A Roadmap for Sustainable Ecosystems of CSE Software

Roscoe A. Bartlett, Oak Ridge National Laboratory

06/26/2015

Introduction

The computational science & engineering (CSE) community is constantly developing and improving diverse algorithms and software for solving complex simulation and analysis problems. The best algorithms implemented as software packages by different experts from many institutions must be integrated into single executables to solve the most challenging problems. For this to happen, the CSE community must be able to affordably develop, integrate, and deploy software built from an ecosystem of these packages in a sustainable way over many decades and many changes in computer architectures (e.g. exascale). Some of the challenges and proposed solutions are the focus here.

Consider a small ecosystem shown in Figure 1 which includes six independently developed and released packages `A`–`F` and four applications `App1`–`App4` that use different sets of them. Here, a **Package** is defined as a piece of software that is released, built, and installed as a single unit. Using stacks of packages like this is difficult for many reasons: these different packages can be released at different schedules, releases can break backward compatibility or introduce new bugs (sometimes subtly and without warning), one or more packages can be difficult to build from source on any given platform, or support for a given package can suddenly disappear. It is because of these challenges that the CSE community has struggled to develop and reuse such stacks of software packages. For example, it is the challenges around a package dependency graph about the size of the one in Figure 1 that motivated the formation of the IDEAS Project [1].

A possible roadmap is outlined for how to achieve these ecosystems in a sustainable way that builds on experience that comes from development and integration projects related to Trilinos [4], SIERRA (SNL), CASL VERA [5] in many others over the last 15+ years in the DOE labs.

A Roadmap for Sustainable Ecosystems of CSE Software

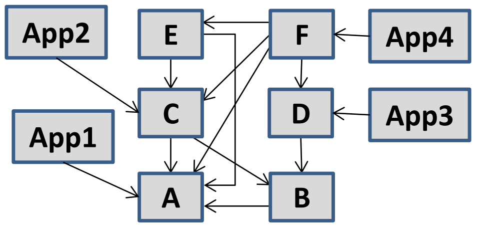

Figure 1: Small ecosystem of packages and apps

There are four different parts to the challenge of developing and maintaining these ecosystems of packages and applications. For each challenge area, past, on-going, and future work to address the issues is mentioned.

1) Lifecycle and software quality of individual packages: The needs of research and the needs of customers with respect to the ecosystem must be balanced. Also, CSE software that is too immature cannot directly participate in a large ecosystem. To address this, a lifecycle model for research-based CSE software based on modern Lean/Agile software engineering principles has been developed [6]. This lifecycle model defines four different lifecycle maturity levels: **Exploratory** (**EP**: primary purpose is to explore alternative approaches), **Research Stable** (**RS**: strong verification tests, clean design and code base), **Production Growth** (**PG**: improving input checking and error reporting, better formal documentation, better regulated backward compatibility with fewer incompatible changes, increasingly better portability and space/time performance , expanding usage in more customer codes), and **Production Maintenance** (**PM**: primary development includes mostly bug fixes and performance tweaks). Legacy software can be grandfathered in by adopting the **Legacy Software Change Algorithm** [6]. Packages in the PG and PM stages are most appropriate to directly participate in larger ecosystems. However, lower maturity packages (e.g. RS) can participate as well as long as they are wrapped under higher maturity packages. This lifecycle model is being further refined as part of the IDEAS Project [1].

2) Sustainability of software packages: What would happen if all support for a software package in the ecosystem disappeared? Would downstream customers or the CSE community be able to adapt? In order to sustain an ecosystem of software packages, one must be able to build all of the non-standard software from source and the software should have the properties of **Self-Sustaining Software** [6]. That is, the package must have an open-source license, be exceptionally well tested (with strong automated tests), have clean structure and code, have minimal controlled internal and external dependencies, and all these properties must apply recursively to upstream dependencies (stopping at standards like C/C++ compilers, MPI, etc.). Packages without these properties represent a risk to the ecosystem.

3) Maintaining compatibility of packages in the ecosystem: This is most difficult of the challenges and the area where the current CSE community is the most lacking. There are several different models to maintain the integration between different software packages ranging from continuous integration (CI), to almost CI, to punctuated upgrades against released versions [7]. While CI is fast and efficient, it requires massive coordination and is the least scalable. Alternatively, punctuated upgrades against released versions is the most scalable but presents great challenges in maintaining compatible sets of released packages. The **Semantic Versioning Standard** [2] was created to help define the rules by which packages name their releases **X.Y.Z** in a way that helps to determine compatibility between different versions of the software. Given this standard, the CSE community can define processes and requirements for package releases needed to sustain these ecosystems. For example, there is a trade-off between how many consecutive backward compatible releases a package puts out and how many prior non-backward compatible releases need to be supported. A package that never breaks backward compatibility needs only support the current release. However, a package that breaks backward compatibility with every release may need to support many prior releases. For example, if all of the packages shown in Figure 1 put out releases at the same cadence, then if package `A` breaks backward compatibility with every release, then the developers for package `A` must support the current and prior two releases in order to make the release of package `F` feasible. One can define a number of processes and rules like this based on simple analysis of the package dependency graph. These processes and requirements are being developed as part of the IDEAS Project [1].

4) Building a compatible set of packages for a given application from source: Because CSE packages tend to break backward compatibility often and are very actively developed, it is not practical (or scalable) to try to put out binary releases of pre-built libraries using "package management" systems like, for instance, `apt` and `yup` for Linux distributions. The only scalable approach is to build the needed packages from source for a specific set of versions for a given set of application codes. The popular approach of wrapping the heterogeneous build systems (e.g. autotools, raw makefiles, raw CMake) for different packages in a single build driver presents many portability challenges. To overcome this, for example, the ASC SIERRA project threw away the native build systems for their 30+ upstream packages and wrote uniform build files with SIERRA itself and it proved to massively improve the portability of SIERRA. Another example is Trilinos, CASL VERA, and other related projects which use CMake TriBITS [3] to provide uniform builds for hundreds of packages in different contexts. Work is under way to extend the TriBITS into a more general meta-build system that will scale to the largest imaginable ecosystems of packages in the CSE community.

References

[1] Interoperable design of extreme-scale application software (IDEAS). https://ideas-productivity.org/.

[2] Semantic Versioning 2.0.0. http://semver.org/.

[3] TriBITS: Tribal Build, Integrate, and Test System. http://tribits.org/.

[4] The Trilinos project. https://trilinos.org/.

[5] Virtual Environment for Reactor Applications (VERA). http://www.casl.gov/VERA.shtml.

[6] R. A. Bartlett, Michael A. Heroux, and James M. Willenbring. Overview of the tribits lifecycle model: A lean/agile software lifecycle model for research-based computational science and engineering software. *e-science*, 2012 IEEE 8th International Conference on E-Science:1–8, 2012.

[7] R.A. Bartlett. Integration strategies for computational science. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 35 –42, 23-23 2009.