Large Scale Porting through Parameterization

David Tilbrook, Russell Crook – Siemens Nixdorf Information Systems Ltd.

ABSTRACT

The advent of open systems and standards, while beneficial, has not eliminated the difficulty of maintaining and transporting large scale software systems across many varying platforms.

In this paper we discuss the need and criteria for a effective porting strategy, one that allows the rapid and inexpensive retargeting of large scale software systems to many widely varying platforms while not compromising the integrity of that software on any previously supported platform.

"*Getting Tigger down*", said Eeyore, "and *not hurting anybody.* Keep those two ideas in your head, Piglet, and you'll be all right."

A. A. Milne, *The World of Pooh*, 1957, pp216, McCelland & Stewart Ltd.

A key component of any porting strategy is the methodology used to determine, represent, use, and validate specifications of the target system's characteristics and site or system dependent build and run time controls. The standards efforts (e.g., POSIX, ANSI C) are attempting to eliminate the large number of discrepancies that exist among systems today. However, the problem will always exist, for reasons that are discussed.

Hence, the main objective of this paper is to present and justify the methodology that we use.

This methodology is in production use on several commercial products in Sietec. Its benefits include relieving the programmer from the burden of needing detailed knowledge of all the idiosyncrasies of the various target environments. It is sufficiently powerful that it accommodates many flavours of BSD, System V, and DOS.

Introduction

Porting is important for a vendor in the open systems market. There are many reasons for this:

- Rapid advances in technology are creating new platforms at an astounding rate. It is essential that existing software be made available on new platforms as quickly as possible.
- Being able to port to existing customer equipment has clear financial and marketing benefits.
- Heterogeneous networks are becoming both larger and more common. The vendor's software product must run, and run well, in these environments.
- Large scale portability allows deploying the software on platforms with best price/performance.
- The reliability of the code is improved, as the differing environments provide different checks and constraints on the software.
- Widespread portability gives a leverage on testing. If the software works in some environments but not others, attention can be more quickly focused on the relevant areas. Additionally, different environments may have different testing tools. Exposing the problem on a platform with better testing tools can lead to more rapid repair.

All of the above are needed for both mature products and those currently undergoing large scale development. Responsiveness to market need is critical for competitive reasons in this environment. This means that the porting process must be fast, inexpensive, and robust. That the process should be fast and inexpensive should not need any further explanation or justification. In fact both these criteria will be sacrificed if necessary to ensure the process is "robust".

To explain what we mean by a "robust" porting process, assume that there is a software system called **Z** and that there are three platforms (*alpha*, *beta* and *gamma*) on which **Z** is to be supported. **Z** was ported to *alpha* in the past, but has not been or cannot be reconstructed or tested for sometime. **Z** is currently being maintained and tested on *beta*, and is under continuous development. **Z** has never been ported to *gamma*, but the sales department has told an important customer that **Z** already runs on it.

Our problem is then to port **Z** to *gamma*, quickly and completely, while ensuring that:

1) the modifications made to port **Z** to *gamma* do not adversely affect the ongoing

development on *beta*, yet can be easily and reliably integrated into the source once the port has been completed;
2) development done on *beta* (also promised by the salesman) can be added to *gamma* quickly and reliably without requiring any additional work beyond recompilation and test;
3) there is a high degree of confidence that the modifications made to port to *gamma* and enhance **Z** on *beta* will work on *alpha*.

In this paper, we will describe part of our porting strategy – how we specify the target system's characteristics – and explore some of the ramifications for the overall strategy, particularly with respect to achieving robustness as described above.

Why adherence to standards is not sufficient

The advent of software standards for open systems (POSIX, ANSI C) has improved the situation, but has far from solved it. The problem of developing and maintaining software on multiple platforms persists. There are many reasons for this persistence:

1) Standards compliance cannot be enforced and is frequently weak. If non-compliance is found, the software being ported is forced to adapt, not the other way around.
2) The standards themselves are often moving targets, and, despite the best of intentions, cannot be complete.
3) Any sizable system requires the specification of a large number of controls and settings that depend of factors far beyond the scope of any standard (e.g., **-O** vs. **-g**, the directory into which the system is to be installed).
4) Standards usually define a minimal system and we need to be able to use extra "non-standard" facilities offered by the platforms that might improve performance or security.
5) There are still a large number of potential clients using non-standard platforms, that we do not want to ignore.

Hence we believe that adherence to standards is not sufficient, thus, the problem of the specification, determination, use, and validation of the platform and configuration dependent variations between systems must be solved.

The next section discusses our environment, principles we believe to be important, examples of specifications that must be handled, some constraints on a solution, and the basics of our approach.

Some of this work parallels the goals and objectives of Larry Wall's **metaconfig**, and Glenn Fowler's **#include <feature/*.h>** systems. However, our requirements and constraints are sufficiently different to require yet another solution, as will be discussed later in the paper.

Our environment

A brief description of our environment and the challenges to be faced:

We have approximately a dozen software engineers working on various libraries, daemons, and utilities comprising some 2000 C source files and 500,000 lines of code. The developers tend to do their development and testing on only one or two of the available platforms.

Over the past three years, there has been an average of 50 files changed a day.

These changes have been made and tested on all of the internal platforms averaging six different platforms, and nine different configurations.

In the past year, the software has been moved to ten new environments (six in the last four months). Some of these ports have been on very short notice.

Finally, most development is done on platforms that are not (officially) supported in the released product.

Assuring functional consistency and robustness is a challenge. We require a consistent and comprehensive porting strategy that works well within this environment because we have to "port" and test fifty deltas a day to nine different configurations, while ensuring that the changes will not break any of the other dozen or so supported platforms.

How we solved the problem

Principles

The following are fundamental principles of our porting strategy:

1) We do not port the software itself; instead, we configure a *platform base* or portability layer on which all code is based.
2) Testing must be frequent and widespread. If the portability layer is correctly configured and changes to product software uses that layer correctly, then a successful test of a change on a single platform **should** be sufficient to ensure that the change will be semantically correct on all platforms. Obviously testing on only one platform is not sufficient in practise, therefore we test all nine standard configurations continuously.

To facilitate using these principles, we also mandate

- The use of exactly the same application source files for all platforms (the "one true source"), which in turn means
- avoiding **#ifdef** in application code – especially those that deal with system dependencies. This applies to application header files as well; system dependent values should be inherited from the platform base.

Types of problems encountered

There are many different types of platform differences that give rise to porting difficulties. Some are listed below, with common examples:

Include file problems: Differences in location information (are the *open()* flags in *file.h*, *fcntl.h*, or even provided at all) incompatibilities amongst vendor headers (multiple, distinct definitions of *NULL*), ordering dependencies due to lack of idempotency, etc.

Different names for the same function: *strchr()* vs. *index()*, *bcopy()* vs. *memcpy()*, etc.

Standard libraries that aren't: For a program that uses terminal capabilities, do you need *–ltermcap*, *–lterminfo*, *–lcurses*, or some combination?

Functions that have different types on different platforms: *char* sprintf* (BSD) vs. *int sprintf* (Sys V).

Presence or absence of a capability: Is *lstat()* available? Does *rm* of a symbolic link delete the link or the file behind it?

Runtime environment: Is the user's login name in *$LOGNAME*, *$USER*, or even available?

Construction differences: Is *ranlib* available? Is *__STDC__* defined, and can you believe it if it is?

Bugs: If the *rename()* function exists, does it work, and how well does it work? [1]

Differing tool interfaces and semantics: Is the debugging flag for *cc*, *–g* or *–gx*? Is the *ar –o* flag supported?

Attributes of a Viable Solution

There are several attributes that a solution within this problem domain must possess:

Extensibility: It has been our experience that every new port introduces a new variation that has not been seen in any of the previous ports. To preserve portedness of old systems over time despite changes we must therefore extend capabilities without breaking old ports.

Recreatable: We must preserve port information for old systems over time so that we can recreate that port or extend as required as new requirements arise.

Locale independence: The mechanism must be host, site, and user independent. For example, we construct our DOS version on a UNIX system. This implies that no aspect of any specific host is needed to maintain this mechanism.

Ease of use: Given the rapidity with which new ports must be done, it is necessary to have a mechanism that allows easy addition or corrections of these parameters. This also encourages ready experimentation.

The extensibility criteria dictates that this cannot be a fully automated process, with all relevant information determined at compile or run time. Additionally, some values cannot be determined automatically, as they are almost a matter of taste and local custom. Such preferences should be specified with the same mechanisms as the platform constraints.

Configuring the Portability Layer

The foundation of the portability layer is a set of configured header and data files. The portability layer also contains a compatibility library, run time configuration tools and techniques and data, and a highly configurable construction system used to perform system constructions, but all these components are built using the foundation.

The portability layer foundation is built from the following ingredients:

1) A parameters file for the target system to be constructed.
2) A set of prototypes for files to be configured.
3) A program called *strfix* which, for a given prototype and the parameters file produces the configured information.

Descriptions of each of these ingredients follows:

The parameters file: The parameters file contains the configuration or discrepancy specifications as *name/value* pairs. The name is a upper case C identifier, and the value is an arbitrary string. Inclusion of other parameter files is supported to allow the inheritance of common or base systems values, as well as other shared information. An annotated example is provided in a later section.
Multiple specifications for a given name are allowed, with the last specification taking effect.

Prototype files: A prototype is a standard text file with embedded strings of the following forms:

```
@<name>@
@<name><operator><argument>@
```

where `<name>` is a possible parameters file setting, and `<operator>` is used to indicate special interpretation such as ":default" to specify a default value.

strfix: The program **strfix** reads a prototype file, replacing embedded **@<name>...@** strings with corresponding values from the parameters file. All other text is passed through unchanged.

The process

Creating the portability layer foundation is simply a matter of applying **strfix** against all the prototype files using the parameters file to produce a set of configured files which are copied to their proper locations.

Validation

Validating the portability layer is done, in part, by using it to compile and install the portability layer tools, which are then applied to rebuild itself. However, this is not an exhaustive test, consequently there are a number of regression tests that attempt to provide complete coverage.

This process is obviously dependent on the correct functioning of the **strfix** program. To minimize the chances that a new platform will force a change to **strfix**, its functioning has been kept very simple and easily tested.

Requirements of the construction system to support this strategy

This strategy has obvious implications for both the use of header files and the characteristics of the construction system. These will be addressed in a later section.

Prevention of gratuitous timestamp propagation

Obviously, every file in the portability layer depends upon the parameter file. Just as obviously, lots of programs depend on the portability layer. Therefore, unless other steps are taken, a cosmetic change in the parameters file would result in the complete and gratuitous reconstruction of the entire system.

Aids in preparing the parameters file

Whereas other approaches try to probe the system to determine the settings of various system values (and then automatically use these values), we do very little interpretation of the host environment, other than a program to extract required manifests from *sys/param.h* — a file we are anxious to avoid.

Such probe programs are particularly vulnerable to unanticipated values (or new parameters) forcing a coding change in the probe program, which then makes it difficult to assure that the new program would work correctly on previously ported systems.

Nearly all of our values depend on the parameters file. Instead of probing, we have tools to help the user to prepare and correct the parameters file.

Construction system implications

We are highly dependent upon having a construction system that will guarantee that a construction rule will be automatically applied whenever it is necessary – dependencies are automatically tracked and a changed dependency list or recipe forces reapplication.

The second requirement is that we adopt a programming style that uses our generated header files in lieu of the system provided header files.

If a discrepancy arises in a standard host header file, this ensures that we do not have to change the host header file. Hence we provide header file wrappers for all system header files that are used in the application code.

Note that this provides the necessary insulation from system dependencies that allows the programmers to ignore the underlying header structure. They need only use the generated header files rather than the system headers directly.

Annotated Examples

To describe in full the parameterization system would require the inclusion of a lot of documentation. Therefore, the following brief annotated examples are presented to clarify some of the issues presented in this paper[1].

The following is the parameters file for the optimized, X11R4 BSD4.3 side configuration of our product.

```
# SID @(#)mips4.5b-nix 1.18 ...
include      DefaultConf
include      Sites/snitor
include      Platforms/mips_4.3b
CONFIG       mips
C_OPT        -O -systype bsd43
HOSTNAME     helium
OPTIONS      DTMACH NO_MAN
RDIST        # nixtdc:/u/dtree/mips
XSYS_VERSIONX11R4
```

The *include* lines act as one would expect. Note that the last setting specified is the one that takes effect, therefore the setting for OPTIONS will override those specified in the default specification file *DefaultConf*.

Also note the presence of an SCCS SID line. This file is source and is subject to all the normal source controls. This file is the only specification used to parameterize the construction and this file is the only one that will differ from source used to build any other configuration. To build and install the specified configuration the only required initial human action is to specify this file to the initial configuration setup command. Once that specification has been stated, the file is an inherent part of the source and is subject to the same dependency tracking and rules as any other source file. A change to the file itself or any of its component files will result in the rerunning of any of the construction processes that use it as an input.

The following lines are a subset of the *TreeConfig* prototype file which itself is a *strfix* input file used to configure the construction process.

```
# SID @(#)TreeConfig.D 1.11 ...
# This file configured from @__FILE__@
OPTIONS @OPTIONS@
@{ @BLIT:false@
```

[1]Some example lines are truncated to ensure that they fit within the two column format requested by the programme committee.

```
@| true
         BLIT true
@| false
@| *
   @! BLIT(@BLIT@) must be true, ...
@}
XSYS_VERSION @XSYS_VERSION@
```

When this file is processed by *strfix*, given the example parameters file, the following will be output:

```
# SID @(#)TreeConfig.D 1.11 ...
# This file configured from /n3/...
OPTIONS DTMACH NO_MAN
XSYS_VERSION X11R4
```

The *@OPTIONS@* and *@XSYS_VERSION@* in the original strings in the prototype file have been replaced by the settings specified in the parms file. The `@{` through `@}` lines are a case statement based in the value of `@BLIT:false@`. If the *BLIT* parameter is `true`, `false` or unspecified (defaults to `false`) the lines immediately following the `@| true` or `@| false` line are processed up to the next `@|` or `@}` line. The `@|` arguments are one or more shell-like regular expressions, hence `*` will match everything, thereby processing any value that is not `true`, `false`, or unspecified. The `@!` string causes *strfix* to abort with a diagnostic thereby providing a quick and fool-proof check that the parameter is one of the legitimate values.

This is not an untypical example of a prototype file. Many are not C source and many are themselves used to configure other aspects of the system. The BLIT parameter is also a good illustration of a fundamental principle of our approach. It is used is one and only one place in the prototype files and it need not appear in any parameters file other than one for a system that indeed supports a BLIT. Thus no previous configuration file needs to be changed. Furthermore, the addition of this parameter will not change any existing generated file unless the BLIT parameter's value is true, thereby ensuring that no previous port is broken.

A typical parameters file, when the includes are unfolded, contains the settings for about 130 parameters. The number varies according to the target system as only variations for the default values are usually specified in the parameters files themselves and new parameters are created during most ports. This sounds intimidating, but for the most part just including the base system file (e.g., bsd4.[123], unix5.[0-4]) is sufficient to get started. The validation process quickly finds inappropriate settings that are fairly easily fixed by adding the appropriate override to the platform file. To list all the parameters is beyond the scope of this paper. However, they can be roughly partioned into the following groups:

Site information: the site addresses and phone numbers used within various packages to build new source files;

System configuration: the name of the system, the flags to be used to compile it (e.g., **-O** vs. **-g**); the target location for the installed product; the name to be used to access the installed product (frequently not the same as the target location[2]).

Header file mappings: the header file used to retrieve various types, settings, defines, etc.;

Tool names and availability: the name of the compiler, loader, *yacc*, etc. to be used in the construction process, as well as the names (and full path if desirable) of various useful facilities whose names may differ across systems (e.g., *rsh* vs. *rcmd*);

Compiler characteristics: Can one use prototypes? Can one use prototypes for function pointers? Some compilers can do one but dump core when one attempts the other. Is *char* signed or unsigned? How does one specify the use of an alternative C preprocessor? Tools are provided to help the installer find out answers to some of these questions, but we never depend on them working correctly.

Routine mappings: which routine should be used to copy memory? To move memory? Is there a *dup2* routine?

Supported bugs: Does *rename* work? and so on.

Before leaving this examples section, our solution to a particularly difficult header file mapping is illustrated.

The specification of which header files contain the definitions of the *tm* and the *timeval* structs is one which cannot be based on loose specification of the base system (e.g., bsd vs. unix5). The two structs are frequently used in the same source and the inclusion of the appropriate header files (e.g., *sys/time.h* and *time.h*) is not a matter of simply *#include*ing both. In some situations one includes the other and the second is not idempotent (that is, it may not be included twice). On other systems both files have to be included in a specific order and may or may not require the previous inclusion of *sys/types.h*, which itself is sometimes not idempotent.

We solve this problem by creating our own *envir/time.h* header file which contains the specification of both required structs, plus those prototypes that are sometimes not specified. This is a configured file that uses two parameters: **TIMEVAL_H** and **TM_STRUCT_H** which respectively name the header files to be used to access the definitions for the *timeval* and *tm* structs

[2]We test systems before we install them.

respectively, with one minor caveat. If one header file enforces the inclusion of the other, the other is assigned the empty string. The following lines are included as part of our prototype *time.h* file:

```
/*
 * include our own idempotent
 * types.h wrapper
 */
#include<envir/types.h>
/*
 * include timeval header
 * if necessary
 */
@TIMEVAL_H#i@
/*
 * header containing struct tm
 * if not same as above
 */
@{ X@TM_STRUCT_H:sys/time.h@X
@| X@TIMEVAL_H@X
@| *
   @TM_STRUCT_H#isys/time.h@
@}
```

The `@<name>#i...@` string causes *strfix* to output a `#include` line if the parameter is defined or a default value is specified. The above may look baroque, but the resulting file just consists of the comments and the required includes. Also note the absence of *#ifdefs*.

Evaluation.

How does this strategy meet the criteria given earlier?

- The simple text file is portable, and **strfix** is simple and almost immune to environmental idiosyncrasies.
- We can add new parameter files that allow us to use old configurations with new perturbations. For example, a quick look at the machine (or its documentation) will indicate whether we should start with a BSD 4.x base, or System V, or something else.
- Usually no application C code or header file changes are required. We will not discuss creating of a portability library to compensate for system deficiencies; it is a simple application of the parameter file to select or provide appropriate functionality or provide name mapping.
- In the last year, we have ported our major software product to ten new platforms. Six of these were in the last four months, for an average of one port every three weeks. These ports included our first encounters with X11R4 and System V Release 4. The porting itself took an average of a day, with testing taking a week. During these porting efforts, development efforts continued at their normal rate on the application code.
- Through this mechanism we have virtually eliminated the use of **#ifdefs** in C code. During the last year, with its ten ports, there were no **#ifdefs** added anywhere in the application code or header file. Those **#ifdefs** that remain are either in taste or capability selection (e.g., build with debugging code in or out) or are based on settings in the parameter file. When it is required to alter a setting for a specific platform or host, the parameter file is changed, and not the C code. The importance of this to our efforts should be obvious:
  + Since we do not change the code, we virtually eliminate the possibility of breaking an old port.
  + By not creating platform-specific blocks of code, the regression tests remain accurate.
- The ability to use these techniques in a cross-compilation environment allowed porting the code to a DOS/Windows environment without forcing an unfamiliar development environment on the developers. A probe-based mechanism would not have readily permitted this. As importantly, it permitted application of multiple developers to the porting effort without jeopardizing source code consistency. Although this effort did in fact require substantial code changes due to the radical environmental differences between DOS and UNIX (filename syntax, environment variables, unusual C environment), these changes were applied to the one true source for both the DOS and UNIX environments, and then continually tested in both environments on an ongoing basis.

How well does this work?

Very well indeed. This approach has succeeded in all UNIX platforms tried to date (over fifty at last count). Our approach is now being used to support our application code in a DOS/Windows environment.

Products at other Siemens sites have adopted this approach by converting their software to use our portability layer. One such product, which had previously only worked on one platform was ported to three new platforms in two months.

The application developers are, in our experience, very happy to suffer the style and coding practises in exchange for not having to understand the arcane topology of all the systems[3] encountered. The effort required of the developers to use the

[3]This became *very* important during the DOS/Windows work. Running the regression tests under DOS/Windows was all the contact most of the developers had with the DOS environment.

portability layer is very small by comparison.

Most ports of our software (up to and including running of the automated regression test suite) do indeed take only a day or so. The exceptions occur when the applications have made non-portable assumptions (e.g., using X11R3 and porting to a platform with X11R4). Even in these cases, the ability to rapidly mutate the platform base layer to adapt to the new environment without invalidating previous ports[4] is of great benefit.

It is worth noting another large benefit – anything that is a text file can make use of the platform base. This obviously applies to applications written in languages other than C, but it also applies to shell scripts, construction system recipes, application data files (e.g., X resource files), etc.

It is worthwhile casting our recent experiences into the **Z** software mold mentioned previously, with past platforms *alpha* which may not be testable for a while, present platforms *beta* on which most ongoing development occurs, and future platforms *gamma*. Platforms that were *gamma* systems have become *alpha* systems since the equipment was here for short term evaluation only; some of the future *gamma* systems[5] will be new corporate platforms, and will become *beta* platforms. Some of the *alpha* systems are now *beta* systems, as equipment has been repaired or returned. During all this activity, software development continued at close to its normal rate.

In such an environment of flux, it is clear that we cannot afford to freeze application development, port the code by modifying it and then test it on the relevant platforms. Attempting to modify the application for porting purposes while letting application development proceed has obvious quality problems. We are convinced that the more usual approaches would not suffice in our environment.

What would be done differently?

- Documentation of an individual parameter and the expression of its use is currently weak. This is especially important since each new port (so far) has introduced new parameters.
- Comprehensive validation of a parameter's setting is sometimes delayed until late in a system's construction due to prerequisites. We need a better framework for specifying and executing parameter regression tests.
- Similarly there needs to be an easy to use framework for adding aids to help the user determine the correct settings, although most of the time a simple guess is sufficient.

[4]In this case, the changes have to be tested both in an old (X11R3) and new (X11R4) environments to be considered safe.

[5]We have three of these anticipated in the next six weeks.

Other Approaches

Config

This paper would be incomplete without discussing a widely used approach to the problem addressed by this paper, namely Larry Wall's *config* and *metaconfig* systems. *config* is sufficient for the distribution of a small shareware system to users who are willing to invest the required time and effort to fix it when it goes wrong. However, *config* cannot be considered as the mechanism to be used to do large professional systems due to a number of deficiencies.

- It requires user interaction, which is time-consuming and error-prone, and most importantly cannot be expressed as an administered source file, something that we believe to be essential. It also rules out the possibility of rerunning the configuration stage as part of any construction, again something that we believe is essential.
- The use of probes to determine the appropriate settings for the equivalent of our parameters has several drawbacks. When a probe is in error, the probe mechanism itself must be altered to accomdate the fix. Frequently the probes themselves are constructed with implicit assumptions about the target system. When these assumptions are incorrect, major surgery is often required. Hence, previously valid probing assumptions may be upset by the change to the mechanism, jeopardizing the validity of previous ports.
- The probe information cannot be managed historically. The probe evaluation depended on the state of the host system at the time *config* was executed and therefore its replication cannot be guaranteed. Regenerating a configuration for an unavailable platform (say, an older release of an operating system) for support purposes becomes problematic.
- The addition of a new parameter or correction of an old one has severe performance implications when constructing large systems. Our experience with *config* is limited[6], but *config* users who do use it stated that the actual products of the *config* process are two configured files (one for C and the other for sh). This means that the simple correction of a parameter will require the entire recompilation of the complete system, something that one cannot afford when running four to five thousand compiles across nine different platforms.

[6]One of the authors tried to use it to install *rn* but gave up when it failed. The cost of trying to adapt a much hacked 1800 line shell script was considered to be far more than the benefits of being able to use *rn*. For comparison, we normally use a ten line text file to install 800 programs and 30 libraries.

- The dependence on the host system eliminates the possibility of doing cross compilation, something that we must have to adequately deal with inadequate systems such as DOS. Our approach is to provide probes that may be used to determine and/or test the appropriate value for a parameter, but to never incorporate its running as part of the construction process.

Fowler's #feature mechanism

Glenn Fowler, the creator of the fourth make, has an approach to the configuration process that to our knowledge and that of one of his colleagues has not been documented. Briefly, the use of:

```
#include <feature/name.h>
```

within a C program, and the dynamic dependency tracking of *make4*, will trigger, if necessary, the creation of the named header file by running the associated probe. This shares some of *config*'s weaknesses with respect to the dependence on automated probes and the host environment, but avoids some of *config*'s major flaws. As stated the system is, as yet, undocumented but shows promise.

Conclusions

Porting is extremely important to us, and our techniques have proven to be profitable for us.

This paper addresses only one aspect of the porting problem – that of the specification of parameters for a system. We have been led to this strategy by the requirements of today's environment of open systems and need of rapid ports. The parameterization and characterizations of systems in this way has proven sufficient to handle all porting problems we have seen in the past ten years. Indeed, expectations are now so high we have the situation that all ports are expected to be done in a day, even though they may involve substantial rework and testing to deal with new challenges.

Bibliography

[1] David Tilbrook, *rename(''open'', ''swinging_to_and_fro'');* EurOpen Newsletter, 1991.

[2] David Tilbrook & John McMullen, *Washing Behind Your Ears: Principles of Software Hygiene, EurOpen Nice Conference, Oct. 1990.*

Author Information

By the time this paper is published, David Tilbrook will have started his new position as Vice President, Technology at CS Computing Services in Toronto. For the last three years he has been a consulting engineer at Sietec and the manager of the Software Hygiene Research group. His primary research interest is Software Hygiene and the Software Process. David has served as the programme chair for four EurOpen conferences, a Usenix conference and the Software Management Workshop, and is the chair for the 1993 Uniforum Canada conference on Software Hygiene. In 1985, David was awarded an Honourary Lifetime membership to EurOpen (and a much treasured Swiss army knife).

David's new address is unknown at this time due to the fact that his company is relocating to as yet at unknown location in downtown Toronto but he may be contacted via Russell or `dt@sni.ca` for the time being.

Russell Crook has worked at Sietec since 1988 as a Project Leader within the Imaging and Data Storage groups, working on problems in large scale data storage and management. He has a long standing interest in computer chess, including work on the Treefrog chess program, which won the 1974 CACM computer chess tournament.

Russell may be reached at:

Sietec Open Systems Division
2235 Sheppard Avenue East
Suite 1800
Willowdale, Ontario
Canada
M2J 5B5

or at `rmc@sni.ca`.