Scaling Up the 3Cs Model
A Schema for Extensible Generic Architectures

Larry Latour and Curtis Meadow
University of Maine
Orono, Maine, 04469

larry@gandalf.umcs.maine.edu

Abstract

The 3C model was first developed in [Trac 89] and reviewed at last year's workshop [SMTR 90]. This position paper responds to that review, and in doing so outlines a framework for thinking about 3C aspects of *an architecture* of components. Such components can be said to be *normalized* in that they capture one concern of the architecture in their content.

Keywords: 3C model, generic architectures, normalized components.

1 Introduction

The 3C model was presented and reviewed at our previous workshop, *The Third Annual Workshop: Methods and Tools for Reuse* [SMTR 90]. Will Tracz summarized his perspective on the discussion:

> The 3C model was accepted in part by most of the attendees. The most critical critisism focused on the granularity of the modules. I believe people were trying to scale up the model to apply to larger subsystems and were having difficulty in expressing the semantics. Another area that needed further clarification is the process of differentiating the roles of importation ("with" clause in Ada) and parameterization (genericity in Ada) with regard to interface design.

Partly in response to this need to "scale up" the 3C model, and partly to explore related issues such as the differing (actually complementary) roles of importation and parameterization, we at the University of Maine have been exploring how the model can be used to design and craft:

- integrated component libraries, and
- generic software architectures

By an integrated component library we mean a library (1) whose interface, or *concept*, is defined by an organized collection of module abstractions, and (2) whose implementation, or *content*, is defined by a collection of "normalized" subcomponents, each of which encapsulates a single design concern and derives its *context* from both the environment in which the library is used and from other library subcomponents. Such subcomponents are mixed and matched to realize the library interface abstractions.

Such an integrated library has many of the same properties as a generic architecture. That is, it provides not only a collection of components but an integrated framework for the construction of a family of systems from such components.

We have proposed a schema for thinking about such integrated frameworks, called the *layered generic architecture schema*, or LGA schema. In it we attempt to deal with issues related to integrated collections of components (concept organization, separation of concerns, contextual dependencies between components, etc.). We present an overview of the schema and discuss issues relating to it in the following sections.

2 The 3C Model and "Normalized" Components

As a starting point to building such frameworks, the 3C model provides us with a way to think about the various aspects of a single component. As such, it forms the basis for constructing "normalized" components within a framework. We take the term *normalization* from the relational database world, since:

- *intuitively* we would like to think of a normalized component in the same sense that we think of a normalized relation. That is, just as a normalized relation captures the essence of a single entity with no embedded subentities, so does a normalized component capture the essence of a single concept and implementation concern, with context factored out and either imported or provided through parameterization.

- *formally* we would like to be able to provide rules for discovering and factoring out context in the same way in which we have formulated rules for normalizing relations, i.e., by looking for various forms of structural replication and factoring them out. An interesting avenue of exploration, and one in which we have just begun to think about, is the formalization of such rules.

3 Related Work

A good deal of work has recently been done on the design and construction of integrated libraries/generic architectures. The CAMP (Common Ada Missile Packages) [CAMP 85] effort provided one of the first such collections, approximately 200 normalized parts organized into a number of subsystem architectures along with tools for instantiating such architectures. Stepanov and Musser [Muss 88] have had an ongoing project constructing integrated component libraries consisting of normalized subcomponents and a framework for instantiating them. Their initial Ada work was in the basic data structures domain, with a library of parts and a textbook [Muss 88] describing them. Uhl and Schmid [Uhl 90] have developed a systematic catalogue of reusable abstract data types, in which they have similarly explored these issues. Bassett [Bass 87] has developed a collection of code frames used to construct a wide variety of EDP systems. Batory [Bato 91] has constructed a collection of normalized database components with tools for composing them into a number of database management system variants.

Along with the above efforts specifically designed to provide integrated software libraries/architectures, we have looked at a number of efforts that have similar characteristics. The UNIX environment, for example, can be viewed as a multi-level collection (system calls, library facilities, composition tools) of normalized components along with rules and tools for composing and tailoring them for a particular domain. Similarly, the X windows system is a multi-level collection of components (Xlib components, Widgets, Intrinsics, Resource files) along with rules and tools for their composition. In fact, the philosophy of the X effort, to provide a collection of "look and feel" independent services, is similar to our approach, i.e., to provide an extensible, adaptable collection of building blocks and rules for composing them into flexible, evolving library interfaces/systems.

From these studies and our own experiments using Booch component [Booc 87] implementations and Stepanov and Musser's generic components, we derived a schema for such architectures, allowing us to isolate and deal with a number of orthogonal architectural issues. Such a schema is built on and is complementary to the 3C "schema" for dealing with single component issues.

4 The LGA Schema

As a starting point in the development of our schema, we draw on the 3C model of component structure. We observe that the separation of concepts from their content has been relatively well explored and widely practiced, but that the isolation of contextual dependencies has not been as well explored, nor has it been subjected to a formal analysis.

It seems reasonable to assume that careful isolation of contextual dependencies in the modular structure of an integrated library/generic architecture can produce a framework that is not only capable of being instantiated to a large number of actual components/systems, but that can be easily extended and adapted by "plugging in" newly constructed modules embodying different contextual information. As we mentioned earlier, this approach has to an extent been explored in the CAMP and Stepanov and Musser work.

Our schema divides the modules in an LGA into several classes, similar to the abstraction classes defined by Stepanov and Musser. These classes are contextual abstractions, abstract algorithms, auxiliary abstractions, base abstractions, and view abstractions.

Contextual abstractions encapsulate various contextual design decisions such as data representation, storage management, concurrency control, device handling, persistence, and communication. They typically are characterized by hard coded design decisions with "invariant context" provided by with statements, but they can be services provided by an entirely different "lower-level" architecture, such as a database management system or network architecture.

Abstract algorithms incorporate the algorithmic content of the architecture. They are parameterized by a collection of theories, each describing a class of abstract data types. They typically fall into two categories:

- *Horizontal extensions*: such algorithms provide additional functionality to theories in the same way that operations are added to extend object classes in an object oriented environment.

- *Vertical extensions*: such algorithms combine theories to form structures that can be used together with an abstraction function to build base and auxiliary abstractions. They are vertical extensions in the sense that they provide the glue to perform vertical composition, i.e., the glue to map a representation onto an abstraction.

Base abstractions are the canonical abstractions of the domain (the concepts of the domain), while *View abstractions* are derived from the base abstractions by mapping one language onto another. The terms *view* and *base* were again taken from the dbms domain, as they capture a similar mapping notion. Code in a View "implementation" is typically concerned with renaming procedures and types, redistributing procedure functionality, and tailoring a concept to a particular protocol of use. Such code has little "algorithmic content" in the sense that algorithms have simple complexity and little local resource utilization.

Auxiliary abstractions are structures that are convenient to build in order to provide concepts to algorithm theories not provided by existing contextual abstractions. Both base and auxiliary abstractions are constructed by composing a representation from auxiliary and contextual abstractions, defining an abstraction function, and then choosing a collection of generic algorithms that realize the abstraction function.

Note that a generic architecture embodies two types of content: algorithmic and structural. The development of algorithmic content is a *programming in the small* issue. It is concerned with the implementation of a single algorithm or family of algorithms, organized according to the structural and behavioral properties of the algorithms. Structural content is a *programming in the large* issue. It refers to the "scaffolding" or framework of the architecture, and is implemented by a particular modular decomposition. Generic architectures incorporate the structural content of a domain in the structure of the architecture, and isolate the algorithmic content of the domain in abstract algorithms.

5 Importation vs. Parameterization

Parameterization provides us with a way to build normalized components in a manner completely independent of any contextual dependencies. We only need be concerned with defining the theories that form the underlying representation (along with the abstract machine of the language) on which the component content is specified. The binding of actual concepts to these theories is then done at a later, system build, time. We have noticed though, that part of the process of limiting the domain of a generic architecture includes the early binding of context by using Ada *with* statements rather than generic parameters. This seems to be a similar issue to that of establishing the boundary between context and content. That is, it is a design decision driven by domain scoping concerns rather than by representation concerns.

6 Conclusions

We have attempted both to fit prior efforts into our framework and to develop a number of examples on our own in order to verify the schema structure. In doing so we have found, as others have before us, that Ada is not the ideal design language for describing such generic architectures. We are therefore exploring language formalisms in this regard, including efforts such as Goguen's module interconnect language work [Gogo 83], Tracz's LILEANNA work [Trac 90], and modular extensions to ML [Harp 86].

References

[Bass 87] Bassett, P.G. [1987] Frame-Based Software Development. *IEEE Software*.

[Bato 91] Batory, D., and O'Malley, S., "Genvoca: Reuse in Layered Domains", *Proceedings of the First International Workshop on Software Reusability*, Dortmund, Germany, July, 1991.

[Booc 87] Booch, G. [1987] *Software Components with Ada*. Benjamin/Cummings Publishing Co.

[CAMP 85] CAMP [1985]. *Common Ada Missile Packages*, McDonnell Douglas Astronautics Company, St. Louis, MO, 3 Volumes.

[Gogo 83] Goguen, J.A. [1983]. LIL - A Library Interconnect Language. *Report on Program Libraries Workshop*, SRI International.

[Harp 86] Harper, R., Milner, R., and Tofte, M. [1988] The definition of Standard ML, Version 2. ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, University of Edinburgh.

[Muss 88] Musser, D.R., and Stepanov, A.A. [1988]. Generic Programming, Proceedings of ISSAC-88 and AAEC-6, Rome, Italy.

[SMTR 90] *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, CASE Center, Syracuse University, June, 1990.

[Trac 89] Tracz, W.J., and Edwards, S. [1989]. Implementation Working Group Report. *Reuse In Practice Workshop*, Software Engineering Institute, Pitt, Pa.

[Trac 90] Tracz, W.J. [1990] *Formal Specification of Parameterized Programs in LILEANNA*. Ph.D. Dissertation, Stanford University (to appear, 1992).

[Uhl 90] Uhl, J., and Schmid, H.A. [1990] *A Systematic Catalogue of Reusable Abstract Data Types*, Lecture Notes in Computer Science (#460), Springer-Verlag.

7 About the Authors

Larry Latour received his Ph.D. degree in Computer Science from Stevens Institue of Technology in 1985. He is an Associate Professor of Computer Science at the University of Maine, Orono, Me. His research interests include database transaction systems, software engineering environments, and software reuse. He developed a hypertext-based system, SEER, for describing the many views of a software component (usage, specification, and implementation), and he is currently interested in implementation architectures and their relation to the output of domain analyses.

Curtis Meadow received his M.S. degree in Computer Science from the University of Maine in 1990, and is currently a lecturer in the Computer Science Department at Maine. His research interests include object oriented software development, programming languages and compilers, and software reuse. His thesis examined aspects of layered generic architectures discussed in this position paper.