ASE BuildSystem Manual

Matthew G. Knepley

ASE BuildSystem Manual

by Matthew G. Knepley

Table of Contents

Chapter 1. Introduction

The BuildSystem from ASE is intended to be a Python replacement for the GNU autotools. It actually encompasses somewhat more, as it supports integrated version control and automatic code generation. However, the most useful comparisons will come from **autoconf**, **make**, and **libtool**. The system is not designed to be monolithic. Thus each component may be used independently, meaning logging, configuration, and build are all separate modules which do not require each other. This allows a user to incremenetally adopt the most useful portions of the package.

Chapter 2. Configure

2.1. Configure Design Sketch

The system is based upon an autonomous unit, objects of class `config.base.Configure`, which are responsible for discovering configuration information for a particular package or purpose. The only interface which must be supported is the `configure` method, as shown below. Support for lower-level operations such as compiling and linking will be discussed in section ???.

```
class Configure () :
    def configure(self) :
```

This collection of configure objects is managed by a `config.base.Framework` object. As we will see in section ???, the framework manages all dependecies between modules and output of configure information. The framework is itself a subclass of `config.base.Configure` for which the `configure` method manages the entire configuration process. In order to associate a module with the given framework, it also provides the `require` method, discussed in section ???. Thus, the minimal framework interface is given by:

```
class Framework (config.base.Configure) :
    def require(self, moduleName, depChild, keywordArgs = {}) :
    def configure(self) :
```

This design allows user modules to be seamlessly integrated into the framework without changing the paradigm, or even any of the original code. Modules can be specified on the command line, or left in special directories. Although it is common to derive from `config.base.Configure`, the only necessity is that the user provide a `configure` method for the framework to execute.

The framework does provide the traditional output mechanisms from **autoconf**, namely `#define` statements and file substitutions, to which we add make variables and rules. However, the preferred interaction mechanism is to use member variables directly from the configure objects. This is illustrated in section ???

2.2. Running configure

The first step in running configure is to show the help:

```
bash$ framework.py -help

Python Configure Help
   Comma seperated lists should be given between [] (use \[ \] in tcsh/csh)
    For example: --with-mpi-lib=\[/usr/local/lib/libmpich.a,/usr/local/lib/libpmpich.a\]
----------------------------------------------------------------------------------------
Script:
  --help                : Print this help message                                        current: 1
  --h                   : Print this help message                                        current: 0
Framework:
  --configModules       : A list of Python modules with a Configure class                 current: []
  --ignoreCompileOutput : Ignore compiler output                                         current: 1
  --ignoreLinkOutput    : Ignore linker output                                           current: 1
  --ignoreWarnings      : Ignore compiler and linker warnings                            current: 0
  --doCleanup           : Delete any configure generated files (turn off for debugging)   current: 1
  --with-alternatives   : Provide a choice among alternative package installations       current: 0
  --search-dirs         : A list of directories used to search for executables           current: []
  --package-dirs        : A list of directories used to search for packages              current: []
  --with-batch          : Machine uses a batch system to submit jobs                     current: 0
```

The options shown will depend upon the modules loaded with `-configModules`. For instance, we will normally load the compiler module, which reveals the host of optios controlling preprocessors, compilers, and linkers.

```
bash$ framework.py -configModules=[config.compilers] -help

Python Configure Help
   Comma seperated lists should be given between [] (use \[ \] in tcsh/csh)
    For example: --with-mpi-lib=\[/usr/local/lib/libmpich.a,/usr/local/lib/libpmpich.a\]
----------------------------------------------------------------------------------------
Script:
  --help                           : Print this help message
  --h                              : Print this help message
Framework:
  --configModules                  : A list of Python modules with a Configure class
  --ignoreCompileOutput            : Ignore compiler output
  --ignoreLinkOutput               : Ignore linker output
  --ignoreWarnings                 : Ignore compiler and linker warnings
  --doCleanup                      : Delete any configure generated files (turn off for debugging)
  --with-alternatives              : Provide a choice among alternative package installations
  --search-dirs                    : A list of directories used to search for executables
  --package-dirs                   : A list of directories used to search for packages
  --with-batch                     : Machine uses a batch system to submit jobs
Compilers:
  --with-cpp=<prog>                : Specify the C preprocessor
  --with-cc=<prog>                 : Specify the C compiler
  --with-cxx=<prog>                : Specify the C++ compiler
  --with-fc=<prog>                 : Specify the Fortran compiler
  --with-gnu-compilers=<bool>      : Try to use GNU compilers
  --with-vendor-compilers=<vendor> : Try to use vendor compilers (no argument all vendors, 0 no vendo
  --with-64-bit-pointers=<bool>    : Use 64 bit compilers and libraries
  --CPP=<prog>                     : Specify the C preprocessor
  --CPPFLAGS=<string>              : Specify the C preprocessor options
  --CXXPP=<prog>                   : Specify the C++ preprocessor
  --CC=<prog>                      : Specify the C compiler
  --CFLAGS=<string>                : Specify the C compiler options
  --CXX=<prog>                     : Specify the C++ compiler
  --CXXFLAGS=<string>              : Specify the C++ compiler options
  --CXX_CXXFLAGS=<string>          : Specify the C++ compiler-only options
```

```
--FC=<prog>                      : Specify the Fortran compiler
--FFLAGS=<string>                : Specify the Fortran compiler options
--LD=<prog>                      : Specify the default linker
--CC_LD=<prog>                   : Specify the linker for C only
--CXX_LD=<prog>                  : Specify the linker for C++ only
--FC_LD=<prog>                   : Specify the linker for Fortran only
--LDFLAGS=<string>               : Specify the linker options
--with-ar                        : Specify the archiver
-AR                              : Specify the archiver flags
-AR_FLAGS                        : Specify the archiver flags
--with-ranlib                    : Specify ranlib
--with-shared                    : Enable shared libraries
--with-shared-ld=<prog>          : Specify the shared linker
--with-f90-header=<file>         : Specify the C header for the F90 interface, e.g. f90_intel.h
--with-f90-source=<file>         : Specify the C source for the F90 interface, e.g. f90_intel.c
```

The syntax for list and dictionary option values is identical to Python syntax. However, in some shells (like **csh**), brackets must be escaped, and braces will usually have to be enclosed in quotes.

The modules indicated with `-configModules` are located using PYTHONPATH. Since specifying environment variables can be inconvenient and error prone, it is common to provide a driver which alters `sys.path`, as is done for PETSc. In fact, the PETSc driver

- Verifies PETSC_ARCH
- Checks for invalid Cygwin versions
- Checks for RedHat 9, which has a threads bug
- Augments PYTHONPATH
- Adds the default PETSc configure module
- Persists the configuration in `RDict.db`
- Handles exceptions

2.3. Adding a module

As we discussed in the introduction, all that is strictly necessary for a configure module, is to provide a class named `Configure` with a method `configure` taking no arguments. However, there are a variety of common operations, which will be illustrated in the sections below.

2.3.1. Using other modules

We will often want to use the methods or results of other configure modules in order to perform checks in our own. The framework provides a mechanism for retrieving the object for any given configure module. As an example, consider checking for the `ddot` function in the BLAS library. The relevant Python code would be

```
import config.base

class Configure(config.base.Configure):
  def __init__(self, framework):
    config.base.Configure.__init__(self, framework)
    self.compilers = self.framework.require('config.compilers', self)
    self.libraries = self.framework.require('config.libraries', self)
    return

  def configure(self):
    return self.libraries.check('libblas.a', 'ddot', otherLibs = self.compilers.flibs,
                                fortranMangle = 1)
```

The `require` call will return the configure object from the given module, creating it if necessary. If the second argument is given, the framework will ensure that the returned configure object runs *before* the passed configure object. Notice that we can use the returned object either to call methods, like `check` from `config.libraries`, or use member variables, such as the list of Fortran compatibility libraries `flibs` from `config.compilers`.

The underlying implementation in the framework uses a directed acyclic graph to indicate dependencies among modules. The vertices of this graph, configure objects, are topologically sorted and then executed. Moreover, child objects can be added to the framework without respecting the dependency structure, but this is discouraged.

2.3.2. Adding a test

A user could of course perform all tests in the object's `configure` method, but the base class provides useful logging support for this purpose. Consider again the BLAS example, which will now become,

```
  def checkDot(self):
    '''Verify that the ddot() function is contained in the BLAS library'''
    return self.libraries.check('libblas.a', 'ddot', otherLibs = self.compilers.flibs,
                                fortranMangle = 1)

  def configure(self):
    self.executeTest(self.checkDot)
    return
```

Passing our test module to the framework,

```
docs$ PYTHONPATH=`pwd` ../config/framework.py --configModules=[examples.blasTest]
```

we produce the following log output in `configure.log`. Notice that it not only records the method and module, but the method doc string, all shell calls, and any output actions as well.

```
================================================================================
TEST checkDot from examples.blasTest(/PETSc3/sidl/BuildSystem/docs/examples/blasTest.py:10)
TESTING: checkDot from examples.blasTest(/PETSc3/sidl/BuildSystem/docs/examples/blasTest.py:10)
  Verify that the ddot() function is contained in the BLAS library
      Checking for functions ['ddot'] in library ['libblas.a'] ['-lfrtbegin', '-lg2c', '-lm',
       '-L/usr/lib/gcc-lib/i486-linux/3.3.5', '-L/usr/lib/gcc-lib/i486-linux/3.3.5/../../..',
       '-lm', '-lgcc_s']
sh: gcc -c -o conftest.o  -fPIC  conftest.c
Executing: gcc -c -o conftest.o  -fPIC  conftest.c
sh:
sh: gcc  -o conftest   -fPIC  conftest.o  -lblas -lfrtbegin -lg2c -lm
  -L/usr/lib/gcc-lib/i486-linux/3.3.5 -L/usr/lib/gcc-lib/i486-linux/3.3.5/../../.. -lm -lgcc_s
Executing: gcc  -o conftest   -fPIC  conftest.o  -lblas -lfrtbegin -lg2c -lm
  -L/usr/lib/gcc-lib/i486-linux/3.3.5 -L/usr/lib/gcc-lib/i486-linux/3.3.5/../../.. -lm -lgcc_s
sh:
Defined HAVE_LIBBLAS to 1 in config.libraries
```

2.3.3. Checking for headers

Often, we would like to test for the presence of certain headers. This is done is a completely analogous way to the library case, using instead the `config.headers` module. Below, we test for the presence of the **curses** header.

```
import config.base

class Configure(config.base.Configure):
  def __init__(self, framework):
    config.base.Configure.__init__(self, framework)
    self.headers = self.framework.require('config.headers, self)
    return

  def checkCurses(self):
    'Verify that we have the curses header'
    return self.headers.check('curses.h')

  def configure(self):
    self.executeTest(self.checkCurses)
    return
```

Running this test

```
docs$ PYTHONPATH=`pwd` ../config/framework.py --configModules=[examples.cursesTest]
```

produces the following log output.

```
================================================================================
TEST checkCurses from examples.cursesTest(/PETSc3/sidl/BuildSystem/docs/examples/cursesTest.py:9)
TESTING: checkCurses from examples.cursesTest(/PETSc3/sidl/BuildSystem/docs/examples/cursesTest.py:9)
```

```
  Verify that we have the curses header
Checking for header: curses.h
sh: gcc -E   conftest.c
Executing: gcc -E   conftest.c
sh: # 1 "conftest.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "conftest.c"
# 1 "confdefs.h" 1
# 2 "conftest.c" 2
# 1 "conffix.h" 1
# 3 "conftest.c" 2
# 1 "/usr/include/curses.h" 1 3 4
# 58 "/usr/include/curses.h" 3 4
# 1 "/usr/include/ncurses_dll.h" 1 3 4
# 59 "/usr/include/curses.h" 2 3 4
# 99 "/usr/include/curses.h" 3 4
typedef unsigned long chtype;
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 295 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 296 "/usr/include/features.h" 2 3 4
# 318 "/usr/include/features.h" 3 4
#...
... W* win,int* y, int* x, _Bool to_screen);
extern _Bool mouse_trafo (int*, int*, _Bool);
extern int mcprint (char *, int);
extern int has_key (int);
extern void _tracef (const char *, ...) ;
extern void _tracedump (const char *, WINDOW *);
extern char * _traceattr (attr_t);
extern char * _traceattr2 (int, chtype);
extern char * _nc_tracebits (void);
extern char * _tracechar (int);
extern char * _tracechtype (chtype);
extern char * _tracechtype2 (int, chtype);
# 1203 "/usr/include/curses.h" 3 4
extern char * _tracemouse (const MEVENT *);
extern void trace (const unsigned int);
# 4 "conftest.c" 2

Defined HAVE_CURSES_H to 1 in config.headers
```

Alternatively, we could have specified that this header be included in the list of header files checked by default.

```
import config.base

class Configure(config.base.Configure):
  def __init__(self, framework):
    config.base.Configure.__init__(self, framework)
    self.headers = self.framework.require('config.headers, self)
    self.headers.headers.append('curses.h')
    return

  def checkCurses(self):
    'Verify that we have the curses header'
    return self.headers.haveHeader('curses.h')

  def configure(self):
    self.executeTest(self.checkCurses)
    return
```

In addition, the base class does include lower level support for preprocessing files. The `preprocess` method takes a code string as input and return a tuple of the **(stdout,stderr,error code)** for the run. The `outputPreprocess` method returns only the standard output, and `checkPreprocess` returns true if no error occurs.

2.3.4. Checking for libraries

We have already demonstrated a test for the existence of a function in a library. However the `check` method is much more general. It allows the specification of multiple libraries and multiple functions, as well as auxiliary libraries. For instance, to check for the `MPI_Init` and `MPI_Comm_create` functions in MPICH when the Fortran bindings are active, we would use:

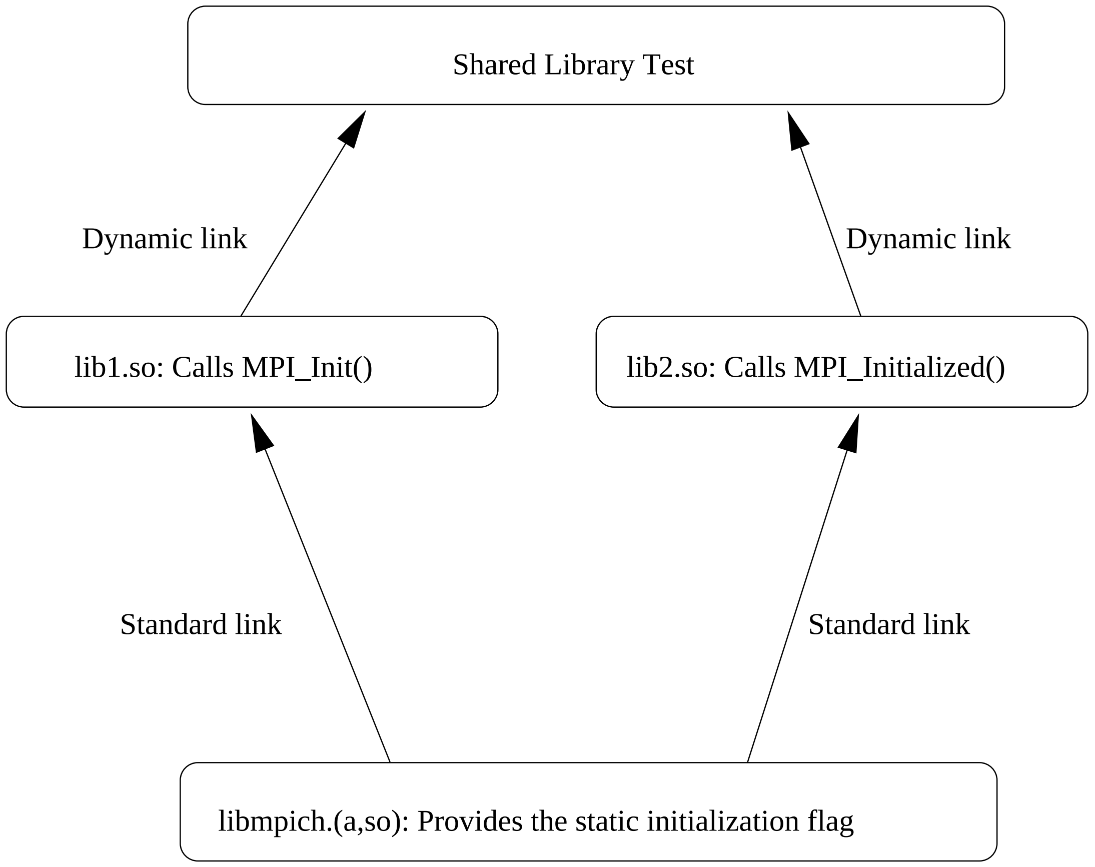
```
self.libraries.check(['libmpich.so', 'libpmpich.so'], ['MPI_Init', 'MPI_Comm_create'],
                     otherLibs = self.compilers.flibs)
```

As in the BLAS example, we can also turn on Fortran name mangling. The caller may also supply a function prototype and calling sequence, which are necessary if the current language is C++.

It is also necessary at some times to determine whether a given library is a shared object. This can be accomplished using the `checkShared` method, as we demonstrate with the MPICH library in a call taken from the MPI configure module in PETSc.

```
self.libraries.checkShared('#include <mpi.h>\n', 'MPI_Init', 'MPI_Initialized',
                           'MPI_Finalize', checkLink = self.checkMPILink,
                           libraries = self.lib)
```

The theory for the check is that a shared object will have only one copy of any global variable. Thus functions such as `MPI_Initialized` will render consistent results across other libraries. The test begins by creating two dynamic libraries, both of which link the given library. Then an executable is constructed which loads the libraries in turn. The first library calls the initizlization functions, here `MPI_Init`, and the second library calls the initialization check function, here `MPI_Initialized`. The check function will return true if the given library is a shared object. This organization is shown in figure ???


Shared Library Test
Dynamic link
Dynamic link
lib1.so: Calls MPI_Init()
lib2.so: Calls MPI_Initialized()
Standard link
Standard link
libmpich.(a,so): Provides the static initialization flag

The lower level interface to compiling and linking in the base class mirrors that for preprocessing. The `outputCompile` and `checkCompile` methods function in the same way. The code is now broken up into four distinct sections. There are includes, the body of `main`, and a possible replacement for the beginning and end of the `main` declaration. The linking methods, `outputLink` and `checkLink`, are exactly analogous.

There are also some convenience methods provided to handle compiler and linker flags. The `checkCompilerFlag` and `checkLinkerFlag` try to determine whether a given flag is accepted by the processor, while `addCompilerFlag` and `addLinkerFlag` will do that check and add any valid flag to the list of default flags.

2.3.5. Checking for executables

The `getExecutable` method is used to locate executable files. For instance, this code would allow us to locate the **valgrind** binary.

```
self.getExecutable('valgrind')
```

If the program is found, a member variable of the same name will be set in the object to the program name, and a make macro defined to it as well. We can opt for these to contain the full path by using the `getFullPath` argument. In addition, we can change the name of the member variable and macro using the `resultName` argument.

We also have control over the search path used. If we give no arguments, the default path from the environment is used. This can be overridden with a new path using the `path` argument, either as a Python list or a colon separated string. Furthermore, the default path can be added to this custom path using the `useDefaultPath` argument. For instance, this call

```
self.getExecutable('valgrind', path=['/opt/valgrind-1.0'], getFullPath=1,
                    useDefaultPath=1, resultName='grinder')
```

will check for **valgrind** first in `/opt/valgrind-1.0` and then along the default path. If found in the first location, it will set `self.grinder` to `/opt/valgrind-1.0/valgrind` as well as define GRINDER to the same value in makefiles.

As in the cases of preprocessing, compiling, and linking, the lower level operations are also exposed. The `checkRun` method takes in a code string and returns true if the executable runs without error. The `outputRun` method returns the output and status code. Both methods us the safe execution routine `config.base.Configure.executeShellCommand` which accepts a timeout. Moreover, there commands can run in the special batch mode described in section ???.

2.3.6. Output results

The BuildSystem configure includes the traditional output methods employed by **autoconf** to enable communication with **make**. Individual configure modules use the `addDefine` method to add C `#define` statements to a configuration header and the `addSubstitution` to setup substitution rules for specified files. For instance, to activate the ParMetis package, we might provide

```
self.addDefine('HAVE_PARMETIS', 1)
```

and then for the make process

```
self.addSubstitution('PARMETIS_INCLUDE', ' '.join([self.libraries.getIncludeArgument(i)
                                                   for i in self.include]))
self.addSubstitution('PARMETIS_LIB, ' '.join([self.libraries.getLibArgument(l)
                                              for l in self.lib]))
```

The actual output of this data is controlled by the framework. The user specifies the header file using the `header` field of the framework, and then the file is created automatically during the configure process, but can be output at any time using the `outputHeader` method. Furthermore, the `addSubstitutionFile` method can be used to tag a file for substitution, and also specify a different file for the result of the substitution.

In the **autoconf** approach, separating the the defines and substitutions for different packages becomes troublesome, and in some cases impossible to maintain. To help with this, we have introduced *prefixes* for the defines and substitutions. The are strings, unique to each module, which are prepended with an underscore to each identifier defined or substituted. These are set on a per object basis using the `headerPrefix` and `substPrefix` members. For instance, in our ParMetis example, if we instead used the code

```
self.headerPrefix = ’MATT’
self.addDefine(’HAVE_PARMETIS’, 1)
```

in our configuration header we would see

```
#ifndef MATT_HAVE_PARMETIS
#define MATT_HAVE_PARMETIS 1
#endif
```

Note that the value of the prefix is used at output time, not at the time that the define or substitution is set.

Another extension of the old-style output mechanisms adds more C structure to the interface. The `addTypedef` method allows a typedef from one typename to another, which in **autoconf** is handled by a define. Likewise `addPrototype` can add a missing function prototype to a header. Since these are C specific structures, they are output into a separate configuration header file, which is controlled by the `cHeader` member variable.

Extending in a different direction, we allow makefile structures to be specified directly rather than through substitutions. Using `addMakeMacro`, we can add variable definitions to the configuration makefile, whereas `addMakeRule` allows the user to specify a make target, complete with dependencies and action. As an example, we will replace our ParMetis example from above with the following code

```
self.addMakeMacro(’PARMETIS_INCLUDE’, ’ ’.join([self.libraries.getIncludeArgument(i)
                                               for i in self.include]))
self.addMakeMacro(’PARMETIS_LIB, ’ ’.join([self.libraries.getLibArgument(l)
                                          for l in self.lib]))
self.addMakeRule(’.c.o’, ”, [’${CC} -c -o $@ -I${PARMETIS_INCLUDE} $<’])
self.addMakeRule(’myApp’, ’${.c=.o:SOURCE}’, [’${CC} -o $@ $< ${PARMETIS_LIB}’])
```

which will produce

```
PARMETIS_INCLUDE = -I/home/knepley/petsc-dev/externalpackages/ParMetis/include
PARMETIS_LIB = -L/home/knepley/petsc-dev/externalpackages/ParMetis/lib/linux-gnu -lparmet
```

in the file specified by the `makeMacroHeader` member variable, and

```
myApp: ${.c=.o:SOURCE}
      ${CC} -i $@ $< ${PARMETIS_LIB}
```

in the file specified by the `makeRuleHeader` member variable.

The above output methods are all specified on a per configure object basis, however this may become confusing in a large project. All the prefixes and output filenames would have to be coordinated. A common strategy is to use the framework for coordination, putting all the output into the framework object itself. For instance, we might have

```
self.framework.addDefine('HAVE_PARMETIS', 1)
```

which would allow the define to appear in the headre specified by the framework with the framework prefix.

2.4. Configuring batch systems

It is not uncommon for large clusters or supercomputing centers to have a batch execution policy, making it difficult for configure to execute the few tests that depend on executing code, rather than compiling and linking it. To handle this case, we provide the `--with-batch` argument. The code to be run is collected in a single executable which the user must submit to the system. This executable produces a *reconfigure* script which may then be run to fully configure the system.

When configure is run with the `--with-batch` option, the following message will appear.

```
petsc-dev$ ./config/configure.py --with-batch
```

produces the following log output.

```
=================================================================================
    Since your compute nodes require use of a batch system or mpirun you must:
 1) Submit ./conftest to your batch system (this will generate the file reconfigure)
 2) Run "python reconfigure" (to complete the configure process).
=================================================================================
```

The user must then execute the `conftest` binary, and then run the **python reconfigure** command.

If a user defined test relies upon running code, he may make it suitable for a batch system. The `checkRun` method takes the `defaultArg` argument which names a configure option whose value may substitute for the outcome of the test, allowing a user to preempt the run. For instance, the `config.types.checkEndian` method contains the code

```
if self.checkRun('', body, defaultArg = 'isLittleEndian'):
```

which means the `isLittleEndian` option can be given to replace the output of the run. However, this does the require the user to supply the missing option.

To automate this process, the test should first check for batch mode. Using the `addBatchInclude` and `addBatchBody` methods, code can be included in the batch executable. We return to the endian test to illustrate this usage.

```
  if not self.framework.argDB['with-batch']:
    body = ''
    /* Are we little or big endian?  From Harbison & Steele. */
    union
    {
      long l;
      char c[sizeof(long)];
    } u;
    u.l = 1;
    exit(u.c[sizeof(long) - 1] == 1);
    ''
    if self.checkRun('', body, defaultArg = 'isLittleEndian'):
      endian = 'little'
    else:
      endian = 'big'
  else:
    self.framework.addBatchBody(
      ['{',
       '  union {long l; char c[sizeof(long)];} u;',
       '  u.l = 1;',
       '  fprintf(output, " \'--with-endian=%s\',\\n",\
            (u.c[sizeof(long) - 1] == 1) ? "little" : "big");',
       '}'])
    # Dummy value
    endian = 'little'
```

The batch body code should output configure options to the `output` file descriptor. These are collected for the new configure run in the `reconfigure` script.

Chapter 3. Build

The build operation now encompasses the configure, compile, link, install, and update operations.

3.1. Running make

All options for both configuration and build are given to `make.py`. Thus, the simplest build is merely

```
petsc-dev$ ./make.py
```

The help is also given by `-help`, but this time it will also include build switches.

```
petsc-dev$ ./make.py -help

Script Help
-----------
Script:
  --help                        : Print this help message                                     cu
  --h                           : Print this help message                                     cu
Make:
  -forceConfigure               : Force a reconfiguration                                     cu
  -ignoreCompileOutput          : Ignore compiler output                                      cu
  -defaultRoot                  : Directory root for all packages                             cu
  -prefix                       : Root for installation of libraries and binaries
SIDLMake:
  -bootstrap                    : Generate the boostrap client                                cu
  -outputSIDLFiles              : Write generated files to disk                               cu
  -excludeLanguages=<languages> : Do not load configurations from RDict for the given languages    cu
  -excludeBasenames=<names>     : Do not load configurations from RDict for these SIDL base names   cu
```

3.2. Makers

The build operation now encompasses configure, compile, and link operations, which are coordinated by objects of class `maker.Maker`. This object manages:

- configuration,
- build,
- install, and
- project dependencies

All options, no matter which component they are intended for, are given uniformly to `make.py`.

3.2.1. SIDLMaker

This is a subclass which handles source generation from SIDL.

3.3. Builders

The build operation now encompasses the configure, compile, and link operations.

3.4. LanguageProcessors

The build operation now encompasses the configure, compile, and link operations.

3.5. Interaction with Configure

The pickled configure is loaded by Maker, and then the config.compile objects are jacked into the Builder.