The Layers of CSE Software Sustainability

James M. Willenbring, Sandia National Laboratories, North Dakota State University

1. Software Sustainability

There are many aspects to software sustainability. Not surprisingly, different definitions emphasize some aspects over others. Sehestedt, et al. define software sustainability as “cost efficient maintainability and evolvability” [1]. The proposal for the Software Sustainability Institute [2] said that software sustainability is the “capacity of the software to endure”, while Katz offered “sustainability means that the software will continue to be available in the future, on new platforms, meeting new needs” [3]. Each of these definitions is insightful and useful, but also very interesting, particularly in their differences. The first definition treats cost efficiency as a first order concern, while the last two do not explicitly address it. The definition from the Software Sustainability Institute does not require the software to meet new needs, only to endure.

Rosado de Souza, et al. refer to two categories of software sustainability - intrinsic sustainability, pertaining to characteristics of the software, and extrinsic sustainability, pertaining to the software development environment [4]. This is a useful dichotomy in that it captures both the importance of software quality measures, such as maintainability, extensibility, and portability, as well as the reality that many factors beyond the software itself impact sustainability.

The Software Sustainability Institute definition appears more intrinsically focused than the others in referencing “the capacity of the software”. Based on this, it may be reasonable to consider software with certain attributes to be sustainable, although the software might not be sustained for other reasons. The Katz definition appears intrinsic/extrinsic neutral, requiring only that the software continue to be available. Taken to the extreme, one could say that a software project deemed too big to fail by a company with sufficient resources could be considered sustainable regardless of other factors. Sehestedt appears to require some measure of both intrinsic and extrinsic sustainability, as internal and external factors both impact cost efficiency.

My purpose is not to analyze the relative merits of these definitions. Rather, my purpose is to establish that software sustainability has many factors and is difficult to define succinctly.

2. Computational Science and Engineering (CSE) Software Sustainability

CSE Software sustainability is as important as it is complicated. CSE software commonly builds on other software in the software stack. For example, an application might call a preconditioner prior to a linear solver. For an application using a third-party solver and/or preconditioner, it is important that the third-party software be sustainable, otherwise the application team may end up investing a lot of resources in using the third-party software, or may end up writing their own.

Reimplementing existing functionality is a common solution for avoiding the headaches associated with using third-party software. This can greatly hinder developer productivity, and even the sustainability of a project, but can be the best choice. The CSE community is generally apprehensive about having dependencies on third-party software. Much of this apprehension is warranted; developers commonly struggle with existing dependencies. Further, maintaining interoperability between the dependencies a project has can be very challenging.

This kind of sustainability concern is different than the sustainability of the application code itself in many respects. For that reason, I propose three layers of sustainability for CSE software. Each layer poses different, but related challenges. I argue that we cannot provide a sustainable and productive environment for researchers to leverage the work of others with the ultimate outcome of production-quality codes without addressing all three layers of sustainability.

2.1.The First Layer

CSE software projects benefit from different kinds of sustainability and related efforts in multiple ways. Intrinsic sustainability of the code base means the project is maintainable and extensible. Fewer resources must be invested per feature, and for maintenance, allowing funding proposals to offer more for a set budget. Extrinsically sustainable CSE software has sufficient budget and staffing, as well as sound and scalable tools and processes.[1] I consider sustainability at this level to be the first layer of sustainability. More formally, I define the first layer of CSE Software Sustainability for a project to be those aspects of sustainability relating directly and specifically to the code base and project circumstances, such as staffing, funding, tools, processes, etc.

Code teams typically have a lot of impact on this layer. How functionality is designed and tested is clearly impacted by the team. Even funding stability, while not in the direct control of the team, is influenced by the team. Measures of coupling, cohesion, complexity and other common software quality metrics can be useful in the analysis of the first layer of sustainability.

2.2. The Second Layer

In CSE, it is common for applications and libraries to have dependencies on CSE libraries. An application development team needing a Krylov solver, for example, ideally should not develop their own, but rather utilize one or more solvers developed, maintained, and supported by experts in Krylov methods. However, every dependency on third-party software, particularly software that the development team has little influence on, represents a risk.

The size of the risk varies based on a multitude of factors, such as if the software is a required or optional dependency, if there multiple options or a close substitute for filling the dependency,

[1] Although I refer to projects as “sustainable” and mention characteristics of those projects that are frequently associated with sustainability, no project is perfectly sustainable, nor completely unsustainable, and likewise projects do not fully possess or lack any the various attributes of sustainability.

and if the software implements a common interface. Additionally, the sustainability of the software factors heavily into the size of the risk associated with depending on it.

Sustainability issues related to the dependencies of a software project make up the Second Layer of CSE Software Sustainability for that software project. To illustrate the kinds of dependency issues found in the second layer, consider the simple example in Figure 1 from the xSDK [F] showing the dependencies of three applications.

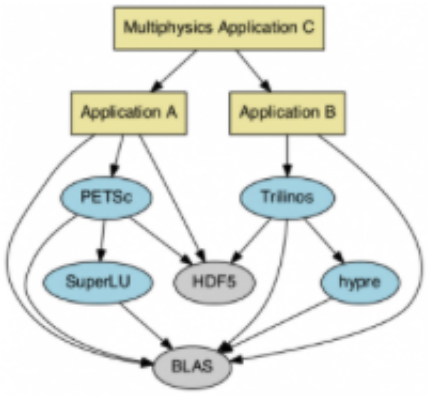

Figure 1

Application A depends on PETSc, and PETSc and the Application both have direct dependencies on HDF5. This requires a little coordination as there needs to exist a version of PETSc that uses a version of HDF5 that is also interoperable with Application A's direct HDF5 calls.[2]

Additionally, Application B depends on Trilinos, which also depends on HDF5. This is a little simpler situation than the previous one because Application B does not have a direct dependence on HDF5. However, there is also a third application, Multiphysics Application C, that depends on both Application A and Application B. This means that now there needs to be versions of Application A, PETSc, and Trilinos that all use a single version of HDF5 and are generally otherwise interoperable (e.g., no duplicate symbols, build with compatible compilers, etc). This is the Second Layer of Sustainability from the viewpoint of Multiphysics Application C. Provided Multiphysics Application C has sufficient visibility and priority for all of the code teams involved, it is realistic for this simple, single example to not only to have compatible versions that exist, but also to maintain frequent compatibility with new versions of the codes moving forward.

2.3. The Third Layer

Some of the challenges associated with achieving sustainability at the second level are a result of the many different contexts in which CSE software is used. A popular math library, for example, might be used in conjunction with dozens of other software packages in a dizzying number of combinations. To illustrate this, consider a 4th application, Application D that depends on Application A and Trilinos, but additionally uses the Trilinos interface to SuperLU, which Application B did not (that interface was not depicted in Figure 1). Now, for both Multiphysics Application C and Application D to work, in addition to all of the previous requirements, there needs to be versions of PETSc and Trilinos that can use a common version of SuperLU, and these same versions also need to be compatible with a version of HDF5 that can be used for both as well as Application A. Considering only the slightly larger set of CSE

[2] Software engineering solutions could make this unnecessary depending on the nature of the dependencies, but require additional maintenance and are beyond the scope of this discussion.

software packages shown in Figure 2, the potential combinations of dependencies quickly becomes unmanageable.

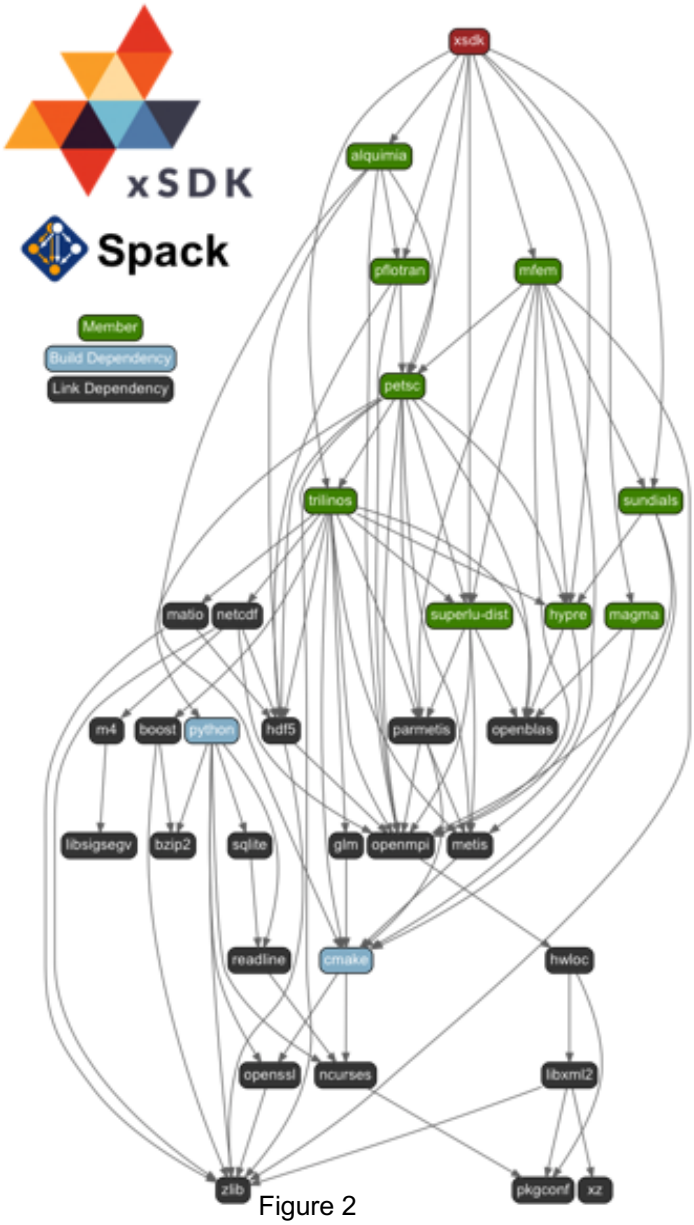

Figure 2

This necessitates the concept of a Third Layer of Sustainability for CSE software, a layer that is concerned with the interoperability of a well-defined ecosystem of software, extending beyond the usage of a single application. One such set of software is the ECP Software Technologies software products and their dependencies, a subset of which is included in a grouping of software called The Extreme-Scale Scientific Software Stack (E4S). It is not practical to consider *all* CSE software, as there is no definition of what that means and measuring interoperability would not be feasible or particularly useful.

Sustaining a CSE software ecosystem does not require universal, indefinite sustainability. Rather, what is required depends on the importance of each software product to the ecosystem and the customers of those products. Over time, products may be added and removed, but a healthy ecosystem will reduce the negative impact of changes through testing, versioning, and common interfaces.

3. Summary and Next Steps

Sustainability within the context of CSE software is complicated, yet necessary for community members to have confidence in adopting code written by other teams of developers, which reduces effort duplication and positively impacts developer productivity. I have proposed three Layers of Sustainability for CSE software to assist in framing the issues that must be overcome to achieve or even define sufficient sustainability. In future work, I want to better flesh out the three layers as well as the challenges and potential solutions associated with each. Additional future work will also look at the applicability of software metrics for each layer, and define additional sustainability metrics for the second and third layers.

Bibliography

[1] Stephan Sehestedt, Chih-Hong Cheng, and Eric Bouwers. 2014. Towards quantitative metrics for architecture models. In *Proceedings of the WICSA 2014 Companion Volume* (WICSA '14 Companion). ACM, New York, NY, USA, , Article 5, 4 pages. DOI: http://dx.doi.org/10.1145/2578128.2578226

[2] Stephen Crouch, Neil Chue Hong, Simon Hettrick, Mike Jackson, Aleksandra Pawlik, Shoaib Sufi, Les Carr, David De Roure, Carole Goble, and Mark Parsons. Nov-Dec 2013. *"The Software Sustainability Institute: Changing Research Software Attitudes and Practices,"* Computing in Science & Engineering , vol.15, no.6, pp.74,80. DOI: 10.1109/MCSE.2013.133.

[3] Daniel Katz. 2016. Defining Software Sustainability. Retrieved from https://danielskatzblog.wordpress.com/2016/09/13/defining-software-sustainability/.

[4] Mário Rosado de souza, Robert Haines, and Caroline Jay. 2014. Defining Sustainability through Developers' Eyes: Recommendations from an Interview Study. DOI: https://doi.org/10.6084/m9.figshare.1111925.v1