LBNE Software Installation System

Brett Viren

Physics Department

BROOKHAVEN
NATIONAL LABORATORY

LBNE Collaboration Meeting 2013/09

Outline

LBNE Software Ecosystem

Where we are:

- Distributed, multi-country collaboration, diverse expertise.
- So far, a largely Fermilab-centric mind-view towards software and computing.
- Existing LBNE software requires a huge number of supporting packages, including leading-edge C++ compiler.
- No automated, cross platform, build system exists
- Fermilab installation managed largely by one person[1] with responsibilities to other experiments.
- We lack an overabundance of software expertise in the collaboration.

What we need:

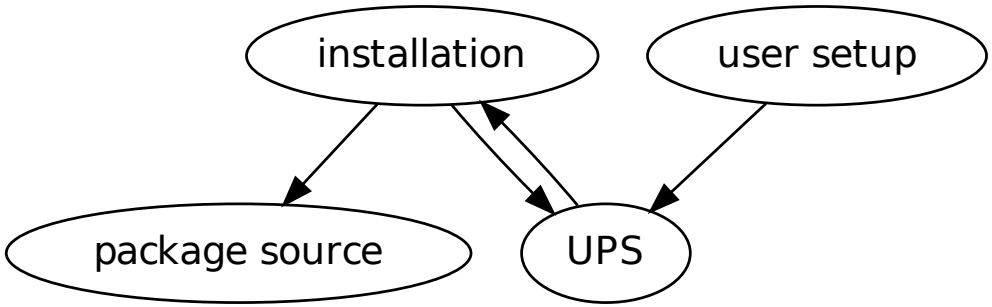
→ Control over the software and computing we rely on

→ Get the existing software working at home institutions and laptops

→ Facilitate further software development

[1] Thanks Lynn Garren!

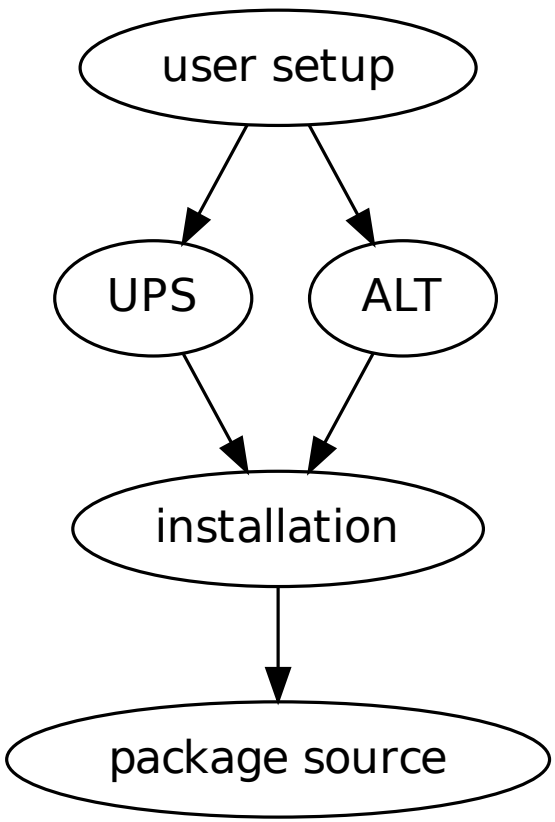
Note on role of UPS: current and desired dependencies



FNAL build system: UPS is required for both installation and usage.

Want: Installation system not dependent on UPS, allow alternative environment management systems.

(arrows indicate direction of dependence)



Current LBNE Suite

An incomplete (?) list of the packages making up LBNE software:

build tools	`cmake, gmp, ppl, mpfr, mpc, isl, cloog, gcc`
externals	`libxml2, sqlite, python, libsigcpp, libxml2, python, sqlite, tbb, xerces-c, lhapdf, mysql, postgresql, log4cpp, boost, fftw, cppunit, gccxml`
HEP	`root, geant4, cry, genie, pythia, globes` (so far up through here)
framework	`cpp0x, cetlib, fhicl-cpp, messagefacility, art, artdaq(?)`
LBNE	`larsoft, g4lbne`

Comprises, ∼20GB of build+install area
(comparison: 2-3× that for the Gaudi-based Daya Bay software).

Anything missing?

Strategy

Develop a (meta) build system which:

- Has a simple user interface driving an expert system.
- Captures all that is needed to install on all supported platforms in one location.
- Provides flexibility to install different suites or variations (debug vs. opt) and following different installation policies.
- Driven by LBNE needs but designed for general purpose use and comes with "batteries included" where practical.
- Maintains "source code provenance", use pristine, upstream source packages as input, but allow for application of any needed patches.
- Supports simple "schema evolution" of a the definition of the software suite, easy incorporation of future suite upgrades, version controlled installation meta data.
- No hidden build failures, fail early, fail often.
- Makes use of multi-CPU installation machines.
- Based on community supported, Free Software tools.

Desires for the build system

What I really want is to be able to say to you:
"To use the LBNE build system all you must do is:"

1. Download "**it**"
2. Run "**it**"
3. ???
4. Profit.

Due to the large size of the LBNE software suite, the "???" entails a few hours of you going and doing something else while the gears churn.

*What is "**it**"?*

"It" is called: "**worch**".

What? What the heck is that?

worch = **w**af + **orch**estration

You are loosing me!

- This horrid name was the working title which just stuck.
- It "orchestrates" building a suite of software by interpreting a configuration file into **waf** tasks and calling **waf** for the heavy lifting.

From the **worch** github page[2]:

"*Let the orchestration* **waf** *through the suite.*"

Okay, that's cheesy. So, what the heck is **waf***?....*

[2] `https://github.com/brettviren/worch`

What is **waf**?

waf[3]:

- is not an acronym, nor a name with any particular meaning
- is a Python-based (v2.3-v3.1) framework for building build systems
- can be used as a `make/autoconf/cmake` replacement or as a layer on top to orchestrate a variety of "native", per-package build systems
- provides a task scheduling and dependency engine
- packaged as a single, self-contained, downloaded Python executable

```
$ wget http://waf.googlecode.com/files/waf-1.7.12
$ chmod +x waf-1.7.12
$ alias waf=$(pwd)/waf-1.7.12
$ waf --version
waf 1.7.12 (c4b82f4337ebdf5236e844791a9faa346770d6db)
$ waf --help
```

[3] `http://code.google.com/p/waf/`

Aside: Synergy with ATLAS

With perfect timing, **Maxim Potekhin** pointed me to **waf** and **Sebastien Binet** of ATLAS at the exact moment that I was becoming frustrated with my hand–written attempt at a scheduling/dependency engine.
SB is using **waf** as the basis for a next-generation build system for the ATLAS experiment. His work includes:

- a command line user interface: **hwaf**
- replacing CMT with **waf**
- migrating Gaudi + LCGCMT (in process) "`tdaq`" subsystem (done)
- its use for bootstrapping/creating analysis packages,
- contributions to **worch** itself.

Note of caution for LBNE and Fermilab: ATLAS looked at CMake to replace CMT but gave up on it after two months of effort.
I still support replacing SRT with CMake in LArSoft, but for large-scale meta-build systems we should take heed.
We should investigate **hwaf**/**waf** to see if would be a worthy basis for LBNE "native" package builds.

Example: getting **worch**

```
$ git clone https://github.com/brettviren/worch.git
$ cd worch/
```

- Repository includes a main **waf** "`wscript`" file, the "`orch`" python module, various example configuration files and documentation (including this presentation).
- Will investigate packaging **worch** + **waf** into a single file like **waf** itself is.

Github is the primary repository due to collaboration with ATLAS and some personal intentions to use **worch** in other experiments/contexts.

- A symmetric mirror in Fermilab Redmine is possible.

Example: installation of the "simple" example suite

```
$ waf --out=tmp-simple --prefix=install-simple --orch-config=examples/simple/*.cfg  configure
Setting top to                           : /data3/bv/w/worch
Setting out to                           : /data3/bv/w/worch/tmp-simple
Orch configuration files                 : "examples/simple/buildtools.cfg", "examples/simple/gnuprograms.cfg", "examples/si
Orch configure envs                      : "", "cmake", "hello", "bc"
'configure' finished successfully (0.057s)

$ waf
Waf: Entering directory `/data3/bv/w/worch/tmp-simple'
Supported features: "dumpenv", "prepare", "makemake", "patch", "tarball", "vcs", "cmake", "autoconf"
[ 1/18] cmake_seturl:  -> tmp-simple/urlfiles/cmake-2.8.8.url
[ 2/18] cmake_download: tmp-simple/urlfiles/cmake-2.8.8.url -> tmp-simple/downloads/cmake-2.8.8.tar.gz
[ 3/18] cmake_unpack: tmp-simple/downloads/cmake-2.8.8.tar.gz -> tmp-simple/sources/cmake-2.8.8/bootstrap
[ 4/18] cmake_prepare: tmp-simple/sources/cmake-2.8.8/bootstrap -> tmp-simple/builds/cmake-2.8.8-debug/cmake_in
[ 5/18] cmake_build: tmp-simple/builds/cmake-2.8.8-debug/cmake_install.cmake -> tmp-simple/builds/cmake-2.8.8-d
[ 6/18] cmake_install: tmp-simple/builds/cmake-2.8.8-debug/bin/cmake -> install-simple/cmake/2.8.8/debug/bin/cm
[ 8/18] hello_seturl:  -> tmp-simple/urlfiles/hello-2.8.url
[ 8/18] bc_seturl:  -> tmp-simple/urlfiles/bc-1.06.url
[ 9/18] hello_download: tmp-simple/urlfiles/hello-2.8.url -> tmp-simple/downloads/hello-2.8.tar.gz
[10/18] bc_download: tmp-simple/urlfiles/bc-1.06.url -> tmp-simple/downloads/bc-1.06.tar.gz
[11/18] bc_unpack: tmp-simple/downloads/bc-1.06.tar.gz -> tmp-simple/sources/bc-1.06/configure
[12/18] hello_unpack: tmp-simple/downloads/hello-2.8.tar.gz -> tmp-simple/sources/hello-2.8/configure
[13/18] bc_prepare: tmp-simple/sources/bc-1.06/configure -> tmp-simple/builds/bc-1.06-debug/config.status
[14/18] bc_build: tmp-simple/builds/bc-1.06-debug/config.status -> tmp-simple/builds/bc-1.06-debug/bc/bc
[15/18] bc_install: tmp-simple/builds/bc-1.06-debug/bc/bc -> install-simple/bc/1.06/debug/bin/bc
[16/18] hello_prepare: tmp-simple/sources/hello-2.8/configure -> tmp-simple/builds/hello-2.8-debug/config.statu
[17/18] hello_build: tmp-simple/builds/hello-2.8-debug/config.status -> tmp-simple/builds/hello-2.8-debug/src/h
[18/18] hello_install: tmp-simple/builds/hello-2.8-debug/src/hello -> install-simple/hello/2.8/debug/bin/hello
Waf: Leaving directory `/data3/bv/w/worch/tmp-simple'
'build' finished successfully (4m5.157s)
```

Visualization

The "simple" example suite.
Of note:

- 3 packages
- 2 atomic build groups
- File-based deps
- Inter-step deps
  artificial dep shown:

  `bc_install` → `hello_prepare`

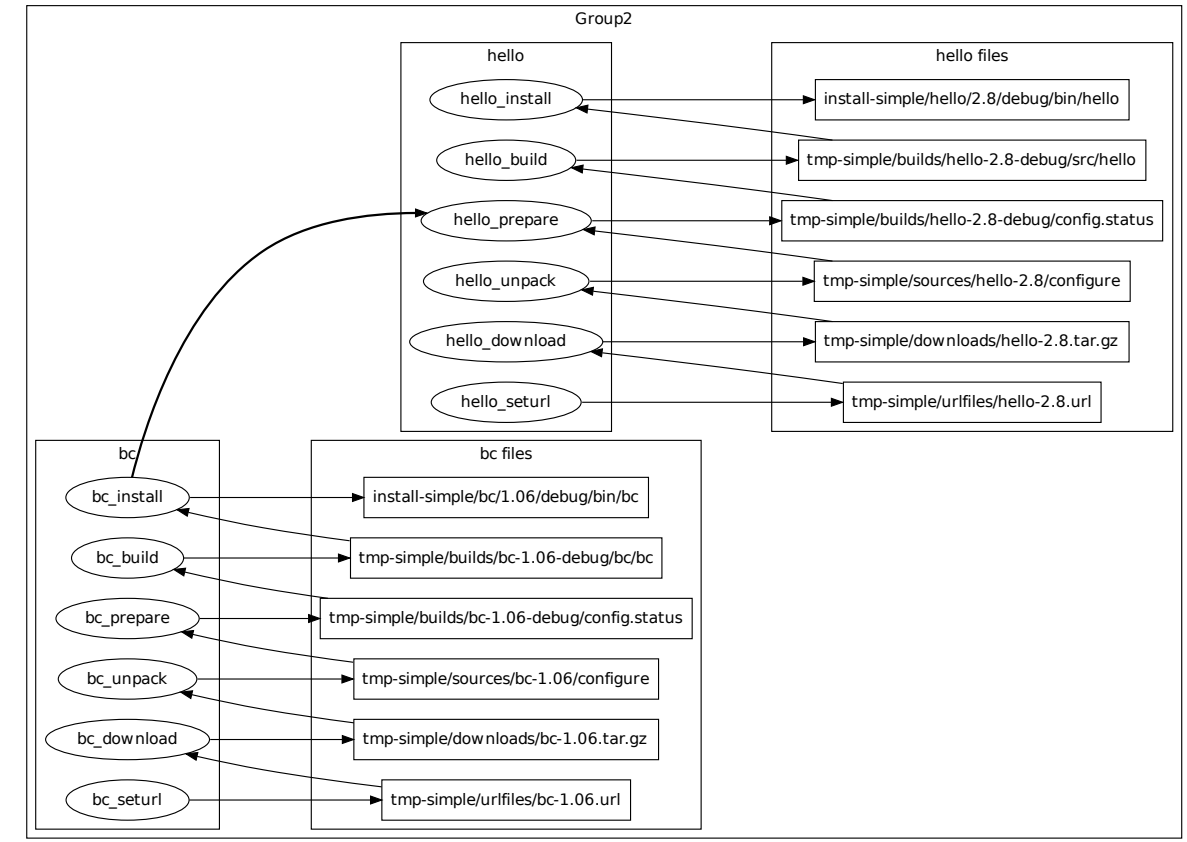
(Arrows indicate reverse-dependency,

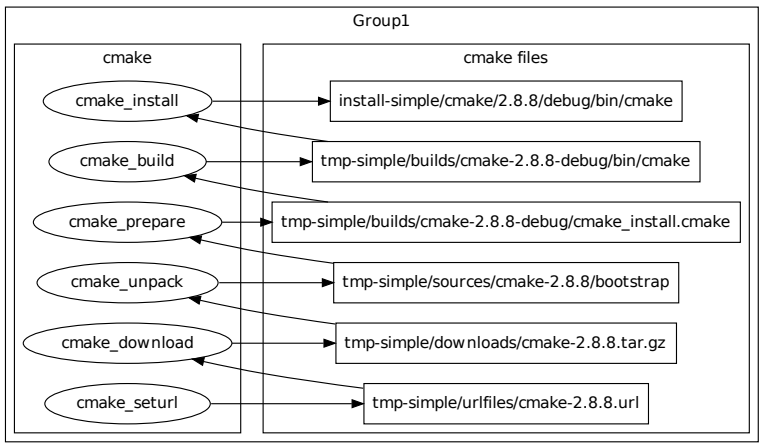
or processing flow).

To generate:

```
$ waf [...] configure
$ waf --dot=simple.dot
$ dot -Tpdf \
  -osimple.pdf simple.dot
```

`dot` is from `http://graphviz.org`





Example: running **waf** to build LBNE's suite

Read configuration file(s) and set installation destination:

```
$ waf --prefix=/path/to/install/area \
      --orch-config==examples/main/art.cfg configure
```

Build everything, with as many as 10 waf tasks at once:

```
$ waf -j10
```

Which shows:

```
[ 3/89] cmake_seturl:  -> tmp/urlfiles/cmake-2.8.8.url
[ 4/89] libxml2_seturl:  -> tmp/urlfiles/libxml2-2.8.0.url
[ 4/89] sqlite_seturl:  -> tmp/urlfiles/sqlite-3.7.17.url
[ 4/89] python_seturl:  -> tmp/urlfiles/python-2.7.3.url
[... etc for download, unpack, build, install ...]
[87/89] root_prepare: tmp/sources/root/CMakeLists.txt -> tmp/builds/root-5.
[88/89] root_build: tmp/builds/root-5.34.09-debug/CMakeCache.txt -> tmp/bui
[89/89] root_install: tmp/builds/root-5.34.09-debug/bin/root.exe -> ../inst
'build' finished successfully (1h25m17.527s)
```

(this build only went up through ROOT)

Visualize the LBNE suite build

Three groups: *buildtools*, *compiler* and *externals* includes up through "standard" HEP packages:

`cmake`, `libxml2`, `sqlite`, `python`, `cmake`, `gmp`, `mpfr`, `mpc`, `isl`, `ppl`, `cloog`, `gcc`, `libsigcpp`, `tbb`, `xrootd`, `ROOT`, `Geant4`, `xercesc`, `log4cpp`, `mysql`.

In progress *framework* and *lbnesoft* groups with `cry`, `genie`, `globes`, `art`, `art_externals`, `g4lbne`, `larsoft`.

(That things are too small to see is the point)

Configuration

If **worch** "orchestrates" the install then its configuration file is the conductor's notes. One mainly interacts with **worch** through the configuration files in order to:

- Determine installation layout policy, Eg:
  - match UPS "product area" conventions
  - share/interleave different build options (`opt/debug`), platform labels
  - build on fast local disk/ramfs, install to AFS/NFS
- Specify list of packages to install
- Group them into atomically installed sets
- Define "features" implementing installation "steps"

  features *tarball, vcs, patch, autoconf, cmake, makemake*

  steps `seturl, download, unpack, patch, prepare, build, install`
- Set parameters governing details of their build and installation
- Declare explicit inter-step dependencies
- Set package build and use environment variables

Basic configuration file syntax

Standard Python (`ConfigParser`) configuration file syntax (a.k.a. "INI" format).

```
# this is a comment
[section]
key = value

[section2]
key = newvalue
key2 = value2
key3: equals-or-colons
```

Consists of a number of named **sections** containing a number of configuration **items** consisting of **key**/**value** pairs separated by "=" or ":". Sections, keys and values are interpreted as strings.

Extension: hierarchy of configuration parameters

```
[start]
groups = group1, group2
key1 = val1
key2 = val2
key3 = val3

[group group1]
packages = package1, package2
key2 = grpval2

[package package1]
key1 = pkg1val1

[package package2]
key1 = pkg2val1

[keytype]
groups = group
packages = package
```

Hierarchy connected by section names

- Results in `package1` having keys:
  - `key1 = pkg1val1`
  - `key2 = grpval2`
  - `key3 = val3`
- Likewise for `package2` but with `key1 = pkg2val1`
- Special `keytype` section defines special variables to be interpreted as references to typed/named sections.
- Additional keytypes can be defined and parsed but **worch** uses just `groups` and `packages`.

Parameters in configuration

Configuration items form variables that can be referenced in other items. Reference a variable like "{keyname}".

```
[start]
tags = debug
install_dir = {PREFIX}/{package}/{version}/{tagsdashed}
gcc_version = 4.8.1
groups = buildtools, compiler, externals
#...
[group compiler]
install_dir = {PREFIX}/gcc/{gcc_version}/{tagsdashed}
packages = gmp, ppl, mpfr, mpc, isl, cloog, gcc
#...
[package gcc]
version = {gcc_version}
#...
```

Variables are resolved late so "flow down" the hierarchy to take on meanings specific to each "leaf" keytype (package in this case). Some variables and some other's default values are provided by **worch**.

Simple example - main file

```
# worch/examples/simple/simple.cfg
[start]
groups = buildtools, gnuprograms
includes = buildtools.cfg, gnuprograms.cfg
group = gnuprograms
tags = debug
features = tarball autoconf makemake
download_dir = downloads
source_dir = sources
build_dir = builds/{package}-{version}-{tagsdashed}
install_dir = {PREFIX}/{package}/{version}/{tagsdashed}

[keytype]
groups = group
packages = package
```

Simple example - buildtools group file

```
# worch/examples/simple/buildtools.cfg
[group buildtools]
packages = cmake
build_target = bin/{package}
install_target = bin/{package}

[package cmake]
version = 2.8.8
source_url = http://www.cmake.org/files/v{version_2digit}/{sou
unpacked_target = bootstrap
prepare_cmd = ../../{source_dir}/{source_unpacked}/bootstrap
prepare_cmd_options = --prefix={install_dir}
prepare_target = cmake_install.cmake
export_PATH = prepend:{install_dir}/bin
```

Simple example - gnuprograms group file

```
# worch/examples/simple/gnuprograms.cfg
[group gnuprograms]
packages = hello, bc
source_url = http://ftp.gnu.org/gnu/{package}/{source_package}
environment = group:buildtools
unpacked_target = configure
prepare_target = config.status
build_target = bin/{package}
install_target = bin/{package}

[package hello]
version = 2.8
depends = prepare:bc_install

[package bc]
version = 1.06
```

Design Overview

worch consists of these layers:

configuration parses the configuration file into a dictionary of groups each consisting of a dictionaries of packages. Configuration items come from:

- configuration file(s)
- hard-coded items required a package's *features*
- a global set of required items

features each package has zero or more feature that map to a **worch** Python method which produces one or more **waf** tasks, based on configuration, to accomplish part of a package installation (eg: `tarball`, `autoconf`, `makemake`).

`wscript` **waf** uses `wscript` files (think `Makefile`) as entry points. Normally **waf** will use the single `worch/wscript` file but this will delegate if any `worch/<package>/wscript` files exist. So, if a building a package is too hairy to fit into existing **worch** one can roll-your-own

Example: trivial **worch** feature

A feature is a method which constructs **waf** tasks (task generators) based on its set of required configuration items. Eg, from: `worch/orch/features/feature_dumpenv.py`:

```
from .pfi import feature
@feature('dumpenv')
def feature_dumpenv(info):
    '''
    Dump the environment
    '''
    info.task('dumpenv', rule = "env | sort")
```

All modules like `orch.features.feature_*` are auto-loaded by **worch**. They declare tasks and inter-task (non-file) dependencies via the given `info` object.

Some `task()` options:

- `rule` specify the shell command to run or a Python function
- `source` specify a file (**waf** Node) this task relies on
- `target` specify a file (**waf** Node) this task produces

Example: a feature with requirements

- Features provide list of required configuration items, with defaults
- "Standard" requirements are reused where appropriate.
- Requirements are available as data members or as format variables through the `info` object.

From: `worch/orch/features/feature_autoconf.py`:

```
requirements = {
    'unpacked_target': 'configure',
    'source_unpacked': None,
    'prepare_cmd': '../../{source_dir}/{source_unpacked}/configure',
    'prepare_cmd_options': '--prefix={install_dir}',
    'prepare_target': 'config.status',
    'build_dir': None,
}
@feature('autoconf', **requirements)
def feature_autoconf(info):
    def prepare_task(task):
        cmd = info.format("{prepare_cmd} {prepare_cmd_options}")
        return exec_command(task, cmd)

    info.task('prepare',
              rule = prepare_task,
              source = info.unpacked_target,
              target = info.prepare_target)
```

Dependencies

Tasks may depend on other tasks in a number of ways:

groups All tasks of all packages in a group must finish before those of following groups begin.

files The source/target files may form task dependencies

depends explicit `depends` configuration items in the form of "`<step>:<otherpackage>_<otherstep>`" make the current package's step depend on another package's step.

insertion a feature may insert a step by name in front of another using names of the form `<package>_<step>`.

Steps

An open-ended but limited set of "step" names are used to provide common touchstones. As more functionality is added to worch this set may enlarge. New features should strive to reuse existing steps if they are conceptually compatible.

seturl	write the source archive into a file - this primes the dependencies based on files
download	download the source archive (tarball or VCS repository) given the URL
unpack	unpack the source archive (if tarball)
patch	apply a patch to an unpacked source archive
prepare	prepare the source for building (eg, autoconf, cmake)
build	build the package (eg, "make")
install	install the package (eg, "make install")

Status and Future

Status:

- Basic functionality implemented and tested
- Configuration exists for simple examples and extensive LBNE-targeted groups: *buildtools*, *compiler* and *externals* (includes "standard" HEP packages)
- Sci. Linux 6 initial target (and somewhat Debian "jessie")

Future:

- Finish incorporating art, LArSoft, genie, globes and g4lbne
- Test on SL5, Ubuntu and Mac OS X
- Add support for producing UPS "products area" and binary packages
- Hook into LBNE repositories for continual integration (buildbot)
- In parallel with above, solicit volunteers for testing and use.
- Continue to collaborate with ATLAS, look for "synergies".