span of the copyright, we would have got cars steered with joysticks, cars steered with levers, and cars steered with pedals. Each car user would have had to choose a brand of car to learn to drive, and it would not be easy to switch.

The expiration of the copyright would have freed manufacturers to switch to the best of the known interfaces. But if Ford cars were steered with wheels and General Motors were steered with pedals, neither company could change interface without abandoning their old customers. It would take decades to converge on a single interface.

Reason #6: Users Invest More Than Developers

The plaintiffs like to claim that user interfaces represent large investments on their part.

In fact, the effort spent designing the user interface of a computer program is usually small compared to the cost of developing the program itself. The people who make a large investment in the user interface are the users who train to use it. Users have spent much more time and money learning to use 1-2-3 than Lotus spent developing the entire program, let alone what Lotus spent develop the program's interface *per se*.

Thus, if investment justifies ownership, it is the users who should be the owners. The users should be allowed to decide – in the marketplace – who may use it. According to *Infoworld* (mid January 1989), computer users in general expect user interface copyright to be harmful.

Reason #7: Discrimination Against Software Sharing

User interface copyright discriminates against freely redistributable software, such as freeware, shareware and public domain software.

Although it *may* be possible to license an interface for a proprietary program, if the owner is willing, these licenses require payment, usually per copy. There is no way to collect this payment for a freely redistributable program. The result will be a growing body of interfaces that are barred to non-proprietary software.

Authors of these programs donate to the public the right to share them, and sometimes also to study and change their workings. This is a public service, and one less common than innovation. It does not make sense to encourage innovation of one sort with means that bar donation of another sort.

Reason #8: Copyright Will Be a Tool For Extortion

The scope of interface copyright is so vague and potentially wide that it will be difficult for any programmer to be sure of being safe from lawsuits. Most programs need an interface, and there is usually no way to design an interface except based on the ideas you have seen used elsewhere. Only a great genius would be likely to envision a usable interface without a deep resemblance to current practice. It follows that most programming projects will risk an interface infringement suit.

The spirit of "Millions for defense, but not a cent for tribute" is little honored in business today. Customers and investors often avoid companies that are targets of suits; an eventual victory may come years too late to prevent great loss or even bankruptcy. Therefore, when offered a choice between paying royalties and being sued, most businesses pay, even if they would probably win a suit.

Since this tendency is well known, companies often take advantage of it by filing or threatening suits they are unlikely to win. As long as any interface copyright exists, this form of extortion will broaden its effective scope.

Reason #9: Useful Innovation Is Inhibited

Due to the evolutionary nature of interface development, interface copyright will actually retard progress.

Fully fleshed-out interfaces don't often arise as *tours de force* from the minds of isolated masters. They result from repeated implementations, by different groups, each learning from the results of previous attempts. For example, the Macintosh interface was based on ideas tried previously by Xerox and SRI, and before that by the Stanford Artificial Intelligence Laboratory. The Xerox Star also drew on the interface ideas that came from SRI and SAIL. 1-2-3 adapted the interface ideas of Visicalc and other spreadsheets. dBase drew on a program developed at the Jet Propulsion Laboratory.

This evolutionary process resembles the creation of folk art rather than the way symphonies, novels or films are made. The advances that we ought to encourage are most often small, localized changes to what someone else has done. If each interface has an owner, it will be difficult to implement such ideas. Even assuming the owner will license the interface that is to be improved, the inconvenience and expense would discourage all but the most determined.

Users often appreciate small, incremental changes that make programs easier or faster to use. This means changes that are upwards compatible, or affect only part of a well-known interface. Thus, on computer keyboards, we now have function keys, arrow keys, a delete key and a control key, which typewriters did not have. But the layout of the letters is unchanged.

However, such partial changes as this are not permitted by copyright law. If any significant portion of the new interface is the same as a copyrighted interface, the new interface is illegal.

Reason #10: Interface Developers Don't Want Interface Copyright

At the 1989 ACM Conference on Computer-Human Interaction, Professor Samuelson of the Emory School of Law presented a "mock trial" with legal arguments for and against user interface copyright, and then asked the attendees – researchers and developers of user interfaces – to fill out a survey of their opinion on the subject.

The respondents overwhelmingly opposed all aspects of user interface copyright, by as much as 4 to 1 for some aspects. When they were asked whether user interface copyright would harm or help the field, on a scale from 1 (harm) to 5 (help), the average answer was 1.6.†

The advocates of user interface copyright say that it would provide better security and income for user interface designers. However, the survey shows that these supposed beneficiaries would prefer to be let alone.

Do You Really Want a User Interface Copyright?

For a business, "locking in" customers may be profitable for a time. But, as the vendors of proprietary operating systems have found out, this generates resentment and eventually drives customers to try to escape. In the long run, this leads to failure.

Therefore, by permitting user interface copyright, society encourages counterproductive thinking in its businesses. Not all businesses can resist this temptation; let us not tempt them.

Conclusion

Monopolies on user interfaces do not serve the users and do not "promote the progress of science and the useful arts." User interfaces ought to be the common property of all, as they undisputedly were until a few years ago.

What You Can Do

- Don't do business as usual with the plaintiffs, Xerox, Lotus, and Apple. Buy from their competitors instead; sell their stock; develop new software for other computer systems rather than theirs, and port existing applications away from their systems.
- Don't work for the "look and feel" plaintiffs or accept contracts from them.

† See the May 1990 issue of the *Communications of the ACM,* for the full results.

- Join the League for Programming Freedom – a grass-roots organization of programmers and users opposing software patents and interface copyrights. (The League is not opposed to copyright on individual programs.) Annual dues are $42 for employed professionals, $10.50 for students, and $21 for others. We appreciate activists, but members who cannot contribute their time are also welcome.

  Phone us at (617) 243-4091, send Internet mail to the address `league@prep.ai.mit.edu`, or write to:

  League for Programming Freedom
  1 Kendall Square #143
  P.O. Box 9171
  Cambridge, MA 02139

- Give copies of this paper to your friends, colleagues and customers.
- In the United States, write to your representatives and to these Congressional subcommittees:

  House Subcommittee on Intellectual Property
  2137 Rayburn Bldg
  Washington, DC 20515

  Senate Subcommittee on Patents,
  Trademarks and Copyrights
  United States Senate
  Washington, DC 20510

- The European Community has adopted a directive whose most natural interpretation imposes copyright on all kinds of interfaces, even on programming languages. Since the other countries of Europe are considering joining the EC, they also are in danger of being covered by the directive.

  Other, benign interpretations of the directive are also possible, but they are unlikely to be chosen by judges unless the governments of the individual EC countries explicitly mandate them. Convincing the governments requires political pressure from the programmers and users of Europe.

  Lobbyists working on this issue say that most legislators are unfamiliar with computers and do not understand how harmful interface copyright could be. Thus, what programmers need to do is to educate their legislators.

  One idea is to start teaching your representative the basics of using 1-2-3. Once the representative sees how much work is involved in learning to use a command language, explain that you have only taught one tenth of the subject. This should drive the point home.

  Political effectiveness requires organization. Leagues for Programming Freedom now exist in Finland, Germany, the United Kingdom, the Netherlands, Norway, and Switzerland. (In the UK, the Edinburgh Computing and Social Responsibility organization also deals with this issue.) Ask the League in the US for the address of your nation's League – or for advice and assistance in forming one.

Against Software Patents

(February 28, 1991)

The League for Programming Freedom
league@prep.ai.mit.edu

Abstract

This paper describes how the extension of patents to cover software techniques, algorithms and features threatens programmers' freedom to write software. It explains why patents in the software field yield comparitively little benefit in the form of additional published techniques, algorithms and features, while greatly hindering the persuit of software development; and, in addition, how they favor the largest companies against all others. Conclusion: the application of patent law to software is unsound public policy.

Introduction

Software patents threaten to devastate America's computer industry. Patents granted in the past decade are now being used to attack companies such as the Lotus Development Corporation for selling programs that they have independently developed. Soon new companies will often be barred from the software arena – most major programs will require licenses for dozens of patents, and this will make them infeasible. This problem has only one solution: software patents must be eliminated.

The Patent System and Computer Programs

The framers of the United States Constitution established the patent system so that inventors would have an incentive to share their inventions with the general public. In exchange for divulging an invention, the patent grants the inventor a 17 year monopoly on its use. The patent holder can license others to use the invention, but may also refuse to do so. Independent reinvention of the same technique by others does not give them the right to use it.

Patents do not cover specific systems: instead, they cover particular techniques that can be used to build systems, or particular features that systems can offer. Once a technique or feature is patented, it may not be used in a system without the permission of the patent-holder – even if it is implemented in a different way. Since a computer program typically uses many techniques and provides many features, it can infringe many patents at once.

Until recently, patents were not used in the software field. Software developers copyrighted individual programs or made them trade secrets. Copyright was traditionally understood to cover the implementation details of a particular program; it did not cover the features of the program, or the general methods used. And trade secrecy, by definition, could not prohibit any development work by someone who did not know the secret.

On this basis, software development was extremely profitable, and received considerable investment, without any prohibition on independent software development. But this scheme of things is no more. A change in U.S. government policy in the early 1980's stimulated a flood of applications. Now many have been approved, and the rate is accelerating.

Many programmers are unaware of the change and do not appreciate the magnitude of its effects. Today the lawsuits are just beginning.

Absurd Patents

The Patent Office and the courts have had a difficult time with computer software. The Patent Office refused until recently to hire Computer Science graduates as examiners, and in any case does not offer competitive salaries for the field. Patent examiners are often ill-prepared to evaluate software patent applications to determine if they represent techniques that are widely known or obvious – both of which are grounds for rejection.

Their task is made more difficult because many commonly-used software techniques do not appear in the scientific literature of computer science. Some seemed too obvious to publish while others seemed insufficiently general; some were open secrets.

Computer scientists know many techniques that can be generalized to widely varying circumstances. But the Patent Office seems to believe that each separate use of a technique is a candidate for a new patent. For example, Apple was sued because the Hypercard program allegedly violates patent number 4,736,308, a patent that covers displaying portions of two or more strings together on the screen – effectively, scrolling with multiple subwindows. Scrolling and subwindows are well-known techniques, but combining them is now apparently illegal.

The granting of a patent by the Patent Office carries a presumption in law that the patent is valid. Patents for well-known techniques that were in use many years before the patent application have been upheld by federal courts. It can be hard to prove a technique was well known at the time in question.

For example, the technique of using exclusive-or to write a cursor onto a screen is both well known and obvious. (Its advantage is that another identical exclusive-or operation can be used to erase the cursor without damaging the other data on the screen.) This technique can be implemented in a few lines of a program, and a clever high school student might well reinvent it. But it is covered by patent number 4,197,590, which has been upheld twice in court even though the technique was used at least five years before the patent application. Cadtrak, the company that owns this patent, collects millions of dollars from large computer manufacturers.

English patents covering customary graphics techniques, including airbrushing, stenciling, and combination of two images under control of a

third one, were recently upheld in court, despite the testimony of the pioneers of the field that they had developed these techniques years before. (The corresponding United States patents, including 4,633,416 and 4,602,286, have not yet been tested in court, but they probably will be soon.)

All the major developers of spreadsheet programs have been threatened on the basis of patent 4,398,249, covering "natural order recalc" – the recalculation of all the spreadsheet entries that are affected by the changes the user makes, rather than recalculation in a fixed order. Currently Lotus alone is being sued, but a victory for the plaintiff in this case would leave the other developers little hope. The League has found prior art that may defeat this patent, but this is not assured.

Nothing protects programmers from accidentally using a technique that is patented, and then being sued for it. Taking an existing program and making it run faster may also make it violate half a dozen patents that have been granted, or are about to be granted.

Even if the Patent Office learns to understand software better, the mistakes it is making now will follow us into the next century, unless Congress or the Supreme Court intervenes to declare these patents void.

However, this is not the whole of the problem. Computer programming is fundamentally different from the other fields that the patent system previously covered. Even if the patent system were to operate "as intended" for software, it would still obstruct the industry it is supposed to promote.

What Is "Obvious"?

The patent system will not grant or uphold patents that are judged to be obvious. However, the system interprets the word "obvious" in a way that might surprise computer programmers. The standard of obviousness developed in other fields is inappropriate for software.

Patent examiners and judges are accustomed to considering even small, incremental changes as deserving new patents. For example, the famous *Polaroid vs. Kodak* case hinged on differences in the number and order of layers of chemicals in a film – differences between the technique Kodak was using and those described by previous, expired patents. The court ruled that these differences were unobvious.

Computer scientists solve problems quickly because the medium of programming is tractable. They are trained to generalize solution principles from one problem to another. One such generalization is that a procedure can be repeated or subdivided. Programmers consider this obvious – but the Patent Office did not think that it was obvious when it granted the patent on scrolling multiple strings, described above.

Cases such as this cannot be considered errors. The patent system is functioning as it was designed to do – but with software, it produces outrageous results.

Patenting What Is Too Obvious to Publish

Sometimes it is possible to patent a technique that is not new precisely because it is obvious – so obvious that no one would have published a paper about it.

For example, computer companies distributing the free X Window System developed by MIT are now being threatened with lawsuits by AT&T over patent number 4,555,775, covering the use of "backing store" in a window system that lets multiple programs have windows. Backing store means that the contents of a window that is temporarily partly hidden are saved in off-screen memory, so they can be restored quickly if the obscuring window disappears.

Early window systems were developed on computers that could not run two programs at once. These computers had small memories, so saving window contents was obviously a waste of scarce memory space. Later, larger multiprocessing computers led to the use of backing store, and to permitting each program to have its own windows. The combination was inevitable.

The technique of backing store was used at MIT in the Lisp Machine System before AT&T applied for a patent. (By coincidence, the Lisp Machine also supported multiprocessing.) The Lisp Machine developers published nothing about backing store at the time, considering it too obvious. It was mentioned when a programmers' manual explained how to turn it on and off.

But this manual was published one week after the AT&T patent application – too late to count as prior art to defeat the patent. So the AT&T patent may stand, and MIT may be forbidden to continue using a method that MIT used before AT&T.

The result is that the dozens of companies and hundreds of thousands of users who accepted the software from MIT on the understanding that it was free are now faced with possible lawsuits. (They are also being threatened with Cadtrak's exclusive-or patent.) The X Window System project was intended to develop a window system that all developers could use freely. This public service goal seems to have been thwarted by patents.

Why Software Is Different

Software systems are much easier to design than hardware systems of the same number of components. For example, a program of 100,000 components might be 50,000 lines long and could be written by two good programmers in a year. The equipment needed for this costs less than $10,000; the only other cost would be the programmers' own living expenses while doing the job. The total investment would be less than a $100,000. If done commercially in a large company, it might cost twice that. By contrast, an automobile typically contains under 100,000 components; it requires a large team and costs tens of millions of dollars to design.

And software is also much cheaper to manufacture: copies can be made easily on an ordinary workstation costing under ten thousand dollars. To produce a complex hardware system often requires a factory costing tens of millions of dollars.

Why is this? A hardware system has to be designed using real components. They have varying costs; they have limits of operation; they may be sensitive to temperature, vibration or humidity; they may generate noise; they drain power; they may fail either momentarily or permanently. They must be physically assembled in their proper places, and they must be accessible for replacement in case they fail.

Moreover, each of the components in a hardware design is likely to affect the behavior of many others. This greatly complicates the task

of determining what a hardware design will do: mathematical modeling may prove wrong when the design is built.

By contrast, a computer program is built out of ideal mathematical objects whose behavior is defined, not modeled approximately, by abstract rules. When an if-statement follows a while-statement, there is no need to study whether the if-statement will draw power from the while-statement and thereby distort its output, nor whether it could overstress the while-statement and make it fail.

Despite being built from simple parts, computer programs are incredibly complex. The program with 100,000 parts is as complex as an automobile, though far easier to design.

While programs cost substantially less to write, market and sell than automobiles, the cost of dealing with the patent system will not be less. The same number of components will, on the average, involve the same number techniques that might be patented.

The Danger of a Lawsuit

Under the current patent system, a software developer who wishes to follow the law must determine which patents a program violates and negotiate with each patent holder a license to use that patent. Licensing may be prohibitively expensive, or even unavailable if the patent is held by a competitor. Even "reasonable" license fees for several patents can add up to make a project infeasible. Alternatively, the developer may wish to avoid using the patent altogether; but there may be no way around it.

The worst danger of the patent system is that a developer might find, after releasing a product, that it infringes one or many patents. The resulting lawsuit and legal fees could force even a medium-size company out of business.

Worst of all, there is no practical way for a software developer to avoid this danger – there is no effective way to find out what patents a system will infringe. There is a way to try to find out – a patent search – but searches are unreliable and in any case too expensive to use for software projects.

Patent Searches Are Prohibitively Expensive

A system with a hundred thousand components can use hundreds of techniques that might already be patented. Since each patent search costs thousands of dollars, searching for all the possible points of danger could easily cost over a million. This is far more than the cost of writing the program.

The costs don't stop there. Patent applications are written by lawyers for lawyers. A programmer reading a patent may not believe that his program violates the patent, but a federal court may rule otherwise. It is thus now necessary to involve patent attorneys at every phase of program development.

Yet this only reduces the risk of being sued later – it does not eliminate the risk. So it is necessary to have a reserve of cash for the eventuality of a lawsuit.

When a company spends millions to design a hardware system, and plans to invest tens of millions to manufacture it, an extra million or

two to pay for dealing with the patent system might be bearable. However, for the inexpensive programming project, the same extra cost is prohibitive. Individuals and small companies especially cannot afford these costs. Software patents will put an end to software entrepreneurs.

Patent Searches Are Unreliable

Even if developers could afford patent searches, these are not a reliable method of avoiding the use of patented techniques. This is because patent searches do not reveal pending patent applications (which are kept confidential by the Patent Office). Since it takes several years on the average for a software patent to be granted, this is a serious problem: a developer could begin designing a large program after a patent has been applied for, and release the program before the patent is approved. Only later will the developer learn that distribution of the program is prohibited.

For example, the implementors of the widely-used public domain data compression program `compress` followed an algorithm obtained from the journal *IEEE Computer*. (This algorithm is also used in several popular programs for microcomputers, including `PKZIP`.) They and the user community were surprised to learn later that patent number 4,558,302 had been issued to one of the authors of the article. Now Unisys is demanding royalties for using this algorithm. Although the program `compress` is still in the public domain, using it means risking a lawsuit.

The Patent Office does not have a workable scheme for classifying software patents. Patents are most frequently classified by end results, such as "converting iron to steel;" but many patents cover algorithms whose use in a program is entirely independent of the purpose of the program. For example, a program to analyze human speech might infringe the patent on a speedup in the Fast Fourier Transform; so might a program to perform symbolic algebra (in multiplying large numbers); but the category to search for such a patent would be hard to predict.

You might think it would be easy to keep a list of the patented software techniques, or even simply remember them. However, managing such a list is nearly impossible. A list compiled in 1989 by lawyers specializing in the field omitted some of the patents mentioned in this paper.

Obscure Patents

When you imagine an invention, you probably think of something that could be described in a few words, such as "a flying machine with fixed, curved wings" or "an electrical communicator with a microphone and a speaker". But most patents cover complex detailed processes that have no simple descriptions – often they are speedups or variants of well-known processes that are themselves complex.

Most of these patents are neither obvious nor brilliant; they are obscure. A capable software designer will "invent" several such improvements in the course of a project. However, there are many avenues for improving a technique, so no single project is likely to find any given one.

For example, IBM has several patents (including patent number 4,656,583) on workmanlike, albeit complex, speedups for well-known

computations performed by optimizing compilers, such as register coloring and computing the available expressions.

Patents are also granted on combinations of techniques that are already widely used. One example is IBM patent 4,742,450, which covers "shared copy-on-write segments." This technique allows several programs to share the same piece of memory that represents information in a file; if any program writes a page in the file, that page is replaced by a copy in all of the programs, which continue to share that page with each other but no longer share with the file.

Shared segments and copy-on-write have been used since the 1960's; this particular combination may be new as a specific feature, but is hardly an invention. Nevertheless, the Patent Office thought that it merited a patent, which must now be taken into account by the developer of any new operating system.

Obscure patents are like land mines: other developers are more likely to reinvent these techniques than to find out about the patents, and then they will be sued. The chance of running into any one of these patents is small, but they are so numerous that you cannot go far without hitting one. Every basic technique has many variations, and a small set of basic techniques can be combined in many ways. The patent office has now granted at least 2000 software patents – no less than 700 in 1989 alone, according to a list compiled by EDS. We can expect the pace to accelerate. In ten years, programmers will have no choice but to march on blindly and hope they are lucky.

Patent Licensing Has Problems, Too

Most large software companies are trying to solve the problem of patents by getting patents of their own. Then they hope to cross-license with the other large companies that own most of the patents, so they will be free to go on as before.

While this approach will allow companies like Microsoft, Apple and IBM to continue in business, it will shut new companies out of the field. A future start-up, with no patents of its own, will be forced to pay whatever price the giants choose to impose. That price might be high: established companies have an interest in excluding future competitors. The recent Lotus lawsuits against Borland and the Santa Cruz Operation (although involving an extended idea of copyright rather than patents) show how this can work.

Even the giants cannot protect themselves with cross-licensing from companies whose only business is to obtain exclusive rights to patents and then threaten to sue. For example, consider the New York-based Refac Technology Development Corporation, representing the owner of the "natural order recalc" patent. Contrary to its name, Refac does not develop anything except lawsuits – it has no business reason to join a cross-licensing compact. Cadtrak, the owner of the exclusive-or patent, is also a litigation company.

Refac is demanding five percent of sales of all major spread-sheet programs. If a future program infringes on twenty such patents – and this is not unlikely, given the complexity of computer programs and the broad applicability of many patents – the combined royalties could exceed 100% of the sales price. (In practice, just a few patents can make a program unprofitable.)

The Fundamental Question

According to the Constitution of the United States, the purpose of patents is to "promote the progress of science and the useful arts." Thus, the basic question at issue is whether software patents, supposedly a method of encouraging software progress, will truly do so, or will retard progress instead.

So far we have explained the ways in which patents will make ordinary software development difficult. But what of the intended benefits of patents: more invention, and more public disclosure of inventions? To what extent will these actually occur in the field of software?

There will be little benefit to society from software patents because invention in software was already flourishing before software patents, and inventions were normally published in journals for everyone to use. Invention flourished so strongly, in fact, that the same inventions were often found again and again.

In Software, Independent Reinvention Is Commonplace

A patent is an absolute monopoly; everyone is forbidden to use the patented process, even those who reinvent it independently. This policy implicitly assumes that inventions are rare and precious, since only in those circumstances is it beneficial.

The field of software is one of constant reinvention; as some people say, programmers throw away more "inventions" each week than other people develop in a year. And the comparative ease of designing large software systems makes it easy for many people to do work in the field. A programmer solves many problems in developing each program. These solutions are likely to be reinvented frequently as other programmers tackle similar problems.

The prevalence of independent reinvention negates the usual purpose of patents. Patents are intended to encourage inventions and, above all, the disclosure of inventions. If a technique will be reinvented frequently, there is no need to encourage more people to invent it; since some of the developers will choose to publish it (if publication is merited), there is no point in encouraging a particular inventor to publish it – not at the cost of inhibiting use of the technique.

Overemphasis of Inventions

Many analysts of American and Japanese industry have attributed Japanese success at producing quality products to the fact that they emphasize incremental improvements, convenient features and quality rather than noteworthy inventions.

It is especially true in software that success depends primarily on getting the details right. And that is most of the work in developing any useful software system. Inventions are a comparatively unimportant part of the job.

The idea of software patents is thus an example of the mistaken American preoccupation with inventions rather than products. And patents will encourage this mistaken focus, even as they impede the development work that actually produces better software.

Impeding Innovation

By reducing the number of programmers engaged in software development, software patents will actually impede innovation. Much software innovation comes from programmers solving problems while developing software, not from projects whose specific purpose is to make inventions and obtain patents. In other words, these innovations are byproducts of software development.

When patents make development more difficult, and cut down on development projects, they will also cut down on the byproducts of development – new techniques.

Could Patents Ever Be Beneficial?

Although software patents in general are harmful to society as a whole, we do not claim that every single software patent is necessarily harmful. Careful study might show that under certain specific and narrow conditions (necessarily excluding the vast majority of cases) it is beneficial to grant software patents.

Nonetheless, the right thing to do now is to eliminate all software patents as soon as possible, before more damage is done. The careful study can come afterward.

Clearly software patents are not urgently needed by anyone except patent lawyers. The pre-patent software industry had no problem that was solved by patents; there was no shortage of invention, and no shortage of investment.

Complete elimination of software patents may not be the ideal solution, but it is close, and is a great improvement. Its very simplicity helps avoid a long delay while people argue about details.

If it is ever shown that software patents are beneficial in certain exceptional cases, the law can be changed again at that time – if it is important enough. There is no reason to continue the present catastrophic situation until that day.

Software Patents Are Legally Questionable

It may come as a surprise that the extension of patent law to software is still legally questionable. It rests on an extreme interpretation of a particular 1981 Supreme Court decision, *Diamond vs. Deihr.*†

Traditionally, the only kinds of processes that could be patented were those for transforming matter (such as, for transforming iron into steel). Many other activities which we would consider processes were entirely excluded from patents, including business methods, data analysis, and "mental steps." This was called the "subject matter" doctrine.

Diamond vs. Deihr has been interpreted by the Patent Office as a reversal of this doctrine, but the court did not explicitly reject it. The case concerned a process for curing rubber – a transformation of matter. The issue at hand was whether the use of a computer program in the process was enough to render it unpatentable, and the court ruled that it was not. The Patent Office took this narrow decision as a green light

† See "Legally Speaking" in *Communications of the ACM,* August 1990.

for unlimited patenting of software techniques, and even for the use of software to perform specific well-known and customary activities.

Most patent lawyers have embraced the change, saying that the new boundaries of patents should be defined over decades by a series of expensive court cases. Such a course of action will certainly be good for patent lawyers, but it is unlikely to be good for software developers and users.

One Way to Eliminate Software Patents

We recommend the passage of a law to exclude software from the domain of patents. That is to say that, no matter what patents might exist, they would not cover implementations in software; only implementations in the form of hard-to-design hardware would be covered. An advantage of this method is that it would not be necessary to classify patent applications into hardware and software when examining them.

Many have asked how to define software for this purpose – where the line should be drawn. For the purpose of this legislation, software should be defined by the characteristics that make software patents especially harmful:

- Software is built from ideal infallible mathematical components, whose outputs are not affected by the components they feed into.

  Ideal mathematical components are defined by abstract rules, so that failure of a component is by definition impossible. The behavior of any system built of these components is likewise defined by the consequences of applying the rules step by step to the components.

- Software can be easily and cheaply copied.

Following this criterion, a program to compute prime numbers is a piece of software. A mechanical device designed specifically to perform the same computation is not software, since mechanical components have friction, can interfere with each other's motion, can fail, and must be assembled physically to form a working machine.

Any piece of software needs a hardware platform in order to run. The software operates the features of the hardware in some combination, under a plan. Our proposal is that combining the features in this way can never create infringement. If the hardware alone does not infringe a patent, then using it in a particular fashion under control of a program should not infringe either. In effect, a program is an extension of the programmer's mind, acting as a proxy for the programmer to control the hardware.

Usually the hardware is a general purpose computer, which implies no particular application. Such hardware cannot infringe any patents except those covering the construction of computers. Our proposal means that, when a user runs such a program on a general purpose computer, no patents other than those should apply.

The traditional distinction between hardware and software involves a complex of characteristics that used to go hand in hand. Some newer technologies, such as gate arrays and silicon compilers, blur the distinction because they combine characteristics associated with hardware with others associated with software. However, most of these technologies can be classified unambiguously for patent purposes, either as software or as hardware, using the criteria above. A few gray areas

may remain, but these are comparatively small, and need not be an obstacle to solving the problems patents pose for ordinary software development. They will eventually be treated as hardware, as software, or as something in between.

What You Can Do

One way to help eliminate software patents is to join the League for Programming Freedom. The League is a grass-roots organization of programmers and users opposing software patents and interface copyrights. (The League is not opposed to copyright on individual programs.) Annual dues for individual members are $42 for employed professionals, $10.50 for students, and $21 for others. We appreciate activists, but members who cannot contribute their time are also welcome.

To contact the League, phone (617) 243-4091, send Internet mail to the address `league@prep.ai.mit.edu`, or write to:

League for Programming Freedom
1 Kendall Square #143
PO Box 9171
Cambridge, MA 02139

In the United States, another way to help is to write to Congress. You can write to your own representatives, but it may be even more effective to write to the subcommittees that consider such issues:

House Subcommittee on Intellectual Property
2137 Rayburn Bldg
Washington, DC 20515

Senate Subcommittee on Patents,
Trademarks and Copyrights
United States Senate
Washington, DC 20510

You can phone your representatives at (202) 225-3121, or write to them using the following addresses:

Senator So and So
United States Senate
Washington, DC 20510

Representative Such and Such
House of Representatives
Washington, DC 20515

Fighting Patents One by One

Until we succeed in eliminating all patenting of software, we must try to overturn individual software patents. This is very expensive and can solve only a small part of the problem, but that is better than nothing.

Overturning patents in court requires prior art, which may not be easy to find. The League for Programming Freedom will try to serve as a clearing house for this information, to assist the defendants in software patent suits. This depends on your help. If you know about prior art for any software patent, please send the information to the League at the address given above.

If you work on software, you can personally help prevent software patents by refusing to cooperate in applying for them. The details of this may depend on the situation.

Conclusion

Exempting software from the scope of patents will protect software developers from the insupportable cost of patent searches, the wasteful struggle to find a way clear of known patents, and the unavoidable danger of lawsuits.

If nothing is changed, what is now an efficient creative activity will become prohibitively expensive. To picture the effects, imagine if each square of pavement on the sidewalk had an owner, and pedestrians required a license to step on it. Imagine the negotiations necessary to walk an entire block under this system. That is what writing a program will be like if software patents continue. The sparks of creativity and individualism that have driven the computer revolution will be snuffed out.

Practical Problems with Porting Software

Brian O'Donovan

Digital International B.V.
Galway
Republic of Ireland
odonovan@ilo.dec.com

Abstract

This paper describes some of the problems that are encountered when porting software between truly heterogeneous systems. As a practical example it describes the problems that were encountered when porting a large software system from VMS to ULTRIX.

1. Introduction

Many papers which describe the development of portable software concentrate entirely on issues relating to coding style. However, there are many additional issues to be faced when porting software between heterogeneous systems. For example, it is relatively difficult to set up a development environment which facilitates multi-platform development.

This paper attempts to describe some of the more practical problems which must be overcome when developing software which must work on heterogeneous operating systems. As an example, the paper will describe the problems which we had to overcome during the development of an application which was originally written to run on VMS and then later ported to two flavors of ULTRIX (Digital's UNIX based operating system).

The paper will not describe the application in detail. Instead, it concentrates on the problems which were encountered when porting it between VMS and ULTRIX and how these problems were overcome. It was relatively simple to make the tool portable between the VAX based and MIPS based versions of ULTRIX; therefore the paper will not place much emphasis on this aspect of the development.

2. The Application

Our application is a menu based system to simplify the installation and licensing of software products. Typically a customer would be provided with one or more CDROMs containing software products and a

file containing Product Authorization Keys (PAKs) for these products. When using our system the user will be presented with a menu of all available products. The user can then select which products he/she wants to license and/or install. When the user has selected all the desired products to license and/or install, the system will carry out these actions automatically in batch mode.

If the user was not using our application, he/she would have to manually type the contents of the PAK into the license database. If a large number of PAKs are being registered, this can become a very tedious and time consuming task. Our system can save the system managers' valuable time by automatically transferring the appropriate PAKs from the file to the license database.

The user of our application can also save some time by eliminating the need for someone to manually answer all of the questions that might arise when doing an product installation. This is done by storing the answers to any of the possible installation questions in an *answer file*. Our system will answer any questions that arise during the installation of a product by searching for an answer in the answer file. Customers are normally supplied with at least one answer file per product containing the most commonly used defaults. However, customers can create their own default options by editing these answer files and inserting different answers to the installation questions.

Our application will record a central history of all the products it has licensed and/or installed. This is done by sending messages across the network to a central history server. It also contains a utility for detecting products which were licensed and/or installed manually (i.e., without using our system). Therefore our system can be used to maintain a database of exactly what products are installed and licensed on each node in the network. The information in this central node can be converted into an itimised bill for the customer's software usage.

Our system is quite a large and complex piece of software. It comprises approximately 85 thousand lines of code in 66 different source files.† There were as many as 8 different developers working simultaneously on the system. This meant that it was necessary to adopt well defined work practices in order to avoid any conflicts between changes made by each of the developers. The possibility for conflicts was exacerbated by the fact that developers would make a change on one operating system without necessarily giving full consideration to the effect that this change would have on the operation of the system on the other operating system(s).

Luckily, both VMS and ULTRIX support the LMF license management system. Hence the code to license products could be reasonably similar on both operating systems since the same license manager was being used. However, VMS and ULTRIX have totally different systems for installing software (VMS uses `vmsinstall` while ULTRIX uses `setld`). Therefore, the installation code had to be significantly different on each system.

Our application will probably be further expanded to support other operating systems, (e.g. MS-DOS, Apple Macintosh, Solaris and HP-UX) installation systems (e.g. OSF's ANDF) [OSF91a] and license managers (e.g. HP's NetLS). However, the schedule for the order in which these are supported depends entirely upon customer demand.

† Comment lines are included in the "lines of code" measurement because it is easier to calculate. However, header files were not included. There are an additional 70 header files.

3. Providing Access to the Sources

When you are porting software between heterogeneous operating systems you do not want to end up developing several different pieces of software (one for each of the operating systems). If you have several different pieces of software, you will have a maintenance problem whereby you will need to apply software patches to each variant of the software separately. To avoid this problem, it is important to ensure that (whenever it is practical) you use the same source files for each variant of the software.

On our project all C source files were shared between the ULTRIX and VMS variants of the system except for one module which accomplishes the interface to the operating system. It was not practical, however, to use the same makefile, install procedure, or help files. Although the message cataloging systems on VMS and ULTRIX require that the message source files should be in different formats, we found that it was quite simple to write a utility that converted the message source file from one format to another. Hence we ware able to share the same message source file for both VMS and ULTRIX.

Most projects use a version control system to store and manage the source files. Conflicting updates to the source files are avoided by forcing developers to acquire an exclusive lock on a source file before making any changes to it. In order to ensure that the developers working on ULTRIX did not make any changes that conflicted with the changes made by developers working on VMS, it was essential that they used the same version control library to store the source files.

The project team originally decided to use DEC/CMS [DEC87a] to store and manage the various source files for the VMS version. Unfortunately DEC/CMS is not available for ULTRIX; likewise, neither of the ULTRIX version control tools (RCS and SCCS) are available on VMS. We decided that we should continue to use DEC/CMS to store and manage the source files for the portable version of our system, because there was no better alternative available. This meant that we had to build our own interface to allow the ULTRIX based developers to access the DEC/CMS library which is stored on VMS.

We combined use of both the "VMS/ULTRIX Connection" (UCX) product [DEC90a] and the DECNET communications package to to build our ULTRIX interface to CMS.

> The UCX product allows VMS directories to be NFS exported to ULTRIX systems. We used this product to allow the ULTRIX systems access the source files which are stored on VMS.
>
> DECNET is a communications protocol which is the primary mode of communications for VMS systems. DECNET is also available for ULTRIX systems. We used DECNET for some of the communications between the ULTRIX systems and the VMS systems. In particular we used the `dcp -S` command which allows you to remotely submit a DCL command procedure† for execution on a VMS system.

The ULTRIX to CMS interface was implemented in a makefile. It was decided to do it this way so that the commands to manage the source files would be integrated with the commands to build the executables. In retrospect, however, this was a bad decision because the makefile grew extremely complex (in excess of 600 lines). It would perhaps

† A DCL command procedure is to VMS what a shell script is to UNIX.

have been better to implement the interface in a shell script and then put simple rules into the makefile to call the shell script.

The makefile had three targets for accessing the CMS library.

fetch ⇒ retrieve a read only copy of the current versions of all source files.

reserve ⇒ retrieve a writable copy of the specified source files.

replace ⇒ return the modified source files into the CMS library (as new versions).

When building the retrieve and replace targets the user must supply values for the macro `FILES` (the files to retrieve/replace) and the macro `MSG` (the log message to record in CMS). For example the following command will retrieve writable copies of the source files `source_file.c` and `source_file.h`. It will also store the log message "changing the parameters to the send_packet routine" in the CMS library as the reason for locking the files.

```
% make reserve FILES="source_file.c source_file.h" \
    MSG="changing the parameters to the send_packet routine"
```

This command will cause the following steps to be executed:

- A DCL command procedure will be generated to reserve the files from the CMS library into a spool directory on the VMS system. Although this command procedure will be run on VMS, it is actually generated by the ULTRIX system.
- The command procedure will be submitted for execution on the VMS node by using the `dcp -S` command.
- When the DCL command procedure completes it will create a file named `done.tmp` in the spool directory if no errors were encountered. If errors were encountered a file named `error.tmp` will be created.
- The ULTRIX system has mounted the VMS spool directory via UCX, hence it can see when the `done.tmp` or `error.tmp` file is created. If the `error.tmp` file is created, an error message is returned to the user. If the `done.tmp` file is created, the reserved files are copied from the spool directory into the developer's personal source directory.

A similar procedure is followed for both the `fetch` and `replace` targets.

Our mechanisms for sharing files between VMS and ULTRIX work satisfactorily. However, they still have a number of significant problems.

- It is not possible to reserve or replace files from the project library when either the central VMS system is down or when the UCX software is not running.
- Accessing the project library from ULTRIX is very slow because of the multi-step procedure involved.
- The error handling is very crude. When errors occur on the VMS side it is not possible to determine the cause of the problem without logging into the VMS system and examining various log files.
- The interface only allows access to the default version of the source files. If an ULTRIX based developer wants any other version of the source files they must log into the VMS system and retrieve the file themselves
- CMS has a feature whereby a copy of all newly created versions of the source files are automatically placed in what is called the

reference directory. VMS based developers can ensure that they are using the latest version of the sources by simply including the reference directory in their search path. Unfortunately, this feature does not work for ULTRIX. Hence it is necessary for the ULTRIX based developers to periodically fetch a new copy of each source files in order to ensure that they are using the most recent version available.

The only true way to solve all of these problems would be to have a distributed version control system which is accessible from either VMS based systems or UNIX based systems. As far as we know no such tool exists today. There is a possibility that DRCS [O'D90a, O'D90b] may be ported to VMS. If this is done it should provide an ideal solution to the problems inherent in sharing source code between VMS based systems and UNIX based systems.

4. Coding Techniques

We said earlier that you should aim to use the same source code on all platforms. Unfortunately, it is inevitable that sometimes the code must perform slightly differently on each platform. The way to get the same source file to compile differently on each platform is by using the pre-processor's `#ifdef` construct.

Each compiler will pre-define certain pre-processor macros. For example the VMS C compiler will pre-define the macro `VMS`, all VAX type systems will pre-define the `VAX` macro, all systems derived from the Berkeley variant of UNIX will pre-define the BSD macro, etc.. The following code fragment shows how these pre-defined macros can be used to assign a different value to a variable depending upon which system the code is compiled on.

```
#ifdef VMS

    op_sys_type = 'A';     /* VAX/VMS      */

#else   /* #ifdef VMS */

#ifdef VAX

    op_sys_type = 'B';     /* VAX ULTRIX   */

#else   /* #ifdef VAX */

    op_sys_type = 'C';     /* RISC ULTRIX */

#endif  /* #ifdef VAX */
#endif  /* #ifdef VMS */
```

You will notice that the following simple coding conventions were followed in the previous example:

- There was at least one blank line separating the pre-processor directives from any surrounding C code.
- The `#else` and `#endif` directives were accompanied by a comment which specified which `#ifdef` directive they are associated with.
- The code inside of each pre-processor block is a self contained unit.
- Each pre-processor block of code contains a comment indicating which systems this block of code is used on.

The need for these coding conventions can easily be seen by looking at the following code fragment which does not follow the rules.

```
    if (i>10)
    {
#ifdef VMS
        i += 2;
    }
    else
    {
        i += 1;
#else
        i += 1;
#ifdef VAX
        k = 0;
#else
        k = 1;
#endif
        u++;
#endif
        j = 2;
    }
```

In this example it is very difficult to follow the flow of control. This is because statements which affect how the CPU will execute the program are interspersed with statements which affect how the compiler will compile the code. In addition it is not very easy to see exactly which C statements will execute on each operating system because it is not immediately obvious which `#ifdef` directives are associated with which `#else` and `#endif` directives.

Program listing 1 shows how the code could be re-written in a manner which makes it much easier to understand.

Even when the code is written to our coding standards, an excessive use of the `#ifdef` directive can make the code difficult to understand. This is because it is difficult to follow two simultaneous flows of control. When you are reading code which contains many `#ifdef` statements, you must first try and figure out which C statements apply to the operating system on which you are trying to debug, then you must also try to understand the effect of the C statements themselves. For this reason, we adopted an informal guideline that a single C function should never contain more than three `#ifdef` directives. If a function requires more than three `#ifdef` directives, then different versions of the function should be written and the entire functions should be enclosed inside of `#ifdef` directives.

We find it very useful to use what we call *virtual functions* when we are trying to reduce the number of `#ifdef` directives in our code. Virtual function calls are so called because look like ordinary function calls in the body of the code, but they actually call a different function on each operating system. The use of virtual functions is easiest understood by looking at an example.

Our application frequently needs to search to see if a file exists that matches a particular template (e.g. to find out there any files ending in `.install_guide` in the directory containing the kit). As a result we developed a function called `find_file()` to accomplish this task. The VMS operating system comes with a system supplied routine called `lib$find_file` which does most of the work required by the `find_file()` routine. Unfortunately the ULTRIX operating system does not come with any equivalent routine and hence the VMS and ULTRIX implementations of this function are significantly different.

```
#ifdef VMS

    /*
     * Start of VMS Code
     */
    if (i>10)
    {
        i += 2;
    }
    else
    {
        i += 1;
        j = 2;
    }

#else     /* #ifdef VMS */

    /*
     * Start of ULTRIX Code
     */
    if (i>10)
    {
        i += 1;

#ifdef VAX

        k = 0;    /* VAX ULTRIX  */

#else     /* #ifdef VAX */

        k = 1;    /* RISC ULTRIX */

#endif    /* #ifdef VAX */

        u++;
        j = 2;
    }

#endif    /* #ifdef VMS */
```

Program 1: *Code re-written to be easier to understand*

According to our function naming conventions we named the two versions of the routine `vms_find_file()` and `ult_find_file()`. Because these routines are needed frequently throughout the our system, we could easily find the following code fragment being repeated several times in each of the source files:

```
#ifdef VMS

    status = vms_find_file (...);       /* VAX/VMS code */

#else   /* #ifdef VMS */

    status = ult_find_file (...);       /* ULTRIX code  */

#endif  /* #ifdef VMS */
```

The need for these repeated `#ifdef` directives can easily be eliminated by defining a virtual function `FIND_FILE()` to evaluate to either `vms_find_file()` or `ult_find_file()` depending upon the operating system for which we are compiling. A header file is created with the following lines:

```
#ifdef VMS

/*
 * VMS function definitions
 */
#define  FIND_FILE  vms_find_file‡
#define  GET_TIME   vms_get_time
 ....
 ....†

#else   /* #ifdef VMS */

/*
 * ULTRIX function definitions
 */
#define  FIND_FILE  ult_find_file‡
#define  GET_TIME   ult_get_time
 ....
 ....†

#endif  /* #ifdef VMS */
```

When this header file is included, the code fragment containing the `#ifdef` directive shown earlier can be replaced by the following simple line:

```
status = FIND_FILE (...);
```

5. Compiler Problems

For our example project we had to use three different C compilers. The system was initially developed on VAX/VMS using the VAX C compiler. An ULTRIX version of the VAX C compiler was used for the development of the ULTRIX VAX version. Since VAX C is not available for ULTRIX RISC, we had to use the MIPs C compiler for the ULTRIX RISC version of our system.

Luckily, we ran into relatively few problems caused by differences in the compilers. However, we had to limit ourselves to only use the simple features of the C language (which were the same across all three compilers). The following is a list of the compiler related problems we encountered and how we overcame these problems.

- All three compilers we use support function calls with variable length argument lists. However, there are differences between the compilers with regard to how these functions should be prototyped. In order to avoid compatibility problems, we did not include function prototypes for functions with variable length argument lists.
- According to the ANSI C standard function prototypes are optional. However, each different C compiler has a different way of handling functions which are not prototyped. In order to avoid any potential problems, we insisted that all functions (except for functions with variable length argument lists) should be prototyped in advance of being used.
- There are certain situations when you might want to pass the address of a pointer as a parameter to a function. However, some compilers will sometimes assume that the '&' character

† These dotted lines are to indicate that we defined several other *virtual functions* as well as just FIND_FILE.

‡ You will notice that we adopt the convention that virtual functions have names which are all in uppercase. This distinguishes then from normal function names which are in lowercase.

before the pointer variable was placed there in error and will issue a warning message that the '&' character is being ignored (it is a common programming mistake to place an '&' character before a pointer variable). In order to ensure that the compiler does not make this assumption it is necessary to explicitly declare a variable which is a pointer to a pointer; this variable can then be set to point to the desired pointer.

- There is a feature of the MIPS C compiler which causes it to sometimes "forget" the data type of some function call parameters. This bug occurs predominately in long source files. To overcome this bug, we had to place explicit type casts on the variables in question. This has no effect upon the compilation of the code with other compilers because we are explicitly casting the variables to the same type as they already are.
- The VAX C compiler supports a `globaldef` data type. Variables which are declared with the `globaldef` type are similar to normal C global variables. However, with the `globaldef` type, you can control where in the executable image the space is allocated for the variable. Unfortunately, the `globaldef` type is not supported by any other compiler. Although the original code contained several variables which were declared to be `globaldef` type variables, we found that they could all be replaced with normal global variables.

We initially planned to use the `lint` tool to check for potential portability problems in the source code. However, when we ran `lint` on the original source code we got such a huge amount of output that it was almost impossible to decipher. The use of `lint` could possibly have identified some problems with the code, however, the genuine portability warnings would have been difficult to find in the middle of pages of spurious warnings.

The VAX C compiler has a portability warning feature, when this feature is enabled the compiler will issue additional warning messages about potential portability problems in the source code. We found that enabling this feature enabled us to highlight many potential portability problems. The MIPS C compiler also enforces very strict type checking. In fact, we found several situations where some of our code would produce warnings when compiled with the MIPS C compiler but would pass through the `lint` checker without any errors.

Because of the fact that the compilers themselves were capable of detecting most of the potential portability problems, we decided not to use the `lint` checker.

6. Standards

Both of the operating systems on which we implemented our installation / licensing tool claim to implement open systems standards. However, this did not make it a trivial task to port our software from one system to the other. It is interesting to look at why it is such a major piece of work to port a software package from one standards compliant system to another.

The VMS system is capable of emulating most of the UNIX type system interfaces. In fact, there is now a package available which will provide an almost complete implementation of the POSIX [IEE88a] system interface standard running on the VMS operating system. Likewise the ULTRIX operating system is also capable of providing an almost per-

fect implementation of the POSIX system interface. If the developers of the initial system had restricted themselves to only using POSIX type system calls, the task of porting the software from one system to the other would have been relatively simple.

However, the developers of the original VMS version of the software, chose to use the native VMS operating system interface in preference to the POSIX interface for many functions. This choice was a fairly logical one because the VMS operating system interface offers many useful functions which are not part of the POSIX specification.

It was probably even more significant that the developers of the VMS version chose to use some tools which are not available on UNIX. For example, the VMS version uses the SMG (Screen Management Graphics) package and the VMS message compiler because both of these packages are available with VMS and perform a useful function. Since these packages are not available on ULTRIX, we chose to replace these in the ULTRIX version with the Curses screen management package and the message retrieval system provided as part of the ULTRIX internationalisation tools. The code which interfaced to these packages had to be completely re-written in order to be able to work with the different tools on ULTRIX.

In summary we found that while standard interfaces do provide some help when porting between some systems they are not much use unless the standard interfaces are also the preferred interfaces to each system. The standards would also be much more useful if they specified the interfaces to some of the tools which a programmer is likely to use (e.g. a screen management package) as well as the interface to the operating system itself.

We are aware that the POSIX working group is extending its scope quite considerably. However, even if all of the interfaces proposed by the POSIX group were available on both systems, there could still be significant differences between the two environments.

There are several efforts underway to come up with a much wider set of standard interfaces. Digital's Network Architecture Standard (NAS) [DEC90b], HP's New Wave and OSF's Distributed Computing Environment (DCE) / Distributed Managemeent Environment (DME) [OSF90a, OSF91b] are all examples of the work that is being done. However, none of these has yet to gain the widespread acceptance and/or implementation which would make them de facto standards. Unfortunately, it seems that the task of porting between heterogeneous operating systems will remain difficult for the near future.

7. Future

7.1. Other Systems

VMS and ULTRIX are two significantly different operating systems. Therefore, the problems that we encountered in getting our software to work on both ULTRIX and VMS type systems are illustrative of the type of problems that would be encountered in developing software for any set of truly heterogeneous operating systems.

However, VMS and ULTRIX are much more similar to each other than they are to systems such as MS-DOS. For example, MS-DOS does not support multiple processes or virtual memory. Hence, it will probably

be much more difficult for us to port our application to a single-user system such as MS-DOS.

7.2. Tools

It is obviously desirable that all of the development tools you use should be available on all the platforms you develop on. However, we found that it is absolutely essential that the central code library should be accessible from all of the platforms that are used for development. In our project we had to use our own home grown solution. Hopefully a production quality distributed / heterogeneous version management system will become available soon.

8. Lessons

The following is a list of the main lessons that we learned from porting our software from VMS to ULTRIX:

- It is relatively easy to write code that does the same thing on different systems; the problem is that in many cases you only achieve the same higher level function by doing very different things on each operating system. The challenge is to write one piece of code that achieves a different implementation depending upon the platform.
- The `#ifdef` statement is very useful. However, you must be very careful about when and how you use the `#ifdef` statement, because it can easily destroy the understandability of the source code.
- Although standards help make porting a little easier, they should be enhanced so that many more features are standardized. It is common for compilers and other tools to offer additional features which are not defined by the standards. However, if you are developing software on more than one platform you will probably not be able to use these additional features because they wont be available on all platforms.
- When you are developing software for more than one platform, you should try to use tools that are available on both platforms. This is because developers involved in porting work have to be familiar with the development tools on all of the development platforms. A significant overhead was placed on our project by the fact that, many of the development tools we use are not available on both systems and hence the developers had top learn how to use both sets of tools.
- As well as the problems of initially porting the code, we also face the problem that portable code is more difficult to maintain than code than code that only runs on one platform. This is because any change to the portable code must also be tested on each platform.

9. Acknowledgments

I would like to acknowledge the assistance of Jim Hutton who reviewed a preliminary draft of this paper and provided some useful feedback.

References

[DEC87a] DEC, *Guide to the VAX DEC/Code Management System,* Digital Equipment Corporation (April 1987).

[DEC90a] DEC, *VMS/Ultrix Connection User's Guide,* Digital Equipment Corporation (September 1990).

[DEC90b] DEC, *NAS Handbook: Developing Applications in a Multi-vendor Environment,* Digital Equipment Corporation (1990).

[IEE88a] IEEE, *IEEE Std 1003.1-1988: Portable Operating System Interface for Computer Environments (POSIX),* Institute of Electrical and Electronic Engineers (1988).

[O'D90a] B. O'Donovan and J. Grimson, "Development of a Distributed Revision Control System," pp. 207-214 in *Proc. of the Summer 1990 UKUUG Conference*, London (July 1990).

[O'D90b] B. O'Donovan and J. Grimson, "A Distributed Version Controld System for Wide Area Networks," *Software Engineering Journal* **5**(5), pp. 255-262 (September 1990).

[OSF90a] OSF, *OSF Distributed Computing Environment Rationale,* Open Software Foundataion (1990).

[OSF91a] OSF, *OSF Architecture-Neutral Distribution Format Rationale,* Open Software Foundation (June 1991).

[OSF91b] OSF, *OSF Distributed Management Environment Rationale,* Open Software Foundataion (September 1991).

A Health Information System based on UNIX–Client–Server called PHOENIX

Dr. Reinhard Koller

Amt der O Landesregierung
Austria

Abstract

PHOENIX stands for **P**roject **HO**spital **E**nvironment **UNIX** and is the logo for a project organized by the Government of Upper Austria for 16 hospitals. The project started at the beginning of 1990 and phase I will take 3 years. Phase II will start in 1993.

The project is completely based on the requirements for portability, 4GL Tools, relational databases and standards.

So the system environment consists of:

- Motorola's 88100 Dual RISC Server
- UNIX System V.3
- TCP/IP and NFS
- Informix-Database, Informix-4GL
- Uniplex 7

At the moment the application software includes a patient care system (evidence and administration of patients), office automation, a staff information system, a laboratory and a pathology information system.

In this paper I will tell you about our relevant criterions we have considered in building these complex and large (three of our systems will consist of about 200 terminals each) information systems. I will show which strategic requirements and measures are necessary so that based on this concept the information systems can be enlarged and extended for further applications and necessities and the costs can be reduced by "downsizing" and using client-server models.

This is especially considered under the aspect of the importance and influence of standards, connectivity and multivendorship.

I will finish by giving you an outlook of our plannings for phase II starting in the Spring of 1993.

1. Start of the Project

The project "PHOENIX" started at the beginning of 1990 and phase I will take 3 years. The project is organized by the "Government of Upper Austria" and concerns 16 hospitals. The largest of these hospitals will have about 200 terminals each.

The project was completely based on the requirements for portability, 4GL Tools, relational databases and standards.

Not only lower costs but also a lot of strategic aims – as described below – led to the decision for "Open Systems". The first time we had a lot of problems, especially in establishing and configuring our first servers (some system software products had too many bugs for working, there was no correct spooling for printing available, the first servers based on Motorola's 68030 processors became very slow although there were only 10 people working on 10 terminals).

But as our team became more experienced and had practise in UNIX and the products we are working with, most of our problems faded and as we were convinced to be on the right way we took a lot of efforts to solve all our troubles and problems.

In Summer 1991 we revised our concept in favour of RISC processors and some new system software products which have a better connectivity and better performance in OLTP.

2. System Environment

2.1. Hardware

- Motorola's 88100 Dual RISC CPU with 64 to 256 MB memory per server. Up to 3 servers per system (one database server and application servers)
- Disks: from 2.0 GB up to as many as necessary
- ExaTape (2.3 GB) for data backup plus Streamer (150 MB) for "Logging"
- Modems. "Schoeller Octocam" for remote support (one modem for each server), plus

  "Trailblazer" for fast connections to other systems (19200 Bd) supporting the MNP-5 protocol (one modem for each system)
- Spider Systems, Spider Ports, Spider Router, supporting both TCP/IP and OSI

2.2. System Software

2.2.1. Operating System

- UNIX System V.3 (based on Standard AT&T UNIX)
- NFS from SUN
- NSE (Network Services Extensions) as additional tools
- TCP/IP as network protocol (but most of our LAN products not only support TCP/IP but also OSI)

2.2.2. Programming Equipment

- INFORMIX-4GL RDS as front end
- C (based on ANSII X3J11) in special cases
- INFORMIX-ONLINE 4.1 as backend supporting BLOBS

2.2.3. Database

- INFORMIX-ONLINE 4.1
- INFORMIX-SQL based on ANSI-SQL
- DATA LINK as connection to UNIPLEX and in future to WINGZ
- INFORMIX-STAR and INFORMIX-NET supporting distributed databases (patient database, laboratory database, patho database)

2.2.4. Office Automation

- UNIPLEX 7 supporting both ASCII and X-Terminals (word processing, mailing, spreadsheet, graphics)
- (WINGZ) probably in future for calculating and graphics, as WINGZ supports DATALINK for direct access to INFORMIX and UNIPLEX data
- FRAMEMAKER for designing documents, manuscripts and overheads

2.2.5. Tools

- REFLECTION for VT-220 emulation and file transfer as frontend product
- SMB-SERVER as backend product which also offers the possibility of local printing
- SOFT-PC for running DOS and MAC applications under UNIX without having PCs
- OSF/MOTIF as Graphical User Interface (GUI) running FRAMEMAKER, SOFT-PC and WINGZ

3. Application Software

At the moment the application software consists of the following parts:

- Patient Information System
  - Evidence of patients
  - Accounting
  - Registration of medical services
- Administration System
  - Bookkeeping
  - Purchasing
  - Cost accounting
  - Management information system
- Staff Information System
- Office Automation
  - Integrated text processing (medical letters)
  - Controlling
  - Individual reports and graphics
- Laboratory System (Technical and administrative support)

  Supporting both mono and bidirectional laboratory equipment and having a direct connection to the Patient Information System with the help of INFORMIX-STAR

4. Strategic Aims

4.1. Portability of Applications

The whole application software is designed with the CASE Tool "MAESTRO II" by considering the specifications of the X/Open Portability Guide (issue 3). All parts that aren't defined by the XPG yet are isolated and enclosed in functions so that they can be replaced by standardized functions as soon as these parts are considered in a further issue of the XPG.

The complete software is written under UNIX System V.3 in INFORMIX 4GL and C according to ANSI X3J11. But only the pathology information system and some technical parts of the laboratory information system are written in C. The data is stored in the relational database of INFORMIX including the standardized interface of ANSI-SQL. The backend tool on the database server is INFORMIX-ONLINE.

The multivendor office automation software UNIPLEX 7 has a direct connection to INFORMIX-SQL, so that data can easily be transferred from 4GL applications to text processing.

4.2. Openness of the Information System

By considering system independent standards (OSI as far as available, TCP/IP, ANSI, X/Open's XPG, SQL, UNIX System V.3) we provide the basis for enlargening and extending the hospital information system for further applications. This is especially necessary in the fields of medical applications, where we have started with the laboratory and pathology information system by now.

But enormous demands are coming from the fields of radiology, where pictures should be administrated, transferred and stored. Moreover, using INFORMIX-ONLINE we have the possibilities of using new types of interfaces (the so called BLOBs) – voice, video and special graphics in databases – that users are demanding more and more.

Beyond that we are looking at the possibility of connecting our sixteen systems to one wide area network using X.25 so we can transfer data about our patients from one hospital to another if necessary. This is very interesting for certain medical checkups being made in some special hospital.

4.3. Homogeneous "Look and Feel" of all Applications

We made our decision in favor of INFORMIX and UNIPLEX because of the following reasons:

- Both INFORMIX and UNIPLEX are well known and widespread products in the world of UNIX; so support and further innovations are quite good
- The communication between INFORMIX and UNIPLEX is very good done by INFORMIX-NET and UNIPLEX DATA LINK supporting ASCII text files and SQL
- Both products are available under UNIX and DOS using the same user interface under UNIX and DOS; by now we are using the character based interface with the exception of graphics on some X-terminals

- This homogeneous "look and feel" increases productivity and decreases the nontechnical user's fear of the computer because the user has to learn only "one system"
- There is no difference for the user whether she/he is working on the personal computer or as a UNIX terminal

4.4. Reducing the Costs by "Downsizing"

In our largest hospitals in Linz and Steyr we would have to install a mainframe (for about 200 to 300 terminals in each system).

Instead of this we are using a so called UNIX-UNIX Client-Server concept, installing one UNIX database server (192 MB of memory) running INFORMIX-ONLINE and INFORMIX-STAR and two UNIX application servers (128 MB of memory each server) running INFORMIX-SQL, INFORMIX-4GL, INFORMIX-NET and UNIPLEX 7.

Looking at the costs of hardware and licences this is far cheaper than having one large mainframe. Furthermore, expansion of the system can be done by installing a third or if necessary a fourth UNIX application server in the network. So all previously installed hardware (database server and application servers) can be used in future.

Using personal computers or rather X-terminals in the fields of office automation (e.g. for text processing, calculating and graphics) a lot of work is being transferred from the X-servers to the personal computers.

5. Most Important Experiences

- Necessity of gaining own know how in UNIX, INFORMIX and UNIPLEX
- Revising one year old concepts because of the great innovation in the field of Open Systems
- Spending a lot of time in order to get familiar with the enormous possibilities and parameters of INFORMIX and UNIPLEX

  Otherwise you will only use a small part of the product and you will have a lot of problems with the performance of your database if you haven't studied the TB-monitor carefully.
- Having more confidence in Open Systems and especially in UNIX, because they are much better than their reputation (in commercial projects)

  And by now there are almost as many standard applications available for UNIX as for DOS and the prizing is quite good.

6. Outlook

6.1. Radiology System

At the moment a concept is being made for a so called PACS (Picture Archiving and Communication System) and a RIS (Radiology Information System) based on FDDI.

6.2. Multi-Media

Both the PACS and the RIS require the new picture data types supported by INFORMIX-ONLINE, the so called BLOBS. With the help of this data type we hope that "pictures" not only can be stored in the database but also the communication between the technical and the administrative part of this application can be handled.

The Portability of GNU Software

Joseph Arceneaux

The Free Software Foundation
San Francisco, CA, USA
jla@ai.mit.edu

Michael Tiemann D. V. Henkel-Wallace

Cygnus Support
Palo Alto, CA, USA
{ tiemann | gumby }@cygnus.com

Abstract

In June of 1987, GNU C version 1.0 was released by the Free Software Foundation. Since that time, it has been ported to about 50 host environments and generates code for about 20 machine architectures. In many cases, GNU C was ported to new platforms by volunteers, working with incomplete information, more rapidly than vendor-sponsored ports of AT&T's "portable C compiler".

This talk will describe 4 portability case studies based on well-known GNU software packages: Emacs, GDB, GCC, and BFD. The case studies will present the initial design specifications, the evolution of the design over time, factors that influenced (or mandated) changes to the package, and experience in applying the lessons learned in the design of subsequent programs.

Introduction

Why is GNU software so portable? The answer is because it's free software.

The current official distribution of GNU Emacs, version 18.58, runs on 60 different architectures and 26 different systems, including many types of UNIX as well as VMS and MS-DOS. The GNU C compiler has been ported to everything from a 1-bit DSP to 64-bit super-scalar processors. The GNU debugger works in an enormous number of environ-

ments, both natively and in cross-debugging modes that go "down to the bare iron." Many other GNU programs run not only under these systems, but on the Atari Amiga systems, and even IBM MVS and DOS.

Achieving this versatility required a large amount of expertise and cost: knowledge of many systems was necessary, and hundreds of thousands of dollars worth of hardware was needed to test these ports. However, the GNU project, a relatively small team, has been able to do this work with significantly smaller resources, and with much greater success than most corporations achieve. This paper explains how.

It is important to understand that the primary reason for the technical success of GNU software is its copy-able status. It's not good software that happens to be free; it's good software *because* it's free software.

Free Software's Special Advantages

The free status has given us two levers on the portability of the code. First, since the code is freely redistributable, enormous numbers of people have worked on the code, fixing bugs, adding features, and porting it. Second, because it's intended for frequent reuse, the software has been designed from the ground up for portability.

GNU's "parallel processing" approach is superior to traditional, proprietary models. The people who port GNU to new platforms are highly motivated to have high-quality, useful tools, since they themselves become the users. In addition, there has been a much tighter fit between tool and marketplace: nobody has spent effort porting to a platform on which the tool was unwanted.

This structure has also kept the tools resilient. Each tool has a single maintainer, who is the arbiter of changes. This forces style consistency and reliability (de-stabilizing changes don't go into the tool). On the other hand, although this arbiter is frequently a developer, maintainers also receive changes from others, changes which have already proven successful.

Alone, this sounds just like any other development plan. However, the arbiter is not restricted by any GNU policy; essentially she operates by consensus. When necessary, other foci of development may appear, and later disappear. For instance, when our compiler was ported to run under MS-DOS, it was maintained separately from the "mainline" sources until it was stable. In that time many people used and contributed to that "tributary" of the tool. After it had been in regular use for some time, the two groups decided that the DOS port had sufficient stability and sufficiently widespread usage to warrant convergence. This method allowed both groups to move forward at their own pace, neither pulling the other back, and yet eventually allowed each to take advantage of the other's work.

The rest of this paper discusses approaches used within Project GNU to produce portable software. We identify several important issues in this process, including deciding what to port, writing portable code, making installation a portable process, and achieving retargetability.

As examples, we will look at four components which demonstrate the breadth of approaches used by the GNU Project: the GNU Emacs text editor (`emacs`), the GNU Debugger (`gdb`), the GNU C compiler (`gcc`), and the Binary File Descriptor library (`bfd`). These programs have been chosen to illustrate the portability of GNU software because:

- Each has been ported to more than a dozen platforms.
- Each involved a non-trivial amount of development.
- Each shows innovation in its approach to portability.
- Each continues to be ported to new platforms.

A Few Explanations

We will begin by explaining a few terms. When we say *free software*, we are referring to the the freedom of any user to copy, to give away, to learn from, and to modify or improve the software. This freedom is a primary reason for the wide distribution of GNU software.

Portability of software is impeded by the differences in the hardware and software of different systems: CPUs, memory organization, operating systems, signals provided by an OS, libraries provided, filesystem organization, the terminal interface, etc. When we say that two particular platforms are different, we mean that the two systems differ in one or more of these respects.

An ideal in portability might be a universal host, able to dynamically emulate various machines and thus run many different binaries simultaneously. Although the BFD library moves in that direction, we will say that *portable software* means software which requires little or no modification to run on heterogeneous platforms with no meaningful differences in behavior.

Deciding What to Port

The *modus operandi* of Project GNU involves extensive participation by the user community. Many tasks normally associated with a traditional software vendor are performed by enthusiastic users of GNU software, including hand-holding and bug fixes. This cooperative effort includes contributions of ports to various machines as well as development and maintenance of entire programs. For example, the vast majority of Emacs ports were done by users.

An important part of keeping our software portable is deciding which ports to integrate into a distribution. Adding any significant amount of code will add to the complexity of the software. Therefore, an important criterion for this decision is the extent to which the port will benefit users. If the port is only for a very little-used system, then it may not be worth the trouble and increased code to implement. In such cases the authors of the port frequently distribute their code themselves through networks, floppies, BBSs, tapes, etc., to interested users who may then further redistribute it.

The other major factor affecting the decision of whether to integrate a port is the impact the integration will have on the code. If the contribution is too poorly coded, or is for a bizarre system which would require significant warping of the existing program structure, then it may not be worth it. The effect of such code is not merely immediate, but may also add subtle bugs which only surface later. Furthermore, useful free software will spend most of its life in maintenance mode, and many programmers may work on it. This makes the clarity of the code very important.

Sometimes it becomes apparent that adding code for many ports is not consistent with the design of a program, and that it may be better to

rewrite some aspects of the program. An example is the rewrite of GDB and other binary tools which enabled them to use the new BFD interface. However, it is preferable to avoid such situations and, if possible, design the software to be portable from the beginning. GCC, for example, was designed to use an abstract CPU model.

Another way of making programs very portable is to first implement a machine-independent language, then write most of the system in it. This can also make it easier to extend or customize the program. Emacs Lisp is an example of this technique.

Writing Portable Code

GNU is coded mostly in C, with the exception of various extension languages such as Emacs Lisp or GCC's language for target machine descriptions. C was chosen for several reasons:

- C is a powerful language which provides support for systems programming.
- C is easily implemented across a diversity of architectures and operating systems.
- C is the system language of almost all varieties of UNIX. Coding in C meant that users could immediately use GNU programs.
- Many user programs not distributed as part of GNU would still run under GNU with little or no modification. Local user modifications would be much easier to install under GNU.

In general, using C minimized the difficulties in moving from proprietary versions of UNIX to GNU. Similarly, we chose to emulate the Bourne shell (`/bin/sh`) with BASH (the **B**ourne **A**gain **SH**ell), despite considerations of designing a better shell language.

Having chosen C, we sought to avoid some of the pitfalls associated with traditional UNIX programming. One of the most widely quoted rules of our programming standards is *no arbitrary limits*. This means no static tables or fixed size strings. From [Fou92a]:

> Avoid arbitrary limits on the length or number of *any* data structure, including filenames, lines, files, and symbols, by allocating all data structures dynamically. In most UNIX utilities, "long lines are silently truncated". This is not acceptable in a GNU utility.

Applying this principle makes for more robust and much friendlier programs.

One problem occasionally encountered by GNU programmers is the temptation to use GNU extensions to the C language. This frequently makes the code much cleaner, and in some cases, such as the GNU extended `asm` facility, these extensions can actually make a program more portable. For systems not using GCC it means the program won't run unless someone modifies it. It is often easier, however, to port GCC than to modify the precocious program.

Sometimes it is possible to code in such a way that an extension is conditionally used, depending on the compiler, without breaking anything. The GNU `obstack` facility is coded in this way. In general, however, our policy is to avoid such extensions in large, established programs like Emacs, which run on a great variety of systems, and in GCC, because as part of its bootstrap process it must be compiled with other

compilers. Making GCC easy to port makes the rest of GNU, extensions and all, easy to port.

Another principle we employ is to avoid using low-level interfaces to possibly system-dependent data structures or functions. In general, it is much better to use a high-level abstraction (writing one if it doesn't already exist) and encode the low-level details in libraries or conditionally compiled files.

In general, we attempt to make our code upwardly compatible with Berkeley UNIX. If ANSI C specifies certain behavior, we try to be upwardly compatible with it, and likewise with POSIX. When standards conflict, we attempt to offer compatibility modes for each.

This guideline does not restrict us from implementing extensions prohibited by ANSI or POSIX if we feel they are superior (for example, our C extensions). In this case we provide a `-ansi` or `-compatibility` option to turn them off. We do however, avoid extensions which would break any existing programs or scripts.

Architectural Impediments to Portability

The two main hardware features impacting portability are the processor type and the memory organization. For most GNU programs, the first is dealt with by re-compilation, but there are exceptions. Some GNU programs, such as Emacs and GCC, contain assembly code. Although versions (or substitutes) of such code are provided for supported processors, there can be problems like that of `alloca()` for 68K machines: this function, which increments the stack pointer to acquire space, failed on systems whose compilers accessed the stack through `sp` rather than `fp`.

Memory organization can have a devastating effect on the portability of software. GNU is not intended to support 16 bit machines (although many of our smaller programs do run on 16 bit PCs), but with the growing number of 64 bit machines available, differing sizes of data remain a major source of portability bugs. Thus, the GNU programming principle of "no arbitrary limits" applies to pointer size as well as to array and string sizes.

Another problem presented by differing memory organizations is illustrated by the following code fragment:

```
int c;

while ((c = getchar ()) != EOF)
  write (file_descriptor, &c, 1);
```

It is very important to remember that some machines (for example, the 68K) are big-endian and thus this code will not work (although it will work on little-endian machines). Programs which do things like use bit-fields overlayed on signed numbers must allow for both types of memory. This can be cleanly handled with conditionally defined macros, like those used in Emacs.

Working Around Operating Systems

The greatest amount of diversity to be dealt with in making software portable is that found in operating systems. However, even completely different systems can be dealt with, especially if the software is designed portably from the beginning.

Large divergences between systems may dictate that certain functionality must be optional, dependent upon the system (for example, the subprocess functions of Emacs are not available under VMS), but that need not hinder the main purpose of the program. As long as the upper levels are designed with a sufficiently abstract interface to the parts below, moving code between systems need not be excessively difficult.

Designing for generality can also yield much simpler and more understandable code. For example, designing a simple I/O model, rather than depending on particular system calls, may yield a cleaner structure. It is very difficult, using `#ifdef`, to properly structure the multitude of `ioctl` and `fcntl` calls available on various systems. Also, various functions are frequently present on one system and absent on others. Supplying such functions along with the program ensure that they will never be missing (such functions supplied with GNU programs are frequently superior to those of the system).

Some of the dramatically varying aspects of operating systems that have given us difficulty are:

- Different executable file formats.
- Collisions between header files.
- Different styles of signal handling, and signal semantics.
- Differing filename conventions for different systems.
- Different bugs in different versions of the same system.
- Gratuitous extensions to "standard" features.
- Different window systems.
- Poor or absent documentation for obscure but important details of the system.

While many of these problems can be eliminated with techniques described thus far, we have made other choices in some cases. While we believe it is possible to design an abstract interface to window systems, we have adopted the X window system because it is free and an ad-hoc standard which can be assumed to run a most systems. Using X also means we don't have to build our own window system. Thus, new versions of Emacs, for example, can be expected to provide increased support for X.

Building and Installing Portably

As well as being portable, software ought to *install* easily, with only minor modifications needed to handle the differences between sites. Over the years, several strategies have been used to approach this problem. One of the most entertaining (`configure`, part of the `rn` news reader) was frequently referred to as "the Larry Wall show," since the script printed interesting messages while it explored the system, trying to determine its persuasion.

GNU software has used a variety of approaches to this problem. BASH depends on the C preprocessor to define the system and architecture, and then configures its `Makefile` from there. This is a clean approach which usually works well, but which fails when `cpp` doesn't define enough symbols to uniquely identify the system. So BASH now uses a shell script in combination with the C preprocessor to determine the system type.

Some GNU programs have recently begun to use `configure`, a system of shell scripts which includes all CPU and system types recog-

nized by GNU software. This system will automatically build a `Makefile` and/or C header files tailored to a variety of specifications, including cross environments. For example, the following command:

```
configure sun4 +target=a29k
```

will configure the program such that, when built in a Sun 4 environment, it will produce tools intended for an `a29k`. Here is another example:

```
configure sun4 +ansi +destdir=/usr/local/bin +target=sun3
```

This configures a software package for building under a Sun 4 environment, to be installed in `/usr/local/bin`, and executable on a Sun 3. Furthermore, the executable will expect conforming ANSI C source.

We plan to eventually use `configure` with all of our software.

Case Studies

GNU Emacs

GNU Emacs may be the most portable large system of Project GNU because it runs on more platforms than any other. However, the impact of making it so portable has not had a uniformly desirable impact on the code; Emacs embodies both the best and the worst of portability. We will concentrate on the positive aspects.

How Portable Is It?

GNU Emacs provides one of the most consistent programming environments available on today's computers. A user can leave an Emacs session on a Sun workstation, walk across the hall to a VAX running VMS, and encounter almost no difference running Emacs there. The screens look the same and the commands work the same way. (With the exception, under VMS, of filenames. Strictly speaking, this is also true of UNIX since some older versions don't support long filenames). Personal extensions written by the user also look the same, and furthermore, work without modification on any other platform running GNU Emacs. Users can also expect Emacs to appear similar, if not exactly the same, on a plethora of terminals.

How This Happened

A major force behind this consistency was the early design choice of providing Lisp for the development of editor functionality. Thus, most of the C code for Emacs is concerned with supporting the Lisp world, including defining primitive Lisp functions. This means that the vast majority of extensions to the editor can be written without fear of the dialectal terrors of the local C compiler. Emacs users routinely write vast amounts of sophisticated editing code which works perfectly, with no modification, on wildly different platforms.

While all Lisp code related exclusively to editing is system independent, a few functions are defined differently, depending on the host operating system. Here is an example from Emacs' directory editor `dired`:

```
(if (eq system-type 'vax-vms)
    (vms-read-directory dirname dired-listing-switches buffer)
  (if (file-directory-p dirname)
      (call-process "ls" nil buffer nil
                    dired-listing-switches dirname)
    ...
```

On UNIX systems, `vms-read-directory` is an unbound Lisp symbol because it is defined in a file which is only compiled under VMS. Such conditional compilation also applies to various functions not available on all versions of UNIX.

Emacs also looks either similar or exactly the same on many different terminals, regardless of platform. This is achieved by providing an abstract interface to the display code. Thus the terminal driver can call the function `delete_chars (n)` and expect that, whether or not the terminal supports multi-character deletion, the right thing will happen. Likewise, it matters not to Emacs if the operating system worships `termcap` or `terminfo`; both religions are supported.

To be efficient, Emacs' redisplay must take into consideration the different capabilities and performance of various terminals. This is done by encoding these parameters into a single set of tables which is used by redisplay. Thus, the display algorithms make reference only to these cost tables, and do not refer to specific terminals.

While this level of Emacs is thus fairly system independent, the many differences among systems must appear somewhere, and most are in a file called `sysdep.c`. Here the age and impact of Emacs' multitude of ports can be seen – this file seems to contain almost as many preprocessor directives as C statements. While it may have been much cleaner to separate various sections into different files supporting a few highly varying models, contributors provide `diff` listings; it takes the maintainers much less time to simply to merge those in. Eventually we would like to re-do this, possibly after the first release of our kernel.

More Internal Matters

Early versions of Emacs supported two implementations of Lisp objects: one using a `struct`, and another using an `int`. For the former it was important to know the *endianness* of the machine so that the pointer and type parts of the structure could be properly laid out in memory. For the integer implementation this is irrelevant, as the compiler handles this machine dependency; macros for extraction of the pointer or type merely specify mask and shift operations.

Regardless of the implementation used, all accesses are done with macros so that the following sorts of comparisons may be performed:

```
/* From definition of window-p */
return XTYPE (obj) == Lisp_Window ? Qt : Qnil;
```

Such an implementation has proved quite portable across a broad spectrum of machines. There are a few machines, such as the IBM PC-RT, which require minor adjustments to these macros, but this is rare. While the decision to use Lisp greatly extended the power and flexibility of Emacs, UNIX poses a problem to such implementations. The problem is that all the Lisp code must go into the data segment, and thus is not sharable (on systems which support shared text segments). The solution adopted was to re-map this data into the read-only text segment. Getting this to work can be the most difficult part of an Emacs port. Nevertheless, Emacs does indeed achieve this on most systems which support such an operation, and making this happen on a new system is generally a matter of getting the `#define`'s right.

All such definitions and macros used by Emacs can generally be divided into those determined by hardware (for example, whether integers must be sign extended or not) and those specified by the operating system (such as the provision of sockets, or whether subprocesses are supported). Thus they are divided into two files, a machine description and a system description. Occasionally such a file overrides a standard Emacs definition (such as how to extract an integer from a Lisp object).

One of each of these files (e.g., `s-sunos4.h` and `m-sparc.h`) are specified in another include file, `config.h` which is included in all Emacs C files. `config.h` is also included in the `Makefile` template. Thus, the machine and system descriptions also specify the build process. While this system has worked quite well, we expect to use the `configure` system in future versions of Emacs.

The GNU Debugger

GNU Emacs is a sophisticated program which continues to grow more powerful (and not at all smaller). Between versions 16 and 17 it became impossible to debug Emacs with the standard Berkeley debugger (`dbx`) because its fixed-size symbol table and other built-in limits were no longer sufficient for Emacs. Another problem was the implementation of preemptable redisplay using UNIX signals, something `dbx` would never understand. Furthermore, `dbx` source was not provided by many vendors, and support for these platforms was not present in the Berkeley distribution. A new debugger was needed.

Although GDB has a user interface similar to that of `dbx`, it was designed to be portable. This was important because a goal for GDB was that it be distributed with Emacs, so that Emacs could be ported and debugged on a multitude of platforms.

Designing a Portable Debugger

GDB has been through four major revisions since 1985. The initial design, done for the VAX, specified the following modules: a symbol file reader, an inferior process manager, an expression evaluator, a command interpreter, a terminal handler, and machine-dependent routines. The separation of components into distinct logical units proved to be of key importance in making GDB portable.

One of the more interesting aspects of GDB was the implementation of the machine-dependent interface. A number of functions in the GDB program were simple encapsulations of macros as shown in Figure 1.

By specifying a macro interface to the lowest-level internals of the machine interface and a function interface on top of that, the tasks of porting and adding functionality to the debugger were cleanly separated, so each could be pursued independently, without impacting the other.

Evolution of the Design

GDB version 2 supported the Sun 3 workstation in addition to the VAX. The Sun 3 is different architecturally from the VAX in some important ways (especially byte order), but also has important similarities with that machine: both use a stack for parameter passing, both run flavors of UNIX which support roughly the same symbol table format, and neither architecture uses a long, exposed instruction pipeline. The design of GDB was refined so that all the necessary changes could be kept local to the machine-dependent files of the debugger.

```
/* Record that register REGNO contains VAL.
   This is used when the value is obtained from the inferior or core dump,
   so there is no need to store the value there.  */

void
supply_register (regno, val)
     int regno;
     char *val;
{
   register_valid[regno] = 1;
   bcopy (val, &registers[REGISTER_BYTE (regno)], REGISTER_RAW_SIZE (regno));
}

CORE_ADDR
read_pc ()
{
   return (CORE_ADDR) read_register (PC_REGNUM);
}
```

Figure 1: *Code fragment from GDB*

When GDB was ported to the SPARC architecture, serious rewriting was required because parameters could be passed in registers, register windows could act as stack caches, `call` instructions placed the return address in a register instead of on the stack, and control transfer instructions (branches and calls) could execute instructions in *delay slots*.

There were two ways to deal with these new problems: implement a more specific solution for the SPARC, or generalize the components of GDB to deal with the SPARC. The first approach seemed to be the more expedient, but would have meant cluttering up the implementation with SPARC-specific `#ifdef`'s. Instead, the second approach was used, yielding a much more powerful design: once the SPARC port had been completed, GDB was ported to several other RISC architectures without great difficulty.

GDB also began to serve as a *cross* debugger. Many embedded systems are one-of-a-kind machines consisting of custom hardware and software. An (unfortunate) property of these systems is the lack of a common communications interface between host and target systems. So many developers used GDB, modifying the original remote communications mechanism to support their custom communications protocol, and then considered their problems solved. Unfortunately, every such instance meant another version of GDB.

This same story was repeated with respect to object file formats: one of the most painful dilemmas of using UNIX System V is the choice between `adb` and `sdb`. By replacing the file that read `a.out` file format with one that could read COFF, GDB could be ported to these systems with minimal effort.

When the GDB maintainers announced the coalescence of these versions into a new release, there was an unexpected flood of responses from the user community: over 130 developers had "customized" GDB to support their remote communications files and/or object file formats. In effect, these developers individually felt that a point had been reached where GDB had gained all that it would from being freely redistributable, and that the time had come to only consider making local modifications and enhancements.

Portability versus Maintainability

GDB was designed to be a portable debugger. It was so portable that almost any user could port it, and almost every user did, making it virtually un-maintainable. GDB version 4 was designed to retain the positive aspects of portability, but also to support maintainability. A major component of this design was the BFD library.

BFD

The BFD, or *Binary File Descriptor* library, provides a high-level interface to a variety of different object file (or executable) formats. Currently it runs on a dozen different systems and supports 19 target systems.

The BFD library differs in approach from programs such as GCC or Emacs. First, rather than trying to fit many systems into one model (for example, trying to treat all systems generally like BSD and patching the differences) it defines a new abstraction to solve its portability problems. And second, it can be tailored *at runtime* to certain aspects of multi-platform operation.

Purpose and Functionality

BFD provides a high-level interface to the manipulation of object files (commonly referred to as "point-ohs" or "dot-oh files.") At the time it was started in early 1990, few of the GNU tools other than Emacs ran on systems that weren't substantially derived from BSD. We recognized that there would be a need for COFF (and eventually ELF) support, and probably other formats as well.

Rather than trying to write what would essentially be parallel implementations of many tools, we decided to provide an interface of unstructured objects at the level of symbols, sections, and relocations. This has lead to a greater amount of code reuse than was possible before.

A New Abstraction

Earlier GNU tools were tightly tailored to the Berkeley `a.out` representation. Operations were performed in the order that their data appeared in files; the file structure was apparent in the program structure.

BFD uses a different paradigm, that of considering what operations are meaningful on any object file. An object file is represented in core by an object called a `bfd`. All operations are performed via function calls upon this `bfd`. The set of operations is small; it includes operations such as creating sections, performing relocation, reading section contents, searching for symbols, and reading data (in order to get the byte ordering correct).

We then implemented the `strip` program for `a.out` using these operations. Using concrete applications from the start helped us find holes in the design. Once we had "ported" several applications to BFD, we then produced another back-end, this time for COFF. When operations that we hadn't represented were needed, we added them.

Runtime Behavior

Although BFD must be configured at compile-time for the *host* on which it runs, it needs no compile-time knowledge of which object file format will be used with it. That is selected at runtime through a transfer vector (or branch table). Implementation in C was straightforward, although the bookkeeping was a bit painful.

The interface to an object file format is through a structure that looks like this:

```
struct bfd_target
{
   char *name;
   boolean byteorder_big_p;
   struct bfd_target *(*_bfd_check_format) (bfd *);
};
```

Note that the name is uninterpreted; it is only used to identify a format (referred to as the "target", since "format" could also be construed as "one of archive, core, or object"). Also, in the actual source we use a macro for function definitions so that prototypes are defined for ANSI compilers, and not for others.

Since the format is stored in the `bfd` structure itself, it's simple to create a method-dispatching macro which indexes through this target to call the correct method. This looks like:

```
bfd_send (some_bfd, message_name, (args));.
```

Implementation Implications

Porting BFD has proved to be simpler than we originally expected. Some `#ifdef`'s were still required, but since BFD doesn't use signals, or other greatly varying parts of UNIX, these are minimized.†

Another reason BFD is easy to port is that the interface is precisely described by the branch vector of interface functions. This reduces porting to going through the list of operations enumerated in the vector and writing an implementation for each. As these functions are completely modular, it's a simple matter to use code from existing ports.

One problem with writing retargetable code is that different systems may require different kinds of bookkeeping. This is especially a problem for BFD, where any special book-keeping variables must exist at runtime in all implementations. Furthermore, because multiple `bfd` structures are frequently instantiated, possibly of different types, the back-end programmer might have to maintain all sorts of complex stacks or lists to keep track of the correct instance.

BFD was designed from the beginning to handle this. Each `bfd` contains uninterpreted, target-specific data (in the form of a `void *` in the `bfd` structure). None of the external entry points use these data, but back-ends are free to store in them whatever information they deem necessary. In fact, there are two such blocks, an undifferentiated one and another just for archives. This is so one may use, for example, COFF-style archives with s-record files. If necessary, we can spilt the target-specific data again, although as time goes by it becomes less likely that it will be necessary.

† The `#ifdef`'s were for: Setting the default target, checking the word length or endianness, and to check the names of the flags supplied to `open`.

Successes

Although BFD has been highly successful in making GNU tools more portable, programmers who have been used to programming at a lower level are sometimes a little confused when they start programming with it. For example, they often ask why BFD doesn't provide a way to identify the "text" section, since `a.out` files always have one. BFD doesn't have any built-in knowledge of a "text" section. There *is* often a section called "text" or ".text", but to find an executable section one must identify a section flagged to contain instructions. The "price" of this abstractional confusion is ultimately clearer and more portable code, with fewer system-specific constraints wired in.

One useful discovery about BFD was the test program `copy` a user wrote using BFD. It merely copied a binary into a file. If that didn't work, it meant BFD had a bug. This program turned into `strip`, `nm`, and `objdump`, since those programs essentially perform a copy operation with certain information filtered out. Now it is possible to strip an archive merely by copying it, telling `copy` to discard debugging symbols.

Another useful surprise has been the ability to deal with heterogeneous binaries. This allows users with different binary formats to simultaneously work with libraries in either format. This was achieved as a side-effect of implementing support for the different formats in BFD. This feature is being used by the GNU kernel, which uses BFD to read object files and write core files.

The GNU C Compiler

The GNU C Compiler is an optimizing C compiler that accepts ANSI and traditional (K&R) C code as well as supporting the GNU C extensions. When it was first released, it ran in "native" mode on the VAX and Sun 3 platforms and generated code that was within 5% of the performance of AT&T's Portable C Compiler (`pcc`). Less than five years later, it generates code for 22 processors, runs under 46 major host configurations, and produces code which is typically 10%-40% faster than that generated by `pcc`. Front-ends have been completed for C++ and Objective C, and are in development for Fortran77, Fortran90, Modula-2, Modula-3, Pascal, and Ada. In this section, we use the term "retargetability" to describe target-specific portability issues, and "portability" in describing host-specific portability issues.

Initial Design Specifications

GCC was initially designed to meet two main objectives. The first was to provide a high-quality implementation of C – one conformant to the (at the time, draft-proposed) ANSI standard, with decent compilation speed, and generating optimized code. The second was to write it in such a way that it could be ported and/or retargeted to run on any machine capable of running the GNU operating system, and thus to port the GNU OS to such a platform.

The initial organization of GCC was based on experience gained from GDB and Emacs. As described in the Emacs and GDB sections, machine and system characteristics could be defined in header files, and when code could not be written to be independent of these characteristics, macros defined by these header files would be used. This may sound like a rhetorical prescription, but precisely describing these

parameters is one of the key contributors to GCC's portability and retargetability.

Emacs and GDB distinguish between "system" and "machine" dependencies. The implementation of GCC refines this organization by distinguishing *host* parameters from *target* parameters, and by defining a *machine description* language which can be translated into many other sorts of information. Because the compiler does not contain a built-in interpreter (like that used by Emacs for Lisp), GCC follows the GDB paradigm of allowing the user one C file in which to put target-specific functions (for example, those used to format assembly code).

When two different host machines have the same processor but different operating systems (for example, a VMS VAX and a BSD VAX), most "machine" characteristics are common to both, while "system" characteristics can be quite different: under UNIX, a nonzero exit code means failure, the opposite of VMS. To make the compiler portable, aspects such as these exit codes had to be placed in system-dependent files.

The *host* and *target* files are written in such a way that they can be included by files overriding certain definitions. Thus, "system" differences need not blur *host* and *target* distinctions. By cleanly separating concerns of portability (host parameters), retargetability (target parameters), and functionality (machine description and target-specific functionality), GCC provides a powerful and elegant framework upon which to build.

Within the compiler's implementation is another important separation: that between the representations of syntactic constructs and of machine-specific constructs. Both were designed to be fully general, the former having been successfully extended to handle multiple languages, and the latter a variety of machine architectures. The machine-specific representation was further refined to distinguish machine patterns (such as input and output operands of a given machine operation) from architecture characteristics (such as machine word size, numbers of bits, and numbers of bytes). These abstractions have allowed some users to easily make GCC into a cross compiler for small 8- and 16-bit PCs.

This implementation, GCC version 1.0, was released in June of 1987.

Evolution of the Design

By the time GCC had been available for two years, it had been ported to about a dozen platforms. Virtually all of the ports were *native* ports, meaning that the compiler was built on the system for which it would generate code. The fact that so many native ports were accomplished by amateurs (most commercial compiler companies were charging about US$500,000 for a port of their proprietary software) showed that GCC was a highly retargetable compiler.

As the reputation of GCC spread, especially in fields where performance was critical (such as real-time robotics), users began to configure GCC as a *cross-compiler*. At first there was not much difference between the native and cross configurations: users worked on Motorola 680x0 workstations and developed embedded 680x0 boards. Later, as users upgraded their 680x0 machines to RISC processors such as SPARC and MIPS, GCC's portability underwent more serious tests.

Building a cross compiler from SPARC to 680x0 was not difficult. Structure alignment rules are different, but byte and bit orders are the same. On the other hand, the DEC MIPS platform uses "little-endian"

byte order, which means that operations such as constant folding (which amounts to performing target computations on the host at compile time) needed to be done with care. Because these problems were anticipated from the outset (there were macros for `BYTES_BIG_ENDIAN` and `HOST_BYTES_BIG_ENDIAN`), the "port" involved little more than verifying that the code worked correctly when used.

What was not anticipated was that single users would want to support GCC on multiple machines, generating code for a variety of processors.

Factors that Influenced (or Mandated) Changes

Perhaps the greatest obstacle to performing "host ports" of GCC involved using the host compiler to build GCC for the first time. GCC is designed to expose the user to as few predefined limits as possible. For example, it has no fixed-sized buffers that can overflow when long identifier names or large numbers of `cases` in a `switch` statement are encountered.

GCC was written with the assumption that it would be compiled by compilers of similar implementation quality. This assumption proved to be almost universally false: the macros in GCC's header files often exceeded built-in limits of system compilers, various non-ANSI platforms had header files that were only supposed to be provided in an ANSI environment, and some compiler header files would simply be wrong. A fair amount of effort went into making GCC run on these lowest of the lowest-common-denominator platforms.

Conclusion

GNU software has been extremely portable from the beginning. As we have gained more experience, we feel that our software has become even more portable, an increasingly important quality in today's world of standards. As we extend and improve our software, including its portability, we ask that the technical community continue to help us in this process.

Acknowledgements

Thanks to David MacKenzie for initial thoughts on this subject, and for his expert editing.

Thanks to Arnold Robbins for his thoughts and editing.

Thanks to Karl Berry, Kathryn Hargreaves, Henry Mensch, and Leonard Tower, Jr., for their editing assistance.

References

[Fou92a] Free Software Foundation, *The GNU Coding Standards*, Unpublished, but can be obtained from the address `gnu@prep.ai.mit.edu`, 1992.

Portability in a Research Environment

Andrew Hume

Howard Trickey

AT&T Bell Laboratories
Murray Hill, New Jersey, USA
andrew@research.att.com

Abstract

Software developed within a research environment is both easier and harder to make portable than software developed within a industrial environment. Projects utilising unique aspects of the underlying environment may prove difficult or expensive to port. On the other hand, access to the system source can sometimes solve portability problems locally.

This report describes some of the approaches used within the Computing Science Research Center at AT&T Bell Laboratories for writing portable software.

1. Introduction

This report describes some aspects of how portability interacts with the research done within the Computing Science Research Center at AT&T Bell Laboratories. For some center members, portability is irrelevant – all they need is TeX or a Fortran compiler. For others, particularly those who investigate new directions in paradigms, portability is simply not an issue, as their work is not portable. A third group has a much harder job; their work is mostly portable but is made less so every time they use a novel part of our environment. They are very much aware of the tension between research and standards; standards are fine when your research can build upon them, but when your research explores new ways of doing a particular thing, standards are often a millstone around your neck.

2. Our Environment

The hardware supporting our Center's computing environment is fairly ordinary: we primarily work on VAX 8550s and SGI multiprocessors. There is an ECL MIPS system (6280), a token Sun, a Cray X/MP-28 and some VAX 11/750s we can't seem to get rid of. For terminals, we use mainly MIPS Magnums and Gnots (a terminal designed in our center by Bart Locanthi), although there are several Nexts and NCD X terminals.

Our networking is split between Datakit, a virtual circuit based network, and Ethernet. (There is also an experimental 45Mbit/s link to a few other sites.)

The software environment is rather more mixed. The VAXen run 10th Edition Research UNIX [Hum90a], most of the SGI machines and the Cray run the manufacturer's version of the UNIX system, and the rest run Plan 9, a new operating system developed in our Center [Pik90a].

3. So How Do I Write Portable Software?

It depends a lot on how big the thing is you are trying to make portable, and to how many different systems you wish it to be portable to. There is a spectrum of portability; we give some examples below.

3.1. Approach 1: Use a trivial UNIX system environment

There is a class of programs that just don't need much from the environment. Take as an example *gre*, a new version of *grep*. The sole requirements it has on the environment are

```
<ctype.h>   read     printf   longjmp
open        write    malloc   setjmp
close       fwrite   fflush
```

(The names above are library functions or system calls; names in <> are header files.) Until recently, programs of this kind were extremely easy to port. However, a growing need to work with old C compilers, ANSI C compilers (particularly with strictness flags turned on) and C++ compilers has made the source uglier and more complicated.

The advantages of using function prototypes are too great to throw away, so we write new programs with function prototypes. Rather than litter up the source with `#ifdef`'s to accommodate non-ANSI C compilers (and C++), we use an awk script to convert files to use old-style function headers before distributing the files outside of our center.

3.2. Approach 2: Use a rich UNIX environment

Another class of programs uses much more of the UNIX environment, or in current parlance, the environment defined by ANSI C [ANS89a] and ISO 9945-1 (which was POSIX 1003.1). Take as an example *mk* [Hum87a], a new version of *make*. Some of the routines it needs from the environment include

```
<ctype.h>   execle   getuid   print     strchr
_exit       exit     lseek    printf    strcmp
access      fflush   malloc   read      strcpy
atoi        fgets    memcmp   readdir   strdup
atol        fopen    memcpy   regcomp   strlen
close       fork     memset   regexec   strncmp
closedir    fprintf  mktemp   regsub    system
creat       fputc    open     signal    time
dup2        fwrite   opendir  sprintf   unlink
environ     getgid   perror   stat      utime
errno       getpid   pipe     strcat    wait
write
```

Except for the regular expression routines, these are all covered by the ANSI C and POSIX 1003.1 standards. (The regular expression routines are covered by POSIX 1003.2.)

Most, if not all, providers of UNIX have promised to embrace POSIX in some way or another, so eventually programs written to these standards ought to be very portable. Each new release from the major manufacturers comes closer to POSIX conformance, but most are not quite there yet. And even if the vendors started to ship POSIX conformant systems, there are many users out there who won't have upgraded their operating system and, in fact, may not even do so until they buy a new computer. Thus, any exported code is likely to run into any of the following levels of POSIX and ANSI C conformance:

- Older BSD-based UNIX systems: Most core functions are there, though some string functions have different names. Headers are mostly missing or wrong. Typically, the C compiler is pre-ANSI.
- Older System V-based UNIX systems: Most functions are there. Many header files are there, though they lack function prototypes and may have extra things in them. Typically, the C compiler is pre-ANSI.
- First Pass attempt at POSIX compliance: All or almost all functions are there. Header files are there, but not up to par in various ways (perhaps no prototypes, may include extra stuff that shouldn't be there).
- POSIX and ANSI C compliance. Any extra stuff in header files is either allowed by the standard, or properly protected by some sort of extension feature-test symbol. It appears that SGI's IRIX release 4 reaches this level.

We have tried to anticipate the industry trend towards standard languages and environments. On all of our machines, we have built an ANSI C/POSIX environment called APE, consisting of an ANSI C compiler (including preprocessor) which searches `/v/ape/include` instead of `/usr/include`. The files in `/v/ape/include` are mostly the same across machines, but there are some machine-dependent ones too. Only ANSI C headers may be included if no feature-test symbols are defined. If `_POSIX_SOURCE` is defined, then POSIX headers may be included and more (POSIX-defined) symbols are visible in the ANSI C headers. Note that this is deliberately picky; our goal is that software that runs under APE locally will at least build on any conforming ANSI C/POSIX environment. The downside is that software developed under more user-friendly environments may have a few easy to fix problems, and some potentially more awkward problems, compiling under APE. The easy problems involve using the feature-test symbols, which APE applies as strictly as the standard allows. The harder problems involve system parameters which, in principle, are only known at runtime. One such example is the number of files that a process can have open at once. The symbol `OPEN_MAX`, if present, defines this maximum. Even though the value of this is known (and fixed) in all our implementations, we do not define this symbol because it is not compulsory according to POSIX and thus, we force our programmers to cope with the hardest case (where it is not a compile-time constant). Most often, this just means changed statically declared arrays into `malloc`'ed arrays. The APE compiler loads against a library which is constructed by extracting many functions from the system `libc.a` and providing others from source of our own.

One problem with the POSIX solution is that ANSI C and POSIX 1003.1 don't quite provide enough functionality to implement some programs. Some omissions, such as `popen` and `ftw`, are easy enough to simulate, but it is annoying that they are not there. Standard network access functions are a bigger problem. Eventually, we expect various POSIX subcommittees will solve these problems. Until then, we find it necessary to maintain a small additional library to go along with APE. The feature test symbol for these additional routines and their headers is `_RESEARCH_SOURCE`.

3.3. Approach 3: A little piece of England

During the latter part of the 80s, there was an explosion in the number and variety of UNIX systems. Every day seemed to bring another system with a small (sometimes more) number of gratuitous differences to existing UNIX systems. Rather than design a system to a common subset of these systems, our developers would assume a specific environment and rely on a system-specific library to implement this environment.

This style of portability works quite well with largish subsystems such as *upas*, our Research mail system, and the File Motel, our file backup system. The success of this technique depends on how well the support environment is designed and how much effort is needed to port the support library. In practice, this has never seemed to be much of a problem, but there is a lot of interest outside our Center on automating this process. People seem to like trusting baroque programs, like *config*, which act as oracles on what UNIX system you have and thus, you can (in principle) write one support library conditioned by `#ifdef`'s supplied by such an oracle. (This is largely unused in our Center, mainly because even if there were a reliable way of automatically determining the appropriate `#defines`, and there never seemed to be one that worked on our UNIX system, a single library of source festooned with `ifdef`'ed code seemed rather less attractive than multiple copies of the same library tuned to various systems.)

Perhaps the most elaborate form of this self-configuring approach we know of is that used by some colleagues in another group at AT&T Bell Laboratories. Applications are built using a fairly extensive library common to their group. The installation process first builds this library piece by piece, probing for each routine by compiling a little test program, and either using the installed routine or some portable source. Any installation errors or problems are automatically mailed back to the owner of that piece. This, of course, takes time; it took well over 45 minutes on a VAX 11/750 to install some tree-walking software (equivalent to *ftw*)!

Note that the difference between approach 2 and approach 3 is small but important; with approach 2, the need for a separate environment eventually will go away (except, of course, for target systems which do not support ANSI C and POSIX). With approach 3, you are stuck with your support libraries forever, although with luck, you may be able to implement your support library under POSIX.

3.4. Approach 4: Just say no

There comes a point, though, where you simply cannot port programs. This can stem from an ideological or aesthetic point of view, or from a simple mismatch of system capabilities. An example of the former might be programs that depend on features peculiar to Plan 9, such as a

file system configured to and shared by a process group, or by the way the environment is shared by a process group through the file system. (In principle, these could be nearly simulated, albeit with considerable effort.) An example of a mismatch might be programs that depend on the normal UNIX system semantics of linking and unlinking files. Plan 9 simply doesn't have links (of any kind). And under some circumstances, these programs might fail if the underlying file system is an NFS file system.

4. Some Real Examples

Here are some real examples from our center.

4.1. *gre*

Gre is a fast replacement for *grep*, *egrep*, and *fgrep*. It consists of a sophisticated pattern matching library, including single string, multiple string and regular expression routines, of 3349 lines of 17 C source and header files. The *gre* source, excluding this library, is 977 lines of 9 C source and header files. The program has proven to be very portable; the reasons why are

- It requires very little from the system; the most worrying are `setjmp` and `longjmp` (although these have caused no problem so far). Actually, the routine that caused the most porting problems was `getopt`, so we now just ship it with *gre*.
- The source is designed to be compiled with either ANSI C or C++. However, by adhering to a specific style of declarations, an *awk* script converts the source into old-style C.
- One thing that varies across our computing environment is the preferred buffered I/O system. On Plan 9, we use *bio* by Ken Thompson; on 10th Edition machines, we use *fio* by Andrew Hume; and on the other machines, we use *stdio*. These are handled by a compilation flag (say `USE_STDIO`) set within the makefile and an appropriate header file:

```
#ifdef  USE_STDIO
#define         PR      printf(
#define         EPR     fprintf(stderr,
#define         SPR     sprintf(
#define         WR(b,n) fwrite(b, 1, n, stdout)
#define         FLUSH   fflush(stdout)
#endif
```

  (*Gre* doesn't use buffered input; it uses the system call *read* for all its input.)
- Compilation is controlled by *mk* on our Plan 9 and 10th Edition machines, but an extremely simple makefile is provided for other systems.
- *Gre* comes with an extensive test suite of 212 simple tests and 13 complicated tests. (Every bug generated at least one test case.) The test suite is comprehensive, convenient to run and takes only 35 seconds to run on a VAX 8550.

4.2. File Motel

The File Motel [Hum88a] is an example of a larger system. It is a file backup and restore system based on a central server and multiple clients running on heterogeneous systems. It solves modestly well the main problems facing such systems: heterogeneous compilers, libraries and include files; heterogeneous networking interfaces; heterogeneous backup media; and configurability issues. To give you an idea of the size of the system, a client has a configuration file, 3 shell scripts and 15 executables; one script is in `/usr/bin` and all the other files are in `/usr/lib/backup`. The server has an additional 7 shell scripts and 16 executables, supporting amongst other things a compressed B-tree database. The source for the 10 shell scripts is 376 lines, and the C source is 5065 lines in 38 source files.

The porting strategy chosen for the File Motel is typical for large systems in our Center; the source has almost no `#ifdef`'s (when used, they control features or functions rather than alternate implementations) and instead uses system or feature specific libraries to hide system differences. Unlike *gre*, the File Motel relies upon a specific make tool *mk*, and a specific I/O library, *fio*; it was easier to port both of these than it was to deal with the variability and inefficiency of *make* and *stdio*. The support libraries are

library	files	lines	contents
`libc`	28	2158	fio, getopt, regcomp/regexp
`libfm`	24	1214	db interface, logging, etc
`libcbt`	14	2045	compressed B-tree library
`sys`/*sys*	9	754	(see below)

To some extent, the system specific library reflects when this system was implemented (1986 and 1989). (If this system were reimplemented today, we would assume POSIX 1003.1 compliance and several of the routines would move into `libc`.) The routines in the system specific library are

`dateadjust`	set dates on a file
`dirtoents`	return member names from a directory
`ftw`	file tree walk routine
`nofile`	determine number of file descriptors available
`rx.`*ipc*	establish a bidirectional link to a <machine,service>
`serv.`*ipc*	service requests for a given service
`sysname`	determine the system name
`username`	determine the user's name

We decided, as POSIX has, that you can't unify current IPC interfaces; we simply cope with them. On some systems, we had to implement some other routines as well, such as *mkdir*, *rmdir*, *lstat*, and *dup2*. To support a new system, one simply copies the directory for a similar system and tweaks the implementation. This admittedly crude technique has proved manageable in practice; we have not found a bug common to the system-specific routines.

Configuration issues are controlled by two files; one is on all systems and designates the name of the central server machine and other system configuration parameters, and the other is used in compilation and is included by all the various mkfile's. Nearly all the various configuration options are selected in the latter file; however, the server machine's name is dependent on the underlying IPC mechanism. Here is an example of the compilation options file:

```
# version 10
RANLIB=ranlib
IPC=v10
IPCLIB=-lipc
SYS=v10
FMLIB=/usr/lib/backup
FMBIN=/usr/bin
LIBTYPE=a
NPROC=2
CFLAGS= -DSTRINGH="'<string.h>'"
COMPAT=.compat
WORMFACE=uda
```

4.3. The Research UNIX System

The research UNIX system has not been ported (except experimentally) to a new architecture or machine since it was ported to the VAX in the early 80s. Furthermore, it seems unlikely it ever will be; the job is large and tedious, vendors no longer routinely provide enough information to do such a job, and the people who are most likely to do it are working on Plan 9 instead. However, there is continuing interest and support for the user level programs such as *grep*, *awk*, *troff* etc. So we are leaning towards a solution which is essentially porting the user level commands and libraries (and not the operating system) to the other systems. The problem is maintaining such programs across the different machines and operating systems in our Center. The components of the solution are

APE — As described above. The compiler `lcc` is by Chris Fraser and David Hanson [Fra91a], and the ANSI C preprocessor is by Dennis Ritchie.

mk — *Mk* is a replacement for *make* that runs on all the machines in our Center. It is currently being rewritten so as to better fit within the Plan 9 environment.

dist/ship — These programs aid in the distribution of database files, source and executables across the machines in our environment. *Ship* [Koe84a] works well between the 10th Edition machines. *Dist*, written by Mike Haertel, solves the more difficult problem of handling the non-10th edition machines and Plan 9.

rc — Tom Duff's new shell is the only shell common to (nearly) all our machines. Actually, the issue of a common shell, at least where it impacts installation scripts and *mk* recipes, is one of the main unresolved issues.

4.4. Plan 9

Plan 9 is a *research* project, whose goal is to find a good way of working with modern computers, networks, displays, and storage media. A deliberate attempt was made to question each element of the programming environment, looking for better ways than the traditional UNIX way wherever possible. Software compatibility between Plan 9 and UNIX was not an important issue. So for Plan 9, the only portability question is dealing with the various machine architectures that Plan 9 runs on. This has lead to a style of programming that trades CPU costs for generality. For example, when 32 bit integers are transmitted through communication channels, a byte order, say MSB, is chosen and implemented in the obvious portable way:

```
long v;
unsigned char *p;
v = *p++ << 24;
v |= *p++ << 16;
v |= *p++ << 8;
v |= *p;
```

rather than using an `#ifdef` to determine if we can do it in some faster way for a specific system. We actively use identical source to compile all of the utilities running on MIPS, 68020, 68040, Sparc, and 386 machines.

Plan 9 has its own set of system calls, libraries, header files, compiler, shell, editor, and other utilities. There is some similarity to UNIX, but there are also differences. One difference is that each process group can have a custom namespace. This makes it easy to handle heterogeneous architectures by arranging that, for instance, `/bin` contains binaries appropriate for the architecture that the process group is running on.

What are the advantages of the Plan 9 environment over POSIX? From a programmer's perspective, the big advantage is library and header file simplicity. Most of the system calls and library functions used in normal programming are in `libc.a`, and they are all declared (with function prototypes) in `<libc.h>`. There is a small architecture-dependent file called `<u.h>` that defines types used in various header files, but otherwise Plan 9 sticks to the rule: one library, one header file. Most utilities use only `libc` and `libbio`, a new buffered I/O library. Contrast this with the situation in POSIX, where you typically need six or eight header files, and a manual to remember which ones. For instance, to use the `open` system call, you must include `<sys/types.h>`, `<sys/stat.h>`, and `<fcntl.h>`.

Of course, there are occasions when we want to port software to or from Plan 9. For importing, there is an APE library for Plan 9 that simulates most of POSIX. For exporting, we can use the "little bit of England" approach, and provide a simulation of the Plan 9 environment assuming a POSIX one. In particular, the graphics model is much simpler than other systems and has been implemented using the X window system.

However, the simulation in both directions cannot be perfect. For example, there is no notion of an "effective user" in Plan 9, so `seteuid` has no effect (you can't even fake it). Conversely, the Plan 9 system calls for modifying the namespace would be hard to simulate in a POSIX environment. So far, the problems in importing programs have not been due to the veracity of the POSIX simulation, but rather, with things that POSIX doesn't yet define (networking, for instance). Exporting programs is harder as the more interesting programs tend to use the novel aspects of the Plan 9 environment. For example, the Plan 9 window system, 8.5, relies heavily on how the file system namespace is built and shared by process groups and cannot be easily ported to a normal UNIX system.

5. Conclusion

There is no magic bullet for solving portability problems. Standards such as ANSI C and POSIX have made the chore of porting significantly easier than it used to be, and the situation will continue to improve as these standards become ubiquitous in the market place.

The approaches outlined above describe some ways that applications can be structured for portability. One part of the Plan 9 scheme that deserves special mention is the structuring of header files (an initial machine-dependent file followed by one per library). This is a far superior scheme to that in the standards (although they address rather more concerns than we do).

References

[ANS89a] ANSI, *Programming Language – C*, ANSI X3.159-1989, 1989.

[Fra91a] Christopher W. Fraser and David R. Hanson, "A Retargetable Compiler for ANSI C," *SIGPLAN* **26**(10), pp. 29-43 (October 1991).

[Hum87a] Andrew Hume, "Mk: A Successor to Make," *Summer 1987 USENIX Conference Proceedings*, USENIX (June 1987).

[Hum88a] Andrew Hume, "The File Motel – An Incremental Backup System for UNIX," *Summer 1988 USENIX Conference Proceedings*, pp. 61-72, USENIX (June 1988).

[Hum90a] Andrew Hume and Doug McIlroy, *Unix Research System (10th Edition)*, 1990.

[Koe84a] Andrew Koenig, "Automatic Software Distribution," *Summer 1984 USENIX Summer Conference Proceedings*, pp. 312-322, USENIX (June 1984).

[Pik90a] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, "Plan 9 from Bell Labs.," *Summer 1990 UKUUG Conference Proceedings*, London, pp. 1-9, UKUUG (July 1990).

Inter-Fashion Portability

Barry Shein

Software Tool & Die
Brookline, Massachusetts, USA
bzs@world.std.com

Abstract

Fashion is an invention of civilization to make us unhappy with everything we already own. Many fine programs and shell scripts seem dated simply because they happen to also work on a printing terminal or otherwise do not use one of the latest, fashionable window systems.

This paper will explore a set of simple primitives designed to be executed from shell scripts (or via the simple insertion of `popen` or `system` library calls within existing C programs) which manage various user input/output requirements utilizing modern window systems. This is not a research paper per se, but rather a position paper attempting to outline some views and implementations I have been developing over the past few years, and to try to raise some new questions to the community which have been puzzling me.

I will also speculate on why these programs either haven't been written before or why they are not commonly distributed. Fashion eschews cheating by retrofit: Why fix a hemline when you can buy a whole new dress?

1. Introduction

1.1. What is the problem? Fashion portability defined

We all have many programs which perform rather mundane chores, such as tape backup regimens, mail subject summarizers or various semi-automated operational procedures. These do not seem worthwhile re-working into window systems simply because such interfaces have become fashionable, the effort is not warranted.

What is needed is a way to port programs to the new fashion, as one might add cuffs to a suit or change a hemline. Window systems are still in a volatile state and the fashions continue to change. The issue of modifying a program only for the purpose of changing its user-interface I will define as *fashion portability.*

1.2. How Did We Get Here? How Bad Will This Get?

Once upon a time it was sufficient, in the X11 world, to have a simple menu, for example, constructed directly out of the X11 library. Today one is asked whether their software is Open Look or Motif compatible? Does it use the Xview or Athena or New Wave toolkit? In a few more years a new list of requirements will be imposed upon the fashion of the day. Three-dimensional revolving menus, stereo sound and full-motion video icons, perhaps. How could we have lived without these?

Other window platforms have not fared better. Apple's latest Macintosh OS release seems to now want cartoon bubbles attached to everything. DOS window systems arrive regularly as potential solutions to huge corporate egos. Still others simply fade away with time, Sunview for example. Those who persist in using these obsolete relics are doomed to become about as popular as a paper terminal. The situation seems to be getting worse over time as so-called standards, such as X11, mitose into multiple new standards (de facto or otherwise) such as New Wave and Open Look.

Another, pervasive problem is that of portability in general. Some people still live on dumb ASCII terminals [She90a], others buy what they believed were compatible UNIX systems from different vendors (*they all said they ran X11!*) only to find that toolkits and other support is quite different and incompatible.

1.3. Who Should Be Concerned?

Among all the teapots and tempests some of us need to get some work done. Perhaps the most beleaguered victim of this paned warfare are the system and network administrators and their operations staffs. They are the keepers of the mundane, programs which ordinary mortals don't normally see. But nonetheless they pine for the ability to take advantage of this brave new world of colorful rectangles but generally cannot justify the programming staff a full re-coding effort would require. We can add to them the many applications developers who do not work for organizations interested in spending on developing such windowed applications, but would be interested in using them.

1.4. Why Is This Hard?

1.4.1. Event Drive vs Non-Event Driven Programming

Window programming is normally done using an event-loop model. A set of objects is created and displayed on the screen (perhaps not all simultaneously.) The program then enables interest in some number of events such as a keystroke, a mouse pointer entering a particular area, a mouse button going down or up, etc. The program then enters a loop extracting each event in turn and, generally by means of a case statement, dispatches actions based on the events received.

Non-window programs with a user interface also use a loop as the heart of their control, but generally a loop on a single input stream. The characters (tokens) entered into the input stream are used to switch among possible actions by the program.

1.4.2. The Collision Point Between The Two Models

The major collision between the two programming models involves state. In a windowed interface a user might begin typing text into a dialog box and suddenly move the mouse over to a button for help and click on it. The state of the input must be preserved while the help text is being displayed and scrolled. Other states, such as button choices, must also be preserved during this time.

In a non-windowed interface, generally, only one thing can go on at a time. If you are entering a string and suddenly decide to go off and read some help text you must cancel entering the string and instead enter a command to request help, often requiring you to pop out of the entire action back up to a command level. Interfaces which attempt to provide some help during text input, such as was popular on the Tops-20 command completion interface, only provide brief prompt and noise strings in response to a special character such as '?' and don't actually change state in any significant way.

It is this multiplexing of input types in a windowed system which imposes an entirely different programming style on the problem. Each possible action must be coded in a highly modular fashion and be ready to relinquish control in mid-stream and resume later as if nothing intermittent had occurred. Most non-windowed programs are not prepared to deal with this flexibility.

2. A Basic User Interface Taxonomy

2.1. Let's Make A List

Other than particulars, there are not that many different objects common to popular window systems. Here is a good list to start working from:

- Menus – Short lists of strings to be chosen from.
- Lists – Long lists of strings to be chosen from, often scrolled.
- Text Displays – Program, mail, news and other textual messages, scrolled or not scrolled depending on the length of the message.
- Alerts – A brief message with a small choice of buttons displaying possible actions such as confirming a program exit.
- Dialogs – A brief message with an area to enter a bit of text.
- Editor – A two-dimensional, potentially scrolled panel into which extensive text can be typed and modified.

This list is not exhaustive, there are no gouraud-shaded virtual reality objects, for example. But it does cover the basics I have found useful in converting old, non-windowed programs into new, fashionable, windowed programs. Perhaps the most obviously missing object is a forms manager which would allow the grouping of several objects onto one window panel. I have purposely omitted this as it sorely complicates use of the solution I will describe.

Also missing are entries into new domains such as displaying bar, line, pie and other charts, drawing programs to capture graphical user input, icon management etc.

The choices are heavily weighted towards textual input and output. But so are the expected applications. Another reason for the weighting

towards textual output is to allow implementation of the primitives on non-bitmapped displays.

3. Solution

3.1. A Set of Shell Primitives

What I have done is implemented a set of primitives which follow the above taxonomy of objects. Each program is completely standalone, taking any options it needs either as command-line options, via a standard input stream, or both.

This effort started, intermittently, back when X10 was current. When I implemented and submitted `xman` to the X consortium I also submitted a program called `xmore` which was a windowed analogue to the popular UNIX `more` program. This program appeared on their X10 tapes but was never converted over to X11 entirely.

All programs take standard X11 arguments such as `-display` and `-font`. These are mostly ignored on non-bitmapped implementations, but harmless. All programs return their results as either strings to the standard output or exit values. The intention is that these should be easily run within shell programs, typically as an assignment to a shell variable within backquotes. The `popen` standard I/O library can also be used similarly with these programs from within C code.

A brief overview of the programs and their use:

- `wmore` – Display text read-only, generally intended for textual displays of more than a few lines. Scrollbars are created automatically, as needed. An optional `-s` flag adds a `save` button to the display so the user may save the file to his or her own area.

  There are no outputs from `wmore`.

- `wconfirm` – Put a message on the screen waiting for confirmation. There may be one, two or three buttons. These may have text labels attached on the command line by using `-1 text`, `-2 text` and `-3 text` arguments. If all are omitted one button is displayed with the text OK. The text message to be confirmed is specified on the command line as a string.

  The motivation for the three buttons is the common one button OK, two button YES and NO, and the three button YES, NO, CANCEL. Any text may be attached to the buttons, within reason, but they generally are similar to the structures implied by these examples.

  On exit `wconfirm` puts the text of the button chosen onto the standard output.

- `wdialog` – Get an input string. Puts up a prompt, by default `Input`, or as specified with the `-p` flag. There are two buttons offered, OK and CANCEL. Any other text on the command line is used to initialize the input area (editable.)

  The string typed in is printed to the standard output. In order to distinguish between an empty string and a cancelled input the program exits with zero when OK is chosen, otherwise one.

- `wlist` – List a display of choices and waits for the user to pick one. The choices may be specified either as separate arguments on the command line or through the standard input. For example, the first form might be used for a short list while the second form

```
#!/bin/sh
while true
do
        BACKUP='wlist -T 'Which Backup?' Incr Full Spec'

        case $BACKUP in

        Incr|Full)
                wconfirm 'Tape Mounted and Ready?'
                echo 'backup started...'
                exit 0
                ;;
        Spec)
                wconfirm 'Ready to start Special Backup?'
                echo 'special backup in progress...'
                exit 0
                ;;
        *)
                echo 'Backup cancelled, do call again.'
                exit 1
                ;;
        esac
done
```

Figure 1: *Sample script*

using the standard input might be used to allow choice among an arbitrarily long list of files (`ls|wlist`). If a `-s` option is given then the list is sorted first. The first item in the list is `*CANCEL*` unless a `-c` flag is given, in which case it is suppressed. A `-T text` argument creates a title above the list. Scrollbars are added as needed.

The user's choice is printed to the standard output. If the input is (optionally) cancelled an empty string is returned. Empty strings in the list are silently ignored.

3.2. Commentary

The intention is that these programs be useful within shell scripts or very simply from within programs (via `popen()` or equivalent.) A simple but illustrative example is show in Figure 1.

An attentive reader might note that we have not included a separate menu program. I found that the `wlist` program works well enough due to the automatic addition or not of scrollbars. Short lists look like menus, long lists look like scrolled lists (reminiscent of the file choice box from the Macintosh tool chest, but a bit more flexible.)

3.3. Current Implementations

These programs are currently implemented in Xview (which is quite similar to Sunview in programming style), Athena Widgets and curses.

4. Similar Work

Other than my own `xmore` of several years ago (already described), there are a few pieces of the concept I am presenting already available on the net. There is even a question in the Frequently Asked Questions list posted regularly on the comp.windows.x USENET discussion list which alludes to these (Figure 2).

```
82) Where can I find X tools callable from shell
scripts?  I want to have a shell script pop up menus
and yes/no dialog boxes if the user is running X.

Several tools in the R3 contrib/ area were developed to
satisfy these needs: yorn pops up a yes/no box,
xmessage displays a string, etc. There are several
versions of these tools; few, if any, have made it to
the R4 contrib/ area, though they may still be
available on various archive sites.

In addition, Richard Hesketh (rlh2@ukc.ac.uk) has
posted the xmenu package to comp.sources.x ("v08i008:
xmenu") for 1-of-n choices.

Two versions of XPrompt have been posted to
comp.sources.x, the latter being an unauthorized
rewrite. [R. Forsman]

There is a version of XMenu available from
comp.sources.x; it is being worked on and will likely
be re-released.
```

Figure 2: *Extract from comp.windows.x FAQ list*

These cover similar ground but I felt that a single-minded, cohesive attack would be useful. Note that Xmenu continues to be released *real-soon-now*, I think it has been about four years. There is also no attempt to make these work in multiple windowed and curses environments. Current work is very unsatisfying and not of much use to a truly heterogeneous shop.

The package XVT [Roc89a] is a multi-window system package, now commercialized, which provides a portable subroutine interface to several window systems. My concern is currently limited to UNIX, or at least multi-tasking operating systems. It would not be difficult to adapt the programs in my package and their style to a subroutine library for DOS or Macintosh systems (that is, the specification is simple enough and won't be the problem.) The software described in [Pik91a] and [Pik91b] provides very good overview of the sort of things one needs and one perhaps doesn't in a basic window system.

5. Lessons

The best explanation I have for why a package like this has not shown up earlier is that programmers, when confronted with new technology, make the best the enemy of the good. When given a window system the same part of the brain that used to fire off when laser-printers and multiple font documents first appeared takes over. Every program must be heavily customized with careful widget design and rapt attention paid to placing little icons onto buttons which are artfully customized to the application. That anyone gets any work done is secondary.

In that atmosphere, the idea of generic windowing software goes out the window, so to speak. If an application can't display full-animated fonts in sixteen million colors and use every feature of the window system then, it would seem, it is not worth thinking about. The mundane are left behind. The quest to be fashionable rules supreme.

Technology continues to amaze and astound. With upcoming multimedia technology promising to integrate voice, video and other sensa-

tions managers of software environments would do well to stop for a moment and ask themselves why some basic access to these new tools cannot be provided in a simple manner.

One stellar example of a simple interface to new technology is Sun's /dev/audio device. You can make good use of it with

```
cat /dev/audio > file
```

to record some sound and later

```
cat file > /dev/audio
```

to play it back. At the other extreme are computers like the Macintosh where every idea, no matter how trivial, seems to need another store-bought, customized package.

6. Conclusion

I have raised an issue analogous to the "dusty-decks" problem of days past. The dusty-deck problem was what to do with those old card decks, often containing huge FORTRAN programs, and often still very useful. It wasn't the card media so much as it raised the general software re-use question. Billions of dollars had been poured into writing those dusty-decks, and many were written in a non-portable manner.

Today we are rapidly building an inventory of unfashionable programs. Programs which are still very useful, but don't fit into this windowed, bitmapped, friendly GUI world. I have presented a way to raise those hemlines, cuff those pants, and take them back out for a proud walk with little trouble.

References

[Pik91a] Rob Pike, "A Minimalist Global User Interface," pp. 267-280 in *USENIX Conference Proceedings*, USENIX, Nashville, TN (Summer 1991).

[Pik91b] Rob Pike, "8½, the Plan 9 Window System," pp. 257-265 in *USENIX Conference Proceedings*, USENIX, Nashville, TN (Summer 1991).

[Roc89a] Marc J. Rochkind, "A Unified Programming Interface for Character-Based and Graphical Window Systems," pp. 109-117 in *USENIX Conference Proceedings*, USENIX, Baltimore, MD (Summer 1989).

[She90a] Barry Shein, "Primal Screens," *Sun Expert* **1**(3), Computer Publishing Group (January 1990).

Camera: Cooperation in Open Distributed Environments

Gert Florijn†
Ernst Lippe†‡
Atze Dijkstra‡
Norbert van Oosterom†
Doaitse Swierstra‡

† *Software Engineering Research Centre – SERC*
Utrecht, The Netherlands

‡ *University of Utrecht*
Utrecht, The Netherlands

camera-info@serc.nl

Abstract

The next generation of end-user computing environments will be marked by two features: they must be able to operate in a rapidly evolving distributed environment and they must provide support for cooperation among users. The Camera system, a platform for defining (end-user) workspaces, provides a particular approach to these issues which is based on visible distribution of data and explicit communication among users.

This paper describes the architecture of the Camera system in some detail. Among the topics addressed are the Object Management System, which replaces the traditional file system as a data store, the two-level architecture which introduces a distinction between development activities and the products created during these steps, and the mechanisms for communication and cooperation, which allow users of different, isolated systems to exchange progress and merge the results of different activities. The paper includes a description of a prototype implementation on UNIX, and indicates some of the work in progress.

Introduction

The large-scale introduction of networks combined with the development of more open computer systems offers some interesting promises for the future. In principle, we can expect to see "open distributed environments", rapidly evolving, integrated networks consisting of all kinds of computers, tools and data. People will use these environments

to communicate, share data, exchange progress, synchronise activities, in short, to cooperate.

At the moment, most attention is focussed on the technical "openness" of future systems. Committees and consortia are defining standards on all layers of current computer systems: processors and other hardware, operating systems, communication protocols, user interface systems and styles, and now even inter-application communication models [Gro90a]. Of course, these developments are important. They will solve (some of) the extremely annoying interoperability problems well known to anyone who tries to get some work done on a heterogeneous multi-vendor network.

But, with all this activity (and the related hype) going on, some people seem to forget that things like tool interoperability and system integration are not goals per se. We need these technical provisions to make computers more practical and useful for individual users and to enable groups of computer users to cooperate more smoothly. Thus, besides thinking about machine interoperability, we need to consider *user interoperability* as one of the primary goals of open systems.

The Effects of Distribution

Support for working in distributed environments will be a key ingredient of future open systems. Again, most attention nowadays is focussed on technical issues: physical networks, communication protocols, networked applications, etc. Far less time is spent on the conceptual model of distribution that should be used for these open computing worlds. From an end-user's point of view, however, this model is the key ingredient because it determines how they view their computing environment, and how they communicate and cooperate with others.

Some developments point in the direction of a "heterogeneous, distributed mainframe" model. The idea is that all users share a logical workspace consisting of tools, data, etc., which is mapped transparently on all available machinery. If a new service is introduced somewhere in the network, it can be accessed and used by all people connected to it, regardless where it is located or on which machine it runs. People basically use the same name-space and protocols for accessing local and remote entities.

There are, however, some fundamental problems with this model. In particular, the model only supports people as long as their system is connected to the network. While this may seem a reasonable assumption it has some severe limitations and drawbacks. As more and more computers move from the desktop via the "lap top" and the "palm top" into the background [Wei91a], computers and computer users will become less tied to one physical place. People will take their computers along wherever they go, and use them wherever they want.

This means that computers will not always be linked to the same network and even that they will not be connected to any network for unknown periods of time. Of course, future protocol architectures must find a way of dealing with people and/or computers moving from one network to another without losing their identity. But, on a more abstract level we must consider how end-users deal with the fact that they aren't connected to a network or that their current physical connection is too slow to get the full integration they want. Clearly, the wrong conclusion would be to say that users should not continue work-

ing in such situations and wait until their machines are connected (properly) again.

The model of one shared workspace also has some conceptual problems with respect to cooperation among people. Sharing a workspace with others is only useful if people actually cooperate in some way or another. Cooperation implies more than sharing of data and tools, running remote applications or accessing remote information. Within any team (regardless how closely they cooperate) there exist conventions as to how particular activities are carried out, agreements on which tools are used for particular jobs, conventions on how information is archived, protocols describing how and when others are informed of certain developments, etc. Since people can be involved in many different collaborations, this means that they participate in many of these shared workspaces, each possibly with its own particular conventions. It is clear that this mismatch with the model of one global workspace should be handled in one way or another.

The Camera Project

Given these problems it is essential to investigate alternative (end-user) models for organising distribution and cooperation. In the Camera project at SERC we study a somewhat radical model which is based on explicit communication and non-transparent distribution.

The goal of the Camera project is to devise mechanisms and tools which assist people who use loosely-coupled infrastructures for cooperative development, like the joint writing of a book or the development of a software system. The term "loosely-coupled" indicates that we focus on situations in which the network configuration is dynamic: computers need not be permanently linked to others, or may be linked by slow lines. In particular Camera concentrates on asynchronous cooperation. We want to investigate concepts and technology that help these people keep track of their own progress, and allow them to periodically synchronise and coordinate their work with others.

Our overall approach to the problem is strongly influenced by two choices. The first one is that we accept the autonomy of individual computer users. We aim to support cooperation among individuals and teams who are more or less free to organise their own (computer) workspace, without having to conform to central procedures. The consequence is that coordination and synchronisation of activities cannot be enforced, but can only be stimulated.

The second fundamental choice is that we do not attempt to hide distribution details. We assume that data must always be available within the workspace of a user before it can be accessed. Thus the networked world is divided into two units: the "own" system, and the rest of the world. In this way, the functionality of the workspace does not depend on whether it is connected to a network or not. It can be used even without being connected to any other system.

Of course, these choices imply that there is no implicit data sharing. The basic model is to communicate copies of data and to integrate them into the recipient's workspace. Thus, if an author wants to change something in a chapter prepared by her colleague, she must transfer it into her own workspace before being able to read and/or change it. This style of working resembles the way people currently cooperate using message systems such as electronic mail or bulletin boards.

Replacing a shared repository by explicit copies of data introduces a version management problem. There are potentially many copies of the same data which are being modified by different users in parallel. An individual user must not only track her own activities, but must also explicitly synchronise and merge her progress with that of others.

The Camera System

Research in the Camera project is aimed at developing generic (i.e. application domain independent) mechanisms to support communication and collaboration among these "isolated" systems and to help people solve the corresponding versioning and synchronisation problems. The mechanisms are unified in the design of the Camera system [Lip91a] which represents the working system of one user or a small team of users who cooperate closely. In compliance with the choice for autonomy, each Camera system is fully protected against interference from outside. Users have full control over what is stored and what happens within their system.

The architecture of the Camera system has the following characteristics:

- To improve support for the exchange of data and for merging separated activities, it is necessary that the computing environment has as much knowledge as possible about the semantics of the user's data. On the other hand, since the application domain is open, this need should not restrict the use or expressiveness of the system. To satisfy these needs, Camera users store their data in an extensible Object Management System (OMS). Data is represented as (typed) objects and relationships among these objects. Functionality is represented by methods associated with object types. Methods can be invoked by sending messages to objects.
- In order to help people combine the results of different development activities, users should be able to represent their activities and keep track of the history of these activities. In Camera we support this through a two-level architecture combined with built-in facilities for history management. Development activities are represented as objects in a special OMS called the "Album". The data that is the subject of a development activity, which constitutes an OMS in itself, is versioned. States of such an OMS (called snapshots) together with the changes that were involved are also stored as objects in the Album.
- Within Camera, facilities for communication with other users (i.e. those using other Camera systems) are integrated with the rest of the system. Transferring data does not require different representations, or packing and unpacking. Camera users only have to communicate on the logical level: they can send (collections) of objects to other Camera systems/users who are represented as objects on the Album level.
- Incorporating and combining the results of different development activities is supported for different scenarios. For a producer-client relationship, in which a recipient uses the end-result of work carried out by someone else, it is possible to exchange either complete states of an OMS (snapshots) or a comprehensive part of such a state (called a piece). For true parallel development, the end-results have to be merged. Camera contains sup-

port for merging based on the operations carried out in the development lines.

In subsequent sections we discuss the architecture of the Camera system in somewhat more detail.

The Object Management System

Camera users store their data in an *Object Management System* (OMS), which is an extension and generalisation of the traditional file system. Object management systems have become popular over the past few years, especially within the area of software engineering environments (PCTE [ECM90a] is a good example in this domain). The Camera OMS is inspired by this work, but also incorporates developments from the field of object-oriented databases (see [Kim89a] and [Zdo90a] for an overview).

The Camera OMS data model is truly object-oriented. Data is stored as objects which have a system generated, unique identifier. Objects are instances of classes which define the structure of objects by listing their attributes. Attributes can store values of various primitive data types (e.g. integer, arbitrary length string, symbol). Associations among objects are not modelled through object-pointers but through (n-ary) relationships. Relationships are tuples in a relation and can also contain primitive values. The functionality that manipulates and transforms the data in an OMS is represented as methods associated with classes. The basic interaction paradigm is message passing: users send messages to objects to invoke certain behaviour.

The OMS is extensible: users can define new object classes (inheriting from, possibly multiple, existing classes), relations and methods (using an external language). More dynamics can be added by using triggers, which invoke an action when certain conditions become true, and through (partially) computed relations, where a script determines which tuples make up the relation. Since the OMS is modelled in itself (i.e. classes are objects, etc.), adding new definitions is done by sending messages to meta-objects.

The OMS provides built-in support for derivation management. A derivation manager keeps derived values (such as the compiled form of a program source) up-to-date with the values on which these values depend (e.g. the source code, the compiler, include files, search paths, options, etc.). The main role of a derivation manager is to optimise the amount of computation involved (mostly by caching previously computed values).

In Camera, derived values are modelled as the result of invocations of functional (i.e. side effect-free) methods. The derivation manager caches (in a hidden cache) the results of such invocations and also tracks all dependencies for a given derived value. This is done by logging all attributes that are accessed and other functional methods that are invoked during the computation. The dependency information is made visible in a system-maintained (read-only) relation. Whenever some part of the OMS is changed, some cached derived values may have become inconsistent. Re-computation of such values and further propagation of changes (either on demand or immediately) is then necessary. Propagation stops whenever the end-result of a computation is the same.

There are several advantages of combining the derivation management with the (implementation of) the OMS. First, users do not have to spec-

ify derivations and dependencies explicitly in some separate file as is the case with tools such as Make [Fel79a]. In Camera one only specifies the methods that compute derived values and associates these with the relevant class. Since dependency tracking is implicit and complete† the system guarantees that the derived values are always consistent with all inputs. There is no chance of forgetting implicit dependencies on things like options and tools. Finally, propagation of changes does not depend on criteria such as the last-modification time. If a re-computation results in the same outcome (as when a comment is changed in a source file), propagation stops.

Program 1 shows an example of the use of the OMS. It is an edited script of a session with our current prototype [Dij92a]. We use an Elk Scheme [Lau90a] shell to interact with the OMS. The example shows the definition (by sending a `construct` message to class `Class`) of a class `C-source` which is a sub-type of the predefined class `Edit`. The attribute `value` of class `Edit` is inherited, and contains the actual C source code. The functional method `compile` defines a derived value, the compiled form of the C source stored in the `value` attribute. The method `execute` takes the byte string produced by `compile` and runs it as a UNIX process. As follows from the example, the OMS

```
;; Construct a new class named 'C-Source
> (send (@c 'Class) 'construct 'name 'C-Source 'super (@c 'Edit))

;; Define a new function 'compile on C-source. It has no arguments
;; and returns an arbitrary length byte-string
> (send (@c 'C-Source) 'def-function 'compile '() '(bytestring)
>       (lambda ()
>         ;; Invoke the byte-string that constitutes the compiler,
>         ;; obtained by sending a message to another object, on this
>         ;; C source code, obtained by doing (send (self) 'get-value).
>         ;; Return the byte string produced by the compiler
>       ))

;; Define a new function 'execute on C-source. It has no arguments
;; and an undefined return type.
> (send (@c 'C-source) 'def-function 'execute '() '(**)
>       (lambda ()
>         ;; Invoke the compiled form of this source by executing
>         ;; the byte string returned by (send (self) 'compile)
>       ))

;; Create a place for sources in the name-space (/src) and
;; create a piece of C-code under the name '/src/hello.c
> (send (@ '/) 'dir-add 'src (send (@c 'Object) 'new))
> (send (@ '/src) 'dir-add 'hello.c (send (@ '/class/C-source) 'new))
> (send (@ '/src/hello.c) 'set-value "... The Hello World program ... ")

;; Run the C-source code (with implicit compile)
> (send (@ '/src/hello.c) 'execute)
Hello world

;; List the dependency information for method 'execute for the source.
;; Note that the #[objectid...] value represents the object identifier
;; of the source object. The <-  indicates a dependency.
> (dependency-print (@ '/src/hello.c) 'execute)
(#[objectid 192.87.7.19 7824 17] execute ())
  <- (consistent derived #[objectid 192.87.7.19 7824 17] compile ())
    <- (consistent attribute #[objectid 192.87.7.19 7824 17] value ())
```

Program 1: *Sample interaction with the OMS*

† Dependency tracking is complete because the OMS is self-contained

contains a basic naming scheme for objects, similar to the UNIX file name structure. One specific name is predefined: the symbol / refers to a specific instance of class `Object`. Classes can be found under the directory object `/class`. Resolving an object name is done by sending the method `dir-lookup` to any object. In the example in Listing 1, we use the macro `@` as an abbreviation for `dir-lookup`, and the macro `@c` as an abbreviation to look up classes.

Camera's Two-level Architecture

The Camera system offers support for modelling and tracking development activities. The basic idea is that each activity has its own OMS associated with it. This OMS contains the product data, i.e. the data that is manipulated and modified in that particular activity. Activities are called *environments* in Camera terminology and they are represented as objects in a special OMS called the *Album*. Camera thus has a two-level architecture (illustrated in Figure 1). The Album OMS has (almost) the same data model as the environment OMS's, which means that it is extensible and programmable.

Support for activities is extended by mechanisms for tracking their history. In Camera, version management is applied to the complete object management systems associated with environments.† Camera (logically) stores distinct states of these OMS's in Album level objects called *snapshots*. Thus, the association between an environment and its OMS in fact is an association between an environment and a particular snapshot. In OMS terminology: within the Album OMS there is a relation called "current-snapshot" which links environments to their current snapshot object.

Snapshots represent an end-result or an intermediary stage of a particular activity. Once created, they are immutable. Changes to an existing snapshot lead to a new snapshot, which is the successor of the old one. In addition, the Album also registers *transformations*, the sequence of

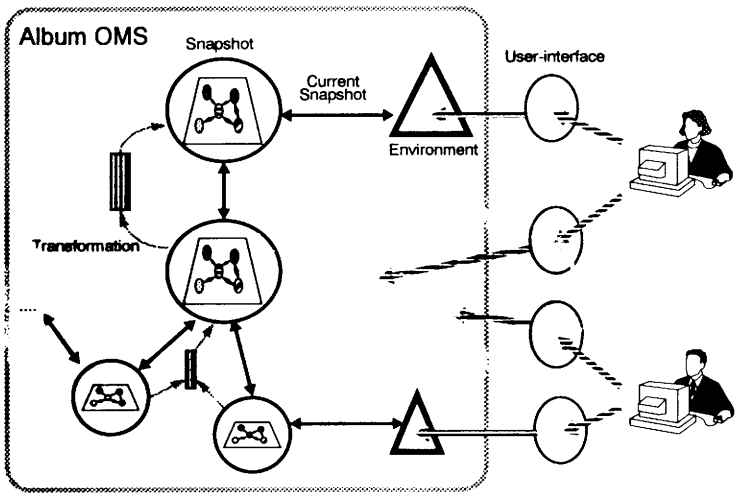


Figure 1: *Camera's two-level architecture*

† A coarse-grained versioning model like this is also found in the Network Software Environment [Cou88a], while Hume [Hum89a] presents a similar model applied to UNIX file systems.

operations that led from one snapshot to another. Transformations are not merely the differences between two snapshots. They contain the messages sent by users which changed the state of the OMS. Transformations are editable and reusable, which means that they can be applied to other snapshots.

A snapshot is self-contained: it contains no references to anything outside it. This implies that all data that is involved in a development step (source code, documentation text, tools, libraries, editors, include files, etc.) is logically part of the snapshot. This has the big advantage that historical states are (likely to be) consistent. Installing a new compiler does not break the reproducibility of old releases, since the compiler with which they were created with is still stored as part of their snapshots.

Incorporating all relevant data in a snapshot can make these quite large. It is clear that it is not feasible to store the different snapshots as distinct copies. The Camera implementation therefore maintains a delta-based, incremental data structure [Lip91b] for storing snapshots.

Doing work within the context of an activity starts with activating an environment. This initiates a transaction in which all operations (message sends) are directed to a temporary, mutable copy of the environment's current snapshot. When the transaction is completed, the new state may be saved as a new snapshot in the Album as a successor of the previous state. If so, the "current-snapshot" relation is updated accordingly and the transformation object is stored too.

Environments also provide the mechanism for accessing historical information and for travelling in time. By associating an environment with an older snapshot (changing the "current-snapshot" relationship) we can work in that old state. Modifications may lead to a new snapshot which is a new successor of the old state. This introduces branches in the revision graph of snapshots.

A Camera system can be used by a team of users. This means that these people share the workspace constituted by the data in the Album OMS. Of course, they all can use distinct environments to represent their own activities. Consequently, at any moment in time, multiple environments can be active. Each of these active environments is isolated from others. Changes carried out within one environment are invisible to the outside world. Even if two environments point to the same snapshot, changes in one environment will not be seen in the other; two distinct successor snapshots will be created. For close cooperation it is possible to share an active environment. This means that changes carried out by one person will also be immediately visible to the other users. This mode of working is similar to current file systems.

Snapshots, transformations and environments are mechanisms. Using them in a particular situation requires policy decisions for determining such matters as whether and when a snapshot should actually be saved for later reference, what extra annotation should be added to snapshots, what relationships exist among environments, etc. The extensibility of the Album OMS allows for the implementation of such policies.

Mechanisms for Communication and Cooperation

A Camera system is inherently isolated from other Camera systems, whether they are running on the same machine, the same local-area network, or in different continents. In order to break this "splendid isolation" we first of all need constructs to communicate with other Cam-

era systems. To support cooperation among users at different Camera systems we must make it easy to exchange the results of development activities and support combination of parallel work.

Within an Album OMS other Camera systems can be represented as objects. For each system a list of possible "routes" to reach that system can be defined. Each route indicates a communication means (e.g. a TCP/IP connection, E-mail) and an address for that type. This information is used by the implementation to determine how to transfer the data. Non-computerised transfer, for instance via floppy discs, is modelled in a similar fashion.

Sending data to another Camera system involves nothing more than the invocation of a message on such a "remote Camera" object with the data to be transferred (a local Album level object) given as an argument. The actual data transfer is done asynchronously. Of course, the time needed to carry out the transfer depends on the network situation. Therefore, users can query the progress of the transfer session. Furthermore, the system informs the users of the completion of a transfer. Incoming data is modelled as a method invocation (executed by the Camera implementation) directed to a general "in-box" object in the Album. This object can decide what to do with the incoming message and the Album object that it contains.

Since communication is defined on the level of Album objects, a basic mechanism to share progress among different Camera systems is by transferring snapshots. Since snapshots are self-contained they contain not only the data that is being developed (the text of a book, the source code of a program), but also all the data needed to use and further develop it (text formatters, compilers, libraries). Given the right circumstances (e.g. similar machine architecture) this means that a snapshot can be used immediately in the receiving Camera system without a need for any reconfiguration or adaptation. This can be useful when people take work along on laptops, or when they use different computers at different places.

Communicating on the level of complete snapshots is not always suitable. The situation in which different machine platforms are used is but one example. In many cases, changes made by one user do not affect a complete snapshot or even all the parts of the product that is being developed. Their work has concentrated only on a specific part of this data (e.g. a chapter of a book). In this situation, transformations can be useful to share progress, because they can be edited and applied to other snapshots. The net changes that were carried out can be transferred and re-applied. This is somewhat comparable to the use of patches [Wal88a] with the aforementioned distinction that transformations are logs of user-level operations, not mere differences between states.

Transformations however do not give the level of abstraction we need because they do not allow us to decompose snapshots into more manageable sub-units. To alleviate this problem we have introduced the notion of *pieces* as a modelling mechanism in the snapshot OMS. A piece is a collection of objects and relationships among these objects They are somewhat similar to complex or composite objects found in some object-oriented database systems [Kim87a]. Pieces in Camera also can have some knowledge about the other objects which are not part of the piece itself, but that are needed to use the data in the piece. These external connections can be modelled as scripts of computed relations. For instance, a piece containing source code could have an external relationship that should be connected to an object representing

a particular compiler. The script of the computed relationship contains the constraints to identify the proper compiler object within an actual snapshot.

Pieces are a key mechanism for cooperation because a piece can be transplanted into other development lines and thus into other snapshots. A transplant involves adding the piece elements to another object system and (automatically) establishing the relationships of the piece with its environment. If a previous value of the piece exists it will be overwritten. If some of the external relationships cannot be established, some functionality may not be available (e.g. the source code cannot be compiled), and the user will be warned accordingly. A copy of a piece can be extracted out of a snapshot into an object on the Album level, and thus be sent to other systems.

Transplanting pieces is mainly useful in producer-client collaborations or in situations in which parallel activities have focussed on distinct sub-parts. In the case of true parallel development however, the changes in one development line must be *merged* with changes made in another. Merging is a difficult problem which, in general, cannot be automated. Changes in distinct development lines may create a conflict and in some cases only the user can decide how this conflict should be resolved. The level of support that can be given for automated merging depends on the knowledge about the semantics of the data to be merged. Instead of merging states, as done by most existing merge-tools, Camera uses an operation-based model [Lip91c] to implement three-way merging, i.e. combining the changes of two development lines based on a common ancestor. The model uses the transformations that constitute the two development lines. The idea is to merge the two transformations by checking whether operations from the two transactions commute, i.e. whether the order in which the operations are applied is interchangeable without altering the end-result, and by identifying and grouping sequences of operations that conflict. These conflicts must be resolved by the user. The algorithm works in an extensible data model since users can specify merge conditions for newly defined operations. Of course, this approach does imply that the quality of merge support depends on the level of detail that the operations provide.

Operation-based merging has several advantages over traditional (state-based) merge tools. Most of these traditional tools support merging of ASCII text files using individual lines as the granule. The tools try to recover the operations that have been carried out, but only have a limited repertoire of operations such as adding and deleting blocks of lines. Single operations that affect multiple lines, such as a global substitute, cannot be identified and the user is confronted with a conflict on each instance. Furthermore, since there is no insight in the actual semantics of the data involved, the result of a merge may be an invalid data structure. Since our merging tool simply re-invokes operations on the OMS, the semantics of the underlying data-structure are guaranteed to remain intact.

We are currently investigating how the Camera system could be used to describe different collaboration scenarios. The idea is that if the activities of a group of users follow a certain plan, this plan can be described in the Album OMS's of the participating Camera systems. A major advantage of putting these process descriptions in the system is that they can be used to automate parts of the communication. Such standard procedures are often found in software development projects, e.g. for handling bug-reports and fixes, for the development and admin-

istration of new releases, etc.† However they can also occur in other domains, such as office systems. Our preliminary research gives us confidence that such process programs can be described as applications in the Album OMS.

A UNIX-based prototype implementation

The Camera system has been implemented on top of UNIX. The prototype consists of several communicating processes (see Figure 2). The key process type is the one that implements an OMS. It is used for implementation of an Album, and also for environments in which a particular snapshot is manipulated. This OMS (process) is implemented partly in C and partly in Elk Scheme [Lau90a]. Users can therefore write new OMS methods in Scheme, and, in principle, in C.

The OMS processes use several servers with which they communicate via UNIX IPC mechanisms. The snapshot storage server stores (versions) of objects. It is also responsible for optimising the index structures that store snapshots. The long-field server manages arbitrary length byte strings which are created in an OMS. The cache manager contains the cache of derived values. The communication manager handles communication with other Camera systems. All the servers use the UNIX file system for background storage.

The simplest user-interface to Camera is currently a small "shell" implemented as a separate Scheme process. However, interfacing other tools with the Camera OMS processes is relatively easy. These processes basically implement a virtual machine with one instruction: send a message to an object. Integrating this with an existing language or tool only requires one to implement this instruction (by making it a remote procedure call to an OMS process). Further integration can be provided by mapping the basic data types of the Camera OMS (strings, symbols, lists, etc.) to the host environment.

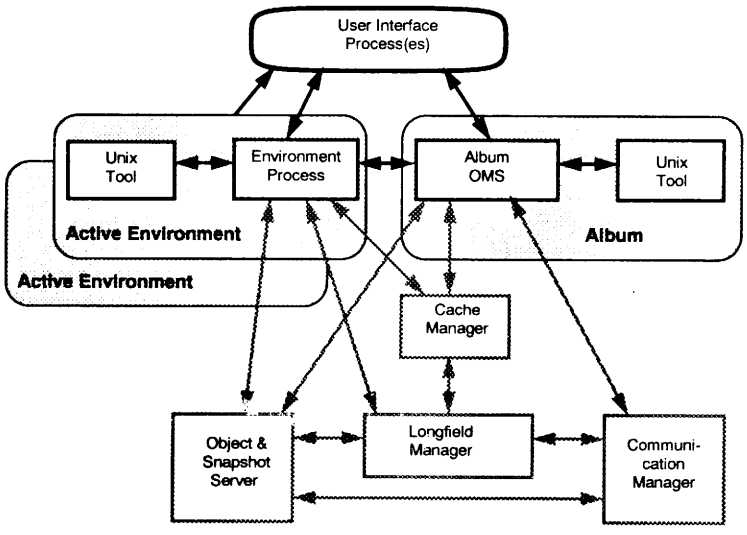


Figure 2: *Processes in the Camera prototype implementation*

† The term process program [Ost87a] is often used to describe these computerised work flow descriptions.

Since this coupling is so small and easily implemented, it means that it is possible to connect existing tools to Camera. We have, for instance, added such a connection to Emacs [Sta86a] and to a Smalltalk-80 system [Gol83a]. This approach provides the opportunity to use Camera as a backing store and extension of these tools.

Several extensions to the functionality of the prototype are currently under construction. Besides a window-based shell we are constructing a piece-editor, a tool which assists users in defining pieces. Work is currently also underway to integrate the Camera OMS with the UNIX file system. The goal is to be able to run existing UNIX tools on the Camera OMS without modifying or recompiling them and without writing tool-specific wrappers. This will be done by creating a special NFS [San85a] server which translates the file operations carried out by these tools into messages sent to OMS objects. This means that tools such as compilers and text formatters can be viewed as functional methods in the OMS. This also means that derivation management is available for these tools.

Conclusions

The Camera system provides platform technology for systems that operate in an open distributed environment. It highlights a particular approach to organising distribution: instead of viewing the whole network as one shared workspace, we assume that each user or team owns a workspace, and that communication and distribution of data among these workspaces is explicit. People must copy data into their own system before they can access it. This model makes the system usable in evolving networks.

The Camera system provides a number of basic mechanisms that can be specialised towards particular application domains. The mechanisms are fundamental, but provide a higher level of abstraction than is commonly found in current systems. The Object Management System offers the modelling power and flexibility of an object-oriented system as a replacement for the traditional file system. The two-level architecture introduces a distinction between development activities and the products produced in them. The history management facilities give the means to save consistent states of products and to re-use them later on. Finally, the available mechanisms for communication and cooperation reduce the amount of work involved in exchanging progress and combining results of different activities.

Using the Camera system implies specialising the definitions for a particular use. Currently, we use the prototype for its own further development. This work is carried out by several people located at SERC and at the University of Utrecht. Since there is no direct network connection between these sites, the system is actually used in a situation that it was designed for. In our case, Camera is used as an end-user system similar to UNIX. We have defined classes in the OMS to represent the kinds of data that we are working with: programs written in Scheme and C, OMS definitions, documentation written in TeX and LaTeX, etc. Since our development process is not strictly structured we have not yet found agreement upon a canonical activity model or history management procedures. The various users define activities and save snapshots as they like it. Likewise, communication patterns are not (yet) structured.

We are, at the moment, investigating other possible application domains for the Camera system. Our interest is in situations where the Camera system would be used as an extension of an existing tool or environment, such as an object-oriented development environment or software design tools. Roughly speaking, the idea is to store the data structures manipulated by these tools as objects in the snapshot OMS by connecting the tools to an environment process. Camera then provides the additional mechanisms for distinguishing activities and catching their history, and for sharing progress including the merging of distinct development lines.

References

[Cou88a] W. Courington, J. Feiber, and M. Honda, "The NSE Highlights, NSE Tackles Large-Scale Programming Issues," *SunTechnology*, pp. 49-53 (Winter 1988).

[Dij92a] Atze Dijkstra, *The Camera Prototype Documentation,* SERC (1992).

[ECM90a] ECMA–European Computer Manufacturers Association, *Standard ECMA-149 Portable Common Tool Environment – Abstract Specification*, December 1990.

[Fel79a] Stuart I. Feldman, "Make – A Program for Maintaining Computer Programs," *Software Practice and Experience*(9), pp. 255-265 (1979).

[Gol83a] Adele Goldberg and David Robson, *Smalltalk-80: the Language and its Implementation,* Addison-Wesley (1983).

[Gro90a] Object Management Group, "Object Management Architecture Guide, Revision 1.0," OMG TC Document 90.9.1 (September 1990).

[Hum89a] Andrew G. Hume, "The Use of a Time Machine to Control Software," pp. 119-124 in *Proc. of the USENIX Software Management Workshop*, New Orleans, Louisiana (April 1989).

[Kim87a] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk, "Composite Object Support in an Object-Oriented Database System," *SIGPLAN* **22**(12), pp. 118-125 (December 1987).

[Kim89a] Won Kim and Frederick H. Lochovsky (Editors), *Object-Oriented Concepts, Databases, and Applications,* ACM Press (1989).

[Lau90a] Oliver Laumann, *Elk – The Extension Language Kit*, 1990.

[Lip91b] Ernst Lippe and Gert Florijn, "Implementation Techniques for Integral Version Management," in *Proceedings of the 1991 European Conference on Object-Oriented Programming (ECOOP)*, ed. Pierre America, Springer Verlag (July 1991).

[Lip91c] Ernst Lippe and Norbert van Oosterom, "Operation-based Merging," Report 91/11, Software Engineering Research Centre (November 1991).

[Lip91a] Ernst Lippe and Gert Florijn, "CAMERA: a Distributed Version Control System," Report 91/01, Software Engineering Research Centre (January 1991).

[Ost87a] Leon Osterweil, "Software Processes are Software Too," in *Proceedings of the Ninth International Conference on Software Engineering* (March 1987).

[San85a] R. Sandberg et al, "The Design and Implementation of the Sun Network File System," pp. 119-131 in *Proceedings of the USENIX Summer Conference* (1985).

[Sta86a] Richard M. Stallman, *The GNU Emacs Manual*, 1985, 1986.

[Wal88a] Larry Wall, *Patch – a Program to Apply Diffs to Original Files*, 1988.

[Wei91a] Mark Weiser, "The Computer for the 21st Century," *Scientific American* **265**(3) (September 1991).

[Zdo90a] Stanley B. Zdonik and David Maier (Editors), *Readings in Object-Oriented Database Systems,* Morgan Kaufman (1990).

Distributed System and Security Management with Centralized Control

Chii-Ren Tsai

VDG, Inc.
Maryland USA

Virgil D. Gligor

University of Maryland
Maryland USA
{ crtsai | gligor }@eng.umd.edu

Abstract

We have designed and implemented a prototype of distributed system and security management for AIX Version 3 on the RISC System/6000 by using an experimental secure remote procedure call (RPC) mechanism [Tsa91a] based on Network Computing System (NCS) [Din87a] and Kerberos [Ste88a]. The prototype consists of distributed SMIT (System Management Interface Tool), distributed audit [Tsa90a] and access control list (ACL) management for AIX systems. Distributed SMIT can manage user accounts, file systems, devices, networks, spoolers and system configuration. Distributed system security management, which includes distributed audit and distributed ACL management, allows the distributed system security administrator to turn on/off auditing, perform audit system management, analyze audit trails and set ACLs on a per-file, per-directory or per-application basis. Based on the experimental secure RPC mechanism and Motif widgets on the X window system [Sch86a], we designed and implemented a high-level, protocol-transparent, integrated interface for the prototype of distributed system and security management.

1. Introduction

We have combined the services of the MIT Kerberos authentication protocol [Ste88a] with vanilla RPCs of Network Computing System (NCS) [Din87a] to support an experimental secure RPC mechanism

The work described herein was performed under contract to IBM and only contains the authors' perspectives. It does not represent, imply or describe any IBM products.

[Tsa91a]. Based on the secure RPCs, we implemented a prototype of distributed system and security management for AIX systems.

The prototype consists of a distributed-system management subsystem, a distributed audit subsystem and an access control list (ACL) management subsystem. The distributed-system management subsystem is an extension of AIX SMIT (System Management Interface Tool), which can be used to handle various system management tasks from system configuration to network management. The distributed audit subsystem can invoke and revoke auditing for remote hosts, locate the audit trail server to which audit trails are transferred, perform audit system management, such as dynamically adding or deleting audit events on a per-user, per-group, or per-system basis and querying the audit status of remote hosts, and trace audit trails. The ACL management subsystem can browse and change ACLs on a per-file, per-directory or per-application basis.

In this paper, we highlight the experimental secure RPC protocol, distributed SMIT and ACL management, the distributed audit mechanism and the ACL structure of AIX. We also compare distributed SMIT with the MIT MOIRA Service Management System [Ros88a]. Furthermore, we present the design and implementation of the prototype of distributed system and security management with Motif widgets on AIX Version 3 for the RISC System/6000. We also take a snapshot of the interface of the prototype. Finally, we discuss the migration of the prototype to the OSF Distributed Computing Environment (DCE) platform.

2. Overview of the Experimental Secure RPC Protocol

The experimental secure RPC protocol is shown in Figure 1 [Tsa91a]. In this protocol, all security enhancements, including authentication, data encryption and decryption, and authorization, are implemented at the client and server stubs, so that the interfaces of secure RPCs are the same as those of vanilla RPCs and RPC runtime is left unchanged. Our secure RPC mechanism uses the Kerberos authentication protocol, encrypts input and returned data, provides data encryption standard (DES) cipher-block-chaining (CBC) checksums for data, and performs access checking against the caller's identity. Therefore, it provides the capability of authentication, data secrecy and integrity, and authorization.

In this protocol, message 1 requests a ticket for the Ticket Granting Service (TGS); message 2 returns a ticket for TGS; message 3 requests a ticket for an RPC server; message 4 returns a ticket for the RPC server; message 5 invokes an RPC by sending a packet that contains encrypted input parameters, the 64-bit DES CBC checksum of input parameters, and a ticket for the RPC server; message 6 returns the encrypted results, timestamp and checksum. The first two messages, performed by the Kerberos command `kinit` or `klogin`, obtain a ticket granting ticket, which is subsequently stored in a ticket file, so that the client stub can retrieve the ticket from the ticket file. The client and the server of a secure RPC can authenticate each other by utilizing the Kerberos mechanism. The RPC server authenticates the client by ensuring that the name in the server ticket, $T_{c,s}$, and the name in the authenticator, A_c, are the same. This protects against the usage of a stolen ticket. The client authenticates the RPC server by the returned timestamp, t_a, which must be the same as the timestamp in the authenticator. Input parameters and the returned data of RPCs are encrypted to maintain the secrecy of the data. To ensure the integrity of the data,

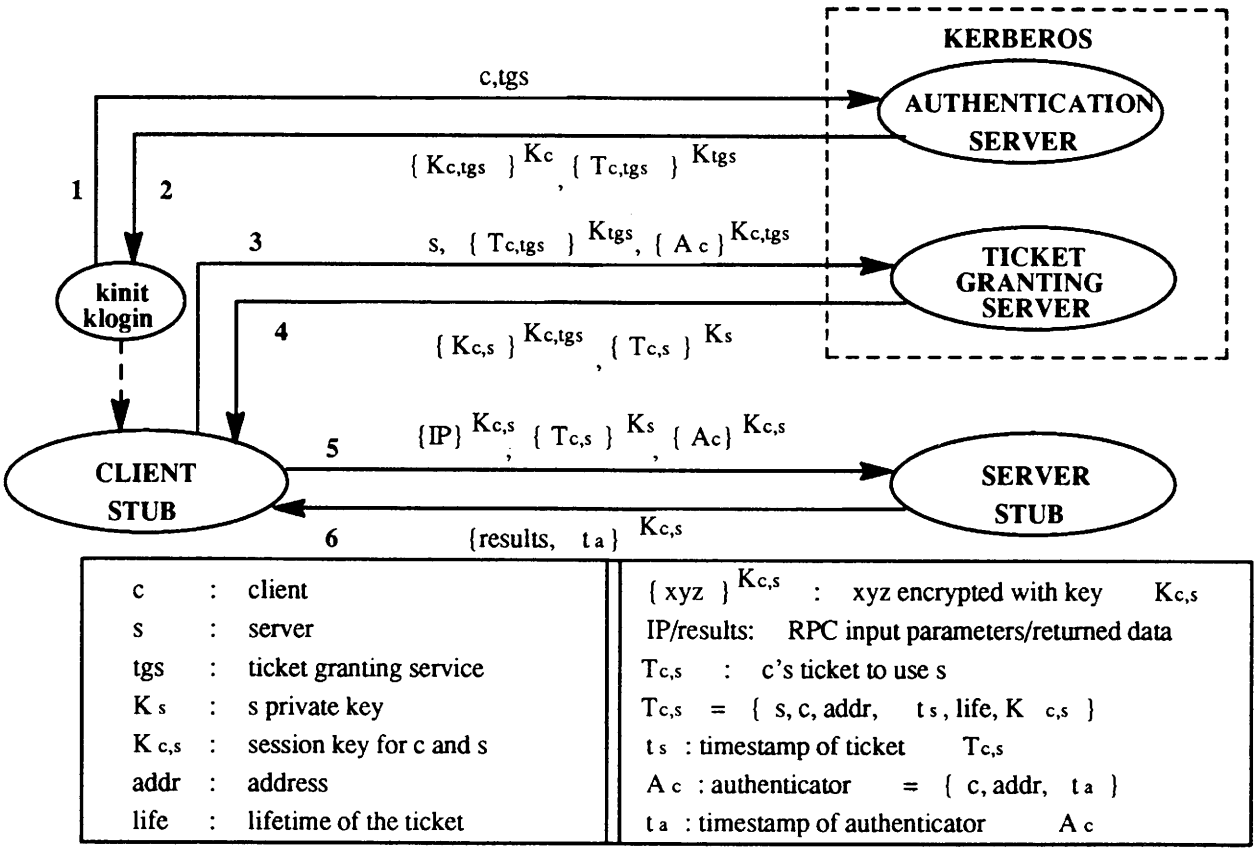


Figure 1: *Secure RPC Protocol*

64-bit DES CBC checksums are provides for the tickets, the input parameters and the returned data. Unlike Kerberos, the server returns t_a instead of t_{a+1} to the client for mutual authentication while providing the same level of security [Bur90a]. The protocol is similar to the Kerberos protocol except that with messages 5 and 6 encrypted input parameters are sent with the server's ticket and authenticator, and with message 6 results are returned with a timestamp.

To enforce access authorization, we implement access checks in the server stub. After authentication in the server is done, the caller identity is checked against the access control list, which is maintained by the server, before the RPC is called. If the access check succeeds, the RPC is executed. Otherwise, the server stub immediately returns an error.

3. Overview of Distributed SMIT and ACL Management

System management tasks in AIX can be handled by using the System Management Interface Tool (SMIT). AIX also provides several system calls and commands for users to browse or change ACLs of objects. In the following sections, we review SMIT and the structure of AIX ACLs and then describe the concepts of distributed SMIT and ACL management. In addition, we compare distributed SMIT with the MOIRA Services Management System in terms of their purposes and system models.

3.1. Distributed SMIT

The AIX SMIT integrates all system management functions in a single tool, which can be used to interactively manage software installation, system maintenance, devices, physical and logical storage, user accounts, communications applications and services, the spooler, resource scheduling, system environment and processes, and applications. SMIT provides a hierarchical screen structure. Users are guided through the use of menus and dialogs to run system management commands. The data objects managed by SMIT are handled by the Object Data Manager (ODM) [IBM90a], which is a data manager intended for the storage of system data. System data managed by ODM include devices configuration information, display information for SMIT, vital product data for installation and update procedures, communication configuration information, and system resource information. All the data are stored either in the `/etc/objrepos` directory or the directory specified by the `ODMDIR` environment variable. SMIT generates and updates two log files, `smit.log` and `smit.script`, for each user. The `smit.log` file keeps additional detailed information that can be used by programmers to extend the SMIT system, while the `smit.script` file records shell-script commands used to perform system management functions. These two files can be used as audit trails.

Distributed SMIT is an administrative function that can provide the SMIT interface and communicate with each system's ODM through a daemon, `dsmitd`, to retrieve and modify system data or to execute configuration functions. The distributed-system administrator interacts with `dsmitd` via secure remote procedure calls. The mechanism of distributed SMIT is shown in Figure 2.

The purpose of Distributed SMIT, to reduce the effort of distributed systems management, is different from that of the MOIRA service management system of the MIT Project Athena. Unlike Distributed SMIT, MOIRA's purpose is to perform centralized data administration, not system administration. Services supported by MOIRA include a name service HESIOD [Dye88a], NFS, mail service and a notification service ZEPHYR [Del88a]. Consequently, MOIRA is designed to provide a unique, consistent view of administration data. To accomplish this

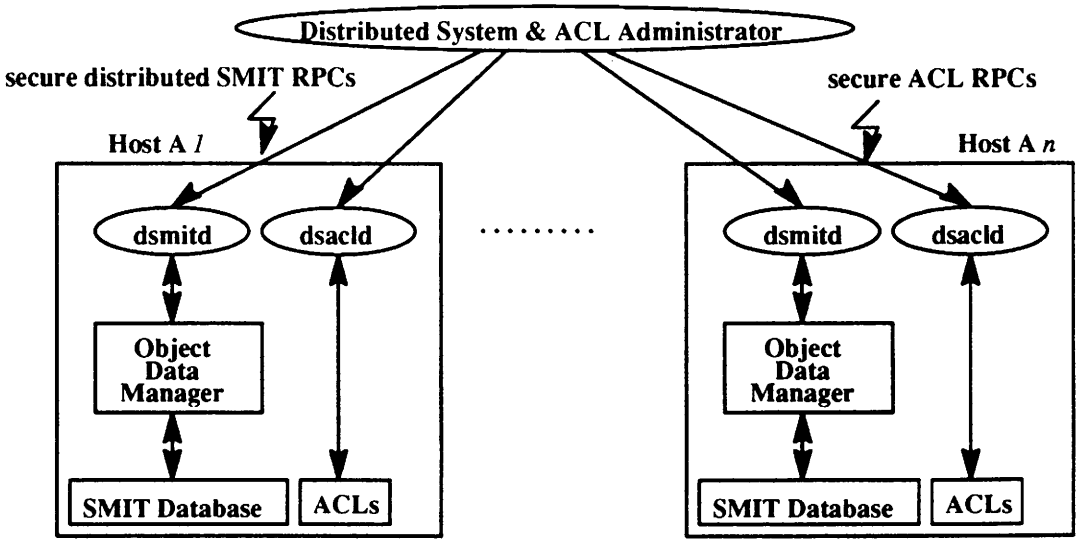


Figure 2: *Distributed* `SMIT` *and* `ACL` *Management*

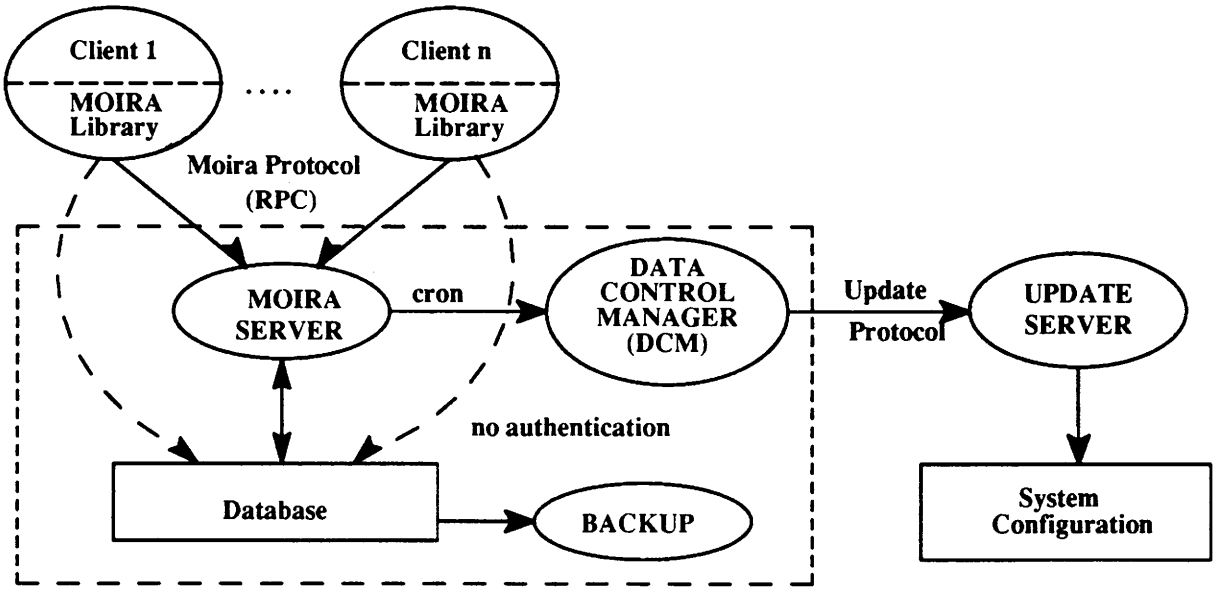


Figure 3: *The MOIRA Service Management System*

goal, all users in MOIRA must communicate with a central server, which is the MOIRA server as shown in Figure 3, to access or modify administration data. The server is responsible for maintaining the integrity and consistency of the data. Clients call the MOIRA library, which uses the MOIRA remote procedure call protocol to send requests to the MOIRA server. The MOIRA server may then access the database to provide its service. The Data Control Manager is responsible for distributing data to servers, which subsequently update the data. Some provisions have been made, so that data updates can survive from server failure to perform action, server crashes, and MOIRA crashes. Consequently, data updates in the MOIRA system are atomic.

3.2. ACL Management

An object's Access Control List defines the access authorization of the subjects in the system. AIX Version 3 implements ACLs on objects such as files, directories, named pipes, message queues, shared memory segments and semaphores, so that users can browse and change the ACL of an object through system calls. The structure of AIX ACLs is shown in Figure 4. Each ACL may consists of *base permissions* and *extended permissions*. Base permissions, which can be modified by `chmod`, contain the setuid, setgid and save text bits, and three sets of access modes for owner, group and others, respectively. Each set of access modes consists of read, write, and execute/search permission bits. Extended permissions consist of an unordered list of Access Control Entries (ACE) [IBM90a]. Each ACE contains a list of identifiers, the type of the ACE and a set of access modes. The ACE types include *permit*, *deny* and *specify*. The permit and deny types indicate that the specified set of access modes is granted or denied, respectively; the specify type means that only the specified set of access modes is granted and others are restricted. The type of an identifier is either USER or GROUP.

Distributed ACL management is the centralized control of ACLs in a distributed system, so that the central ACL administrator can manipulate the ACLs of objects through a daemon, `dsacld`. To simplify ACL

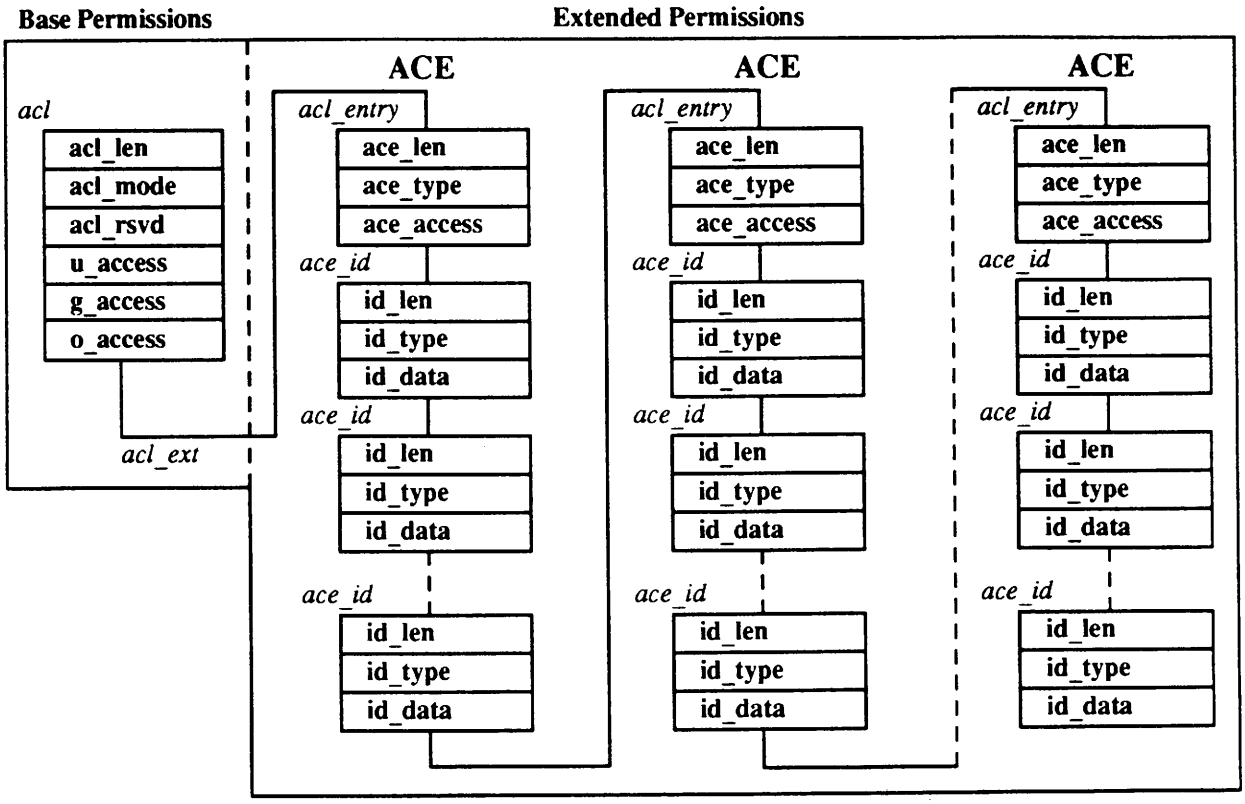


Figure 4: *The Structure of AIX Access Control Lists*

management, ACLs can be changed on a per-file, per-directory or per-application basis, which means that the ACL administrator can change the ACL of a file, the ACLs of all files in a directory, or the ACLs of files categorized to the same application. The mechanism of ACL management is shown in Figure 2.

4. Overview of Distributed Audit Mechanism

The distributed audit mechanism discussed herein is another central administrative function that can perform audit system management, invoke/revoke auditing for each host, instruct each host to transfer its audit trail to a specific site called an *audit trail server,* and trace audit trails [Tsa90a, Tsa91a]. As shown in Figure 5, the central audit administrator invokes/revokes auditing by using the secure RPC mechanism. Each host mounts the audit trail filesystem from the audit trail server over a local directory by utilizing NFS [San85a], so that the host's audit records are compressed and stored in a file, called the audit trail of the host. Consequently, audit trails are collected in the audit trail filesystem of the audit trail server, and the audit administrator can manage these audit trails, or trace user activities or security violations.

The current implementation of the distributed audit system is based on homogeneous stand-alone audit subsystems. To extend it to heterogeneous systems, we need to resolve several problems such as the discrepancies of audit system configurations and audit records. In this paper, we do not address these issues.

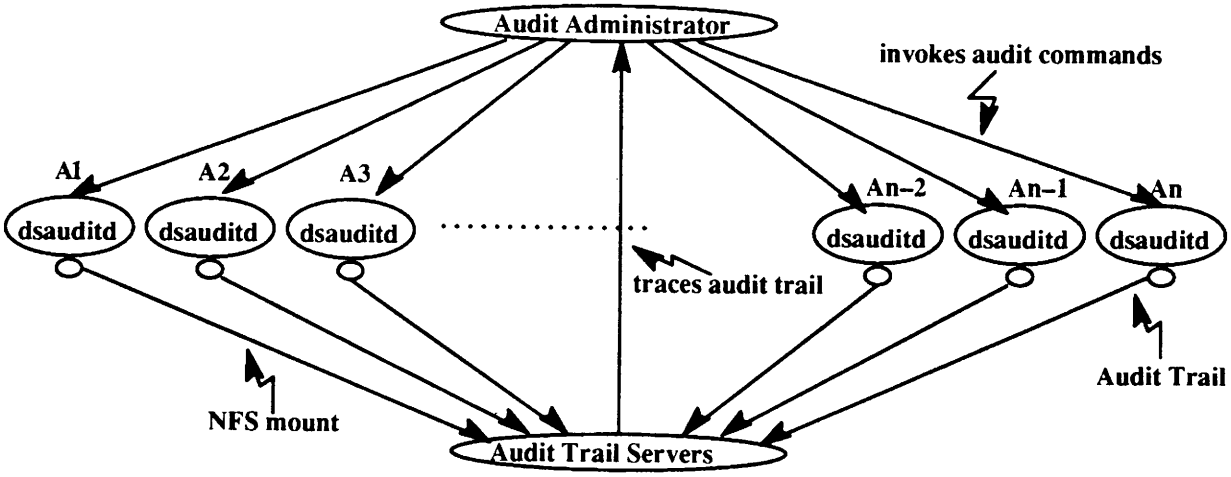


Figure 5: *A Distributed Audit Mechanism*

5. Implementation

We have implemented three sets of secure RPCs for distributed audit, distributed SMIT and ACL management, respectively. Based on the secure RPCs, we also have implemented a distributed audit daemon `dsauditd`, a distributed SMIT daemon `dsmitd`, a distributed ACL management daemon `dsacld`, and a command `xadministrator` to support the central administrator's role. The `xadministrator` is an application based on the Motif/X interface and secure RPCs. An instance of the interface of `xadministrator` for an environment of seven RISC System/6000 systems is shown in Figure 6.

Note that the systems under the control of a distributed-system administrator may not necessarily be connected within the same network. As a matter of fact, the systems shown in Figure 6 are configured into several networks, as shown in Figure 7.

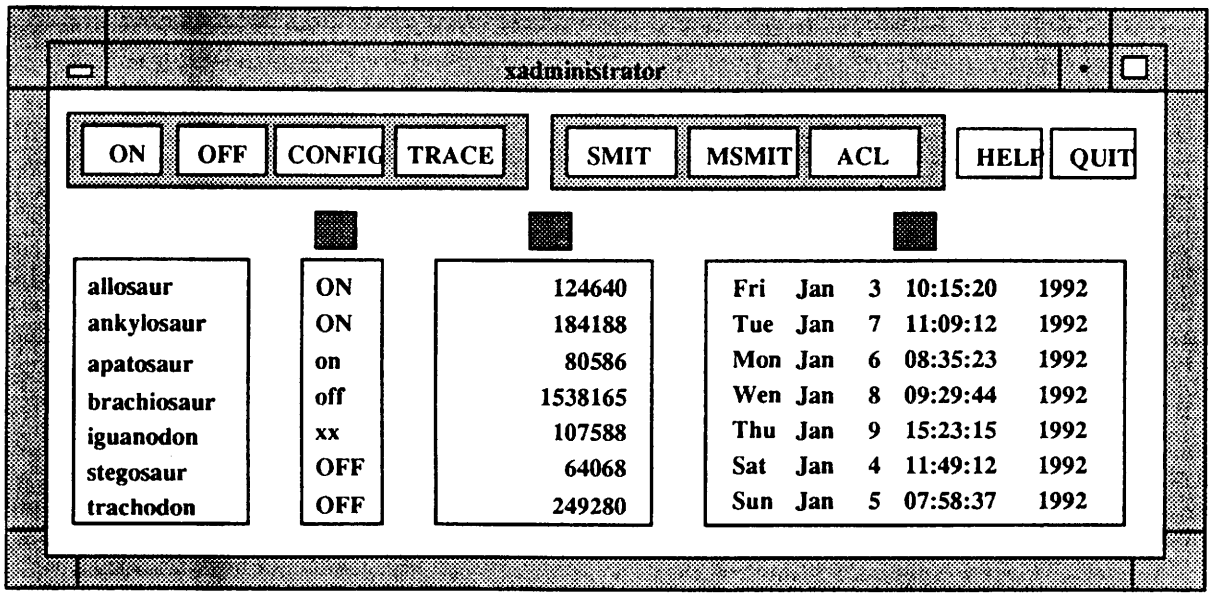


Figure 6: *Distributed System and Security Administrator*

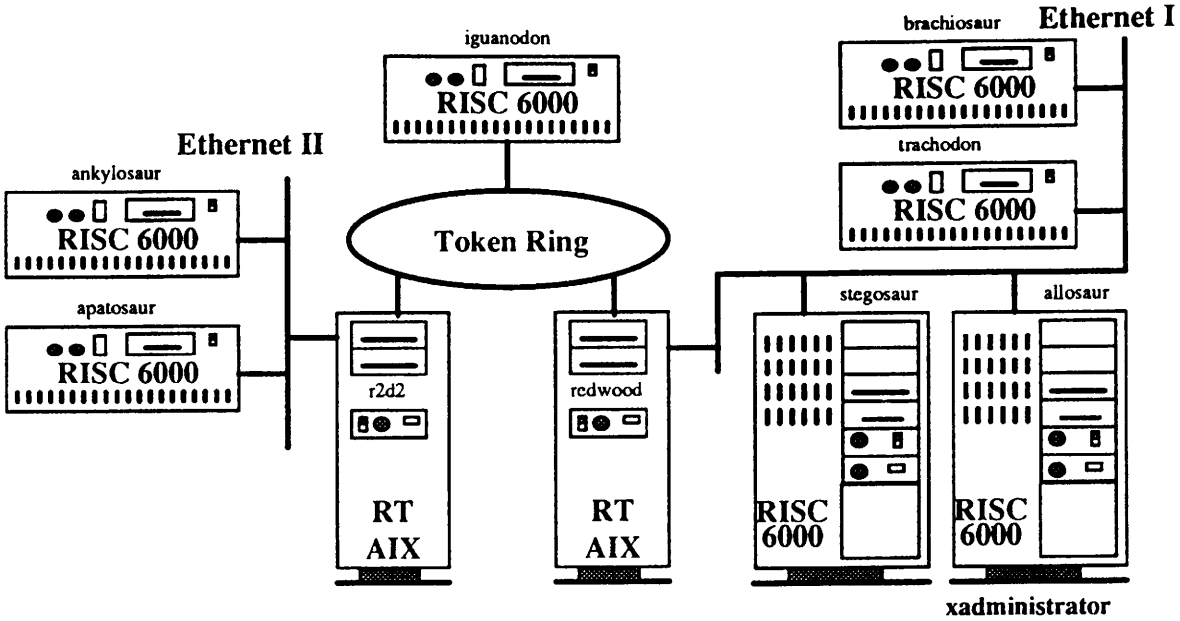


Figure 7: *Network Environment for* `xadministrator`

In Figure 6, there are two sets of main selection menus. The set of menus that contains "ON", "OFF", "CONFIG", and "TRACE" buttons is designed for distributed audit, the other for distributed SMIT and ACL management. MSMIT is a version of SMIT that provides a Motif/X-based interface. Each main selection button is associated with a pull-down list of hosts which are controlled by the `xadministrator`. Each host selection may lead to subsequent submenus or dialog boxes.

The `xadministrator` can turn on/off auditing of any or all hosts through "ON/OFF" buttons. The "CONFIG" button can be used to configure (1) audit trail servers, (2) the high-water-marks of audit trail size, the local audit trail filesystem and the central audit trail filesystem, and (3) audit classes on a per-user, per-group or per-system basis, where an *audit class* is defined as a subset of audit events. Consequently, the `xadministrator` can instruct the audit subsystem to collect the audit records of certain audit events for a specific user, or the users of a specific group, or all the users in the system. The function of the "TRACE" button is to trace an audit trail or selected audit trails through a query dialog. The four panels under the main menus show host names, audit status, audit trail sizes, and the time the auditing was turned on/off, respectively. Audit status and audit trail sizes are updated periodically. The buttons above those two panels are used to instantly update the data in them. The button above the time panel can be used to show system-clock discrepancies and to sychronize each host's clock with that of the local host. The purpose of clock synchronization is to satisfy the Kerberos requirement for loosely synchronized clocks and to maintain the consistency of the timestamps of audit records to allow accurate tracing of users' activities.

Note that the distributed audit system encompasses an integration of stand-alone audit subsystems, so that both the central audit administrator and the local audit administrator can manipulate the audit subsystem. If an action of the central audit administrator changes an audit status, the audit status is displayed in capital letters such as ON and OFF shown in Figure 6; otherwise, it is shown in small letters. Any intervention of the local auditor will immediately signal the central auditor. If the audit rpc daemon of a host was running and is now out of service, its status is marked as "xx." If a distributed audit daemon has never been reached, its status is left blank.

Distributed SMIT and MSMIT support the same functions but different types of interfaces. Distributed SMIT provides a keyboard-driven interface, while distributed MSMIT offers a mouse-driven X interface. To select an object for ACL management, the `xadministrator` can search the file tree of a remote filesystem like a local filesystem. To simplify ACL management, the ACL administrator can browse and change not only the ACL of a single object, but also objects in the same directory or objects of a specific application. We create a file `app.conf` to store the definitions of system applications. Each application occupies a stanza in the file, so that `dsacld` can discover the list of files in each application.

6. Migration to the OSF DCE Platform

Our experimental secure RPCs are based on NCS and Kerberos, which are subsets of OSF DCE, except that we use earlier versions as our platform. We can easily replace our secure RPC with the DCE authenticated RPC [Som90a]. NCS supports local and global location brokers (LLB and GLB) to provide the identity and location information of objects utilizing RPC interfaces. In OSF DCE, the LLB is replaced by the Remote Procedure Call Daemon, and the role of the GLB is simply replaced by the DCE Directory Service [OSF91a]. Also, we can take advantage of the DCE Distributed File Service (DFS) to replace NFS.

The structure of AIX ACLs is different from that of DCE ACLs. Each DCE ACL includes owner privilege and denial entries, inter-realm authorization and optional extensions. A DCE ACL entry contains the type of the ACL entry, the entry class, a set of access modes, an optional key and optional application-specific entries. DCE ACLs support four types of ACL entries: simple, key, foreign key and extended. A simple ACL contains the ACL type and no key; a key entry contains a key that identifies the principal or group to whom the entry applies; a foreign key entry contains a key that identifies both a principal or group and the realm of administrative authority from which the principal or group certification will be accepted; an extended ACL is application-specific and requires the interpretation of a specific data manager. To migrate ACL management to DCE ACLs, we would have to modify the `xadministrator`'s interface to support various types of ACL entries.

7. Conclusions

The trend toward distributed system and security management is aimed at reducing management efforts while still maintaining security, especially when the number of systems is increasing and these systems are geographically distributed. The prototype described here, which supports a central administrator and uses secure remote procedure calls, is an example of this trend.

Acknowledgments

We are grateful to Debby Yakov of IBM for her insightful suggestions and discussions on this paper. We also acknowledge Wen-Der Jiang and Tom Tamburo of IBM for their support of this effort.

References

[Bur90a] M. Burrows and M. Abadi, "A Logic of Authentication," pp. 18-36 in *ACM Transactions on Computer Systems* (Feb. 1990).

[Del88a] C. A. DellaFera, M. W. Eichin, R. S. French, D. C. Jedlinsky, J. T. Kohl, and W. E. Sommerfeld, "The Zephyr Notification Service," pp. 213-219 in *Proceedings of the 1988 USENIX Winter Conference*, Dallas, Texas (Feb. 1988).

[Din87a] T. H. Dineen, P. J. Leach, N. W. Mishkin, J. N. Pato, and G. L. Wyant, "The Network Computing Architecture and System: An Environment for Developing Distributed Applications," pp. 385-398 in *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, Arizona (June 1987).

[Dye88a] S. P. Dyer, "The Hesiod Name Server," pp. 183-189 in *proceedings of the 1988 USENIX Winter Conference*, Dallas, Texas (Feb. 1988).

[IBM90a] IBM, *AIX Version 3 for RISC System/6000: General Concepts and Procedures*, 1990.

[OSF91a] OSF, *DCE Application Development Guide*, March 1991.

[Ros88a] M. A. Rosenstein, D. E. Geer, Jr., and P. J. Levine, "The Athena Service Management System," pp. 203-211 in *proceedings of the 1988 USENIX Winter Conference*, Dallas, Texas (Feb. 1988).

[San85a] R. Sandberg, "Design and Implementation of the Sun Network Filesystem," pp. 119-130 in *Proceedings of the 1985 Summer USENIX Conference*, Portland, Oregon (June 1985).

[Sch86a] R. W. Scheifler and J. Gettys, "The X Window System," pp. 79-109 in *ACM Transactions on Graphics, 5:2* (April 1986).

[Som90a] B. Sommerfeld, *A Mechanism Independent API for Authentication with NCS*, 1990.

[Ste88a] J. G. Steiner, C. Newman, and J. I. Schiller, "Kerberos: An Authentication Server for Open Network Systems," pp. 191-202 in *Proceedings of the 1988 Winter USENIX Conference*, Dallas, Texas (Feb. 1988).

[Tsa90a] C. R. Tsai, V. D. Gligor, and M. S. Hecht, "Potential Pitfalls of a Distributed Audit Mechanism," pp. 91-103 in *Proceedings of the 1990 EurOpen Autumn Conference, (also available as IBM Gaithersburg Technical Report 85.0098)*, Nice, France (October 1990).

[Tsa91a] C. R. Tsai and V. D. Gligor, "Distributed Audit with Secure Remote Procedure Calls," pp. 154-160 in *Proceedings of the 1991 IEEE International Carnahan Conference on Security Technology*, Taipei, Taiwan (October 1991).

The European Forum for Open Systems

EurOpen

Owles Hall
Buntingford
Hertfordshire, SG9 9PL
United Kingdom

Telephone +44 763 73039
Facsimile +44 763 73255
E-mail europen@EU.net