Technical Report
TR-93-16, Department of Computer Sciences, University of Texas at Austin,
November 1993

P++: A Language for Software System Generators†

Vivek Singhal and Don Batory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{singhal,batory}@cs.utexas.edu

Abstract

P++ is a programming language that supports the GenVoca model, 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 suite of interrelated classes and functions. This paper describes the motivations for P++, the ideas which underlie its design, the syntax and features of the language, and related areas of research.

Keywords: program families, large-scale encapsulation, software system synthesis, GenVoca, software system generators.

1 Introduction

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 was the most cost-effective way to proceed.

Recent work on domain-specific software architectures [Cog93, SEI90] 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 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.

Genesis is an example of a generator for the domain of database systems [Bat88]. Genesis consists of a library of components and a generator which are used to specify and synthesize university-quality database management systems (consisting of up to 70,000 lines of code). Defining a database system as a composition of components and generating that system can be done within a half-hour; an equivalent system built from scratch would take months or years of effort. Generators for other domains include Avoca (network protocols) [Hut91, OMa92], Ficus (file systems) [Hei93], Brale (host-at-sea buoy systems) [Wei90], and Predator (data structures) [Bat93].

† This research was supported in part by Applied Research Laboratories at the University of Texas, Schlumberger, and Digital Equipment Corporation.

Although each of the previously mentioned systems 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 [Bat92a]. GenVoca defines a particular style of designing and organizing 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:

- Subsystems are the building blocks of hierarchical software systems.

  Effective software system synthesis requires that large systems be constructed from combinations of subsystems, not just functions or classes. Function encapsulation is *small-scale*; class encapsulation (i.e., the encapsulation of a suite of interrelated functions and variables) is *medium-scale*. Small-scale and medium-scale encapsulations are much too fine-grained for large software system specification and synthesis; software system building blocks require *large-scale* encapsulation, the encapsulation of suites of interrelated classes and functions. We will call a such a building block a subsystem or *component* in this paper.

- Components must import and export standardized interfaces.

  The key to software system synthesis is composition; software generators must be able to compose components easily. When standardized interfaces correspond to the fundamental abstractions of the target domain, composition is significantly simplified. Different components with the same interface are plug-compatible and interchangeable; such components represent alternate implementations of a common abstraction. This enables different versions of systems (i.e., different systems within the same family) to be created easily.

- Component composition and customization should be achieved through parameterization.

  A parameterized component is a compact representation of a family of related components (each member of the family corresponds to a different parametric instantiation). If interfaces are standardized, then each unique interface can be treated like a "type signature" — which can be used to quickly verify the compatibility of component compositions. We can therefore model the imported interfaces of a component as types, so that component composition is really just an example of type 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 this paper, we present 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.

We first present and illustrate the primary features of P++. Each feature described in Section 2 corresponds to or elaborates a basic construct of GenVoca. Section 3 describes some issues on code generation that P++ must address. Readers will note that P++ is a domain-independent language; software system generators are domain-specific. We explain in Section 4 how P++ can be used to support software generators and the implementation trade-offs that are part of the implementation of P++. Finally, Section 5 compares P++ to other programming languages.

2 P++ Features

P++ extends the encapsulation, abstraction, parameterization, and inheritance mechanisms of C++ to include linguistic support for components. Each extension is described in the following sections along with a discussion of its novelty and significance.

It is important to note that components do not necessarily imply that their size (in terms of lines of code) is large. Rather, components correspond to conceptually bigger units of software design (i.e., encapsulation beyond individual functions or individual classes). Consequently, the size of a component can range from tens of lines to many thousands of lines of code. The simplest examples of components that we have encountered occurs in the domain of data structures. The example that we will use to illustrate the linguistic constructs of P++ is taken from the Predator project, a generator for the domain of data structures. Readers will see that a linked list data structure is an elementary, but clear, example of a component and large-scale encapsulation.

2.1 Encapsulation and Abstraction

Component encapsulation and abstraction are fundamental to P++. These linguistic extensions of P++ are syntactically similar to, but semantically different from, nested classes in C++. We first explain the ideas of component encapsulation and abstraction, and then distinguish them from nested classes.

Encapsulation and abstraction are two program design techniques, often used to manage the complexity of large software systems. *Encapsulation* is the technique of consolidating related code and data fragments into atomic units of program construction; *abstraction* separates the interface of a module from its implementation. Current programming languages only support encapsulation and abstraction at the scale of functions and classes; P++ extends this support to components.

2.1.1 Encapsulation

A component consists of data members, functions, and/or classes. This is analogous to a C++ class, which consists of methods and/or data members. As shown in Figure 1, the P++ component declaration and C++ class declaration have similar syntax. This example implements a linked list of integers; the component **`int_linked_list`** consists of the **`element`**, **`container`**, and **`cursor`** classes. A **`container`** object stores a collection of **`element`** objects on a linked list, where each **`element`** includes an integer data item. **`cursor`** objects are used to reference and manipulate the **`element`** objects in a **`container`**.

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

2.1.2 Abstraction

A *realm* specifies the interface of a component: it declares the functions and classes which comprise a component's interface. A realm interface does not reveal the implementation of its functions and classes; that is the job of the component. Every P++ component definition must indicate the name of a realm "to which it belongs". In addition to those functions and classes declared by its realm, a component may also declare private functions, classes, and variables which are used internally by the component's implementation. These private members do not affect the external interface of the component; they are simply implementation details and are hidden by the realm's interface.

Figure 2 depicts sample realm and component declarations: the example of Figure 1 has been rewritten to use the realm construct. The realm, **`int_collection`**, specifies a generic interface for the set of software components that store collections of integers. The realm enumerates the methods of two classes, **`container`** and **`cursor`**, but does not reveal the actual algorithms that implement these methods. The component, **`int_linked_list`**, defines implementation-specific classes and data members. As shown in

```
component int_linked_list
{
  class element
  {
    element (int);            // constructor
    int data;                 // the value of this node
    element *next, *prev;     // adjacent nodes on list
  };

  class container
  {
    container ();             // constructor
    element *head;            // first node of list
  };

  class cursor
  {
    cursor (container *);     // constructor
    void first ();            // container traversal
    void next ();             // container traversal
    int eoc ();               // beyond end of container?
    void insert (int);        // add item
    void remove ();           // delete item
    int& get_value ();        // get item
    element *current_pos;     // current position
    container *cont;          // container iterated upon
  };
};
```

Figure 1: A component declaration.

the figure, the component adds the **head** data member to the **container** class, and the **current_pos** and **cont** members to the **cursor** class; moreover, it declares a new class, **element**, that is private to the **int_linked_list**'s implementation.

2.1.3 Beyond C++

As mentioned earlier, there is a similarity between C++ nested classes and P++ components. In terms of expressiveness, both constructs are roughly equivalent. However, the basic difference arises from their intended usage. Nested classes do not arise in popular object-oriented design methodologies (e.g., [Boo91, Rum91, Teo90]); their intended use in C++ is to facilitate the implementation of the enclosing class. In contrast, a basic tenet of GenVoca is to identify groups of interrelated classes and encapsulate them within component constructs. In short, P++ components encapsulate subsystems (i.e., multiple classes) into atomic units of program construction, whereas C++ nested classes are used for data hiding (i.e. abstraction) within a single class.

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, customization via direct source code modification 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 [Pri93].

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

component int_linked_list : int_collection
{
  class element
  {
    element (int);          // constructor
    int 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 2: Declarations of a realm and a component.

Contemporary programming languages already offer constant and type parameterization of classes [Ell90, DOD83]; P++ extends these parameterization capabilities to components and realms. In addition, P++ components can also be parameterized by realms. Realm parameters are a basic feature of the GenVoca model and a concept borrowed from module interconnection languages [Pri86, Gog86]. In the following sections, we describe these P++ parameterization constructs and their novelty and significance.

2.2.1 Constant and Type Parameters

Generic components and generic realms require constant and type parameters. Consider the example of Figure 2. Both the **int_collection** realm and the **int_linked_list** component operate on integer objects; however, there is nothing intrinsic to their algorithms or interfaces which requires that they manipulate only integers. This realm and this component would be far more useful if the programmer could specify the data type of the objects stored in the **container**. Type parameters remove this limitation. Figure 3 shows how to generalize **int_collection** by parameterizing the type of objects stored in its **container**. The **linked_list** component uses this parameter's value to define the **element** class.

While a component inherits the parameters of its realm, individual components can also have additional parameters, usually in the form of tuning constants. Consider the example of **array**, a **collection** com-

```
realm collection<class T>
{
  class container
  {
    container ();            // constructor
  };
  class cursor
  {
    cursor (container *);    // constructor
    void first ();           // container traversal
    void next ();            // container traversal
    int eoc ();              // past 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 3: Parameterized realm and component declarations.

```
component array <int size> : collection<class T>
{
  class container
  {
    T items[size];            // array of T objects
    int free_slot;            // index of next free slot
  };
  class cursor
  {
    int index;                // index of current object
    container *cont;          // container iterated upon
  };
};
```

Figure 4: Declaration of the **array** component.

ponent which stores its elements in a statically allocated array (see Figure 4). An obvious parameter for this component would be the number of elements to allocate in the array. As shown in the figure, the definition of the **container** class depends on the value of the **size** parameter (an integer constant).

Constant and type parameters are a powerful means for generalizing components and realms. But, as we will see in the next subsection, they are not sufficient; realm parameters are also needed to define generic components.

2.2.2 Realm Parameters

Large software systems are often organized as layered subsystems. Each layer implements a particular set of abstractions; the layer "above" builds upon the abstractions of the layer "below" to implement even more powerful abstractions. The benefit of a layered organization is to manage complexity by successively reducing the original problem into simpler problems.

In GenVoca, layers are components. Each component implements an abstract interface (a realm) and exploits the abstractions provided by some lower realm(s). GenVoca models this arrangement by parameterizing components in terms of realms. When one component is stacked on top of another, then the upper component is parameterized by the realm interface of the lower component.

The technique of parameterizing components by realms is fundamental to GenVoca; moreover, it yields one of the most powerful features in P++. A realm parameter can be associated with any component that implements that realm's interface. For example, suppose components **r1**, **r2**, and **r3** belong to realm **R**, and **s1** and **s2** belong to realm **S**. **r1** and **r2** also happen to be parameterized in terms of realm **S**. These relationships would be expressed in P++ as:

```
component r1<S s> : realm R { ... };
component r2<S s> : realm R { ... };
component r3 : realm R { ... };
component s1 : realm S { ... };
component s2 : realm S { ... };
```

An application program could combine the **r1** and **s1** components to yield a software subsystem (**sys**) that exports the interface **R**:

```
typedef r1<s1> system11;            // combine r1 and s1
system11       sys;                 // create an instance of system11
```

This example demonstrates the ease of specifying system implementations; in this case, **r1** is layered on top of **s1**. However, because **r1** is actually defined in terms of realm **S**, **s2** is also a legal parameter value. In fact, each of the following is a legal subsystem implementation that exports the interface **R**:

```
typedef r1<s2> system12;;            // r1 is layered on top of s2
typedef r2<s1> system21;;            // r2 is layered on top of s1
typedef r2<s2> system22;;            // r2 is layered on top of s2
typedef r3 system33;;                // a component with no realm parameter
```

A more complicated example that uses both realm and type parameters arises in data structure generation [Sir93]. Data structure components can be understood as transformations, where the **linked_list** component of Figure 3 transforms a collection of **T** objects into a collection of **element** objects that are connected by a linked list (see Figure 5a). This interpretation of components is interesting because it provides a uniform framework for modelling a wide variety of data structure features (e.g., binary trees, persistence, compression, etc.). For example, Figure 5b shows how the same collection of **T** objects is transformed by the **array** component of Figure 4 into an array of **T** objects.

When **collection** components behave like transformations, it is natural to combine them to produce more sophisticated data structures. Figure 6 shows a revised declaration of **linked_list**; this declaration includes a realm parameter (**next_layer**) which specifies how the **element**s of the linked list

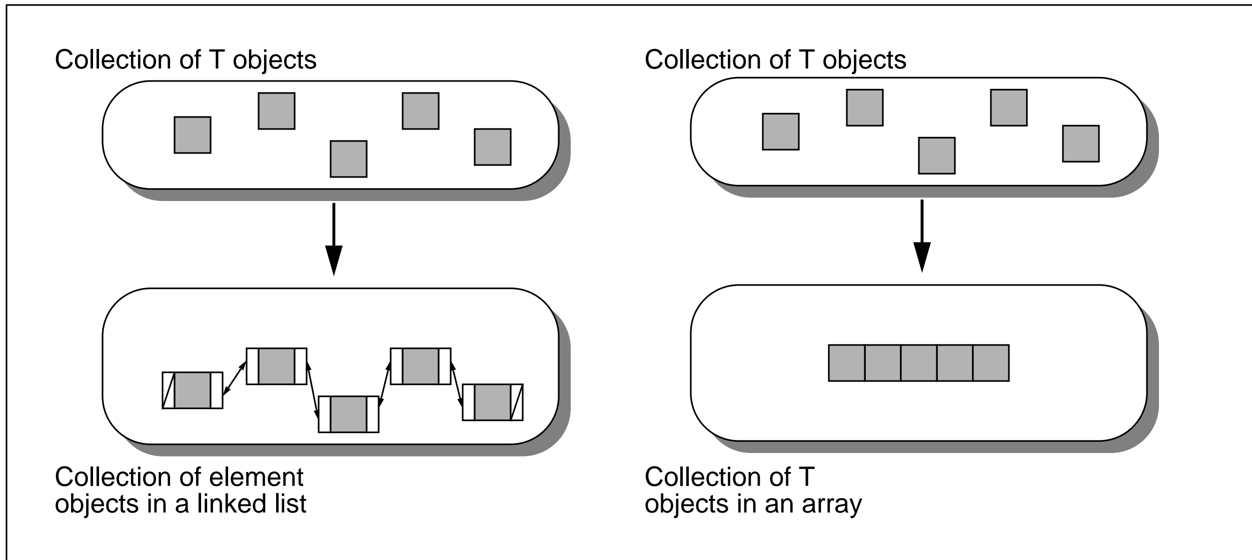

Figure 5a and 5b: The **`linked_list`** and **`array`** transformations.

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

Figure 6: Realm parameter version of **`linked_list`**.

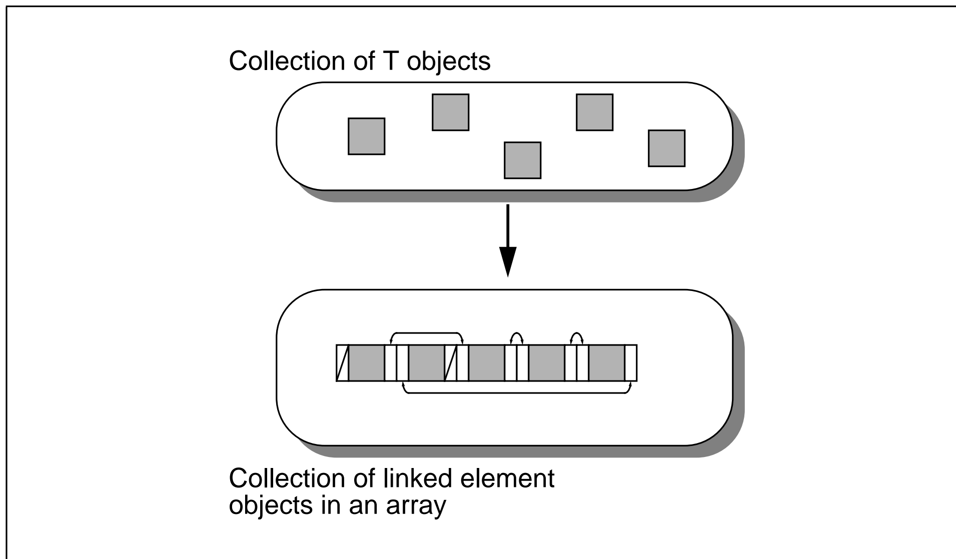

Figure 7: The transformation for **`linked_list<T><array<5>>`**.

should be stored. Since **`array`** belongs to the **`collection`** realm, it is a legal argument for this parameter.

The definition:

```
typedef linked_list<T><array<5>>     list_of_T_in_array;
```

implements a linked list of **T** objects whose members are stored in a five-element array, as shown in Figure 7.

As demonstrated by these examples, very complex hierarchical systems can be expressed in a compact specification. The P++ realm parameterization facility is a powerful language feature for designing and customizing components:

- It encourages the development of components with standardized abstract interfaces. Such components are more likely to be used because they can be easily interchanged with other components of the same realm.
- 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.
- P++ integrates component definition and component combination features in a single language. Other researchers have used module interconnection languages (MILs) to combine components [Pri86, Gog86]. However, such languages typically are different from the language in which components are written.

2.2.3 Beyond C++

There are some similarities between P++ parameterized components and C++ templates — both allow constant and type parameters. The principle difference is that P++ also allows realm parameters. Recall that a realm declares the external interface of one or more components; this means that a realm parameter imposes certain restrictions on what components can satisfy that parameter. Furthermore, this restriction on the parameter value means that the compiler can statically validate a component composition just by verifying the compatibility of each realm parameter and its value. In contrast, there is no provision for C++ templates to impose restrictions on its type parameters. For example, a C++ template that implements a sorting algorithm might require that its type parameter define comparison and equality operators (to be used during sorting). However, there is no C++ syntax for expressing this restriction. This makes it more difficult to determine the validity of template instantiations, and makes it inconvenient to specify complex requirements on parameter values.

There is a more significant difference between P++ and C++. Even though C++ provides limited template facilities, (i.e., constant and type parameters), these features are insufficient for defining customizable families of components. Consider the examples given in Section 2.2.2, which employ both realm parameters and component parameters. In particular, the **linked_list** component (of Figure 6) is not only parameterized by the type of elements on the list, but also by the **collection** realm. That is, **linked_list** belongs to a family of realms; the particular family member is determined by specifying a value for the **T** parameter. Even after **T**'s value is determined, **linked_list** offers yet another dimension of customization: the **next_layer** parameter specifies the component "below" **linked_list**. The C++ template facility is unable to mimic P++'s parameterized realms and components, because C++ templates cannot be parameterized in terms of other templates. C++ does not support "templates of templates", which makes C++ somewhat less flexible than P++.

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 its superclass, unless explicitly overridden (overloaded) by the subclass. In this context, the primary purpose of inheritance is to

support code reuse, at the cost of potentially violating encapsulation. A subclass could violate the data abstraction presented by its superclass in three ways: the subclass might directly access an instance variable of its superclass, call a private operation of its superclass, or directly refer to one of its superclass' superclasses [Sny86].

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. The primary advantage of type hierarchies over implementation hierarchies is interchangeability. Because a subtype's interface is "bigger" than (i.e. a superset of) its supertype's interface, an instance of a subtype can be substituted for instances of the supertype. Consequently, type hierarchies ensure that subtype and supertype implementations are independent, whereas implementation hierarchies cannot.

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 the classes and functions declared by **S**, but also the classes and functions declared by **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 to evolve component interfaces. Because realm inheritance is an instance of type hierarchies, this form of inheritance ensures interchangeability, a property which enhances the likelihood of using the component in a system.

3 P++ Implementation

The P++ compiler implements a superset of ANSI C++. It is actually structured as a P++ to C++ translator; that is, the output of the P++ compiler can be redirected to a C++ compiler to produce executable code. Many of P++'s language constructs do not have direct analogs in C++. Consequently, P++ must simplify these constructs so that they are expressed in terms of C++. The AT&T Cfront compiler (which translates C++ into C) faces a similar dilemma: certain C++ features (e.g. templates, virtual methods) are not directly available in C. In the case of templates, Cfront uses the technique of macro-expansion to resolve all unique instantiations of a parameterized type into separate non-parameterized types. P++ employs a similar approach to resolve parameterized components and realms.

Many of the prototype generators we cited earlier (e.g., Genesis, Avoca, Ficus) accomplish component composition through function calls or dispatch tables; that is, the boundary of every component in a generated system is explicit. The overhead for using composed components is small because the computations that occur inside components dwarf the cost of intercomponent communication. This is definitely not the case for Predator. Computations within data structure components are often a just a few assignment statements; function call overhead for explicit component interfaces would yield unacceptable performance. Predator circumvents this problem by macro-expansion — to eliminate unnecessary function calls and explicit interfaces — and specialization — to generate usage-specific optimized code. P++ must perform similar optimizations in order to achieve high-quality code generation. In a future paper, we will report experiments on P++'s performance and elaborate on the requirements imposed on its implementation. It is our conjecture that Predator's techniques of specialization and judicious use of macro-expansion will be the keys to efficient code generation.

4 Using P++

As mentioned earlier, P++ is domain-independent. Software system generators are domain-specific. We believe that P++ will be one member of a suite of tools — some of which are domain-specific — that comprise a software system generator; P++ provides the substrate for writing and combining components. The following examples illustrate this idea.

Predator is a generator for the data structures domain [Bat92c, Bat93]. P2 is a domain-specific compiler that implements a superset of ANSI C, providing linguistic extensions for components, component compositions, containers, and cursor constructs. Every data structure component in Predator may add rules or tokens to the P2 grammar (for example, the name of a component would be a token of a type expression). Automatically modifying and extending the ANSI C grammar is too difficult; for this reason, P2 is implemented as a pair of preprocessors: the first converts component compositions and container declarations to a form syntactically acceptable to ANSI C, and the second replaces the remaining P2 constructs with their C source equivalents. P2 also performs sophisticated analyses and query optimizations in order to generate efficient code.

Genesis offers another example of a domain-specific tool. Genesis has a layout editor (called DaTE) that provides a graphical interface for selecting and combining components [Bat92b]. The editor not only provides an easy-to-understand depiction of a target system's organization, but it also guides the programmer in building semantically correct component compositions. Since most database systems consist of dozens of components, this tool has substantially simplified database software system specification and has proven to make the task of system construction less error-prone.

In general, software generators that are integrated with domain-specific programming languages, like Predator, will be realized by a pipeline of preprocessors (see Figure 8). The first preprocessor will encapsulate domain-specific optimizations and analyses, and remove domain-specific language extensions. The output of this preprocessor would be P++ source. P++ would then produce C++ source by composing components.

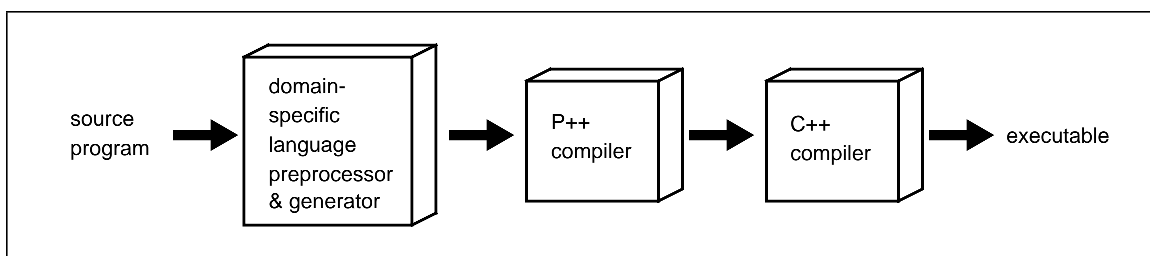

Figure 8: A pipeline of preprocessors for domain-specific generators.

5 Related Work

As mentioned earlier, the conceptual foundations for P++ lie in the notion of program families [Par76] and the construction of families by parameterized programming [Gog86]. The specific concepts offered by P++ arise from the actual experiences and requirements of software system generators (Genesis, Avoca, Ficus, etc.). Nevertheless, there are many programming languages that offer features similar to P++.

Languages like Ada 9X [Bar93] and Modula 3 [Car89] provide constructs for defining components. In addition, in most object-oriented languages (e.g., Eiffel [Mey88], Self [Cha91]) it is possible to emulate abstract component interfaces (realms) using virtual classes, albeit with some performance penalties. Realm inheritance can also be emulated using inheritance hierarchies of virtual classes. The use of virtual

classes introduces run-time overhead because each method invocation from an abstract class requires a virtual table lookup. In contrast, P++ performs compile-time simplification of component compositions, which eliminates this cost.

LIL [Gog86] and FOOPS [Gog93] are examples of languages that are based on Goguen's parameterized programming model. This model provides a formal foundation for understanding the two varieties of parameters: horizontal (e.g. type and constants parameters) and vertical (e.g. realm parameters). The functional programming language ML employs a very different approach to parameterization [Mil90]. ML automatically deduces type restrictions on parameters; consequently, the application of a function is limited only by the usage of its parameters.

P++ is distinguished from other languages by its explicit support for the GenVoca model. Although other languages support some of the features that are needed by GenVoca software system generators, to our knowledge only P++ has the entire requisite set.

6 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 is becoming progressively more important in meeting the challenge of economical software system evolution.

Research in domain analysis and program families has identified 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); a component is a suite of interrelated classes, functions, and data members. P++ is a programming language that extends the encapsulation, abstraction, parameterization, and inheritance capabilities of C++ to components and uses compiler optimizations like inlining and specialization to ensure high-quality code generation.

We believe P++ is a prototype of future languages that support the construction of families of systems and software system generators.

Acknowledgments. The authors would like to thank Marty Sirkin and Jeff Thomas for their detailed comments and thought-provoking criticisms, which helped the ideas of P++ to mature.

7 References

[Bar93] John Barnes. Introducing Ada 9X. Technical Report, Intermetrics Inc., February 1993.

[Bat88] Don Batory. Concepts for a DBMS synthesizer. In *Proceedings of ACM Principles of Database Systems Conference*, 1988. Also in Rubén Prieto-Díaz and Guillermo Arango, editors, *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 1991.

[Bat92a] 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.

[Bat92b] Don Batory and James Barnett, DaTE: the Genesis DBMS software layout editor. In Pericles Loucopoulos and Roberto Zicari, editors*, Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development.* Wiley, 1992.

[Bat92c] Don Batory, Vivek Singhal, and Marty Sirkin. Implementing a domain model for data structures. *International Journal of Software Engineering and Knowledge Engineering*, 2(3):375-402, September 1992.

[Bat93] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings of the ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*, December 1993.

[Boo91] Grady Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, 1991.

[Car89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, Systems Research Center, Digital Equipment Corporation, November 1989.

[Cha91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *OOPSLA '91 Conference Proceedings*, October 1991.

[Cog93] L. Coglianese and R. Szymanski. DSSA-ADAGE: an environment for architecture-based avionics development. In *Proceedings of AGARD 1993*. Also in Technical Report ADAGE-IBM-93-04, IBM Owego, May 1993.

[DOD83] Department of Defense. *Reference Manual for the Ada Programming Language*. Ada Joint Program Office, Department of Defense, ANSI/MIL-STD-1815A, 1983.

[Ell90] Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Gog86] Joseph Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16-28, February 1986. Also in Rubén Prieto-Díaz and Guillermo Arango, editors, *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 1991.

[Gog93] Joseph Goguen and Adolfo Socorro. Module composition and system design for the object paradigm. Technical Report, Programming Research Group, Oxford University Computing Laboratory, August 1993.

[Hei93] John Heidemann and Gerald Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, to appear. Also in Technical Report CSD-930019, Department of Computer Science, University of California, Los Angeles, July 1993.

[Hut91] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1), January 1991.

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

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*, Prentice-Hall, 1988.

[Mil90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*, MIT Press, 1990.

[OMa92] 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.

[Pri86] Rubén Prieto-Díaz and James M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307-334, November 1986. Also in Peter Freeman, editor. *Software Reusability*. IEEE Computer Society Press, 1987.

[Pri93] Rubén Prieto-Díaz and William Frakes, editors. *Advances in Software Reuse: Second International Workshop on Software Reuse*, IEEE Computer Society Press, 1993.

[Rum91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*, Prentice-Hall, 1991

[SEI90] Software Engineering Institute. *Proceedings of the Workshop on Domain-Specific Software Architectures*, Hidden-Valley, Pennsylvania, 1990.

[Sir93] Marty Sirkin, Don Batory, and Vivek Singhal. Software components in a data structure precompiler. In *Proceedings of the 15th International Conference on Software Engineering*, May 1993.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 38-45, November 1986.

[Teo90] T. J. Teorey. *Database Modeling and Design: The Entity-Relationship Approach.* Morgan-Kaufmann, 1990.

[Wei90] David M. Weiss. Synthesis Operational Scenarios. Technical Report 90038-N, Version 1.00.01, Software Productivity Consortium, Herndon, Virginia, August 1990.