6th Annual Workshop on Software Reuse(Owego, New York), November 1993

P++: A Language for Large-Scale Reusable Software Components[*]

Vivek Singhal and Don Batory

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
Tel: (512) 471-9711/9713
Email: {singhal,batory}@cs.utexas.edu

Abstract

P++ is a programming language that supports the GenVoca model [BO92], a particular style of software design that is intended for building software system generators. P++ is an enhanced version of C++: it offers linguistic extensions for component encapsulation, abstraction, parameterization, and inheritance, where a component is a subsystem, i.e., a suite of interrelated classes and functions.

Keywords: open architectures, program families, large-scale reuse, software system synthesis, GenVoca, software system generators.

Workshop Goals: feedback on our research; exposure to other work in software reuse.

Working Groups: reuse process models; domain analysis/engineering; design guidelines for reuse; reuse and object-oriented methods; tools and environments.

[*]Appeared in *Proceedings of the Sixth Annual Workshop on Software Reuse*, Owego, New York, November 1993.

1 Background

Almost two decades ago, Parnas observed that software design was incorrectly taught as a technique which sought a unique program/solution, because, over its lifetime, the program inevitably would evolve into a family of similar programs [Par76]. When programs are not designed for extensibility, the effort and expense needed to modify them is often out of proportion to the changes themselves. A different approach, one that achieves economies of scale, was needed. He argued that since program families are inevitable, designing program families from the beginning is the most cost-effective way to proceed.

Recent work on domain-specific software architectures [CS93, Sof90] has shown that software system generators offer a promising means of economically building families of large, complex software systems. These generators are domain-specific; they implement models (called *domain models*) which show how to construct a family of similar software systems by composing reusable, prefabricated components. The essential themes that Parnas espoused are present in contemporary software generators; generators formalize the design of software families as open architectures, where software system synthesis and evolution can be quick and inexpensive.

Some examples of generators include Genesis (database systems) [Bat88], Avoca (network protocols) [OP92], Ficus (file systems) [HP93], Brale (host-at-sea buoy systems) [Wei90], and Predator (data structures) [BSST93]. Although each of these generators was developed independently and targeted for a different problem domain, all were organized in basically the same way. The GenVoca model captures their common design strategy [BO92]: it defines a particular style of designing and organizing reusable components that enables families of software systems to be defined through component composition. An implementation of a GenVoca model is a software system generator for a particular domain.

Some of the important features of GenVoca are: (1) *Components* are the building blocks of hierarchical software systems (a component is a subsystem or module). (2) Components must import and export *standardized* interfaces. (3) Component composition and customization should be achieved through *parameterization*.

It is well-known that language support for a design paradigm can tremendously simplify the application of that paradigm; object-oriented languages, for example, have been instrumental in popularizing and realizing object-oriented designs. For the same reason, language support for GenVoca is important. In the next section, we briefly describe the P++ programming language, which codifies the GenVoca model. P++ offers several linguistic extensions to C++ to support component encapsulation, abstraction, parameterization, and inheritance. From our experiences with the Genesis and Predator projects, we believe that P++ would have considerably simplified the implementations of these generators.

2 The P++ Language

2.1 Encapsulation and Abstraction

Encapsulation and abstraction are two program design techniques often used to manage the complexity of large software systems. *Encapulation* is the technique of consolidating related code and data fragments into atomic units of program construction. The GenVoca model has demonstrated

```
realm collection<class T>
{
  class container
  {
    container ();               // container
  };
  class cursor
  {
    cursor (container *);       // constructor
    void first ();              // container traversal
    void next ();               // container traversal
    int eoc ();                 // beyond end of container?
    void insert (T);            // add item
    void remove ();             // delete item
    T& get_value ();            // get item
  };
};

component linked_list : collection<class T>
{
  class element
  {
    element (T);                // constructor
    T data;                     // the value of this node
    element *next, *prev;       // adjacent nodes on list
  };
  class container
  {
    element *head;              // first node of list
  };
  class cursor
  {
    element *current_pos;       // current position
    container *cont;            // container iterated upon
  };
};
```

Figure 1: Sample realm and component declarations.

that the scales of encapsulation currently provided by programming languages, namely functions and classes, are inappropriate for building large systems. Instead, GenVoca advocates the use of components, which correspond to subsystems or modules. In P++, a component consists of a suite of interrelated data members, functions, and classes.

Abstraction is a design technique which separates the interface of a component from its implementation. In P++, the realm construct specifies the interface of a component: it declares the functions and classes which comprise a component's interface, without revealing the implementation of those functions and classes.

Figure 1 lists an example from the domain of data structures. First shown is the declaration of the `collection` realm: this is a standardized abstract interface for interacting with data structures that store collections of objects. `collection` declares two classes, `container` (which actually stores the objects) and cursor (which provides methods for manipulating objects inside a `container`). Notice that the realm reveals no implementation details about its classes, thus maintaining the proper separation between a component's abstract interface and concrete implementation.

The next declaration in Figure 1 is that of `linked_list`, a component from the `collection` realm. Because `linked_list` belongs to `collection`, it is obligated to implement (export) the methods listed in `collection`'s declaration. The body of `linked_list`'s declaration introduces a new class called `element`, along with new data members for the `container` and `cursor` classes. This class

```
component linked_list <collection<element> next_layer> : collection<class T>
{
  class element { ... };
  class container { ... };
  class cursor { ... };
};

component binary_tree <collection<element> next_layer> : collection<class T>
{
  class element { ... };
  class container { ... };
  class cursor { ... };
};

component array : collection<class T>
{
  class element { ... };
  class container { ... };
  class cursor { ... };
};
```

Figure 2: A parameterized component declaration.

and these data members are part of `linked_list`'s private implementation; they are not visible to other components or programs.

A fundamental feature of component encapsulation is evident in this example: a component consists of several data members, functions, and/or classes that are intimately intertwined. Because of their interrelated algorithms, the `element`, `cursor`, and `container` classes cannot be designed, implemented, or reused in isolation. P++ provides the programming language support to maintain their cohesion as an atomic unit of system construction.

2.2 Parameterization

The use of components almost always entails some customization. Direct source code modification may be appropriate for functions or classes because of their (typically) small code volume; however, this customization technique is rarely effective for components. Component parameterization, which permits an easy and controlled form of modification, is widely believed to be the prescription for successful component customization [PDF93].

Contemporary programming languages already offer constant and type parameterization of classes; P++ extends these capabilities to components and realms. In addition, P++ uses the model of parameterization to perform component composition: a component `C` may be parameterized by a realm `R`, which means that `C` may be combined with any component that belongs to realm `R`.

The `collection` declaration of Figure 1 shows an example of type parameters. `collection` is defined in terms of the type parameter `T`; this parameter determines the kind of objects stored by the data structure. Furthermore, because the component `linked_list` is defined to be a member of the `collection` realm, `linked_list` is also implicitly parameterized by the type `T` (notice that the definition of `element` depends on the value of `T`).

Figure 2 shows three components of the `collection` realm, namely `array`, `binary_tree`, and `linked_list`. Note that the declaration of `linked_list` has been revised so that it is now parameterized in terms of the `collection` realm. This means that `linked_list` imports the interface defined by `collection`. Any component belonging to this realm would be a legal value for this parameter (e.g. `array`, `binary_tree`, or even `linked_list`). Although not obvious, parameterizing `collection` components by realms forms the basis for generating a vast family of data structures [BSST93].

P++'s realm parameterization facility is a powerful language feature for designing and reusing software components:

- It encourages the development of components with standardized abstract interfaces. Such components are more likely to be reused because they can be easily interchanged with other components of the same realm [BSST93].
- When interchangeable components are available, it is easy to tune the performance of a system: different components can be quickly substituted for one another, thus greatly facilitating the process of finding improved implementations for a system [BSST93].
- P++ integrates component definition and combination features in a single language. Other researchers have used module interconnection languages to combine components [PDN86]. However, such languages typically are different from the language in which components are written.

2.3 Inheritance

Programming languages use inheritance to implement two kinds of hierarchies: implementation hierarchies and type hierarchies [Lis87]. Current object-oriented languages usually support *implementation hierarchies*, where a subclass inherits *both* the interface and the implementation of the superclass, unless explicitly overridden (overloaded) by the subclass. In contrast, when a language implements *type hierarchies*, inheritance is being used to support data abstraction. In this context, a subtype inherits only the interface of its supertype, but not its implementation.

To support type hierarchies, P++ allows new realm declarations to inherit from existing realm declarations. For example, suppose realms `R` and `S` were defined as follows:

```
realm R { ... };
realm S : realm R { ... };
```

These declarations indicate that `S` inherits the interface of `R`; that is, the interface of `S` includes not only its own functions and classes, but also those of `R`. Therefore, `S`'s interface is a superset of `R`'s, which means that all components belonging to realm `S` also belong to realm `R`. Realm inheritance is important because it provides a structured technique for evolving component interfaces. Since realm inheritance is an instance of type hierarchies, this form of inheritance ensures interchangeability, a property which enhances the likelihood of reuse.

3 Conclusions

There is growing recognition that existing software system design techniques are inadequate; they are aimed at producing one-of-a-kind systems that are difficult to modify and evolve. Software system evolution is a critical requirement of future systems, to satisfy the ever-increasing demands of new applications. Concomitantly, the need for software generators that can manufacture families of related systems by composing reusable components is becoming progressively more important, to meet the challenge of economical software system evolution.

Research in domain analysis and program families has identified the system organization concepts needed by software system generators. The scale of these programming concepts (i.e. components) transcend those found in conventional programming languages (functions, classes). The P++ programming language extends the encapsulation, abstraction, parameterization, and inheritance capabilities of C++ to components; we believe P++ is a prototype of future languages that support the construction of families of systems and software system generators.

References

[Bat88] Don Batory. Concepts for a DBMS synthesizer. In *Proceedings of ACM Principles of Database Systems Conference*, 1988.

[BO92] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[BSST93] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings of ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*, Los Angeles, California, December 1993.

[CS93] L. Coglianese and R. Szymanski. DSSA-ADAGE: An environment for architecture-based avionics development. In *Proceedings of AGARD*, 1993.

[HP93] John Heidemann and Gerald Popek. File system development with stackable layers. Technical Report CSD-930019, Department of Computer Science, University of California, Los Angeles, July 1993.

[Lis87] Barbara Liskov. Data abstraction and hierarchy. In *Addendum to the OOPSLA '87 Conference Proceedings*, pages 17–34, October 1987.

[OP92] Sean O'Malley and Larry Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[Par76] David Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[PDF93] Rubén Prieto-Díaz and William Frakes, editors. *Advances in Software Reuse: Second International Workshop on Software Reuse*. IEEE Computer Society Press, 1993.

[PDN86] Rubén Prieto-Díaz and James Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307–334, November 1986.

[Sof90] Software Engineering Institute. *Proceedings of the Workshop on Domain-Specific Software Architectures*, Hidden-Valley, Pennsylvania, 1990.

[Wei90] David M. Weiss. Synthesis operational scenarios. Technical Report 90038-N, Software Productivity Consortium, Herndon, Virginia, August 1990.

4 Biography

Vivek Singhal is a doctoral candidate in the Department of Computer Sciences at the The University of Texas, Austin. He received his S.B. from the Massachusetts Institute of Technology in 1990. His research interests include reuse systems, domain modeling, and object-oriented database management systems.

Don Batory is an Associate Professor in the Department of Computer Sciences at The University of Texas, Austin. He received his Ph.D. from the University of Toronto in 1980, he was Associate Editor of the IEEE Database Engineering Newsletter from 1981-84 and was Associate Editor of ACM Transactions on Database Systems from 1986-1991. He is currently a member of the ACM Software Systems Award Committee, and his research interests are in extensible and object-oriented database management systems and large-scale reuse.