PACT

PORTABLE APPLICATION CODE TOOLKIT

Table of Contents

Last Updated: 12/6/2002

Installing PACT

Disclaimer

Part of this work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Introduction

In a computing landscape which has a plethora of different hardware architectures and supporting software systems ranging from compilers to operating systems, there is an obvious and strong need for a philosophy of software development that lends itself to the design and construction of portable code systems. The current efforts to standardize software bear witness to this need.

PACT is an effort to implement a software development environment which is itself portable and promotes the design and construction of portable applications. PACT does not include such important tools as editors and compilers. Well built tools of that kind are readily available across virtually all computer platforms. The areas that PACT addresses are at a higher level involving issues such as data portability, portable inter-process communication, and graphics. These blocks of functionality have particular significance to the kind of code development done at LLNL. That is partly why the general computing community has not supplied us with these tools already. This is another key feature of the software development environments which we must recognize. The general computing community cannot and should not be expected to produce all of the tools which we require.

At the time that PACT was begun other groups at other institutions were also beginning efforts to address some of these issues. That is the reason that one can find some of the conceptual parts of PACT in the work done at some of these institutions. One of the strengths of PACT is the degree of integration which the component parts can achieve. This affords the application developer a relatively small and efficient set of tools. For this reason, the total capability of PACT is unmatched in other software systems.

Organization of PACT

In its current conception, PACT is a set of nine tools. Eight of these are implemented as libraries, and two of them provide both libraries and executable applications. PACT is entirely coded in C but was designed with a view toward support of other languages, notably FORTRAN.

The design of PACT was and is functionally driven. The main idea is to identify generic blocks of functionality independent of considerations of language and specific applications. When that has been done, the application program interface (API) for a particular block of functionality, e.g. inter-process communication, naturally emerges. With the API specified, the implementation proceeds fairly naturally. The most important concept in making this approach work is that of abstraction barriers. The API defines an abstraction barrier between the general purpose functionality and the applications built on top of it. The underlying implementation can be changed without necessitating changes to the applications using it. This goal has not always been achieved, but the fact is that it remains as a goal and PACT is always evolving toward that goal. Part of the success which PACT enjoys from the perspective of meeting its requirements is due to the use of abstraction barriers. For people interested in a marvelous exposition of these ideas, I highly recommend Abelson and Sussman’s book, Structure and Interpretation of Computer Programs which has been one of the core textbooks in MIT’s Computer Science curriculum. All of PACT has been strongly influenced by that book.

PACT includes the following software libraries and applications:

SCORE	a low level, environment balancing library
PML	a math library
PPC	a process control library
PDB	a portable binary database management library
SCHEME	an interpreter for the Scheme dialect of the LISP language
PGS	a graphics library
PANACEA	a simulation code development system
ULTRA	a 1d data presentation, analysis, and manipulation tool
SX	Scheme with extensions (PDBView and PDBDiff are SX programs)

These tools facilitate the development of software which need portable support for graphics, inter-process communication and networking, and portable binary data handling and storage. It is ideal for the kinds of software development which are done at academic, applied scientific, and engineering institutions.

The point cannot be overemphasized that these tools are part of a coherent approach to solving portability problems. They are modularized which allows application builders to use only the parts needed for particular projects. They cooperate together so that there is maximal code reuse in PACT with its attendant benefits (maintainability, shareability, and flexibility). They are very generic in that each module is constructed to address a specific block of functionality, e.g. graphics, and address it in a general and comprehensive way. The PACT tools are also layered. The higher level tools utilize the functionality of the lower level ones. Roughly, the PACT tools fit in the following hierarchy:

ULTRA SX
PANACEA
PGS SCHEME PPC
PML PDB
SCORE

ULTRA and SX are at the top of the hierarchy and SCORE is at the bottom. This in turn sits on the more raw, vendor supplied, hierarchy:

COMPILER
OPERATING SYSTEM
HARDWARE

Parallel PACT Issues

PACT can be built with or without knowledge of parallel programming considerations. With regard to SMP parallelism it can be: unaware of threads; simply thread safe; or actively using threads. It turns out that two common SMP standards, pthreads and OpenMP, do not always work well together. This seems to be mainly due to vendor implementation issues. Nevertheless, when you are building PACT you should be aware of this so that you can build the PACT libraries to suit your needs. There are two "visible" places that this appears. You will find several config files which differ only by how the SMP parallelism is controlled. Also the naming convention of the libraries tells you which kind of parallelism or which SMP standard is used.

Config files whose names end with -t refer to SMP parallelism with the pthread standard. Config files whose names end with -s (with a few exceptions) refer to SMP parallelism compatible with OpenMP.

Similarly, if PACT is configured for pthreads, then the libraries will have names like, libscore_r.a where the '_r' follows a common convention employed by vendors. On the other hand, if PACT is configured for compatibility with OpenMP compilers/libraries, then the libraries will have names like, libscore_s.a where the '_s' indicates the distinction. A non-SMP version of PACT will have libraries with names like, libscore.a.

Building PACT in Parallel

PACT can be built very quickly by building the packages in parallel. The package organization permits each package to be built in a separate process possibly on a separate machine of the same architecture.

To build PACT from pact/manager you do:

dsys build [-p] [-o #]

There are four main scenarios for a parallel build.

1) A single machine with multiple CPU's. In this case each package will be handed to one of a pool of N CPUs. You would then issue the command:

dsys build -p N

2) A cluster of single CPU machines with a numeric naming scheme. For example suppose you have the foo cluser with hosts foo1, foo2, foo3, and so on. You would then execute the command:

dsys build -p N -o 2

This would use foo2, foo3, ..., fooN+2 for the build.

3) A cluster of multi-CPU machines. Now suppose you have the bar cluster with 2 CPUs each and hosts bar1, bar2, and so forth. The command to use now is:

dsys build -p 4x2 -o 1

This will build two packages each on bar1 thru bar4.

4) A number of machines with the same architecure. Suppose foo, bar, and baz are all of the same CPU, operating system, compilers, and so on. Here you would do:

dsys build -p 3 -o foo,bar,baz

Running Distributed Parallel Applications

Although MPI is the prevalent distributed parallel programming tool, there are a plethora of variations on how to run a distributed parallel application. It can be very confusing for the user of a portable distributed parallel application to remember a number of front end with different arguments.

Furthermore, there is a serious deviation from POSIX and ANSI standard I/O behavior in most MPI implementations which prevents emitting a line of text that is not terminated with a newline ('\n'). Un-terminated lines remain buffered until a newline is sent. This behavior is uneffected by the use of setbuf (for example) which standards dictate causes all output to be immediately sent. This means that your distributed application cannot print a prompt and expect have the response in the same line. This really seems to annoy users (as it should).

Many vendors deny that this is a bug and the community at large has permitted them to get away with this.

In any event, PACT has a solution to both problems. The do-run utility runs with every MPI implementation that we have been able to access. It runs the same way in each case, so the user only has to worry about one command. It is also part of the solution to the MPI I/O buffer bug. The other part involves the application using the SCORE function, SC_mpi_printf instead of fprintf.

The do-run usage is:

```
do-run [-b] [-d] [-h] [-n #]
       [-o ] [-p #] [-q] [-t #] [-v] args

b - run in batch
d - run the code under a debugger
h - this help message
o - output file for DP runs
n - number of nodes to use (MPI)
p - number of processes to use (MPI)
q - suppress do-batch messages in logs
t - number of threads to use (SMP)
v - verbose mode
```

Building Shared Libraries

The PACT libraries can now be built as shared libraries. To do so chose a configuration with the "-sl" qualifier (see configs.std/README) or create your own config file (see SharedLibArg below). After PACT is configured do the following:

```
-> pact shared
```

This will build static PACT libraries, PACT utilities, and then the shared versions of the PACT libraries. This is a new and evolving capability and there are apt to be some rough edges. Please let us know of any problems which you experience so that we can fix them.

The following sections will give a brief overview of the purpose of the component parts of PACT.

Overview

As mentioned above, the PACT system is both modular and layered in design. In each layer the modules are independent of one another, but each layer can depend on any module in the layers beneath it. This consideration has a strong influence on where certain functions reside. In general, if a function has no strong attachment to a functional grouping (e.g. graphics), it goes into the lowest level layer to which all of the other parts of PACT that might be able to use that function have access. In spite of one’s expectations, this doesn’t mean that functions tend to gravitate to SCORE. There seems to be a real, natural level for almost every function in PACT.

To avoid name conflicts and to aid application developers who use PACT, there are some naming conventions followed (although not fanatically so) in all of the coding of PACT. Most functions, external variables, and macros begin with either a two, upper case character designator followed by an underscore or an underscore followed by the designator and an underscore. The main functions, variables, and macros of the API lack the leading underscore. Those functions, variables, and macros which are very useful but require greater understanding of the libraries or are simply not necessary to the main API have the leading underscore. Many software systems would not document this latter category of objects and protect the API with the freedom to change these around. I document these in PACT because they are too useful to brush off.

The list of designators for the PACT tools is:

Package	Designator	Example
SCORE	SC	SC_strsave
PPC	PC	PC_open
PML	PM	PM_decompose
PDB	PD	PD_read
SCHEME	SS	SS_eval
PGS	PG	PG_set_viewport
PANACEA	PA	PA_simulate
ULTRA	UL	UL_set_graphics_mode
SX	SX	SX_draw_plot

SCORE: A Low Level Foundation

SCORE (System CORE) has two main functions. The first and perhaps most important is to smooth over the differences between different C implementations and define the parameters which drive most of the conditional compilations in the rest of PACT. Secondly, it contains several groups of functionality that are used extensively throughout PACT.

Although C is highly standardized now, that has not always been the case. Roughly speaking C compilers fall into three categories: ANSI standard; derivative of the Portable C Compiler (Kernighan and Ritchie); and the rest. PACT has been successfully ported to many ANSI and PCC systems. It has never been successfully ported to a system in the last category. The reason is mainly that the "standard" C library supplied with such implementations is so far from true ANSI or PCC standard that PACT would have to include its own version of the standard C library to in order to work at all. Even with standardized compilers life is not dead simple. The ANSI standard leaves several crucial points ambiguous as "implementation defined". Under these conditions one can find significant differences in going from one ANSI standard compiler to another.

SCORE's job is to include the requisite standard headers and ensure that certain key standard library functions exist and function correctly (there are bugs in the standard library functions supplied with some compilers) so that, to applications which include the SCORE header(s) and load with SCORE, all C implementations look the same. This is a tall order, but in practice once SCORE has been successfully compiled only the areas of graphics, IPC, and binary data handling require special consideration! This has more of an impact on some programmers than on others. Those who prefer to specify only the exact headers to be included in each source file will find SCORE and PACT unusual. At the expense of a slight increase in compile time, the most commonly used headers are always included. This is crucial to getting the C implementation independence.

Typically, the SCORE header scstd.h includes the following:

ANSI	stdlib.h stddef.h stdarg.h float.h
PCC	sys/types.h varargs.h malloc.h
Both	limits.h stdio.h string.h math.h ctype.h signal.h setjmp.h time.h

The single header, scstd.h, smooths over most of the generic problems that arise because of implementation defined behavior in the host C implementation. The remainder of the PACT sources ultimately include scstd.h. This strategy has been extremely successful for PACT and applications which use PACT.

There are basically three other areas which SCORE functions address: memory management; hash table management; and extended string handling.

See Also: SCORE User's Manual

Dependent PACT Libraries: None

PPC: Portable Process Control Library

In operating systems which permit multiple independent processes to run and communicate there are several different notions of how the communication should work. An example is the difference between pipes and sockets. Even worse in the case of pipes is the fact that BSD Unix and System V Unix have substantial differences in the implementations of the pipe mechanism.

The abstract requirements of applications are more straightforward. A process should be able to spawn a subordinate process and open communications with it. The two processes should be able to exchange messages. The processes should be able to monitor one another's status.

PPC defines and implements an API which embodies that simple model. Its interface is made as analogous as possible to the standard C file interface so that the experience which programmers have with files relates directly to interprocess communication. The details of whether pipes or sockets are used are substantially irrelevant to the application developer so long as the necessary functionality is there.

PPC is the least mature of the PACT tools. Ultimately, in conjunction with the binary data handling mechanisms of PDBLib, it will provide, in addition to interprocess communications (IPC) functionality, a highly efficient remote procedure call (RPC) capability. This will permit and facilitate the development of distributed applications in a portable manner.

See Also: PPC User’s Manual

Dependent PACT Libraries: SCORE, PDB

PML: Portable Mathematics Library

There are a great many numerical math libraries that are widely available. The PACT math library is more aimed at providing some useful mathematical structures, functions which deal with them, some geometry routines, interpolation functions, and the odd function from the standard C library which doesn’t work right on all platforms. It should be viewed as a supplement to some of the other available math libraries rather than a would be competitor. At a certain level, the portability issue does arise in that PML should work (and work the same) on all platforms while some math libraries are not so portable or available. The other parts of PACT which need a certain math routine cannot be held hostage to the availability of other libraries. These then are the basic considerations behind PML.

PML defines several structures to take advantage of C’s capabilities in that area. The effect that this produces on applications which use these mechanisms is one of simplifying the passing of data among functions, clarifying interfaces, and preventing certain classes of bugs. The main structures which PML defines are: complex numbers, matrices; sets; and mappings.

The matrix structure encapsulates an array of data with up to 2 dimensions. The set structure describes a collection of data elements in terms of type, set dimension, dimension of the elements, and other information. The mapping structure describes the generalization of a function which is an association of elements in a domain set with elements in a range set. The mapping is a nested structure containing pointers to two set structures.

These all are designed to be as mathematically faithful as possible. That means that they represent the fundamental abstract objects after which they are named. The sets and mappings in particular have a surprising application in the area of visualization. They turn out to be a good medium of exchange between data production systems such a simulation codes, data storage and transmission systems, and data visualization systems.

PML includes several useful matrix solvers. Some of these use the matrix structure and some of the sparse solvers do not. At this point, the math functions in PML have been more driven by specific needs than any systematic attempt at completeness. Perhaps in the future PML will be extended in this direction or perhaps a portable shell over some other math libraries will evolve.

See Also: PML User’s Manual

Dependent PACT Libraries: SCORE

PDB: Portable Binary Database Library

Binary data takes up about a factor of 3 less space than ASCII data and requires about a twentieth the time to read and write. Despite the obvious advantage of storing and transmitting binary data the majority of data is moved between dissimilar hardware/software platforms via ASCII format. The reason is that different hardware/software platforms use a variety of representations of binary data which are tuned to the hardware, operating system, or compiler on the platform.

PDBLib is a library which addresses the issue of binary data portability. It also offers unique facilities for handling structured data including pointers. Almost all solutions to the data portability problem employ a hub and spoke conversion method (e.g. Sun's XDR) in which data is converted on output to a neutral format and converted to the host format on input. This is exactly what happens with ASCII data, that is ASCII is the neutral format. The conversion process is what takes most of the time difference between doing I/O with ASCII data and binary data. PDBLib's approach only does conversions when necessary. In that way an application which will use its own data files or will give them to a machine of the same architecture will not do any data conversions thus saving the majority of the time difference between ASCII and binary I/O.

PDBLib supports structured data with a mechanism similar to the C struct mechanism. Since data objects as diverse as FORTRAN common blocks and C structs are representable by this means, PDBLib's solution is a very general and flexible one. Furthermore, PDBLib supports the use of pointers, so that entire data trees can be read or written with a single PDBLib call. PDBLib is unique in this capability.

It is important to keep in mind that although there are many systems for handling portable binary data (e.g. HDF, netCDF, CDF, and FLUX) all of them have a particular set of requirements to address. PDBLib's requirement is to be as general and flexible as possible. PDBLib is also the only tool of its kind which is an integral part of so far reaching a set of tools. Its integration with the PACT tools is a significant part of its flexibility.

PDBLib is especially designed to be inherently able to work with other languages. It concerns itself with issues such as array index ordering, and call by reference/call by value argument passing.

Another goal consistent with the portability of PDBLib is to be small enough to be useful on a small machine. The library has been used to real advantage on DOS and Macintosh PC's.

Parallel PDB

PDBLib now also support both SMP and distributed parallel I/O. See the PDBLib Users Manual for details.

See Also: PDBLib User's Manual

Dependent PACT Libraries: SCORE

SCHEME: An Interpreter for the Scheme Dialect of LISP

The LISP programming language is one of the most powerful computer languages ever developed. Somewhat ironically, it is also one of the simplest to implement, at least to a certain level. Many applications would benefit by being able to provide their users with the ability to extend the application via a macro or scripting mechanism. LISP turns out to be ideal for this purpose. The language is syntactically simple and can be molded to meet a variety of needs. It also offers more power than most applications care to provide to their users, but one would rather be in that position than in having an underpowered extension mechanism.

Scheme is a lexically scoped, properly tail recursive dialect of the LISP programming language. The PACT implementation is described abstractly in Abelson and Sussman's book, Structure and Interpretation of Computer Programs. It features all of the "essential procedures" described in the Revised, Revised, Revised Report on Scheme (Jonathan Rees and William Clinger, ed) which defines the standard for Scheme.

SCHEME is implemented as a library, however, a small driver delivers a stand alone Scheme interpreter. This is useful for people who want to learn the language as well as people who want to prototype algorithms (one of LISP's strong points).

The PACT implementation features a reference counting incremental garbage collector. This distributes the overhead of memory management throughout the running of Scheme code. It also tends to keep SCHEME from trying to grab the entire machine on which it is running which some garbage collection schemes will attempt to do.

See Also: SX User's Manual

Dependent PACT Libraries: PML, PPC, SCORE

PGS: Portable Graphics System

One of the biggest headaches for portability is graphics. The objective evidence is that the field is immature. One cannot exactly say that there are no graphics standards. The real problem is that there are too many standards. Until such time that the world settles down, there will be the need for a tool like PGS.

PGS is an API that is independent of the underlying host graphics system. All of the graphics portability headaches are confined to PGS and applications which use PGS are completely portable. PGS currently sits on top of X Windows on UNIX platforms, GRAFLIB on NLTSS, Quickdraw on Macintoshes, and Microsoft's graphics library on DOS platforms.

PGS takes a least common denominator approach regarding what graphics functionality it supports. The goal is to run on the widest variety of machines. This lets out high level graphics capabilities such as real time 3D rotations which depend on specific hardware. On the other hand, any rendering capability that can be implemented with reasonable efficiency in software is fair game for PGS. This model will almost certainly change in time as both graphics hardware and software evolve and become ubiquitous across platforms.

PGS adopts a model in which graphics devices such as display windows and PostScript files are represented by a structure which contains all of their state information. Then PGS can manage an arbitrary number of devices simultaneously and any picture can be drawn to any device.

PGS also structures display surfaces with a viewport defined in normalized coordinates, an enclosing boundary where axes are drawn which is defined as a set of offsets from the viewport window, and a world coordinate system attached to the viewport. The enclosing boundary is useful for obtaining a standoff between rendered data such as line plots and the axes used to measure the rendering.

PGS supports both line and text drawing primitives, line and text attributes, and bit maps for handling images and other cell array data. Most functionality in PGS is either primitive operations such as moves and draws or at a very high level such as axis drawing and the rendering algorithms that it supports. These rendering algorithms have two interfaces: one for "raw" data; and one for PML type mappings. This gives a great deal of flexibility to the application developer.

PGS has the following rendering algorithms currently: 1D line plots; 2D contour plots; 2D vector plots; 2D image plots; 3D wire frame mesh plots (for 2D data sets); and Grotrian diagram plots.

See Also: PGS User's Manual

Dependent PACT Libraries: PML, SCORE

PANACEA: Portable Simulation Code Development Environment

An analysis of numerical simulation codes will demonstrate a great deal of commonality. The goal of PANACEA is to implement everything that is generic to "all" numerical simulation codes once and for all. In doing this a model of numerical simulation codes is necessary as the context for defining a collection of services which defines the generic aspects of numerical simulation codes.

Some people like to refer to tools such as PANACEA as back-planes or shells. I think of PANACEA as an environment for code development. PANACEA is not so highly structured that it must be taken in its entirety or left alone. The notions of shell and back-plane are suggestive of that higher degree of structure. PANACEA can be used as just a library of routines (up to a point).

PANACEA provides the following general categories of services: definition and control of a database of simulation variables and packages; access control to the database; support for initial value problem generation; source and initial value data management; output/plot request handling; simulation control; units and conversion handling; and error handling.

PANACEA draws heavily on PDB and PML to provide its services. This gives it tremendous leverage in handling the database, state files, and output/graphics data files. This close coupling keeps the system small and compact while providing a fairly large range of services.

See Also: PANACEA User's Manual

Dependent PACT Libraries: PDB, PML, SCORE

ULTRA II: 1D Data Presentation, Analysis, and Manipulation

ULTRA II is a tool for the presentation, analysis, and manipulation of 1D data sets or curves (i.e. y = f(x)). It supports operations such as addition, subtraction, multiplication, and division of curves; applying functions to curves (e.g. sin, cos); Fourier transforms; and shifting and scaling of curves. The user can control the range and domain of the displayed data as well as such attributes as line color, style, and width.

Either ASCII or binary data can be written or read by ULTRA II. ULTRA II uses the SCHEME interpreter to provide users with a very powerful mechanism to configure and extend ULTRA II to suit their needs. This also keeps the code small by not having to compile in everybody's pet extension. Thus ULTRA II is extremely flexible.

See Also: ULTRA II User's Manual

Dependent PACT Libraries: SCHEME, PGS, PDB, PML, PPC, SCORE

SX: Scheme with Extensions

SX is perhaps the ultimate PACT statement. It is simply SCHEME plus the other parts of PACT. A more precise way to describe it is as a dialect of LISP with extensions for PGS, PDB, PML, and PANACEA. What this yields is an interpretive language whose primitive procedures span the functionality of all of PACT.

Like the SCHEME implementation which it extends, SX provides both a library and a stand alone application. The stand alone interpreter is the engine behind applications such as PDBView and PDBDiff. The SX library is the heart of TRANSL a tool to translate data files from one database format to another. These examples demonstrate the extreme power of the PACT design. The modularization and layering make it possible to use the PACT components like building blocks.

In addition to the above mentioned functionality, SX contains functionality which is the generalization of that found in ULTRA II. This means that as the development of SX proceeds, an SX driven application will be able to perform arbitrary dimensional presentation, analysis, and manipulation tasks.

PDBView is a tool to examine the contents of PDB files. This may be done either by printing all or part of the entries in a PDB file or by displaying the data graphically. As such PDBView already performs the basic presentation tasks which the ULTRA II successor will have.

PDBDiff is a tool that does a binary comparison of two PDB files. The user can specify the precision of floating point comparisons, but by default PDBDiff uses the lesser of the two files' precision in floating point comparisons. For example, comparing a CRAY generated file with 64 bit floats and a SUN generated file with 32 bit floats will be done to 32 bits not 64. PDBDiff can be run in either a batch or interactive mode. In batch mode it can be thought of as very similar to the UNIX diff utility. In interactive mode, the user can browse the files and compare all or part of arbitrary file entries.

See Also: SX User's Manual

Dependent PACT Libraries: SCHEME, PANACEA, PGS, PDB, PML, PPC, SCORE

PACT Documentation

The individual components of PACT have their own documentation which is intended to give application developers and end users complete information about that component. It is important to remember that the parts of PACT are integrated so that they depend on one another. Consequently, it is assumed that readers of any PACT component document may need to refer to the documents for any component on which the desired component depends.

SCORE	none
PML	SCORE
PDB	SCORE
PPC	SCORE, PDB
PGS	SCORE, PML
SCHEME	SCORE, PML, PPC
PANACEA	SCORE, PML, PPC, PDB, PGS
SX	SCORE, PML, PPC, PDB, PGS, SCHEME, PANACEA
ULTRA II	SCORE, PML, PPC, PDB, PGS, SCHEME

The list of PACT Documents is:

PACT User's Guide	UCRL-MA-112087	Current Document
SCORE User's Manual	UCRL-MA-108976 Rev.1	
PPC User's Manual	UCRL-MA-108964 Rev.1	
PML User's Manual	UCRL-MA-108965 Rev.1	
PDBLib User's Manual	M-270 Rev.2	
PGS User's Manual	UCRL-MA-108966 Rev.1	
PANACEA User's Manual	M-276 Rev.2	

ULTRA II User's Manual	UCRL-MA-108967 Rev.1
PDBDiff User's Manual	UCRL-MA-108975 Rev.1
PDBView User's Manual	UCRL-MA-108968 Rev.1
SX User's Manual	UCRL-MA-112315

Interested readers can obtain these documents from the author. Other sources that readers may find useful are:

Abelson and Sussman, *The Structure and Interpretation of Computer Programs*, MIT P
Abramowitz, Stegun, *Handbook of Mathematical Functions*, Dover.
Klinger, et. al., *The Revised4 Report on the Algorithmic Language Scheme*, MIT.
Knuth, D.E. *The Art of Computer Programming, vol I - III*, Addison-Wesley.
Press, Flannery, Teukolsky, and Vetterling, *Numerical Recipes in C*, Cambridge Pre

Availability of PACT Software

Platforms

The complete PACT distribution is currently running on the following platforms:

```
Linux on Intel and Compaq Alpha CPU's
Solaris on SPARC and Intel CPU's
IBM RS6000 under AIX
HP 700 under HPUX
Apple Macintosh family
IRIX64
INTEL 80x86 under DOS
```

The host graphics systems used are:

```
the X Window System (X11 R3 and up)
Quickdraw
Microsoft Graphics library
```

In the past parts of PACT have been ported to the following platforms:

```
DEC VAX under ULTRIX and VMS
INTEL 80x86 under SCO XENIX
```

These ports were not completed due to lack of access to the necessary computers.

Obtaining the PACT Distribution

The PACT distribution is available on the Internet. You can find it and more information at http://www.llnl.gov/def_sci/pact/.

Installing PACT

MAC OS

Installing PACT on a Macintosh system is unfortunately not easy. The process is very laborious due to the lack of useful tools to help automate it. The popular IDE tools currently have no means of operating from an easy to maintain pure text description of a code system. Building "project" files is an onerous task with a graphical interface. Consequently, building PACT is time consuming and involved on these platforms.

MS WINDOWS

Installing PACT on an MS Windows platform is unfortunately not easy. The process is very laborious due to the lack of useful tools to help automate it. The popular IDE tools currently have no means of operating from an easy to maintain pure text description of a code system. Building "project" files is an onerous task with a graphical interface. Consequently, building PACT is time consuming and involved on these platforms.

UNIX

Installing PACT on a UNIX system is comparatively easy to do. The process has been highly automated and refined over many years.

In the standard distribution each of the components resides in a subdirectory of the main pact directory. In addition there is a subdirectory called manager which contains various configuration files and shell scripts to automate the installation of PACT. In the following discussion, it will be assumed that there is a base directory called pact which is a subdirectory of some other directory such as /usr/local/src, so that when the directory score is referred to as pact/score, the directory /usr/local/src/pact/score is meant.

To make it as easy as possible to port/maintain/install PACT on a variety of systems, PACT uses the MAKE utility to control compilation, loading, and other tasks related to management of the system. Due to differences in operating systems and site installations, the PACT system uses a three part Makefile. The first part has site dependent information and this will be the same for all of the components of PACT. This part is called make-def and it is generated by pact-config during the configuration process. The second part contains the specifics of the components and therefore each component of PACT has a separate file called pre-Make. In pact/score, pact/pml, pact/ppc, and so on there is a file called pre-Make which contains dependencies particular to those parts of PACT. The third part contains a number of macros and dependencies common to all of PACT but which depend on the operating system. This file is called make-macros and it too is generated by pact-config during the configuration process.

I have found the mechanism to be very simple and portable. I have used it in preference to such alternate mechanisms as Imake because it is much easier to manage and depends only on standard UNIX facilities.

Building PACT is a highly automated process. It sometimes breaks down for a variety of reasons such as required utilities being in some unknowable directory. In such a case you will have to do some work by hand if only to determine what is necessary to make the automatic build function properly. If you must build PACT by hand, you will have to edit/re-create the Makefiles in each sub-directory, and run make over them by hand. This should be done in the order stated above: SCORE, PML, PPC, PDB, PGS, SCHEME, PANACEA, ULTRA, and SX. Hopefully you will not have to do this.

IMPORTANT: You may use any directory you choose to do the install. One of your own (e.g.; ~myhome/pactdir) or one in /usr/local (e.g.; /usr/local/src). Of course you must have read/write/execute permissions for the install to work. You do not want to have a subdirectory of the directory in which you have put the distribution file named pact. The installation process will mangle it thoroughly if you do!

1) Obtain the distribution file (e.g. pact98_3_17-src) by some means and put it in a place such as /usr/local/src as follows:

```
-> cd /usr/local/src

-> ftp                  OR

   cp /pact98_3_17-src .

-> chmod a+x pact98_3_17-src

-> mv pact98_3_17-src pact-src
```

For the rest of this document the distribution file will be called pact-src (you may or may not want to rename it to this).

If you have obtained the PACT documents as well as the distribution (e.g. pact98_3_17-doc.tar.gz), you may wish to get them out and start printing the ones of interest before proceeding on to the installation which may take minutes to hours.

```
-> mkdir pact-docs

-> mv pact98_3_17-doc.tar.gz pact-docs/pact-doc.tar.gz

-> cd pact-docs

-> gzip -d pact-doc.tar.gz

-> tar xvf pact-doc.tar

-> lpr <document>.ps ...
```

2) Run the distribution file (the C Shell must be installed on your system). Running the distribution file without arguments will give you a usage summary. Select the arguments you wish and run it again. For example:

```
-> pact-src
               .
```

```
              <usage summary>
                       .
                       .

    -> pact-src -i /usr/local/pact auto
```

This does the following things: configures the distribution, compiles all packages, runs all of the PACT tests, and finally installs it in the designated place (unless "-i none" is used). On a fast machine all of the above will take a few minutes and on a slow machine it will take a few hours.

NOTE: The directory designated for installation, if any, must contain at least the following subdirectories: include, bin, lib, and scheme and it and the subdirectories must be writable by you.

As the installation proceeds, most messages from component programs and scripts are routed into a file called Build.Log. When the installation completes the name is changed to <site>.BUILD and the file is moved up one directory level so as to be out of the way in case another attempt or a build on a different system is to be done next. You may wish to check this file for problems or information about the process itself. There should be no error or warning messages in a normal installation.

3) If the installation fails consult the README file in the pact/manager directory or skip down to the PACT Manager section below.

4) Now that PACT is installed. You need to set two environment variables in order to use ULTRA, SCHEME, SX, PDBVIEW, or PDBDIFF. The SCHEME and ULTRA environment variables must be set to point to the directory in which the scheme files were installed. The default for this directory is /usr/local/scheme and in the above example it would be /usr/local/pact/scheme (see the SCMDir variable if in doubt).

```
    -> setenv SCHEME /usr/local/pact/scheme

    -> setenv ULTRA /usr/local/pact/scheme
```

These should probably be put into your .cshrc or your .profile so that they appear when you log into your system.

The PACT documentation has additional information about these environment variables. This example will serve to get you started.

PACT Configuration

PACT Manager

This section details some the details of the management of the PACT sources. Most PACT users will not need to read this material unless interested. It is included here to help you build PACT on a UNIX system if the automatic build process fails or to give you some information which may be helpful when building on an MS Windows or Macintosh system.

I refer to the scripts and programs which automate the check out, compilation, testing, check in, and distribution of a code system as the manager for that code system. PACT has a manager. All of its scripts are C Shell scripts. It has a master script called *dsys* which provides a user level interface for the scripts which make up the manager. Each operation is referred to as a target. The main targets for *dsys* are:

dsys -h

This prints a summary usage guide for dsys.

dsys config

This target configures the PACT sources for the platform as described by the command line arguments. It makes the make-def and make-macros files (described below) appropriate to the platform as well as any other scripts and programs needed to build PACT such as the binary file translation spokes for SX.

dsys remove

This target removes all binaries (archives and executables) for the platform on which you are working currently. It will not remove binaries for a different platform being built from the same set of sources. It can be run at the same time as any other manager operation being run on a different platform.

dsys build

This target traverses the PACT source hierarchy and brings all of the binaries up to date with the sources. All archives will be brought up to date in pact/dev/*system*/lib and all executable programs will be linked in pact/dev/*system*/bin. Header files will be installed in pact/dev/*system*/include and Scheme interpreter files will be installed in pact/dev/*system*/scheme.

Each component of PACT can be listed so that it can be built and installed individually. This gives the user flexibility to build or develop only the parts he/she cares about. By default all components are built.

dsys test

PACT has an extensive test suite and this target runs the various testing scripts in the PACT hierarchy. The tests include not only pass/fail tests to let you know that all the parts of PACT function as expected but some tests give timings for performance comparisons and memory usage statistics.

There are other target which are mostly of use to PACT developers. The ones listed above should be the only ones needed while installing PACT and these only if the PACT distribution fails to build itself as described in the next section.

It is an important feature of the PACT manager and the organization of PACT that from a single set of sources, PACT can be configured, built, and tested on multiple platforms at the same time. Binaries are kept in platform specific directories and other temporary files have the platform designation, referred to as *system*, built into their names to avoid conflicts.

PACT Makefiles

The PACT manager uses a three part model of a Makefile which controls the construction of each PACT component. Each Makefile consists of: a part which is site dependent and contains all of the special paths for the entire installation process; a part which is specific to the PACT component; and a part containing common macros some of which are operating system dependent. These parts are referred to as make-def, pre-Make, and make-macro respectively. The pre-Make files reside with the PACT components whose compilation and installation they control. The make-def and make-macro files live with the manager. They are constructed in the configuration process.

Configuration File Syntax

The syntax of a configuration file is fairly simple. Blank lines and lines beginning with a '#' signal the end of a section or are ignored. There are the following main section types: MakeRule, CEnvironment, CDeclarations, and VariableDef. The sections are begun by a line containing the section type. The default section type is VariableDef.

You can have any number of instances of the main sections and their specifications are all concatenated together. Specifications in the CEnvironment and CDeclarations sections result in #define's and variable or function declarations in the scconfig.h file. MakeRule specifications allow sites to add targets and macro definitions to the make-def (and hence Makefile) files for the system. VariableDef specifications go into the construction of the make-def, trconfig.h, and spokes.scm files.

In addtion to the main section types there are the following special section types: .c.i:, .c.o:, .c.a:, .l.o:, .l.a:, .f.o:, and .f.a:. These are to define the make rules for preprocessing a C source file, compiling a C source file into an object file, compiling a C source file "directly" into an archive file (if your system's make is capable of doing so), compiling a LEX source file into an object module, compiling a LEX source file directly into an archive file (make willing), compiling a FORTRAN source file into an object file, and compiling a FORTRAN source file directly into an archive file (make willing) respectively. These sections are begun by a line containing the section name.

All sections are ended by blank or comment lines (lines beginning with '#').

Configuration Parameters Described

This group of parameters is used by the configuration building and make processes to distinguish various operating system characteristics as well as the utilities which come with a UNIX/POSIX system.

Some of these are implied by PACT coding and are listed here to give you an idea of what they are for (just in case you need to look in the code).

BuildStrategy

UNIX systems with a sufficiently smart make can save lots of disk space by building the objects directly into library archives. Use "SpaceSaving" if your make is good enough and "none" otherwise. PACT-ANALYZE will detect this for you.
Default is "SpaceSaving"

CCompiler

The name of the C compiler to be used.
Default is "cc"

CDebug

C compiler flags to specify debuggable packages.
Default is "-g"

CFLAGS

C compiler flags to be used for all PACT packages. This is for those options which do not control debugging or optimization. Use CDebug or COptimize for those purposes.
Default is ""

COptimize

C compiler flags to specify optimized packages.
Default is "-O".

CSpokes

A list of C codes binary file translation spokes to be built with SX. An NACF spoke is included with the distribution.
Default is "nacf".

FCompiler

The name of the FORTRAN compiler to be used.
Default is "f77"

FDebug

F77 compiler flags to specify debuggable packages.
Default is "-g"

FFLAGS

F77 compiler flags to be used for all PACT packages. This is for those options which do not control debugging or optimization. Use FDebug or FOptimize for those purposes.
Default is ""

FOptimize

F77 compiler flags to specify optimized packages.
Default is "-O"

GraphicsFlag

Any special graphics flag (not usually needed with X11).
Default is ""

GSYS

The host graphics system being used (almost always X).
Default is "X"

INSTALL

Place holder for either "install <options>" or "cp" depending on whether or not your system has install.
Default is "cp"

InstBase

The directory in which the PACT manager expects to find lib, bin, and include directories into which to install PACT.
Default is "/usr/local"

LDFLAGS

Special loader/linker flags.
Default is ""

LXFLAGS

Flags for C compilation of LEX generated files.
Default is ""

MachDepGraphicsInc

Special specifications for the C compiler to find graphics related include files (e.g. -I/usr/include/X11). This is not often needed.

Default is ""

MachDepGraphicsLib

Special specifications for graphics libraries to be loaded (e.g. /usr/lib/libX11.a).
Default is ""

MachDepInc

Special specifications for the C compiler to find non-graphics related include files (e.g. -I/usr/local/include). This is not often needed.
Default is ""

MachDepLib

Special specifications for non-graphics libraries to be loaded (e.g. -lbsd). Default is ""

NEED_MEM_TRACE

PACT supports a layer of memory management over the system memory manager which provides diagnostic capabilities as well as useful features for applications. On some systems, PACT can understand the workings of the memory manager sufficiently to require a smaller amount of overhead and look more deeply in the system heap. Other systems are not so and PACT must do more bookkeeping work. Set this parameter to TRUE to indicate the latter case and FALSE to indicate the former.
Default is "TRUE"

RANLIB

Place holder for either "ranlib" or "touch" depending on whether or not your system has ranlib.
Default is "touch"

RM

Place holder for "rm <options>" depending on the options your system's rm has.
Default is "rm -f"

SchemeSpokes

A list of Scheme coded binary file translation spokes to be used with SX. A netCDF spoke is included with the distribution.
Default is "netcdf"

ScmDir

A directory in which to install scheme source files. This should also be a subdirectory of InstBase.
Default is "scheme"

UPDATE

Place holder for either "ranlib" or "touch" depending on whether or not your system needs ranlib to update archives which have been copied.
Default is "touch"

C Environment Parameters

This group of parameters is used in C header files to set up and define the C compilation environment for the PACT sources. These are specified in a CEnvironment section of a configuration file.

C Language Standard

These parameters define which C standard is to be assumed when compiling PACT sources. Use at most one.

ANSI

Conforming to ANSI C (this is the default).

PCC

Conforming to the Portable C Compiler (KR 1rst edition).

FORTRAN Language Standard

ANSI_F77

Conforming to ANSI FORTRAN 77.

System Dependent Parameters Which Must Exist

The following parameters are implied by other selected options:

byte

Type "void" if supported (default for ANSI compilers), type "char" if no voids (default for PCC compilers).

UNIX

UNIX/POSIX Operating System (as opposed to DOS or MAC).

X11R4

PACT uses X11 R4 or later graphics (implied by "GSYS = X").

Other System Dependent Parameters

AIX

This indicates the presence of the IBM UNIX variant.

BINARY_MODE_R

Binary file mode parameter for read only mode. Default is "rb".

BINARY_MODE_RPLUS

Binary file mode parameter for open for update mode. Default is "r+b".

BINARY_MODE_W

Binary file mode parameter for create mode. Default is "wb".

BINARY_MODE_WPLUS

Binary file mode parameter for create for update. Default is "w+b".

BSD

This indicates the presence of BSD UNIX 4.2 or later.

DOS

This indicates the presence of IBM or Microsoft DOS.

FIXNUM

C type corresponding to FORTRAN integer defaults to long. Don't set this.

HAVE_SELECT_P

Indicate the presence of a working "select" system call iff there is no working STREAMS package available. PACT-ANALYZE detects this for you.

HAVE_SOCKETS_P

Indicate the presence of a working socket library. Default is off. If you have X windows you have this.

HAVE_STREAMS_P

Indicate the presence of a working STREAMS package. Default is off. PACT-ANALYZE detects this for you.

HPUX

This indicates the presence of the HP UNIX variant.

MAC

This indicates the presence of Apple Macintosh OS.

MPW

This indicates the presence of MPW C 3.0 or later.

MSC

This indicates the presence of Microsoft C 5.0 DOS (16 bit) or later.

Register

Some C compilers take a dim view of the use of the register allocation specifier. Set to "register" if appropriate, and "" if not. Default is "".

SGI

This indicates the presence of Silicon Graphics machines.

SYSV

This indicates the presence of UNIX System V.

THINK_C

This indicates the presence of Symantec THINK C 5.0 and later.

TURBOC

This indicates the presence of Borland C++ 3.0 DOS (16 bit) or later.

UNICOS

This indicates the presence of Cray Research C for UNICOS.

Send questions and comments to rodrigues2@llnl.gov