YAML2RPM – Another Attempt at Taming Scientific Software

Philip Papadopoulos, PhD.
ppapadop@uci.edu
Director, UCI Research Cyberinfrastructure Center (RCIC)

Nadya Williams
npw@uci.edu
RCIC

What is The Research Cyberinfrastructure Center?

- Created at the recommendation of UCI Research Cyberinfrastructure Vision document
- Build and maintain computing/data infrastructure for all UCI researchers to improve competitiveness and operational efficiency
- Support grant proposals (entire process)
- Shared-funding model
  - Campus $$ **pay for baseline computing/data**
  - Researchers pay **cost recovery for additional cycles/storage**

Key Resources and Functions

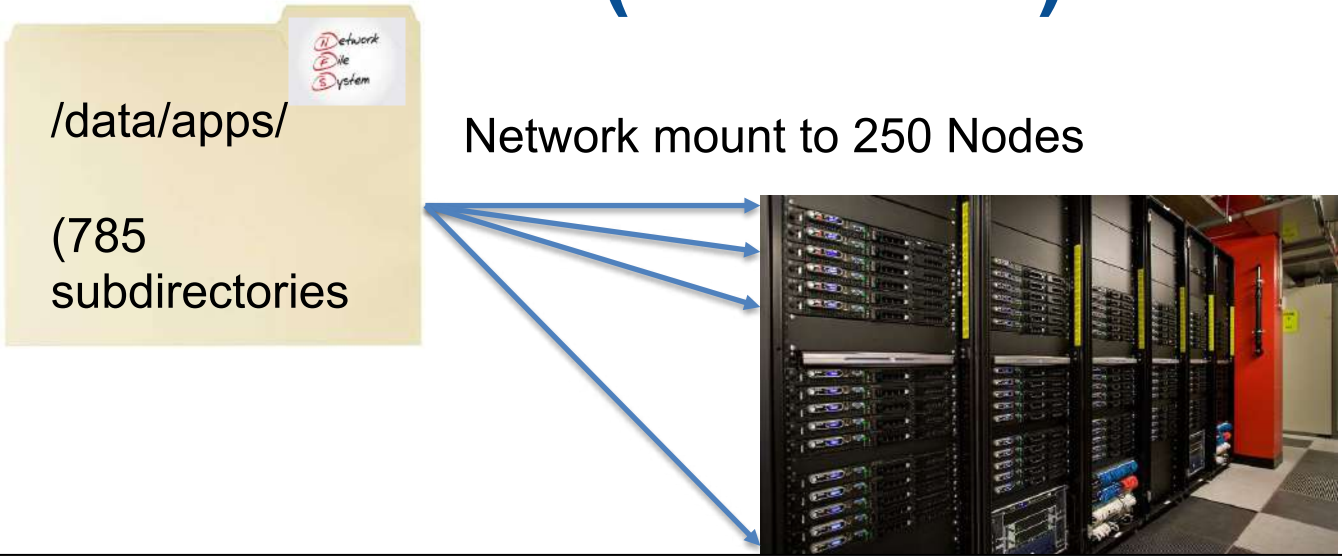
- Campus Research Storage Pool
  - Expandable, Reliable, On-line storage for research data
  - Two copies of all data at all times (2+ PB)
- Cluster Computing
  - HPC "Condo" (~10K cores). HPC^3 coming soon (4K+ cores)
- Parallel File System Storage
  - High-performance, lowest cost, single copy of data (2+ PB)
- Science Application Software Integration
  - More than 700 open-source domain-specific applications available
- Support Research Grant Process
  - Concept through implementation
  - Can include one or more of the following:
    - Hardware recommendation, Facilities, Data management, writing, fractional support of RCIC staff, Co-Investigator

A Software Perspective…

- UCI runs a shared HPC cluster for campus researchers
- A very diverse community of researchers
- Care and feeding of an Applications “Zoo”:
  - Have built and must maintain/update:
    - 780+ Unique Scientific Software packages
    - 1200+ when considering multiple version
    - Note: CentOS7 builds/maintains ~9000 packages.
- Existing cluster is CentOS6
  - New cluster is CentOS7
  - Will need to be agile, CentOS8 is out and users will demand it in a few years
  - Need to rebuild applications for each revision.

➢ Application complexity is on par with a supercomputer center, but we are not the same size resource (or staff)

Current (inherited) State



/data/apps/

(785 subdirectories

Network mount to 250 Nodes

- How was Application X configured/compiled? (some have compile scripts)
- What's affected if you update application X? (can't easily answer this)
- When was something installed/updated? (partial answer: can look at file system snapshots/backups)
- What if users need different versions of the same software? (supported via environment modules)
- What if some applications should not be available on all nodes (shared FS makes this very problematic)

Current state works, but …

Can we tame our software stack?

Claim: Formally packaging software can address many of these issues

- Query what's installed (manifest)
- Assure that no two software packages conflict in the file system (reduce contention)
- Determine when something was added/removed/updated (tracking)
- Query if the current state on disk is different than it is supposed to be. (verification)
- If properly built, when a prerequisite software is removed, all dependent software is also removed (consistency)
- Ability to update software version in place (manageability)

Management is improved dramatically if you can use the packaging system that is NATIVE to the OS

  - ➔ If using CentOS/RedHat use RPM format
  - ➔ if using Debian/Ubuntu use DPKG format

Why do many cluster admins/owners not build their own packages?

- “It’s too difficult to create a package for each piece of software, I have enough on my hands figuring out how to compile X”
- “Python/Perl/R/Anaconda/Brew manage their own packages, that’s most of my work”
- “I just manage one system, packaging is overkill”
- “My users want so much software that I just get it done as fast as possible. “
- “I don’t see anything wrong with compiling into a shared NFS area”

Packaging isn’t Glamorous..

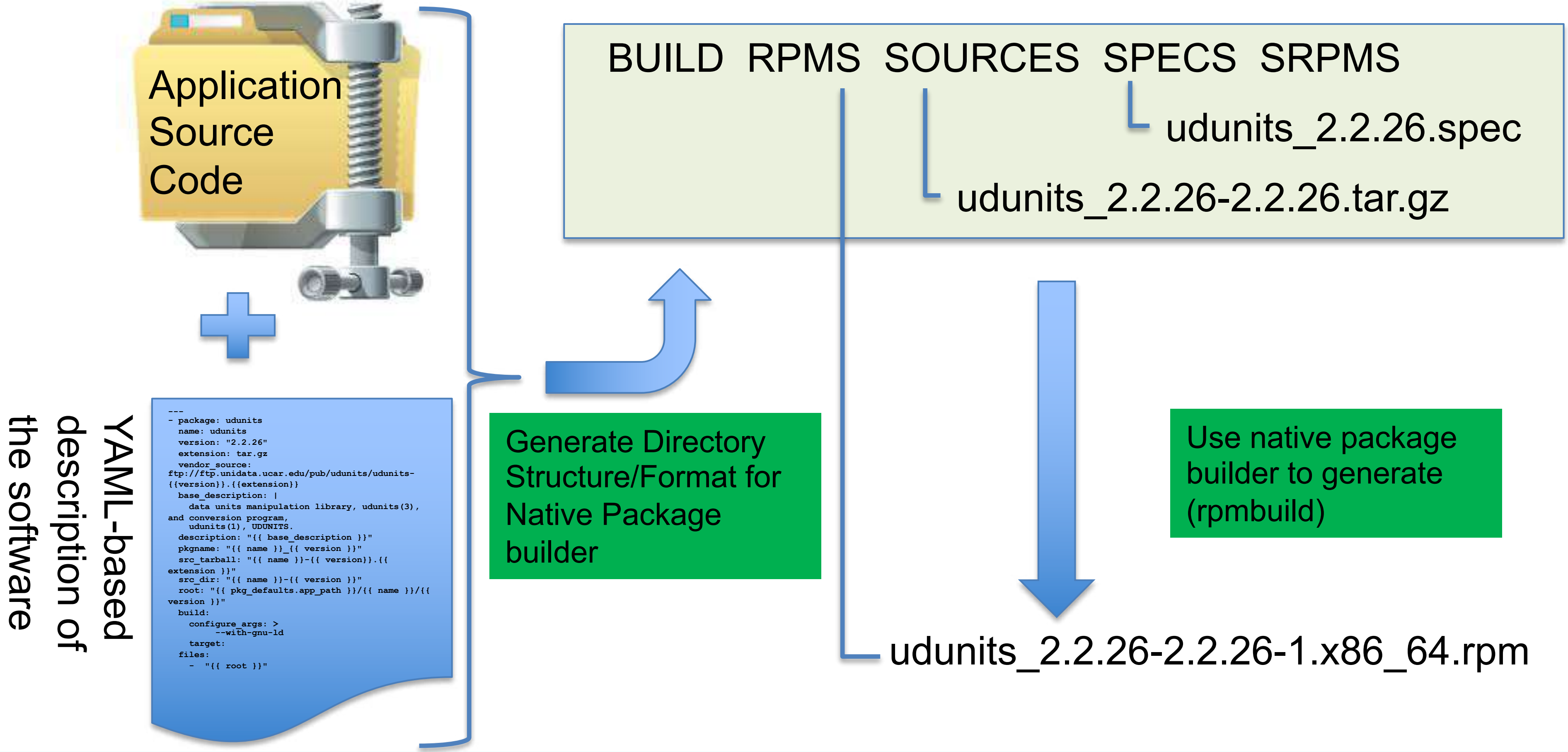
- Can we build a system such that: *it is practical for a a small team of people to build an application stack that:*
  - Contains hundreds (to thousands) of discrete software components
  - Has consistent subsets of software to put into containers
  - Is repeatable/extensible
  - Works in concert with the OS-provided packaging system
- Not glamorous – but, walking this path can significantly improve stability/predictability.

An overriding goal

- If one can figure out how to compile package X (including its dependencies)
  - It should be straightforward to produce a binary package.
  - (often the most time-consuming step of scientific software builds is tracking down what prerequisites are needed)

Simplifying Package Building

YAML2RPM: ***f*(Source code, description file) = package**



Application Source Code

YAML-based description of the software

```
---
- package: udunits
  name: udunits
  version: "2.2.26"
  extension: tar.gz
  vendor_source:
ftp://ftp.unidata.ucar.edu/pub/udunits/udunits-
{{version}}.{{extension}}
  base_description: |
    data units manipulation library, udunits(3),
and conversion program,
    udunits(1), UDUNITS.
  description: "{{ base_description }}"
  pkgname: "{{ name }}_{{ version }}"
  src_tarball: "{{ name }}-{{ version}}.{{
extension }}"
  src_dir: "{{ name }}-{{ version }}"
  root: "{{ pkg_defaults.app_path }}/{{ name }}/{{
version }}"
  build:
    configure_args: >
         --with-gnu-ld
    target:
  files:
    -  "{{ root }}"
```

Generate Directory Structure/Format for Native Package builder

BUILD RPMS SOURCES SPECS SRPMS

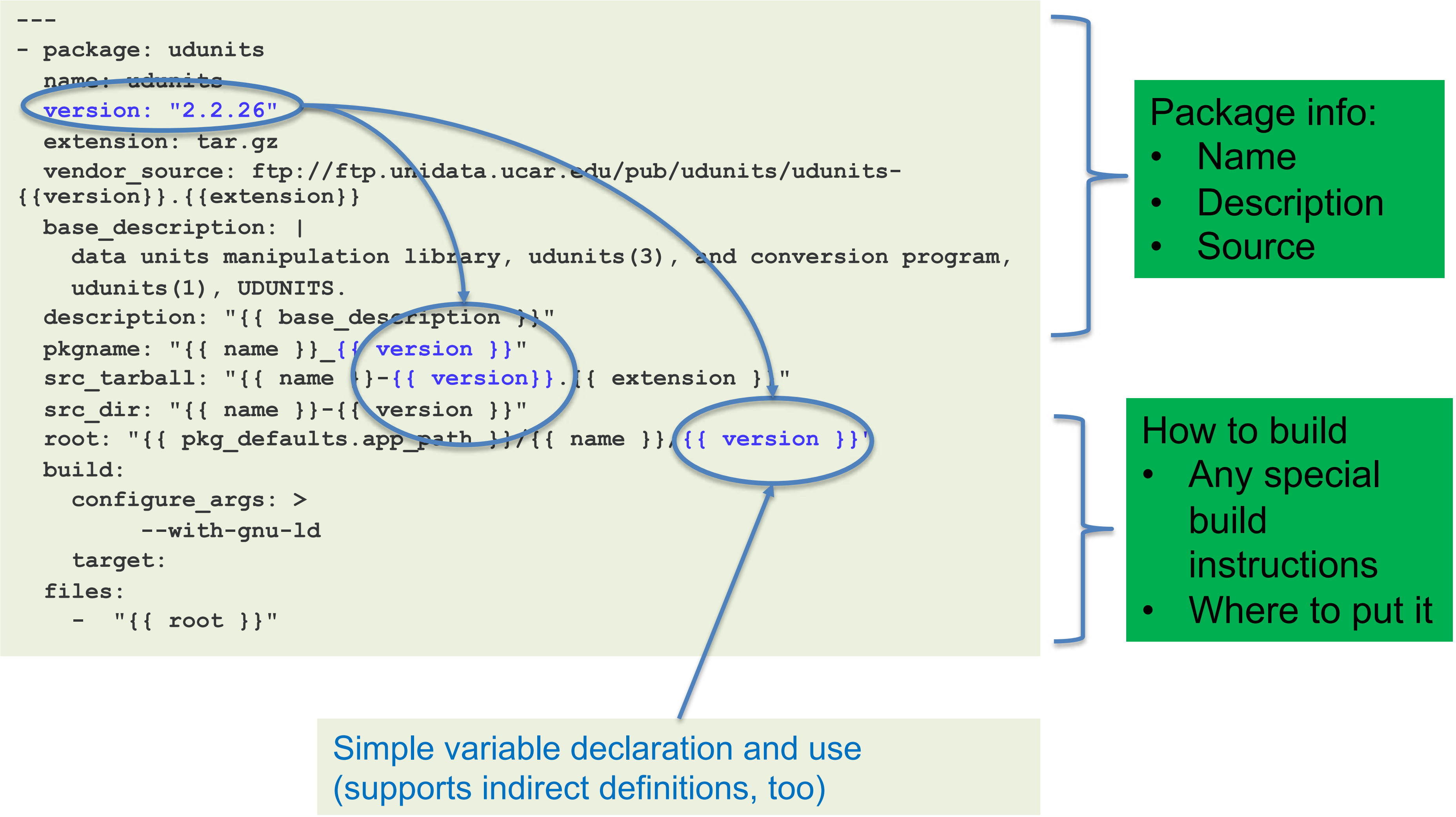
udunits_2.2.26.spec

udunits_2.2.26-2.2.26.tar.gz

Use native package builder to generate (rpmbuild)

udunits_2.2.26-2.2.26-1.x86_64.rpm

A small (real) example



```
---
- package: udunits
  name: udunits
  version: "2.2.26"
  extension: tar.gz
  vendor_source: ftp://ftp.unidata.ucar.edu/pub/udunits/udunits-
{{version}}.{{extension}}
  base_description: |
    data units manipulation library, udunits(3), and conversion program,
    udunits(1), UDUNITS.
  description: "{{ base_description }}"
  pkgname: "{{ name }}_{{ version }}"
  src_tarball: "{{ name }}-{{ version}}.{{ extension }}"
  src_dir: "{{ name }}-{{ version }}"
  root: "{{ pkg_defaults.app_path }}/{{ name }}/{{ version }}"
  build:
    configure_args: >
         --with-gnu-ld
    target:
  files:
    -  "{{ root }}"
```

Package info:
- Name
- Description
- Source

How to build
- Any special build instructions
- Where to put it

Simple variable declaration and use
(supports indirect definitions, too)

Note: There is nothing RPM-specific in this description. That’s a feature!

Generated RPM is usable on any CentOS 7 System

```
# rpm -qip udunits_2.2.26-2.2.26-1.x86_64.rpm
Name        : udunits_2.2.26
Version     : 2.2.26
Release     : 1
Architecture: x86_64
Install Date: (not installed)
Group       : System Environment/Base
Size        : 631790
License     : University of California
Signature   : (none)
Source RPM  : udunits_2.2.26-2.2.26-1.src.rpm
Build Date  : Fri 06 Sep 2019 11:16:30 AM PDT
Build Host  : admixdev-0.local
Relocations : (not relocatable)
Vendor      : RCIC @ UC Irvine
Summary     : udunits_2.2.26
Description :
data units manipulation library, udunits(3), and conversion
program, udunits(1), UDUNITS. . Configured with ./configure --
prefix=/data/apps/udunits/2.2.26 --with-gnu-ld. Modules loaded
for compilation: .
```

```
# rpm -qp --provides udunits_2.2.26-2.2.26-1.x86_64.rpm
libudunits2.so.0()(64bit)
udunits_2.2.26 = 2.2.26-1
udunits_2.2.26(x86-64) = 2.2.26-1
```

- It is a real RPM.
- Can be queried for what it requires and what it provides
- Install in a package repository and use it to
  - Install on physical HW
  - Be part of a container
  - Verify in a test environment

Drawing upon ideas from others…

- YAML format
  - Used by Kubernetes, Ansible (config mgmt.), and others
  - Already existing python-based parsers
  - Human-readable
- {{ var }}
  - This is NOT part of any YAML spec, but Ansible uses this for declaring/resolving variables.
  - We wrote our own custom (recursive) variable resolver
- Rocks Rolls
  - Extract the subdirectory and automated creation of RPM Spec files (directives to build packages) and required RPM structure
  - Re-uses this tooling without requiring build host to be a Rocks-defined system (all development on vanilla CentOS)

How is this different from other efforts

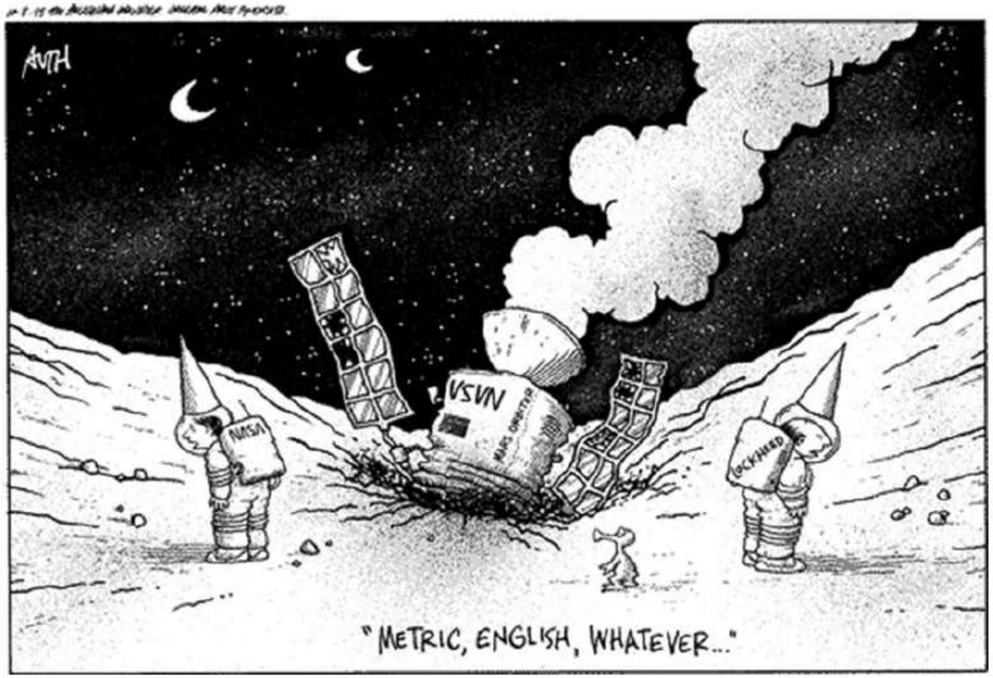
- SPACK - https://spack.io/
  - Self-contained package format, dependency resolver. You must code in Python to generate a package
- Anaconda (https://www.anaconda.com) – focused on datascience. Own packaging format/resolver/installer. Will happily install an OS-specific package on anything (resulting in broken installations)
- Brew (https://brew.sh/) "missing" package manager for MacOS. Also works on Linux. You must code in Ruby.

➢ These create binary packages that might conflict with OS-based packages (installers are unaware of each other)

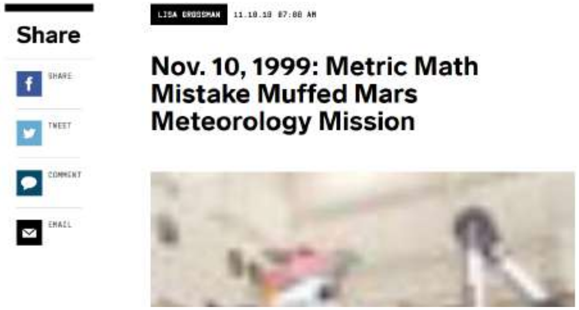
➢ YAML2RPM packages are installed with the OS-based package manager (not with yet another installer)

  ➢ Kickstart, Docker Build, Singularity Recipe, Ansible, others all know how to install RPMS

Don’t let this happen to your cluster …

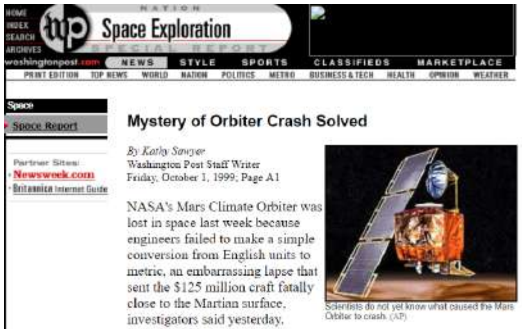


Remember the Mars Climate Orbiter incident from 1999?



LISA GROSSMAN 11.10.10 07:00 AM

Nov. 10, 1999: Metric Math Mistake Muffed Mars Meteorology Mission



Mystery of Orbiter Crash Solved

By Kathy Sawyer
Washington Post Staff Writer
Friday, October 1, 1999; Page A1

NASA's Mars Climate Orbiter was lost in space last week because engineers failed to make a simple conversion from English units to metric, an embarrassing lapse that sent the $125 million craft fatally close to the Martian surface, investigators said yesterday.

Scientists do not yet know what caused the Mars Orbiter to crash. (AP)

“RPM, Brew, SPACK, Whatever …”

Easy things should be straightforward … complicated should be possible

- The simple udunits example follows a commonly-used build recipe

```
1.   ./configure  <configure arguments>
2.   make
3.   make install
```

- The build instructions in the YAML file are therefore very terse, with this pattern *implied*

```
build:
   configure_args: >
        --with-gnu-ld
   target:
```

```
root: "{{ pkg_defaults.app_path }}/{{ name }}/{{ version }}"
```

Complex builds are possible

- ✓ Custom build scripts
- ✓ Patching source code
- ✓ Locally-generated code
- ✓ Cmake-based
- ✓ Using alternative versions of support software (e.g., updated version of gcc, via environment modules)
- ✓ Unconventionally-named source archives and/or build trees
- ✓ environment-modules support
- ✓ Filtering of RPM auto-generated requires/provides to keep alternative versions (e.g. updated Python) well contained
- ✓ Reuse of build templates for generating multiple versions (e.g. gcc 6.5.0 and gcc 8.3.0)
- ✓ Simple bootstrapping when build of Software X requires Software Y to first be built and installed.
- ✓ Multiple YAML files in a single directory to build logical groups of packages

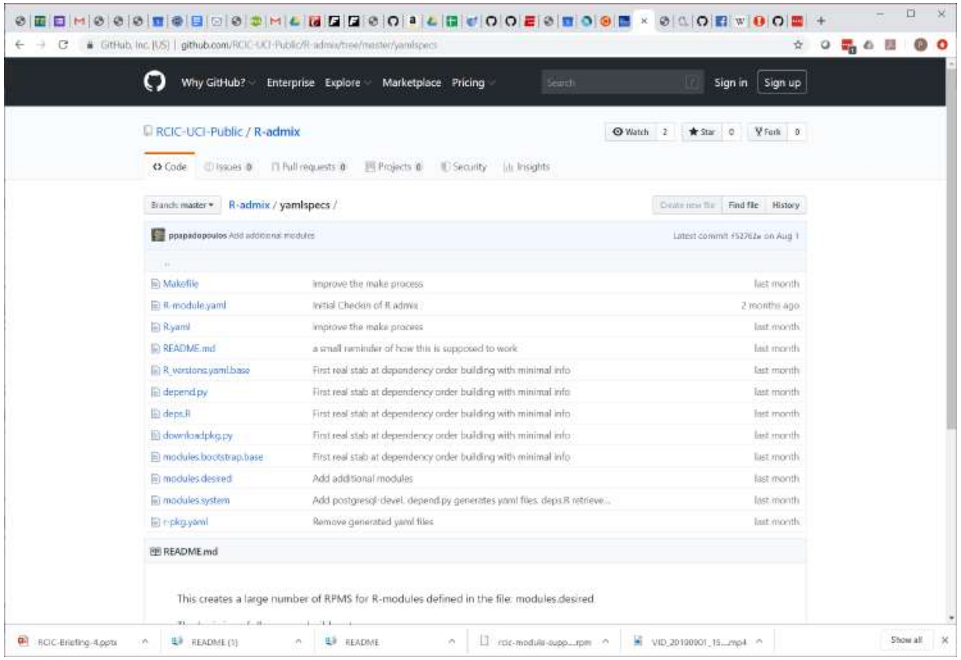
Software Admix – Logical Package Groupings

admixture (noun) ad·mix·ture | \ ad-ˈmiks-chər - something added by mixing

Ones we are working on now:

- gcc compilers, biotools, fileformats, R, Perl, Python, chemistry, …
- Admixes can generate 10s to 100s of related RPMs

The R Admix– A more complicated Example



RCIC-UCI-Public / R-admix

R-admix / yamlspecs /

ppapadopoulos Add additional modules — Latest commit f52762e on Aug 1

Makefile	improve the make process	last month
R-module.yaml	Initial Checkin of R admix	2 months ago
R.yaml	improve the make process	last month
README.md	a small reminder of how this is supposed to work	last month
R_versions.yaml.base	First real stab at dependency order building with minimal info	last month
depend.py	First real stab at dependency order building with minimal info	last month
deps.R	First real stab at dependency order building with minimal info	last month
downloadpkg.py	First real stab at dependency order building with minimal info	last month
modules.bootstrap.base	First real stab at dependency order building with minimal info	last month
modules.desired	Add additional modules	last month
modules.system	Add postgresql-devel. depend.py generates yaml files. deps.R retrieve...	last month
r-pkg.yaml	Remove generated yaml files	last month

README.md

This creates a large number of RPMS for R-modules defined in the file: modules.desired

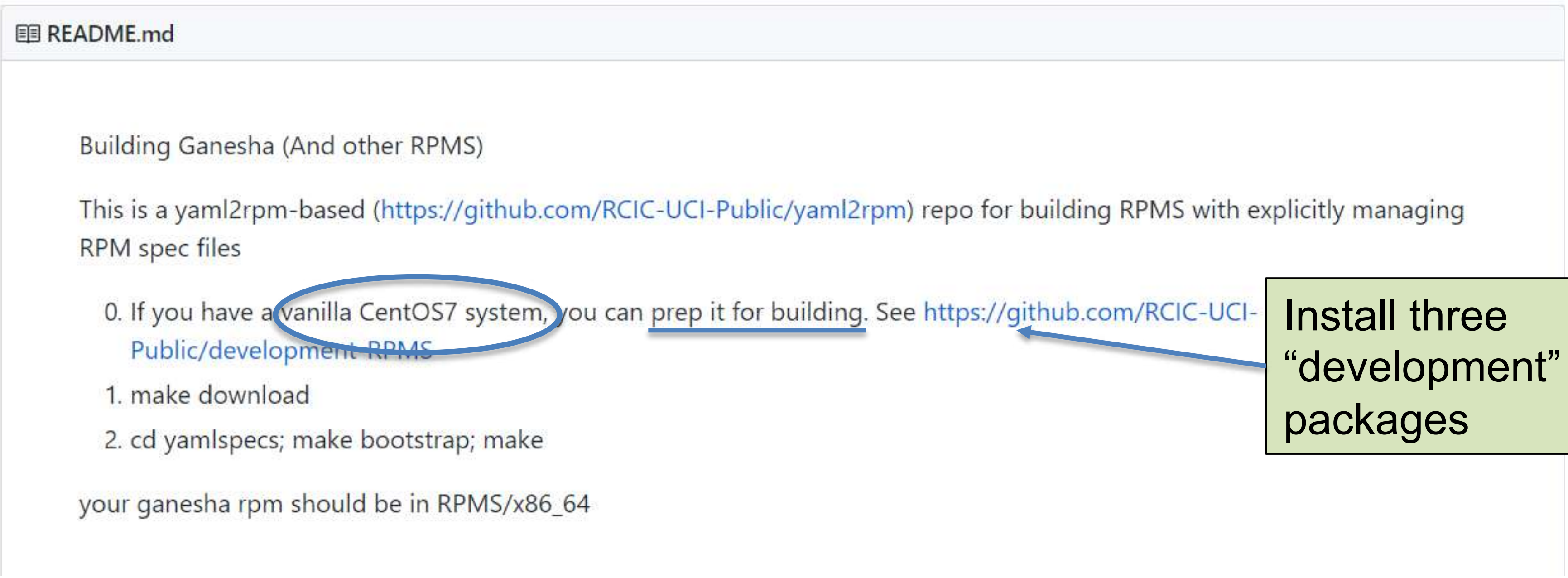
Generates ~ 190 RPMs

Full Dependency resolution and recording. Can install a coherent subset of R modules. Ideal for containers.

- 13 files in Repository
- 78 additional R modules deemed useful. Listed in a single file
- Builds R, and Environment Module (bootstrap)
- Uses CRAN (R online archive) to find all dependencies and encodes in the package
- Creates package yaml files for each module and builds in proper order

Getting Started

- Sample Admix: ganesha-admix
  - https://github.com/RCIC-UCI-Public/ganesha-admix
- YAML2RPM on GitHub
  - https://github.com/RCIC-UCI-Public/yaml2rpm



README.md

Building Ganesha (And other RPMS)

This is a yaml2rpm-based (https://github.com/RCIC-UCI-Public/yaml2rpm) repo for building RPMS with explicitly managing RPM spec files

0. If you have a vanilla CentOS7 system, you can prep it for building. See https://github.com/RCIC-UCI-Public/development-RPMS
1. make download
2. cd yamlspecs; make bootstrap; make

your ganesha rpm should be in RPMS/x86_64

Install three “development” packages

What's Next

- Still in development
  - Needs some documentation on how to handle "outlier" cases.
- Futures:
  - Checkout N admixes – automatically build in dependency order (need to define Admix-admix dependencies)
  - Generate most YAML files from a database (continue to distill package specifics to bare minimum required)
  - Code backend generator for DPKG (Debian) format