On the Role of Self-Similarity in Component-Based Software

Rob Armstrong and Benjamin A. Allan †
{rob,baallan}@ca.sandia.gov

David E. Bernholdt and Wael Elwasif ‡
{bernholdt,elwasifwr}@ornl.gov

Abstract

This is a speculative work meant to stimulate discussion about the role of subsumability in self-similar software structures for computational simulations. As in natural phenomena, self-similar features in framework structures allow the size and complexity of code to grow without bound and still maintain apparent coherence. As in crystal growth, the coherence may be maintained by the application of a repeated pattern, or patterns may, as in fluid mechanical turbulence, be scaled by size and nested. Examples of these kinds of patterns applied to component systems in particular will be given. Conclusions and questions for discussion will be drawn regarding the applicability of these ideas to component architectures, complexity, and scientific computing.

1 Introduction and Motivation

One of the key ideas behind component-based software engineering and computational frameworks, is that of aiding software developers in getting a grip on the burgeoning complexity of their software systems. Encapsulating code into components and taking advantage of their plug-and-play nature are typically viewed as the primary means to this end. Viewed pessimistically, however, this is only a delay – until the component assemblies become unmanageably complex. Self-similarity, a common motif in nature, may provide insights which will help address the complexity problem in more scalable ways. In this note, we consider manifestations of self-similarity in existing HPC scientific software environments and component software systems and how self-similarity might benefit such systems.

2 Self-Similarity in Component Software Structures

As a definition we will say that a component system is self-similar if every composition can also be used as a component in the same system. More formally, the structure of an arbitrary software framework can be thought of as a set of objects ($C$) and a set of morphisms ($\gamma_i, \sigma$) among them. We identify the set of morphisms ($\gamma_i$) that preserve the structure of the system at a single scale:

$$ C_0 \xleftarrow{\gamma} C_0 $$

Further there exists a scaling map, $\sigma$, that carries every element of $C$ into the next highest scale:

$$ C_0 \xrightarrow{\sigma} C_1 $$

---

*This work has been supported in part by the U. S. Department of Energy’s Scientific Discovery through Advanced Computing (SciDAC) initiative, through the Center for Component Technology for Terascale Simulation Software, of which Oak Ridge and Sandia National Laboratories are members.

†Sandia National Laboratories, Mail stop 9158 7011 East Ave, Livermore CA, 94550-0969, USA. This is an account of work performed under the auspices of Sandia Corporation and its Contract No. DE-AC04-94AL85000 with the United States Department of Energy.

‡Oak Ridge National Laboratory, P. O. Box 2008, MS 6016 Oak Ridge, TN 37831–6016 USA. Oak Ridge National Laboratory is managed by UT-Battelle, LLC for the US Dept. of Energy under contract DE-AC-05-00OR22725.
where \( C_1 \) is of the next highest scale from \( C_0 \). Further, to assert the first property of self-similarity, the exact same morphisms must exist among \( C_1 \) as they do for \( C_0 \):

\[
C_1 \xrightarrow{\gamma_1} C_1 \tag{3}
\]

The second requirement of self-similarity is that \( C_0 \) and \( C_1 \) are isomorphic. These requirements arise from the fact that the software system must remain identical in form and function at every level of scale mapped by \( \sigma \). Note, however, that \( \sigma \) is not usually this isomorphism. Equations 2 and 3 imply a hierarchy of scales:

\[
\begin{array}{c}
C_0 \xrightarrow{\sigma} C_1 \xrightarrow{\sigma} C_2 \xrightarrow{\sigma} \cdots \\
\{\gamma_0\} \downarrow \quad \{\gamma_1\} \downarrow \quad \{\gamma_2\} \\
C_0 \xrightarrow{\sigma} C_1 \xrightarrow{\sigma} C_2 \xrightarrow{\sigma} \cdots
\end{array} \tag{4}
\]

This hierarchy, in component systems, almost always takes the form of a container of components that, externally, is itself a component (see Fig. 1). In the rest of this note we will confine ourselves to this case, mindful that it is a subset of the more general formulation.

### 3 Examples

To aid and motivate reasoning about subsumable architectures in components, we offer two examples. The first is the Common Component Architecture that is designed to view every application composed of CCA components as a component itself. The second, as a counter example, considers a containerized architecture based on the build utility `make`.

#### 3.1 The Common Component Architecture

The Common Component Architecture (CCA) [1] incorporates self-similarity into its design in several ways. First, services provided by the CCA framework can be implemented as components which use the ServiceRegistry interface to register certain interface instances as ubiquitous (that is, made available to other components without first requiring a user-defined connection to the using component). Second, components can be created which encapsulate assemblies of other components. Using the BuilderService interface, ports from the interior component assembly can be exported by the encapsulating component. The same mechanism even allows the parent component to programmatically assemble the interior components. Third, arbitrary code, including components, can manipulate a CCA framework through the AbstractFramework service. The parent code can create component assemblies within the encapsulated framework. As with component assemblies encapsulated within components (above), it is possible for interfaces from the interior components to be exported by the code that manipulates the framework.

To provide a concrete example [2], one can easily envision using these mechanisms in the construction of a coupled multi-physics simulation. Normally, each different sub-application would be composed of numerous components. Forming the several sub-applications into a single coupled simulation could become quite challenging simply based on the total number of components and interfaces that must be tracked. If, on the other hand, each sub-application were encapsulated as a single component, exposing only the particular interfaces needed for the coupling, assembly of the full simulation would be comparatively straightforward. The CCA allows this to be done either as a parent component directly managing the assembled components of the sub-application (using BuilderService), or with each sub-application running in a separate framework instance and wrapped up as a component (using AbstractFramework together with BuilderService).
The BuilderService and Abstract-Framework ideas, which are the core of the CCA’s self-similarity, arose from a desire to provide programmatic mechanisms to create and control component-based applications. These features of the CCA were more readily accepted because they reuse the basic concepts of the well-defined component model rather than because they create a self-similar architecture. The ServiceRegistry idea came from a desire to modularize the framework itself.

3.2 A Recursive Structure for make

As an illustration of the subtle difficulties encountered in retrofitting existing applications with a recursive structure, we consider the make build utility [3]. Many complex software packages use make in a self-similar (recursive) fashion to great advantage. However attempts to encapsulate the builds for multiple packages under the control of a single Makefile tend to fail.

For concreteness, we consider several separate packages, all built with the make and make install commands. Following the pattern of Fig. 1, we will regard make and install as methods and the package as a component. Figure 2 illustrates the simplest possible scheme of this kind: when the container’s make is called, it invokes make on all of its constituents, and similarly with install. This arrangement has all of the properties identified with self-similar structure mentioned above.

Even in this simple case, the results will probably not be satisfactory. The probable reason for packaging several make subpackages into a single build is that they are related or inter-dependent in some way. Neither make nor the self-similar make container we have constructed expose these dependencies in a way that they can be exploited. For example one component may need it’s make and install run before another one’s make is run. Although we are free to order the execution of the components arbitrarily, blindly invoking make on all of the container’s constituents will probably result in an error.

The fundamental problem is that make does not provide very effective mechanisms to specify dependencies external to the Makefile (component). For recursive use of make within a single package and a single directory hierarchy, experienced programmers recognize that the dependency mechanism is often barely adequate to satisfy the needs of a complex software build. Make provides even less support for the expression of dependencies external to the package, and therefore few Makefiles include them. Autoconf and the various Linux package management systems are, in part, a response to this limitation of make.
4 Discussion and Questions

In this note, we have outlined some thoughts on the nature and possible ramifications of self-similarity as a design dimension in component frameworks. While the notion of self-similarity is not a novel idea (it has been studied in areas such as fractals, network traffic, biological systems, cybernetics, among others), consideration of it in the design and analysis of component frameworks is a relatively new approach [4].

An analysis of make using self-similarity ideas and comparisons with component models helps us understand its failure to scale for an important use case. By comparison with make, the CCA component model has more explicit external interfaces (“Ports” and port types) and stronger provisions for expressing dependencies through the uses-provides pattern. This coupled with the BuilderService and AbstractFramework self-similar container interfaces potentially yields more satisfactory results. Though examples of all of the features of self-similarity have been demonstrated in CCA and exist in some other frameworks, we are not yet aware of them being routinely used in any applications.

Further consideration of self-similarity (especially in the context of computational frameworks) poses some interesting questions:

- What new capabilities would be available to component-based applications in a self-similar framework that would otherwise be impossible or difficult to achieve?
- How should support of self-similarity be considered in the design of computational components frameworks? How much added complexity (to the framework) would supporting partial or complete self similarity entail? Would the benefits (added application capabilities) outweigh this extra complexity?
- While containers that themselves are components may be necessary to harness the complexity of component-based applications, would the pervasive use of such a feature result in growing complexity in the design of components’ external interfaces and render the whole approach inaccessible to outside developers?
- While nature frequently uses self-similarity, it also frequently displays an organizational complexity (and failure mode complexity) far beyond what any single human being (and often large groups of them) can comprehend. Is taking advantage of self-similarity the solution to software complexity? What are its limits likely to be? What will “break” it?

References


