Software Integration Challenges

Todd Gamblin
Advanced Technology Office
Lawrence Livermore National Laboratory
tgamblin@llnl.gov

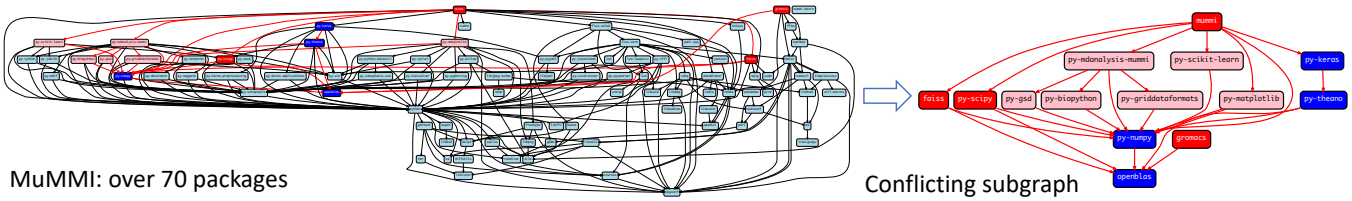

Figure 1: A particularly troublesome conflict in LLNL's MuMMI code, with over 70 dependency libraries

1 INTRODUCTION

Component-based software design allows software developers to build increasingly complex systems, but software complexity is reaching its limits. With the widespread availability of open-source libraries, the number of components integrated into typical applications has exploded, and the integration problem has become more challenging. Integrating tens or hundreds of components, or *packages*, together consumes an increasing fraction of developers' time. Package managers offer a partial solution, and they have been reasonably successful whiting siloed (single distribution, single language, single compiler, etc.) ecosystems. Unfortunately, in scientific computing, the ecosystem is far more diverse. There is no one CPU or GPU architecture, OS, or compiler—there are many. Complex programming environments, machine learning stacks, GPUs, and potentially other types of accelerators must all be supported and integrated together, but introducing these complexities makes it even harder to assemble a software stack that works.

At a fundamental level, relationships among software packages are not well understood. In most package management systems, a package may depend on another, but the characteristics of that relationship are often not known or specified beyond a simple version constraint. If *A* depends on *B*, can *A* work with all versions of *B*? Specific ones? Can *A* work with *B* on some platform that *B* does not support? What if the two packages are built with different compilers that use different OpenMP implementations? What if *B* is built with special flags? Can we build a version of *A* that *is* compatible with the OS's version of *B*? More often than not, developers don't know the answers to these questions, and they must simply try configurations until one works.

Collegeville Workshop 2020, July 21-23, 2020, Virtual Event

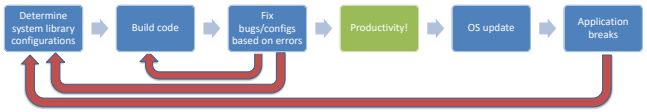

Figure 2: The process of integrating libraries over time

```
  opengl:
    paths:
      opengl@1.7.0: /usr
    buildable: False
  openglu:
    paths:
      openglu@1.3.1: /usr
    buildable: False

  # Lock down which MPI we are using
  mvapich2:
    paths:
      # clang mvapich2
      mvapich2@2.3%clang@9.0.0 arch=linux-rhel7-ivybridge: /usr/tce/packages/mvapich2/mvapich2-2.3-clang-9.0.0
      # gcc mvapich2
      mvapich2@2.3%gcc@8.1.0 arch=linux-rhel7-ivybridge: /usr/tce/packages/mvapich2/mvapich2-2.3-gcc-8.1.0
      # intel mvapich2
      mvapich2@2.3%intel@19.0.4 arch=linux-rhel7-ivybridge: /usr/tce/packages/mvapich2/mvapich2-2.3-intel-19.0.4
    buildable: False
```

Figure 3: MuMMI developers pin the versions of around 20 system packages

2 INTEGRATION COMPLEXITY

To understand the problem better it helps to look at an actual development workflow. Figure 1 shows part of LLNL's MuMMI code. MuMMI integrates molecular dynamics models, AI surrogate models, and other components to model drugs, recently including potential treatments for COVID-19. The code itself comprises over 70 packages. The team uses the Spack [7] package manager to integrate them in a single build. In the MuMMI deployment workflow 2. the team first identifies dependencies they want to use on the host machine, and they then write a configuration file that tells Spack about these packages 3. They build their code with these settings, fix any bugs that arise, and run. As shown in

the figure, there are many ways this can go wrong. For example, the team frequently updates package versions to get new features and functionality. While this may seem simple, any update can cause cascading problems. For example, the team once needed a newer version of the Keras package:

(1) Team "just" needed a new version of Keras,
(2) which needed a new version of Theano,
(3) which needed a new version of Numpy,
(4) which needed a newer version of OpenBLAS.
(5) Team was using system OpenBLAS, which was too old
(6) Team had to build many versions of OpenBLAS to find one compatible with GROMACS, SciPy, and FAISS
(7) Finally, rebuilt the entire stack for ABI compatibility.

This is a complex, cascading issue involving incompatibilities among 8 packages. Figure 1 shows the sub-graph containing conflicting packages in red, both in and outside of the 70+-package MuMMI code. Dealing with these types of cascading issues is daunting. Debugging this particular problem took 36 person-hours, and this is far from the most serious problem the team has dealt with. The need for new package versions arises frequently, causing similar pain. OS updates are a constant source of frustration, as they change package versions underneath the application software. Developers must adjust their system configuration file (Figure 3) and rebuild with each update. Incompatibilities between the proprietary system PMI package (a dependency of MPI) and the team's builds cost hundreds of hours to find and fix.

In HPC, these issues have been called *software collapse* [8]. However, dependency problems are not unique to scientific computing. A study of 26.6 million software builds performed by 18,000 developers at Google showed that 50-60% of build issues (for both C++ and Java) were dependency-related (Figure 4). A similar study of 491 developers at ING showed that dependencies were the single largest contributing factor to release delays, regardless of whether the teams made frequent or infrequent releases (Figure 5). A further study showed that developers avoid critical upgrades due to costs associated with dependencies [10].

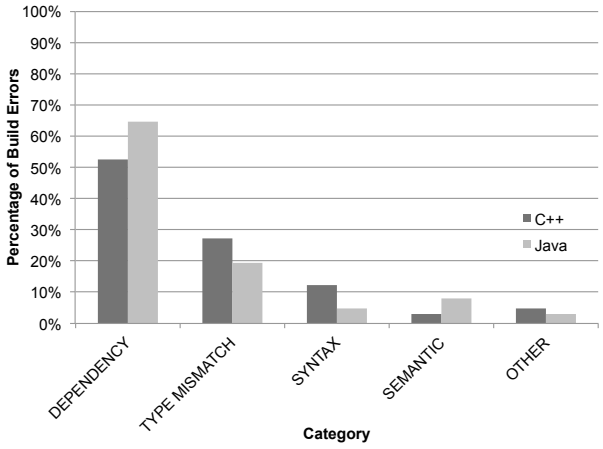

Figure 4: Types of errors in 26.6 million builds at Google [12]

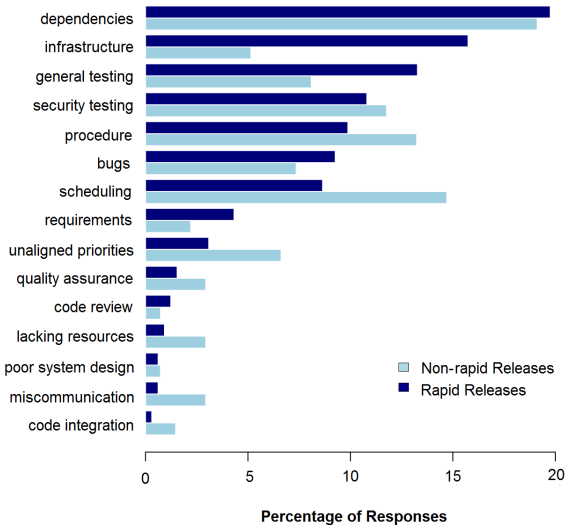

Figure 5: Factors perceived to cause release delays among 491 developers at ING [9]

3 EXISTING STRATEGIES

Integrating dependencies is fundamental to modern software development, so you might think that there would be good, established practices for dealing with it. Unfortunately, not even Google has answers for this. In their 2020 book *Software Engineering at Google* [14], Winters et al. call dependency management "one of the least understood and most challenging problems in software engineering," and they say "we definitely cannot claim to have all the answers here; If we could, we wouldn't be calling this one of the most important problems in software engineering." While they do not claim to have a complete solution, they do outline some of the existing practices and their trade-offs. Figure **??** shows them.

3.1 Bundled distribution.

The most common strategy is to use a curated, bundled software distribution. The curators pick a set of versions and package configurations, ensure they work together, and push them out. The model works—it's the one nearly every Linux distribution uses, as well as projects like xSDK or E4S (ECP's stack). The disadvantage is that it is confining. To ensure compatibility, the versions are mostly fixed, so teams cannot easily choose other versions as the MuMMI team needed to.

	Bundled Distribution	Semantic Versioning	Live at Head
Examples	Linux distributions (Red Hat, Debian) E4S, xSDK, Anaconda Spack with locked versions	Spack NPM, Cargo, Go Most language dependency managers	Google, Facebook, Twitter
Idea	Curate a large set of mutually compatible dependencies	Use uniform version convention, Solve for compatible set	Everything in one repository, Developers test changes with all *dependents*
Pros	Stability (if software is included)	Frequent updates Only relies on local information Works in theory	Frequent updates Stability, consistency All changes tested
Cons	**Infrequent updates** **High packaging/curation effort** **Lack of flexibility**	**Versions are coarse** **Developers over-constrain/over-promise** **Errors start to dominate at scale**	**Doesn't scale beyond a single organization** **High computational cost of testing** **Lack of flexibility (typically just one target env.)**

Figure 6: Existing strategies for managing dependencies [14]

3.2 Semantic Versioning.

The second strategy, which most language-specific package managers are moving towards (npm for Javascript, Cargo for Rust, Ruby Gems, and many others), is to use *semantic versioning*. Systems like this use the version and potentially other information (feature flags, etc.) as a proxy for compatibility. SAT solvers are used to find valid configurations from large package databases. Unfortunately, studies have shown that version compatibility metadata is provided by humans and that it is frequently inaccurate [4, 6], and it does not work well at large scale (where the likelihood of a breaking change in a stack increases). In HPC, where packages can be built in many different ways and theree is more than just version information to consider, compatibility is even harder to assess accurately. Spack [7] uses this approach, but it frequently runs into limitations with its current compatibility metadata. It does not currently model enough build parameters to cover all aspects of binary compatibility, so builds can break if users create too many new configurations.

3.3 Live at Head.

The last strategy, and the one currently used at Google and other large tech companies, is to "live at head". In this model, all changes to software and to configurations are checked into a single repository, and every change is tested before it is merges into the single configuration. Interestingly, this model puts more responsibility on individual package developers to test their changes *because* any change can be checked against any package that uses it. While this ensures a tested stack, it has two main drawbacks. First, like distributions, it limits version flexibility – you must use what is in the repository. Second, the resource requirements to test every change this much in DOE are huge. Google has one main environment and one tool chain where their production software runs, but DOE has tens or hundreds. The combinatorial burden to do this everywhere seems too great, at least without careful planning. Finally, this approach requires a single repository, at least as currently implemented. Scientific software lives in many different repositories and spans communities, and the level of coordination and trust required for a single large repository is not there.

4 ISOLATED COMMUNITIES

Currently, each ecosystem, package manager, or other separately curated body of software tends to develop its own preferences and defaults. Developers who add packages to semantic versioning-based systems tend to lock their versions to values they know will work *for themselves*. Distributions make assumptions about how their code will be compiled, what environment it will build in, what versions are provided, etc. Each project ends up self-consistent, but there are no guarantees about compatibility or versioning *across* distributions. Ultimately, this means that we cannot easily leverage effort from different packaging systems. The world must be reinvented again and again, but differently, for each use case. If we try to combine packages, they will either not work, or conflict due to overly narrow constraints (Figure 7).

5 RECOMMENDATIONS

HPC needs ways to better reuse software, and the main obstacle to doing that is insufficient compatibility information for binaries. HPC users need flexibility to build their own versions and configurations of software, so it is likely that the distribution model and live-at-head will not work well for HPC. The versioning model of packaging, however, is currently limited by what humans can specify. To solve this, we make three major recommendations below.

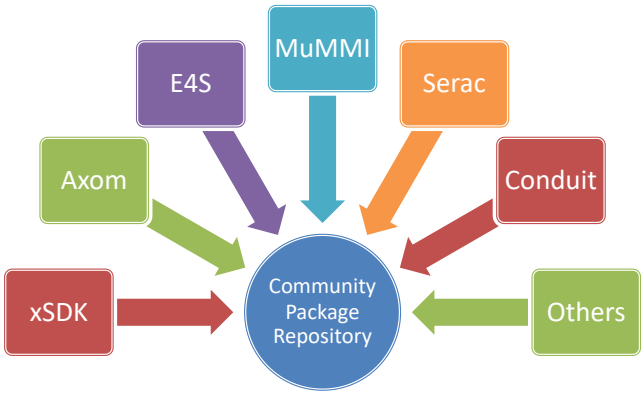

Figure 7: Mixing is hard: if all projects mandate specific versions/configurations, conflicts will eventually arise that prevent all packages from building.

5.1 Compatibility modeling

If we want to enable greater code reuse and better reasoning about compatibility, we need better compatibility *models*. For any binary, we should be able to determine whether it can be linked with another, i.e. whether its ABI is compatible with another. To do this we need much deeper information about language and parallel runtime libraries, types that used by particular binaries, etc. DWARF provides much of this information but is too heavyweight to include in all binaries. We need better ways to embed this type of information in ELF and other binary formats to enable binary reuse *after* compilation.

5.2 Binary analysis

We need analysis that can inspect existing binaries and determine compatibility information of interest. If successful, this would allow us to more easily reuse system libraries (as we could look at their interfaces and *know* whether they were compatible with what we are trying to build. Again, we need better information in the binaries to accomplish this.

5.3 Better solving

We need ways to automate the reasoning behind software integration and builds. SAT solvers of current package managers can find *valid* builds, but they use human-generated version information to do this, and they cannot make guarantees about ABI. We need tools that can solve on the interface (ABI) information directly, and that can find not just a *valid* configuration from a database of packages, but one that is *sound* and guaranteed to link. Prior work has shown that finding compatible versions is NP-complete [1–3, 11, 13], and that modern dependency networks are very complex [5], but there have also been tremendous advances in solver technology over the past decade or two. The time is right to start looking at building software as a true reasoning and quality optimization problem instead of leaving this to the humans. With the right set of solvers, we could find not only *compatible* package combinations, but *optimal* ones.

REFERENCES

[1] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.

[2] R. D. Cosmo. EDOS deliverable WP2-D2.1: Report on Formal Management of Software Dependencies. Technical report, INRIA, May 15 2005. hal-00697463.

[3] R. Cox. Version SAT. https://research.swtch.com/version-sat, December 13 2016.

[4] A. Decan and T. Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 2019.

[5] A. Decan, T. Mens, and M. Claes. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, ECSAW '16, New York, NY, USA, 2016. Association for Computing Machinery.

[6] J. Dietrich, D. J. Pearce, J. Stringer, A. Tahir, and K. Blincoe. Dependency versioning in the wild. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, pages 349–359. IEEE Press, 2019.

[7] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and W. S. Futral. The Spack Package Manager: Bringing order to HPC software chaos. In *Supercomputing 2015 (SC'15)*, Austin, Texas, November 15-20 2015. LLNL-CONF-669890.

[8] K. Hinsen. Dealing with software collapse. *Computing in Science & Engineering*, 21(3):104–108, 2019.

[9] E. Kula, A. Rastogi, H. Huijgens, A. van Deursen, and G. Gousios. Releasing fast and slow: an exploratory case study at ING. In M. Dumas, D. Pfahl, S. Apel, and A. Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 785–795. ACM, 2019.

[10] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.

[11] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 199–208, 2006.

[12] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, pages 724–734, 2014.

[13] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 178–188, USA, 2007. IEEE Computer Society.

[14] T. Winters, T. Manshreck, and H. Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020.