Configuring and Building with Python

Matthew G. Knepley
Dmitry A. Karpeev
Argonne National Laboratory

Problems with Autoconf

"Designed for small packages without package dependencies"

- No namespaces*
  - Resulting in bizarre name-mangled variables
- No hierarchy*
  - Hard to use or manipulate configuration of a dependency
- Unreadable implementation*
  - Often has hidden dependencies
- Obscure dependency structure
- Poor persistence*
- Poor extensibility*
- Unsafe

*Shell is a bad language

Configuration Pieces

- config.base.Configure
  - Base class for all configure objects
- config.framework.Framework
  - Organizes configure
- config.[compilers,libraries,…]
  - Common tests
- User modules

Base Class Interfaces

- Executing tests
- Preprocessing, Compiling, Linking, Running
  - Provide both **output** and **check** forms
  - Maintain a language stack
  - Also have methods to validate flags
- Checking for executables
- Output results
  - Defines
  - Substitutions ➔ Prototypes, Make Macros, Make Rules
  - Typedefs
  - Can be prefixed for primitive namespacing

New Output Paradigm

- All data stored in configure objects
  - Allows access through methods
- Objects retrieved using framework.require()
  - Can give None for successor
- Project frameworks are pickled in RDict
  - Objects can be retrieved from dependencies

Framework Interface I

- Maintains configure object graph
  - Use addChild() or getChild() to add a node
  - Altered with require(moduleName, successor)
- Handles output
  - Use addSubstitutionFile() and configureHeader
  - Allows output prefixes
- Can be completely pickled
- Provides help with --help, --h

Framework Interface II

- setup()
  - Inherited from Script class
  - Called recursively on children
- cleanup()
  - Substitute files
  - Output configure header
  - Produce report from child __str__() methods
  - Log filesystem actions

Configure Help

- Classes must implement
  - setupHelp(help)
- Help is a class which maintains a table
  - addArgument(section, argName, argObj)
- Example from MPI

```
help.addArgument('MPI', '-with-mpi=<bool>',                          nargs.ArgBool(None, 1, 'Activate MPI'))
help.addArgument('MPI', '-with-mpi-dir=<root dir>',                  nargs.ArgDir(None, None, 'MPI root directory'))
help.addArgument('MPI', '-download-mpich=<no,yes,ifneeded>', nargs.ArgFuzzyBool(None, 0, 'Install MPICH'))
```

Standard Configure Library

- Compiler and linkers
- Autoconf-type checks
  - Headers
  - Types
  - Functions
  - Libraries
  - Compatibility library (only for the brave)
- Python
- Backward compatibility (not recommended)

PETSc Library

- BLAS/LAPACK
- MPI, LAM
- ParMetis, HYPRE, Matlab
- Mumps, Spooles, SuperLU, UmfPack
- Sowing, c2html, lgrind
- Preferred compiler flags

User Modules

- Interface
  - Derive from config.base.Configure
  - Implement configure()
  - Add dependencies with require()
  - Run all tests with executeTest()
- Can specify with --configModules
- PETSc determines them dynamically

Simple Driver

```
import config.framework as framework

framework = framework.Fromwork(sys.argv[1:], loadArgDB = 0)
try:
  framework.configure(out = sys.stdout)
  framework.storeSubstitutions(framework.argDB)
  framework.argDB['confCache'] = cPickle.dumps(framework)
except TypeError, e:
  print 'Error in command line argument',e
except Exception, e:
  print 'Configuration crash',e
```

ParMetis Example

In __init__():

```
self.libraries = self.framework.require('config.libraries', self)
self.mpi       = self.framework.require('PETSc.packages.MPI', self)
```

In checkLib():

```
found = self.libraries.check(libraries, 'ParMETIS_V3_PartKway',
otherLibs = ' '.join(map(self.libraries.getLibArgument, self.mpi.lib)))
```

Build Pieces

- Persistence and Argument Processing
- Logging
- Version Control
- Help
- Scripting
- Building libraries and executables
- Building entire projects
- Miscellaneous

Persistence

- RDict.py
  - Dictionary interface
  - Allows typed entries
- Can be distributed
  - Local dictionary can have a parent
  - Can establish daemon servers
  - Uses pickle as the line protocol

Argument Processing

- nargs.py args.py
- Defines RDict types
  - string, bool, fuzzy bool, int, real
  - directory, library, executable
- Can run interactively or batch
- Arguments can be temporary or persistent
- ArgumentProcessor is an object with RDict
  - Defines setupArguments() in addition to setup()

Logging

- logging.py
  - Incorporates some screen manipulation
  - Allows both level and section classification
- Not crucial to use this one
  - Python logging module is fine

Version Control I

- Bitkeeper is going away
  - Moving to Darcs?
- Necessary for EVERY project
- Can be adapted for generated code
- Needs to be integrated into build system

Version Control II

- sourceControl.py
- Operations
  - edit(), add(), revert(), commit()
  - changeSet()
- Queries
  - getNewFiles()
  - getEditedFiles(), getClosedFiles()
  - getChangedFiles(), getUnchangedFiles()

Help System

- Entries have a:
  - Section, e.g. MPICH
  - Name, e.g. with-mpich-dir=<dir>
  - Type, e.g. ArgBool()
- Can produce a formatted table
  - Also reports default value

Scripting

- Provides safe executeShellCommand()
  - Has a timeout
  - Captures all output with select()
- Override setupHelp() to add arguments
  - Called from setup()
- Some import utilities

Dependency Tracking

- builder.DependencyChecker
  - Uses sourceDatabase.py
  - __call__() returns True if source should be rebuilt into target
  - update() marks source up-to-date
- Two current implementations
  - md5
  - timestamps

Configurations

- A **configuration** is script.LanguageProcessor
  - Has set of config.compile.processor.Processor
    - For example, config.compile.C.Linker
    - These hold and manage flags
- Configurations are persisted in Rdict
- Tracks output files
  - Needs some improvement

Build Operations

- Maintains language and configuration stacks
- Normal operations
  - compile(), link() (preprocess() coming)
  - Incorporates dependency checks
  - Source database holds dependency information
    - Automatic determination coming

Building Libraries

- builder.py, script.py
- Dependency tracking
- Configurations
- Build operations

Building Projects

- maker.py
- maker.Make is a skeleton
  - Has a setup-configure-build cycle
  - Fully implements configure
    - Activated by presence of configure.py
  - Allows project dependencies
    - No action taken by default

Simple Builds

- Simple make-like syntax
- Builtin suffix rules
- Builtin target types
  - Library (static, shared, dynamic)
  - Executable
- Exposes configure object
- Arbitrary code is Python (not shell!)
  - Easy to override functionality in makefile

Building SIDL Projects

- maker.SIDLMake
  - Fully implemented system
  - Cascades configure information from dependencies
  - Dependencies are built after configure
  - Integrates version control
  - Handles multiple languages
    - C, Python, C++, F77

Miscellaneous

- importer.py
  - Allows modules to use several directories
  - Allows a more dynamic path
- graph.py
  - Adjacency list representation of directed graphs
  - Traversals are all generators
  - Used for configure graph and project dependencies

References

- bk://sidl.bkbits.net/BuildSystem
  - Contains configure and build classes
- bk://petsc.bkbits.net/petsc-dev
  - Also at http://www.mcs.anl.gov/petsc
  - Show fully implemented configure system