A Programming Language for Writing Domain-Specific Software System Generators

by

Vivek P. Singhal, S.B.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 1996

A Programming Language for Writing Domain-Specific Software System Generators

Approved by
Dissertation Committee:

Acknowledgments

This dissertation was possible only with the guidance, wisdom, encouragement, and patience shown to me by Don Batory. He acted not only as advisor and mentor, but also as a good friend. His hospitality and generosity will always be remembered.

Thanks to the members of the Predator research group, for their insightful criticisms and engaging debate: Dinesh Das, Marty Sirkin, and Jeff Thomas. I also thank Sheetal Kakkad and Mark Johnstone, who always offered me assistance whenever I needed it.

I greatly appreciate the steady stream of support and encouragement my family gave to me over the years.

And finally, I could not have completed this work without the love and devotion shown to me by Aparna, my future wife. Her unfailing patience gave me the energy to finish what I started.

A Programming Language for Writing Domain-Specific Software System Generators

Publication No. ______________

Vivek P. Singhal, Ph.D.
The University of Texas at Austin, 1996

Supervisor: Don S. Batory

Automating routine programming tasks is an effective way to increase the productivity of software development. Software system generators have the potential to achieve this goal: customized software systems can be quickly and easily assembled from component libraries. Our research demonstrates that for generators to be successful, component libraries must be scalable. Scalability enables libraries to be small, because the components of the library implement distinct and largely orthogonal features. These components can be combined to yield an enormous family of software systems and subsystems. Generators thus become tools for combining components to manufacture these systems and subsystems.

In GenVoca, the programming model that forms the foundation of our research, components act as large-scale refinements which simultaneously transform multiple classes from one abstraction to another. Because GenVoca advocates a novel style of program organization, there is little language or tool support for this paradigm. We have developed a programming language called P++, which extends C++ with specialized constructs to support the GenVoca model. It permits components to be treated as transformations which can simultaneously refine several classes in a consistent manner. To validate the utility of this language, we solved a “challenge problem” in software reuse: we reimplemented the Booch C++ Components data structures library as a scalable P++ library. We were able to reduce the volume of code and number of components by approximately a factor of four, without compromising the performance of generated systems.

Table of Contents

Chapter 1

Introduction

Large software systems are becoming increasingly more complex, expensive, and time-consuming to build. Researchers at the 1968 NATO Software Engineering Conference coined the term "software crisis" to describe this situation [Nau68]. Twenty-eight years later, software practitioners still face the same problems. Although advances in object-oriented design, high-level programming languages, and reuse libraries offer some relief, large software system development still poses a formidable challenge.

Of the many software engineering and reuse strategies that we have studied, one approach seems very promising. Software system generators, a kind of software development tool, are domain-specific systems which can rapidly construct entire system implementations out of prefabricated components. The goal of such systems is to "industrialize" the software industry — i.e. to introduce ideas like modularity, interchangeability, and standardization to software development so that high-performance software may be mass-produced [McI68].

We believe that software system generators will emerge as popular tools for large system development, because they offer a significant advance over existing tools. There are already numerous examples of generators in a wide range of domains, such as databases [Bat88], file systems [Hei90, Hei91], network protocols [Hut91, OMa92], and data structures [Sir93]. Our study reveals that even though the systems produced by these generators are radically different in functionality, the organizations of the generators themselves are remarkably similar. The goal of our research is to understand existing software system generators and employ this knowledge to simplify the design, implementation, and use of future generators.

Early research uncovered basic ideas which underlie the design of software system generators. These ideas have been captured in a programming model called GenVoca [Bat92a] (whose name was inspired by the *Gen*esis generator for databases and A*voca* generator for network protocols). GenVoca offers a simple model and clear notation for representing the systems produced by generators.

The primary barrier to the widespread use of GenVoca generators is the difficulty of their development. The problem is two-fold: (1) creating a building-blocks model of a domain (called *domain analysis* [Pri91]) requires considerable domain expertise, and (2) implementing this model (called a *generator*) involves developing a sophisticated infrastructure for defining and using software components. Our research strives to make the *second* task easier: we have created a programming language which offers specialized capabilities to support the GenVoca model. This language, called P++, offers linguistic features for declaring, defining, and combining software components — the building blocks of large software systems.

The following chapters of this dissertation explore this important software engineering problem, explain our proposed solution, and analyze our experimental results.

In Chapter 2 we describe the difficulty of building large systems and the problems with current software libraries. We explain why software system generators are promising, yet require an enormous effort to construct manually.

We describe the P++ programming language in Chapter 3. It offers features for specifying abstract component interfaces, defining component implementations, composing components to form systems, and modelling components as program transformations. We review these linguistic features and demonstrate how they might be used to support GenVoca programming.

In Chapter 4, we motivate and explain our implementation of the P++ compiler. This chapter includes detailed examples which illustrate the compilation process.

To test the P++ compiler, we applied the principles of the GenVoca model to design a library of P++ components for the data structures domain. Chapter 5 describes this library and compares it to an existing C++ data structures library. Our experiments demonstrate that P++ provides significant advantages over conventional object-oriented languages, when used to build a GenVoca software system generator.

We put the results of P++ in context with other areas of software engineering research in Chapter 6. We describe the relationship between P++ and frameworks, parameterized programming, software architectures, and transformation systems.

Finally, in Chapter 7 we summarize the results of the P++ project and list areas of future research.

Chapter 2

Background

Large software system development is difficult and time-consuming. Software libraries appear to offer a promising approach to solving this problem: rather than constructing large systems from scratch, it is far easier to piece together large systems from existing software modules [Gar95]. Unfortunately, current software libraries are often inadequate for this task. For example, if each module of a software library implements only a simple functional unit, then it will take hundreds or thousands of library modules to construct a large system (assuming that the requisite modules are actually available). Conversely, if modules offer sophisticated functionality, then those modules will be applicable to only a few systems; many systems will be unable to exploit the benefits of that library. Thus, researchers face a dilemma where the productivity gained by module reuse increases with the size of the module, but the specificity of the module (and the likelihood of reuse) decreases with the size of the module. This problem appears to be an inherent barrier to high productivity via module reuse [Big94].

Software systems generators have a solution to this problem. A generator is a tool which can rapidly generate many different implementations of systems for a particular problem domain. A model of software construction, called GenVoca, captures the basic design properties of these generators [Bat92a]. In this chapter, we motivate and describe GenVoca, list some of the difficulties in building Gen-

Voca-style generators, and explain why programming languages should directly support GenVoca ideas.

2.1 Building large software systems

Software libraries are one of the most widespread and successful techniques for speeding software development [Big89]. A library is a collection of code modules that can be used without modification to build new applications. The goal of libraries is to promote software reuse, which lowers the cost of software development by leveraging on the work of prewritten code.

Function and class libraries are the most common kinds of software libraries. Consisting of collections of subroutines or object-oriented classes, these libraries provide features which are useful for a wide range of applications. Usually, each library module is relatively simple, implementing a primitive algorithm or data structure; it is uncommon to find library modules which correspond to complex program subsystems. Examples of function and class libraries include the standard C library [Ker90], X window system [Sch90], NIHCL [Gor90], GNU libg++ [Lea88], the Booch Components [Boo87, Boo90], and the C++ Standard Template Library [Pla95].

Experience has shown that large software systems do not benefit much from function or class libraries. Because the goal of such libraries is to provide only a general purpose set of basic abstractions, the programmer is responsible for selecting, customizing, and combining these simple abstractions to form new software modules that implement sophisticated domain-specific abstractions. Two problems are evident with this approach. First, the task of creating large-scale abstractions by manually implementing them in terms of low-level small-scale abstractions is complex and time-consuming. Second, the implementation of a single large system may ultimately consist of hundreds or thousands of individual functions and classes — correctly combining all of those software modules is difficult and error-prone.

A more effective approach is to assemble large systems out of larger units of program construction, namely program subsystems (also called object-oriented *frameworks* [Cam92]). The purpose of a subsystem is to define a high-level programming abstraction which hides the implementation details of a single feature (or algorithm) of an application. For example, in the domain of database systems, a feature can be a particular data language, a join algorithm, a file structure, a method of concurrency control or recovery, etc. In the domain of data structures, a feature could be a specific data structure, a method for encrypting stored elements, a method for garbage collection, or a method for the persistent storage of data.

Depending on the domain, a subsystem's implementation might consist of just one function or class. Typically, however, a subsystem consists of several classes that cooperate to implement a domain-specific feature. A subsystem is not a stand-alone program; rather, it is designed to be combined with several other subsystems to form an application program.

Building a large system from prewritten subsystems is much easier than using function or class libraries, simply because fewer subsystems need to be selected and combined. A library of subsystems corresponds to a collection of sophisticated software modules that implement large-scale abstractions. The implication, however, is that such a library is useful in fewer contexts, because specialized modules are relevant to fewer applications. Consequently, a library of subsystems is usually targeted for applications in a particular problem domain; a programmer developing an application in that domain can exploit the library to rapidly construct major portions of his target system [Big89].

2.2 Scalable software libraries

Why have existing domain-specific software libraries not already solved the problem of large system development? Our research has revealed that a successful domain-specific library must be *scalable*. That is, it must be possible to add new features to an existing library without expending progressively more effort: introducing a new feature should not require duplicating or modifying the modules already in the library. We found that most contemporary libraries are not scalable, which means that it will be progressively more difficult for such libraries to represent the systems found in their respective problem domains.

We uncovered the library scalability problem during our study of libg++ [Lea88] and the Booch C++ Components [Boo87], two popular data structure libraries. These libraries contain numerous modules which implement different data structures; often, the only difference between two modules is the algorithm used to implement a single data structure feature. Therefore, even though these libraries offer many data structures, each data structure really just represents a unique permutation of several features.

This design style is problematic because it makes library enhancements prohibitively expensive to implement [Bat93]. If one wants to introduce a new data structure feature to a library which already contains n modules, the resulting library would contain $2n$ modules: n modules which incorporate the new feature and n modules which do not. For example, most data structure libraries are populated with different modules that represent different structures in transient memory. To add the feature of persistence (i.e., so that data elements reside in persistent storage) would effectively double the size of the library. For each transient memory data structure module, there would be a virtually identical counterpart for persistent memory. To generalize, a library of up to 2^k modules would be needed to implement all combinations of *k* features (provided, of course, that all combinations of features are meaningful). The problem becomes more acute when several implementations of each feature are available, because the corresponding library would be much larger. Implementing and maintaining such a library would be utterly impractical.

Figure 2.1 depicts the relationship between the number of modules and the number of features in scalable and unscalable libraries. As the number of features increases in an unscalable library, the requisite number of modules grows exponentially. In scalable libraries, however, the number of modules grows only linearly as more features are added.

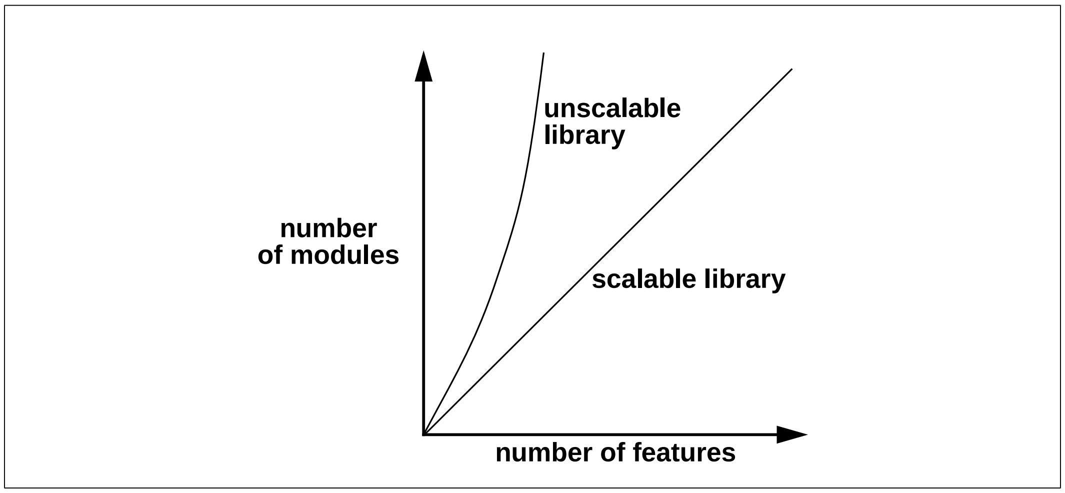


Figure 2.1 Growth comparison of scalable and unscalable libraries.

In hindsight, the solution to this dilemma seems obvious: each module in a library should encapsulate only one feature. Furthermore, a software system with multiple features should be constructed from the set of modules that collectively implement those features. In this way, we make explicit the feature combinatorics of software systems. The size of the library grows linearly as new features are added, but the number of combinations of those modules (to form systems) grows exponentially.

Unfortunately, it is quite difficult to design a library such that its modules are independent of one another. A domain expert must carefully devise module abstractions such that distinct features can be represented as independent software modules. Such a design greatly increases the value of a domain-specific library of software modules, because the library can be subsequently extended with new features without excessive effort.

2.3 Software system generators

An emerging class of software system generators (SSGs) are successfully using scalable subsystem libraries to construct large systems quickly. An SSG is a domain-specific tool which consists of a library of components and a generator for combining those components.[1] An SSG can construct many related systems in a particular problem domain by combining different components from its library. Each component implements a basic feature offered by some systems in the domain. Selecting different component combinations corresponds to building systems with different sets of features.

We have studied several SSGs, including Genesis (domain of databases) [Bat88], Avoca (network protocols) [Hut91, Oma92], Ficus (file systems) [Hei90, Hei91], Brale (host-at-sea buoy systems) [Wei90], Adage (avionics) [Cog93], and Predator (data structures) [Sir93]. We observed that several key design properties underlie their organizations:

1. For the purposes of this discussion, the term *component* is defined to be a suite of interrelated functions and data types which work together to implement some feature of a large software system. Although "component" is a term which is commonly used in object-oriented programming literature, we will use this definition throughout.

- **Software systems are built from combinations of components.** In these SSGs, a generator produces system implementations by combining components from a software library. Little or no hand-coding is necessary, making the task of assembling a system quick and easy. Goguen's research on LIL (library interconnection language) shows that the key to effective software system construction is to use the techniques of parameterized programming [Gog86]. When a parameter is used to "factor out" a common design element of a component (like a constant value or a primitive data type), this scenario is called *horizontal parameterization*. Typically, a horizontal parameter makes only simple adjustments to the behavior of a component. *Vertical parameterization* corresponds to layering progressively more sophisticated programming abstractions in order to implement certain functionality. A vertical parameter is used by a component to relegate part of its algorithm's implementation to a separate component (intuitively, vertical parameterization is used to connect the components in a system). Parameterized programming languages like LIL (and SSGs) simultaneously use horizontal and vertical parameterization to promote the construction of systems from component compositions.
- **Every component from an SSG library has an interface which reflects the fundamental abstractions of the problem domain.** The interface of a component specifies the externally accessible programming entry points which are used to interact with the component. This interface must be carefully designed if the component is to be reusable (i.e. composable, interchangeable, etc.). One must examine many systems from the problem domain, recognize the common architectural traits of those systems, and define standard interfaces which correspond to the basic programming

abstractions. Components designed in this manner will probably be reusable and sufficiently versatile to represent most features found in systems from that problem domain.

- **Several components often have identical (standardized) programming interfaces**, even though each component implements a different algorithm or behavior. This situation arises when several components represent alternate implementations of a system feature. By giving these components the same interface, each component can be interchanged with another without requiring modification of the program code that uses the component. The conclusion is that components must be designed for use in software system generators; dissecting existing software systems to populate a component library is unlikely to be successful.

Although SSGs substantially simplify the task of building large software systems, SSGs themselves are quite difficult to design and build. Considerable expertise and discipline is necessary to craft a scalable library of components and implement a generator for combining those components.

For example, a scalable component library has a small number of components which can be combined in various ways to produce a large number of system implementations. There is little code replication in such a library, because each component represents only one basic feature of systems in the problem domain. Thus, a library of k components can be used to generate 2^k different systems; contrast this with the unscalable data structure libraries we described earlier, which exhaustively implemented all 2^k data structure modules. The components of a scalable library are relatively independent of one another, which means that there are few hidden interdependencies between the library's components. (Two components are "independent" if the presence of one component in a system does not prevent or require the inclusion of the other component in the system.) Without a well-established methodology for designing and building libraries which meet these criteria, writing libraries of components is more difficult than it should be.

Another obstacle to implementing an SSG is devising a mechanism for combining components. It is necessary to create not only a notation for specifying component interfaces and component combinations, but also a tool which generates system implementations out of component descriptions. Experience has shown this task to be non-trivial: fully three-fourths of the programming effort of the Predator data structures SSG involved the development of a tool for combining components.

In summary, SSGs offer a promising approach to rapidly building large software systems. Using libraries of domain-specific components, an SSG can assemble significant portions of software systems, with little or no hand-coding required. The challenge, however, is that component libraries are difficult to design and generators are tedious to implement. Additional help is needed.

2.4 The GenVoca programming model

GenVoca is a model of software construction using components [Bat92a]. It identifies the similarities in component design and organization that are present in all of those SSGs, even though they are targeted to different problem domains. The hope is that understanding the design of existing SSGs will simplify the implementation of future SSGs.

In GenVoca, software components are the building blocks of large software systems. A component is a basic unit of program construction. It supplies an external interface that defines how other components can be connected to it. It can also offer adjustment parameters which are used to customize the behavior of a component. The details of a component's implementation are private — no external program can directly access these hidden details, thus constraining component interaction to its public interface (see Figure 2.2).

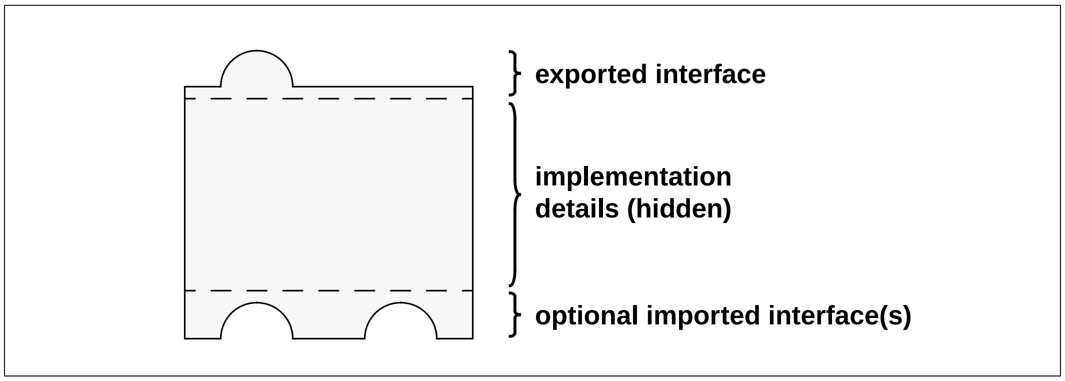

exported interface
implementation details (hidden)
optional imported interface(s)

Figure 2.2 Anatomy of a component.

The term *realm* refers to a set of components that all possess a specific interface. Each component belongs to one realm, and that realm interface specifies the external “appearance” of the component. Often, several components belong to the same realm, which means that all of those components export the same programming interface. Because their interfaces are identical, at a first approximation all components in a realm are interchangeable (see Figure 2.3).

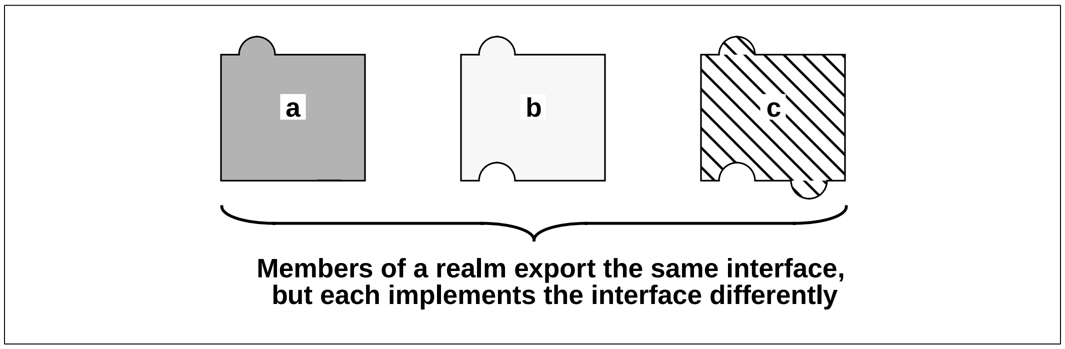

a
b
c
Members of a realm export the same interface, but each implements the interface differently

Figure 2.3 Members of a realm are interchangeable.

In GenVoca, a component represents a forward refinement program transformation. This transformation consists of a set of simultaneous data and function refinements, the purpose of which is to convert an abstract interface into a concrete implementation. The application of these refinements is controlled by certain domain-specific rules, which dictate under what circumstances the refinements are legal, whether optimizations of the refinements are possible, etc.

An abstract interface of a component corresponds to a suite of function and data type declarations; the role of a component is to map each function declaration to a specific algorithm and each data type declaration to a specific representation. If several components combine to form a system, that system can also be modeled as a transformation — a composite transformation corresponding to a sequence of individual component transformations.

To better explain this unusual interpretation of components as transformations, GenVoca offers a simple notation for representing realms, components, and systems. A realm is denoted by a set of elements, where each element represents a component belonging to the realm. The example below lists the components from three realms, *R*, *S*, and *T*:

$$\begin{aligned} R &= \{a, b[R], c[S]\} \\ S &= \{d[T], e, f[S, T]\} \\ T &= \{g\} \end{aligned}$$

Realm *R* has three components (*a*, *b*, *c*), realm *S* has three (*d*, *e*, *f*), and realm *T* has one (*g*). If a component imports another component’s interface, it is designated as a parameter; i.e., the name of the imported realm interface is listed in brackets beside the name of the component. Thus, *b* imports realm *R* and *c* imports realm *S*. Component *f* of realm *S* imports two realm interfaces (*S* and *R*), because it has two parameters. In essence, this notation treats a realm as a type signature. A component from a realm simply exports a type signature, and a component which imports an interface has a parameter corresponding to some type signature. So in the above example, *d* is an object of type *S*, where *d* has a parameter of type *T*.[2]

When component is modelled as a transformation, that means the component simultaneously refines a set of classes and functions (which it imports). This transformation preserves the semantic consistency of the classes and functions: that is, if the classes and functions imported by a component are semantically valid, then the application of a component transformation will yield a (potentially) different set of classes and functions that are also semantically valid.

Using this notation, a software system can be compactly specified as an equation (i.e. a combination of components). For example, here is a system *sys* built from *c*, *d*, and *g*:

$$sys = c[d[g]]$$

Notice that component compatibility is easily checked by verifying that each parameter's type matches the corresponding component's type. Therefore, $c[d[g]]$ is a syntactically valid system because *c*'s parameter type matches *d*'s type, and *d*'s parameter type matches *g*'s type.[3]

Consider the meaning of $c[d[g]]$ when components are viewed as transformations. The GenVoca notation appears to suggest that components are combined much like mathematical functions. This, however, is not the case: GenVoca components are relatively sophisticated, which makes a transformational model more appropriate for understanding component combinations. When two components (like *c* and *d*) are interconnected, they each exchange function, data type, and customization information with one another. The semantics of this exchange are more

2. Formal parameter names have been omitted because these examples are so simple.
3. Note that some combinations of components are syntactically legal but not semantically correct. That is, each pair of components in the system imports and exports compatible interfaces, but the resulting algorithms are invalid for some reason. To verify the semantic correctness of a system, each component must supply domain-specific information that describes the assumptions and restrictions on the use of the component. Details of this process are given in [Bat96].

complicated than functional composition; Figure 2.4 illustrates the exchanges that occur in the system $c[d[g]]$.

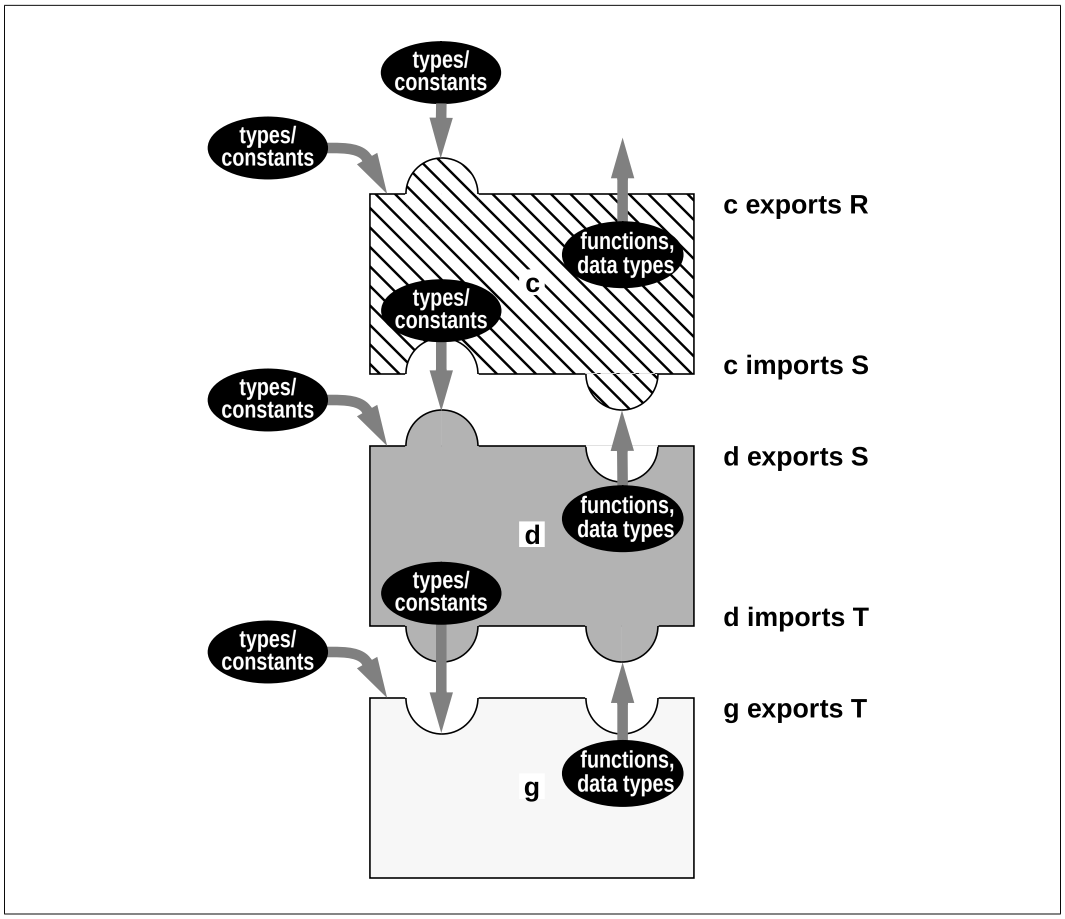


Figure 2.4 The component combination $c[d[g]]$.

A functional interpretation of *sys* would have an innermost-to-outermost evaluation semantics; i.e., component *g* would be evaluated first, then *d*, then *c*. This is not a correct interpretation in GenVoca, where evaluation is outermost to innermost. Thus, the transformations of *sys* start at the top component, *c*, which provides data type information to *d*. In turn, *d* provides its own data types to *g*, which then supplies implemented data types and functions back to *d*, and so on.

Notice that the transformations start at the top component, work their way down to the bottom component, and then back up to the top component. This style of transformation (which we will fully explain in Chapter 3) is certainly different from standard function compositions; consequently, we believe that this transformation mechanism distinguishes GenVoca from traditional programming models.

2.5 The domain of data structures

Throughout the remaining chapters, concepts and results will be explained using examples from the domain of data structures. We will focus on data structures which implement containers of objects — examples include bags, sets, lists, and queues. Why choose this domain? Data structures is a simple, easy-to-understand problem domain. The basic ideas and algorithms are already familiar to most programmers, making data structures an ideal framework for testing and explaining our results. This domain consists of many enumerable data structure “systems”. Moreover, we found that data structures poses the same challenges as the domains of large software systems. The specifications of realms, components, and component compositions for data structures, database systems, communications software, distributed file systems, and avionics software are the same — only the complexity of the *algorithms* differs.

Three classes participate in this problem domain. A *container* is a collection of *elements*, and all of a container’s elements are of a single type. Elements enclosed by a container can be referenced and modified only through runtime objects called *cursors* (also called iterators) [Mus96]. Cursors and contains are well-established concepts in databases; earlier work on Genesis and contemporary work on object-oriented databases strongly influenced our choice of these abstractions. Figure 2.5 shows a container with four elements and a cursor which references one of them.

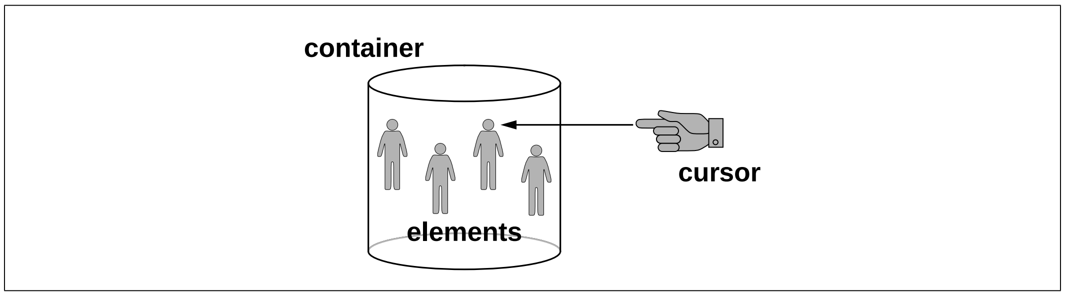


Figure 2.5 Data structure abstractions.

2.6 Language support for GenVoca

A programming language is often a useful vehicle for supporting a programming paradigm. For example, modular programming and object-oriented programming have benefited substantially from language support [Boo87, Boo90]. Although language support offers no guarantee that programs will be well-designed, it does make the application of a programming paradigm much easier.

Because current generators are hand-coded from scratch, each generator's implementation and component combination mechanism is unique. We believe that these problems can be attributed to inadequate language support for GenVoca. It is our contention that such support would make SSGs much easier to build. Specifically, it would be helpful if programming languages provided syntactic constructs for specifying component interfaces and implementations. Without such support, it is the programmer's burden to properly organize his algorithms and data structures into realms and components. In the past, programming languages have served as a catalyst for increasing the scale of programming; for example, procedural languages promoted the use of functional encapsulation, and object-oriented languages made class abstractions easier to use. Therefore, it is logical to use programming languages to promote the next step in encapsulation — software components.

Component combination is another basic trait of GenVoca systems. Components represent abstract to concrete transformations which may be combined in series. As shown in Figure 2.4, a combination of two components involves the mutual exchange of function and data type information. In Chapter 3, we explain why this two-way exchange is more sophisticated than the one-way exchange of information that occurs in traditional function or type parameterization. Consequently, current programming languages are hard-pressed to handle the parameterization constructs needed by the GenVoca model.

As we will see in the next chapter, we modified a programming language's syntax for parameterized types so that it could be used to both customize the behavior of components and combine components. Although in hindsight it appears natural to represent component combinations as instantiations of a parameterized type, this result was not at all obvious. Rather, using parameterized types to uniformly represent Goguen's notions of horizontal and vertical parameterization was actually a key insight. We extended the GenVoca model to incorporate this idea, so that the model could explain not only how to fit components together, but also how to tailor their behaviors.

In summary, SSGs would greatly benefit from the spread of GenVoca ideas into programming languages. Language support for component interfaces, component implementations, and component combinations would reduce the burden of developing GenVoca generators. If SSGs are to become a practical means of building large software systems, this issue must be addressed.

Chapter 3

The P++ language

We have designed a programming language called P++, which offers features that specifically support the GenVoca model. P++ extends the C++ language by providing syntactic constructs for defining and combining realms and components. Each feature extends the capabilities of C++ by increasing the scale of abstraction, encapsulation, or parameterization. Although P++ may appear to provide only incremental enhancements over C++, P++ actually offers powerful features which substantially strengthen the C++ language, making it a suitable framework for building domain-specific software system generators.

We demonstrate how to use these features by giving examples from the domain of data structures. These running examples come from a P++ data structures library which we fully describe in Chapter 5. In this chapter, our focus is to explain the usage and significance of each language feature.

3.1 Motivation

P++ is a superset of C++ [Ell90, Str94]; it supports all existing C++ constructs, as well as certain constructs for defining and using components. Why define P++ in terms of C++? C++ is the most widely-used object-oriented language; its features already are well-defined and understood by many programmers. Moreover, we believe it is important to present GenVoca ideas using a syntax that is familiar to system developers. The choice of a mainstream language ensures that our ideas will be more accessible to them; in contrast, creating a new experimental or prototype language would make system developers reluctant to use such a language. Finally, as we will show in Chapter 4, basing P++ on an existing language happens to greatly simplify the implementation of the P++ compiler.

P++ offers services that would otherwise have to be developed separately for each generator. These features represent a substantial benefit because they drastically reduce the effort required to build a generator, thus permitting a system developer to concentrate on testing and tuning generated systems.

For example, P++ defines a notation for component combinations, which lets a programmer select, customize, and combine a group of components. The P++ compiler can verify the syntactic validity of this combination by checking the compatibility of each pair of imported and exported interfaces. P++ then converts the specification into an actual implementation by combining the appropriate component definitions. In contrast, a hand-coded generator must provide its own syntax for combining components, its own tool for verifying component compatibility, and its own compiler for generating system implementations from component definitions.

The P++ language directly supports GenVoca concepts in order to facilitate the translation of a domain model into a software system generator. Starting with a domain model which defines a suite of realms and components, a system developer can use P++ to write code to represent them. The developer avoids the cumbersome task of emulating realms and components using C++ classes or some other language construct. Instead, he uses special P++ realm and component constructs to directly implement the domain model.

Of course, not all domain models can be easily implemented using P++. If the domain model is not organized along the lines advocated by GenVoca, then P++ will probably offer little benefit. Moreover, even though P++ is useful for writing GenVoca-style componentry, there are certain tasks which the language does not perform. As we will see later in this chapter, not all syntactically legal component combinations are semantically correct; consequently, a software system generator should provide a separate mechanism for analyzing the validity of a system by consulting domain-specific component combination rules. P++ serves as a basic domain-independent development tool for generators, which should be combined with other domain-specific tools to yield a complete software system generator.

3.2 Abstraction

Complexity is the primary obstacle to developing large software systems [Boo91]. To help manage complexity, programming languages offer features to support the design techniques of abstraction and encapsulation.

Abstraction is the technique of hiding implementation details of a module and only revealing its external behavior. Separating behavior from implementation reduces the effort required to understand and use a program fragment. Existing language support for abstraction takes the form of function declarations (function prototypes) and access specifiers for class methods (e.g. *public*, *private*, *protected*). The GenVoca model demonstrates the need for the next step, namely realms.

A realm declaration specifies the programming interface of a component. It lists the functions, enumerated values, classes, and class methods that constitute the entry points of a component. This information describes how a program interacts with a component, without revealing any of the component's implementation details. (Thus, variables and class data members are not part of the realm declaration, because they reveal the memory layout of objects.) Figure 3.1 shows a portion of the P++ grammar which describes the syntax of realm declarations.

```
realm_declaration : opt_tmpl_parameter_header REALM realm_name
                    '{' realm_element_list '}' ';'

opt_tmpl_parameter_header : TEMPLATE '<' tmpl_parameter_list
                            '>'
                          | /* nothing */

tmpl_parameter_list : tmpl_parameter_list ',' tmpl_parameter
                    | tmpl_parameter

realm_element_list : realm_element_list realm_element
                   | realm_element

realm_element : abstract_class_declaration
              | abstract_function_declaration
```

Figure 3.1 The grammar for P++ realm declarations.

The realm construct plays an important role in the design of component libraries. It not only supports a larger scale of abstraction, but it also encourages the use of *virtual machines* and *standardized programming interfaces*.

Virtual machines. The software systems produced by a GenVoca generator typically have a layered organization. An implementation of this kind of system consists of a hierarchy of subsystems that are layered upon one another. Associated with each layer is a pair of virtual machine interfaces, which specify the abstractions that are imported and exported by that layer. By introducing the realm construct, P++ provides a linguistic mechanism for describing and naming these interfaces. Later in this chapter, we show how to associate particular realm names with component implementations.

Standardized programming interfaces. If the components in a GenVoca generator library all have unique interfaces, then that library will have little *generative power.* There would be only a few ways to combine those components, meaning that each component could potentially be used in only a few generated systems. The overall library would be capable of generating relatively few unique system implementations. One way to avoid this problem is to introduce standardized programming interfaces for functionally similar components. That is, if several components provide a different implementations of a basic system feature, then those components should belong to the same realm. These components could then be interchanged for one another in a system's implementation. The realm construct encourages standardization because several components can share a single interface declaration, which in turn increases the generative power of the library.

Figure 3.2 shows the declaration of a sample realm called *ds*. This realm declares two classes, *container* and *cursor*, and the methods which comprise the public interface of each class. The first line of the declaration, *template <class T>*, indicates that *ds* is *parameterized*. The significance of this statement will be explained later in Section 3.4. For now, observe that a realm contains no executable code — only interface specifications.

3.3 Encapsulation

Encapsulation is the technique of grouping related code and data into larger units of program construction [Mey88]. The goal is to increase the scale of programming by creating bigger program building blocks. Existing languages already support functions and classes; however, the GenVoca model scales encapsulation to subsystems (i.e. suites of interrelated components).

```
template <class T>
realm ds
{
  class container
  {
    container ();            // methods for the container class
    container (container &);
    int number_of_objects ();
    void clear ();
    T *member (const T&, int (*compare) (const T&, const T&));
    ...
  };

  class cursor
  {
    cursor (container *);    // methods for the cursor class

    void insert_before (const T&);
    void insert_after (const T&);

    void previous ();
    void next ();

    void goto_first ();
    void goto_last ();
    void goto_nth (unsigned int);

    T *value ();
    ...
  };

  ...
};
```

Figure 3.2 Declaration of the *ds* realm.

The component declaration is the counterpart to the realm declaration. Whereas a realm specifies only an interface, a component supplies an implementation for that interface. In object-oriented terms, a component inherits its interface from a realm, much like a C++ class can inherit an interface from an abstract class; the component overloads the realm's abstract functions and methods with its own concrete implementations for those functions and methods. To implement a realm's interface, a component may augment the declarations supplied by the realm: it may add methods and data members to the classes already declared, and it may introduce new functions, classes, and enumerated values.

Figure 3.3 shows the declaration of the *bounded* component. This component belongs to the *ds<T>* realm, which means that *bounded* is required to implement each method of *ds*. To accomplish this, *bounded* introduces a few private data members and methods to the abstract declarations of *container* and *cursor.* As a result, the complete definition of *container* has parts from several different declarations: the abstract methods declared by *ds* (e.g. *number_of_objects*, *clear*, …); data members defined by *bounded* (e.g. *objs*, *first*, *count*); new methods introduced by *bounded* (e.g. *copy_objs*); and method implementations provided by *bounded*.

As with C++ classes, the implementation of component methods can appear embedded within the component declaration, or separately after the component declaration. For example, *number_of_objects* is defined within the component declaration and *clear* is defined separately. Therefore, the full specification of a component consists of a component construct and all of its separately defined method implementations. The portion of the P++ grammar in Figure 3.4 describes the syntax of component declarations (the full P++ grammar is given in the Appendix).

3.4 Parameterization

Parameterization is a key feature of P++ — it is used to express component customization and combination. P++ customization parameters are declared using a syntax similar to that of C++ templates. A C++ template is really just a mechanism for generating classes. The template specifies only a skeleton for a class declara-

```
template <class T, int size>
component bounded : ds<T>
{
  class cursor;
  class container
  {
    friend class cursor;
    T objs[size];                 // private data members
    int first;
    int count;

    // inline definitions for methods declared by ds
    container () { first = 0; count = 0; }
    int number_of_objects () { return count; }

    // private method which is used internally by bounded
    static void copy_objs (T *dest, T *src, int n);
    ...
  };

  class cursor
  {
    container *const cont;        // private data members
    int index;

    // inline definitions for methods declared by ds
    cursor (container *c) : cont (c) { index = -1; }
    void next ()       { if (index >= 0) ++index; }
    void previous ()   { if (index < cont->count) --index; }
    void goto_first () { index = 0; }
    void goto_nth (unsigned int n) { index = n; }
    void goto_last ()  { index = cont->count - 1; }
    ...
  };
  ...
};

// separate definition of a method declared by ds
template <class T, int size>
void bounded<T,size>::container::clear ()
{
  count = 0;
  first = 0;
}
```

Figure 3.3 Declaration of the *bounded* component.

```
component_declaration : opt_tmpl_parameter_header COMPONENT
                        component_name ':' realm_instantiation
                        '{' component_element_list '}' ';'

realm_instantiation : realm_name opt_tmpl_value_list

opt_tmpl_value_list : '<' tmpl_value_list '>'
                    | /* nothing */

tmpl_value_list : tmpl_value_list ',' tmpl_value
                | tmpl_value

component_element_list : component_element_list
component_element
                       | component_element

component_element : forward_class_declaration
                  | concrete_class_declaration
                  | concrete_function_declaration
```

Figure 3.4 The grammar for P++ component declarations.

tion — to complete the declaration, a programmer must supply a concrete value for each of the template's parameters. This process of generating a customized class from a template is known as *instantiation*. Similarly, parameterized realms and components are just mechanisms for generating nonparameterized realms and components; the conversion is accomplished by instantiating the realm or component.

In P++, a realm or component can be customized by constant or type parameters. Thus, a component with an integer constant parameter could be instantiated with the values 2 or 256. Similarly, a realm with a type parameter could be instantiated with types such as integer or character string. For a concrete example, consider the *ds* realm of Figure 3.2, which has a type parameter called *T*. This parameter lets a programmer generate many different interfaces from one declaration, each corresponding to a different instantiation of *T*. In other words, parameterization provides a compact notation for expressing families of realms or families of components.

Figure 3.3 shows a slightly more complicated example of customization parameters. This case shows the declaration of a parameterized component (*bounded*) which happens to be defined in terms of a parameterized realm (*ds*). The *bounded* component has a type parameter *T* and an integer constant parameter *size*. Both parameters appear in the body of the code for *bounded*; for example, the data member *objs* is defined as an array of objects of type *T*, where the length of the array equals *size*. When bounded is instantiated, the values for these parameters will be inserted into the appropriate locations in the code, thus completing the component declaration.

The header of *bounded* specifies that the component belongs to the *ds<T>* realm (not *ds*). This notation means that when a programmer instantiates the *bounded* component, the *ds* realm is also instantiated with the parameter value *T*. Recall that *ds* really just acts as a realm declaration generator, which must be instantiated before it can be used. The result of instantiating *ds<T>* is a realm which is customized to match the interface that *bounded* exports.

3.4.1 Composition

Component combination is the other use for parameterization in P++. In the GenVoca model, two components may be combined if one component imports the interface of the other. P++ uses the syntax of C++ templates to represent imported interfaces as parameters. We found that this syntax yields an intuitive notation that makes component combination simple and easy to understand.

To denote that a component imports an interface, P++ uses a new kind of template parameter called a *realm parameter.* A realm parameter provides a component with a linguistic mechanism for accessing the interface defined by a realm.

The declaration of a realm parameter consists of a realm name (which indicates the interface being imported) and a symbolic identifier (which serves as a formal parameter name). When the parameterized component is instantiated, a component "value" will be supplied for the realm parameter. The component used as the parameter value must belong to the same realm as the parameter declaration indicates.

```
template <class T, ds<T> rep>
component size_cache : ds<T>
{
  class cursor;
  class container : public rep::container
  {
    friend class cursor;
    int obj_count;

    container () { obj_count = 0; }
    container (container &cn) { obj_count = cn.obj_count; }
    int number_of_objects () { return obj_count; }
  };

  class cursor : public rep::cursor
  {
    container *sc;

    cursor (container *c) : rep::cursor (c) { sc = c; }
    ...
  };
};
```

Figure 3.5 Declaration of the *size_cache* component.

The *size_cache* component of Figure 3.5 demonstrates how realm parameters work. This component has two parameters: a type parameter *T* and a realm parameter *rep*. Because *rep* belongs to the *ds<T>* realm, *rep* corresponds to an interface which declares the *container* and *cursor* classes. The code for *size_cache* uses *rep* to access these classes and their methods. For example, the *cursor* class inherits from *rep::cursor* — the *cursor* class declared by *rep*'s interface. This syntax for accessing a realm parameter's declarations (i.e. using '::') is the same syntax used for accessing nested classes in C++.

To construct a system implementation from a group of components, a programmer must define a *component composition* (also known as a *type equation* [Bat92]). A composition specifies a value for each constant, type, and realm parameter for every component in a system. As we will see, some of these values are explicitly specified by the programmer, and *others are implicitly derived from component definitions*.

Consider the following component composition:

```
typedef size_cache <float, bounded <100> > floatsys;
```

This composition defines a system implementation called *floatsys*, which consists of the components *size_cache* and *bounded*. The *size_cache* component takes two values, *float* (for its type parameter) and *bounded* (for its realm parameter). Because the number of parameters equals the number of values, this portion of the composition is legal. The second component, *bounded*, takes two values. However, the composition only provides one value, *100* (for its constant parameter). Where is the value for its type parameter?

As Figure 3.6 shows, the value for *bounded*'s type parameter comes from the definition of *size_cache*. To understand why, consider the relationship between components and parameterized realms. A parameterized realm must be instantiated before it is used. In particular, a component that imports a parameterized realm must instantiate that realm. By doing so, the component fully constrains the interface that it imports. Moreover, if another component is supplied as a value for that realm parameter, that component must also satisfy the same constraints. In other words, when one component imports a realm, an imported component must provide the same interface as the imported realm.

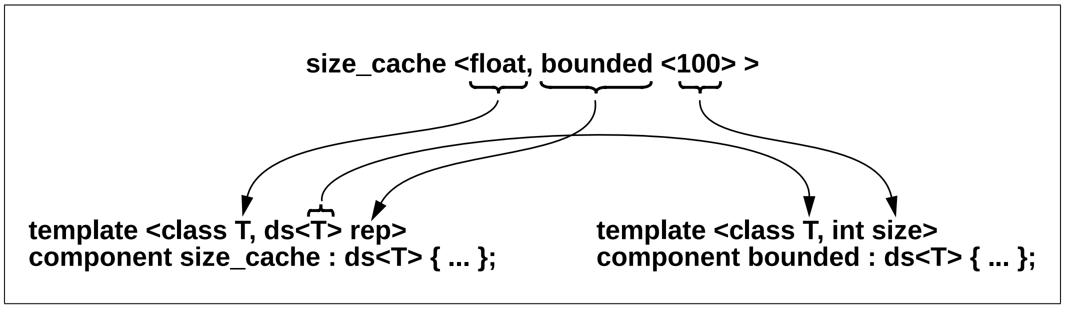

size_cache <float, bounded <100> >
template <class T, ds<T> rep>
component size_cache : ds<T> { ... };
template <class T, int size>
component bounded : ds<T> { ... };

Figure 3.6 Parameter propagation in a component composition.

To ensure that this invariant is maintained, P++ mandates a design convention for parameterized realms and components: if a parameterized component belongs to a parameterized realm, the component should have at least the same parameters as the realm. For example, the *bounded* and *size_cache* components both have a type parameter *T*, just like their realm, *ds*. If this design convention is observed, then the values that are used to instantiate the realm can also be used to instantiate the imported component.

To properly use implicitly specified and explicitly listed parameter values, there is a simple rule to follow. Explicit values must be supplied for each parameter of the outermost component of a system composition. For the remaining components, if a component is defined in terms of a parameterized realm, then the initial parameter values of the component will be supplied by the instantiation of its realm; the remaining values will explicitly come from the system composition.

There is fundamental reason for specifying some parameter values explicitly and others implicitly: a component with a realm parameter must be given the opportunity to dictate the interface it will import. This ensures that the component value supplied for that parameter will export a compatible interface. The parameters of a component fall into two categories: interface parameters — which define how a component's interface is customized or specialized, and implementation parameters — which define how a component's implementation is to be customized or specialized. Values for the initial set of parameters may be explicitly specified (if the component is outermost in the composition) or implicitly computed (in which case the values are obtained from the realm parameter specification). Values for the component-specific parameters must always be explicitly specified.

Therefore, to summarize the example of Figure 3.6, the *size_cache* component is instantiated with *float* for type parameter *T*; this constrains the *rep* parameter to the *ds<float>* interface. For *bounded* to satisfy that requirement, its type parameter *T* must also be instantiated with *float*. The automatic propagation of *float* ensures that *bounded* exports an interface that is compatible with *size_cache*.

3.4.2 Forward declaration

Until now, we have always considered examples where parameter values are passed unchanged to lower layer components. For example, the *size_cache* component of Figure 3.5 (reproduced below) demonstrates a technique which is frequently used in GenVoca definitions. For the declaration of *rep*, the *ds* realm is instantiated with the value *T*. Notice that *T* is actually a type parameter which is declared immediately before *rep*.

```
template <class T, ds<T> rep>
component size_cache : ds<T>
{
  ...
};
```

When a component has a parameter which is a parameterized realm, the instantiation values for the realm are based on the other parameters of the component. In this case, the value *T* used to instantiate *ds* is the same as the *T* parameter of the *size_cache* component. However, it is possible for a component to define a new type, based on a type parameter, which is used to instantiate a realm. The P++ construct which supports this capability is called a *forward declaration*.

A forward declaration allows a component to use a data type before the component has defined it. The initial reference is preceded by the keyword *forward*. The subsequent definition may use any of the type parameters which are declared by the enclosing component definition. Figure 3.7 shows an example of this new construct. The *unbounded* component has a type parameter *T* and a realm parameter *rep* (which belongs to a new realm called *memory*). This declaration instantiates *memory* with a data type called *object*. This instantiation requires a forward declaration because *object* is being used before it has been defined. The subsequent definition of *object* (inside the *unbounded* component definition) is also tagged with the *forward* keyword; this informs the compiler that *object*'s definition is available for use by *unbounded*'s parameters.

```
template <class T, memory<forward object> rep>
component unbounded : ds <T>
{
  ...
  forward class object
  {
    T data;
    object *next;
  };

  class container { ... };
  class cursor { ... };
};
```

Figure 3.7 Declaration of the *unbounded* component.

Why is the forward declaration construct useful? Figure 3.8 depicts the organization of the *size_cache* component. This component inputs a type parameter called *T*, and it outputs this same type as a parameter value for the next component in the composition. In contrast, the *unbounded* component inputs a type parameter *T*, but it outputs a new type called *object* as the parameter value for the

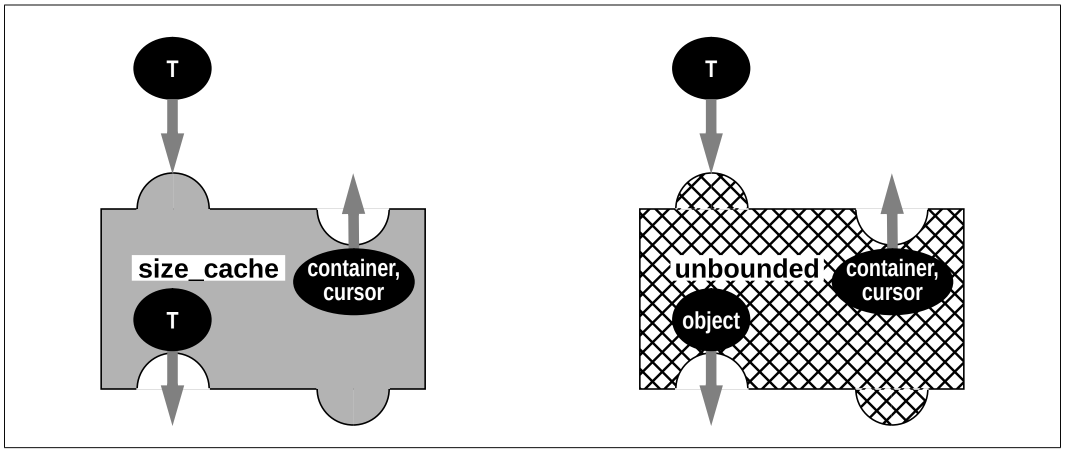


Figure 3.8 Organization of the *size_cache* and *unbounded* components.

next component. The significance is that *unbounded* has transformed its input type into a different output type. In essence, by using a forward declaration, a P++ component acts as a transformation which maps input types to output types.

The intuition intended to be conveyed by Figure 3.8 is that GenVoca components can model both “top-down” and “bottom-up” transformations. The outermost component of a composition corresponds to the top layer and the innermost component is the bottom layer. In our model of the data structures domain, the downward flow represents element transformations, and the upward flow represents container and cursor transformations (similar to standard C++ templates). The use of the forward declaration construct makes it possible to elegantly capture this complex group of type transformations.

It is substantially more complicated (or perhaps even impractical) to emulate this behavior using only C++ templates. For example, to emulate the component composition *floatsys* (reproduced below) using C++ templates, the declarations of Figure 3.9 would be needed.

```
typedef size_cache <float, bounded <100> > floatsys;
```

The first group of declarations separately specifies the implementations for the classes of *size_cache*. The second group is for *bounded*. The third group defines the implementation for the *sys_element* type, then uses that type as an input parameter to define *sys_container* and *sys_cursor*.[1] Not only are these C++ declarations confusing to read and error-prone to write, but they lack the type-checking and encapsulation properties of P++ components.

```
// templates to emulate the size_cache component
template <class element_rep> class size_cache_element { ... };
template <class T, class container_rep>
  class size_cache_container { ... };
template <class T, class container_type, class cursor_rep>
  class size_cache_cursor { ... };

// templates to emulate the bounded component
template <class T> class bounded_element { ... };
template <class T, int size> class bounded_container { ... };
template <class T, class container_type>
  class bounded_cursor { ... };

// instantiations to emulate the floatsys composition
typedef size_cache_element <bounded_element <float> >
  sys_element;
typedef size_cache_container <sys_element,
  bounded_container <sys_element, 100> > sys_container;
typedef size_cache_cursor <sys_element, sys_container
  bounded_cursor <sys_element, sys_container> > sys_cursor;
```

Figure 3.9 Attempting to emulate the *floatsys* composition using C++ templates.

1. Technically, *sys_element* can be supplied as a parameter value to both the *size_cache_container* and *bounded_container* only because the type transformations of *size_cache_element* and *bounded_element* are “compatible”. If not, then the above code would be even more complex.

3.5 Comparison of P++ with C++

Because the new language features of P++ resemble the existing constructs of C++, it is natural to wonder what makes P++ unique. What are the contributions of P++ to programming languages? Why not just emulate these features in existing languages? To our knowledge, no existing language provides the same set of features as P++. For example, Ada packages provide encapsulation capabilities similar to that of P++ components [Bar93] and functional languages like ML have sophisticated parameterization features [Tof90], but none of those languages offer everything that P++ does. This particular combination of features is important, because they are necessary to implement GenVoca component libraries and generators.

To demonstrate that P++ supports language features which are not already available in C++, we highlight the differences between the constructs of both languages. Although it is possible to emulate some P++ features in C++, this effort would be difficult and cumbersome. Moreover, as explained in Section 3.1, P++ automates certain aspects of GenVoca generator development that would otherwise be reimplemented by hand for each generator.

1. There is no concise way to emulate the P++ forward declaration construct in C++. A forward declaration allows a program to use a type before it has been defined. In C++, a program may create only pointers to a forward declared type; it cannot create objects of such a type. More importantly, a C++ program cannot provide a forward declared type as a parameter value for a template. As we have explained, however, this very capability is essential for a component to behave as a forward refinement program transformation.

2. 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 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. (It is possible to envision an alternate system which verifies the validity of each pair of components at runtime, but such a system would naturally suffer a runtime performance penalty.)
3. C++ nested classes are quite similar to 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]); 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) as atomic units of program construction, whereas C++ nested classes are used for data hiding (i.e.

abstraction) within a single class. Since encapsulation and abstraction are semantically distinct ideas, they should be expressed using distinct language constructs.

Chapter 4

The implementation of P++

The P++ compiler translates a P++ program into an equivalent C++ program. This C++ output can be converted into executable code using any standard C++ compiler. The three sections of this chapter provide successively more detailed descriptions of the P++ compiler's implementation. The first motivates the design of the compiler; the second describes its organization, internal data structures, and algorithms; and the third explains details of the P++ to C++ translation process.

4.1 Designing the compiler

Our goals in designing the P++ compiler were to make it portable across machine architectures and operating systems, easy to use, and straightforward to implement. These features would help popularize the use of P++ because the compiler would be easy to install, use, and maintain.

4.1.1 A P++ to C++ translator

Our first task was to determine what output the compiler would produce. A traditional compiler translates program source code into assembler code. Thus, we considered writing a compiler that translated P++ to assembler. To evaluate this approach, we examined the GNU C Compiler [Sta94]. This compiler is split into two subsystems: a “front-end” which translates program source code into an intermediate representation called RTL (register transfer language), and a “back-end” which converts the RTL code into assembler. We eventually dismissed this approach because converting P++ code into RTL — a relatively primitive code representation — seemed too complicated.

An alternative approach was suggested by the AT&T Cfront Compiler. Cfront is a popular C++ compiler, which translates C++ code into C; the C code is then translated by a C compiler into assembler. The benefit of this approach is that any standard C compiler can be used for the assembler code generation phase. Therefore, we could modify Cfront by inserting the P++ syntax into the existing C++ grammar, and adding the corresponding code to translate that syntax into C statements. We ultimately discarded this approach because Cfront was difficult to modify and converting P++ code into C was too complicated.

We finally settled on a different approach, to translate from P++ to C++. The primary advantage is that because P++ and C++ are so similar, the translation effort is relatively small. Most statements in a P++ program would remain unchanged during translation; the real translation effort would involve only the P++-specific constructs. In addition, because the P++ compiler outputs C++ code, any standard C++ compiler can be used to produce the executable. The wide availability of C++ compilers makes the P++ compiler largely independent of machine architectures and operating systems.

4.1.2 Leveraging off an existing C++ parser

The next step was to determine how to implement this translator. We initially considered modifying the GNU C++ Compiler (G++). This would involve isolating and modifying the parsing portion of the compiler, and retargeting the compiler’s front-end to produce C++ instead of RTL. Again, this task proved formidable: G++

consists of 600,000 lines of code, much of which manipulates RTL. Because this code is not cleanly separated from the C++ grammar code, a lot of work would have been necessary to adapt G++ to our task.

Instead we found an alternative system to use as a starting point. CPPP is a C++ parser developed at Brown University [Rei94]. This tool defines only a portion of a normal C++ compiler. It implements only a token scanner, parser, and semantic analyzer for C++ and performs no code generation nor optimization. The goal of CPPP is to provide a development framework for tools which analyze and manipulate C++ source code. Because CPPP provides only the functionality we sought, it served our needs well.

When the CPPP program is executed, it performs three tasks on the input C++ source code. First, it filters the source code through the standard C preprocessor. This step expands all preprocessor directives (e.g. *#include*, *#define*) and produces a new C++ source file. Second, CPPP scans and parses the source file. This produces an internal tree data structure which represents the entire source program. Third, CPPP performs semantic processing on the tree data structure. The result of this processing is a new tree where the type and scope of every identifier have been resolved.

CPPP is a complex program which has a very intricate flow of control. This is not the result of a poor design or implementation, but actually reflects the complexity of the C++ language. For example, the scanner and parser of CPPP are quite complicated because the C++ language has a context-sensitive grammar. A scanner reads the source code of a program character by character. When a complete word is read, the scanner determines if it corresponds to a reserved word of the language or a user-specified identifier name. If it is a reserved word, the scanner returns a special token which corresponds to the particular word; otherwise, it returns a generic token which denotes an identifier.

Usually, when a context-sensitive language is translated into a collection of formal grammar rules (for a compiler generator like *yacc*), the grammar rules have parsing ambiguities (e.g. shift/reduce or reduce/reduce conflicts). The problem is that such ambiguities result in grammars that behave unpredictably. For example, in a program with nested “if” statements, it is not easy to tell to which statement an “else” clause belongs (i.e. the “dangling else” problem); if the grammar is not corrected to resolve this ambiguity, the compiler generator arbitrarily chooses an interpretation for that code. Consequently, parsing ambiguities make compiler development difficult and error-prone.

To avoid this problem, the CPPP scanner inserts extra symbolic tokens into the token stream that it sends to the parser. These tokens provide hints to the parser; they convey extra semantic information that makes it possible to eliminate all ambiguous rules from the CPPP grammar. For example, when the scanner notices the start of a function definition, it inserts an extra token to help the parser distinguish a function definition from a function declaration. In effect, embedded within the scanner is a simplified version of the C++ parser, which is used to select the symbolic tokens to insert.

In practice, it is impossible to correctly insert these extra symbolic tokens without scanning far ahead in the token stream to see what follows. Consequently, CPPP implements a buffering mechanism which allows the scanner to look ahead several tokens before returning a single token to the parser. In addition, it is sometimes necessary to try various interpretations of the token stream before settling on which symbolic tokens to insert. Therefore, CPPP also implements a backtracking mechanism which can undo modifications to the token stream as various interpretations are examined and discarded.

4.2 Developing the compiler

The P++ compiler represents a major development effort. Implementing this system took six months; it entailed understanding and debugging a 34,000 line C++ parser (CPPP), removing 9,000 lines of unnecessary code and comments, and adding 10,000 lines of code to perform P++ to C++ translation. This section summarizes the major development tasks that we performed.

We quickly found that CPPP was not immediately ready to use. The source code files of CPPP were difficult to understand and were somewhat tangled. In addition, CPPP had several C++ parsing bugs, which caused it to reject simple C++ test programs. Consequently, the first weeks of development were spent in debugging CPPP, reorganizing the source code files, and commenting code.

The next step was to modify the C++ grammar to include P++ constructs. This involved adding new grammar rules which correspond to realm declarations, component declarations, constant and type parameters, forward declarations, etc. In addition, for each new grammar rule, we wrote code actions which add representations of P++ constructs to the internal tree data structure of the compiler. Thus, as the P++ compiler parses the input source code, it incrementally builds a tree data structure which represents the entire program.

In CPPP, the scanner and parser are tightly interdependent modules. As described in Section 4.1, the scanner has a backtracking mechanism which tries various interpretations of the token stream as it inserts extra symbolic tokens (as parsing hints). Each interpretation corresponds to a different set of grammar rules. In essence, the scanner predicts which grammar rules will successfully parse the token stream by successively emulating each of these rules. Because the scanner and parser are closely related, adding P++ syntax to the CPPP grammar also entailed modifying the scanner to emulate these new grammar rules. This task constituted the next phase of implementing the P++ compiler.

Once these modifications were complete, we proceeded to update the internal data structures of CPPP to properly support P++ code. When CPPP parses a program, it incrementally constructs a data structure called a *syntax tree*. This data structure provides CPPP with a convenient representation for analyzing and manipulating the program's source code. The syntax tree consists of a collection of nodes, which are organized as a directed acyclic graph. The edges of the graph represent relationships between the nodes, where a source vertex corresponds to a "parent node" and a sink vertex corresponds to a "child node". All nodes except for the "root node" have a parent, and every node has zero or more children.

Each node of the syntax tree corresponds to a code element from the source program. For example, Figure 4.1 shows the portion of a syntax tree that corresponds to the *square_root* function declaration. Associated with each node is a type and optional type-specific information. A node's type indicates the kind of code element represented by the node. As shown in the figure, the *FunctionArgs* node represents the function's arguments and the *Name* node corresponds to a user-specified identifier. Notice that a *Name* node contains additional information: the value of the string identifier being declared (e.g. *square_root*, *num*). The figure also shows that the syntax tree captures the contents of the input source code in sufficient detail to subsequently reconstruct that code. This is what makes the syntax tree a convenient data structure for storing and managing source code.

Our next task, therefore, was to modify the CPPP syntax tree to support the representation of P++ code. To do so, we introduced several new node types, which correspond to new P++ constructs like realm declarations, component declarations, etc. The goal was to precisely capture the information conveyed by P++

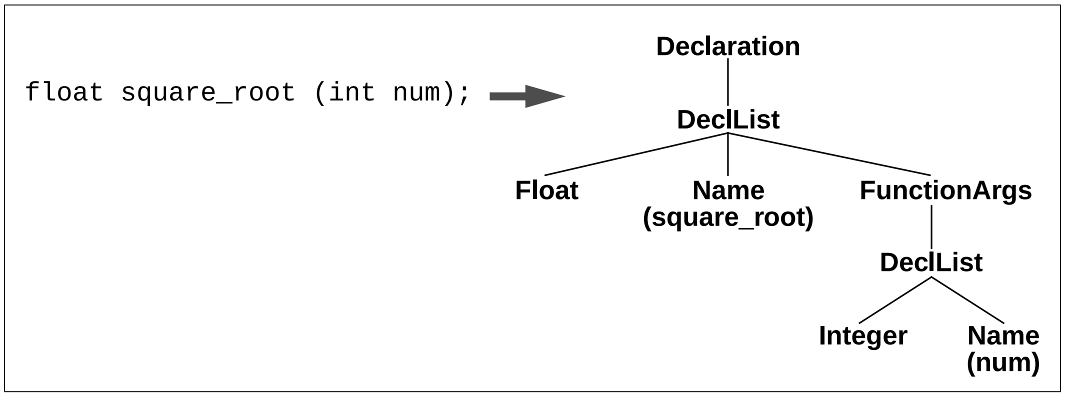


Figure 4.1 Translation of a function declaration into a syntax tree.

declarations so the compiler could subsequently analyze and translate those declarations.

After the input source code has been parsed and the syntax tree constructed, the next compilation step is to transform the syntax tree into a *semantic tree*. A semantic tree is a data structure similar to a syntax tree: it also consists of nodes organized in a directed acyclic graph. The basic difference is that a semantic tree captures the actual meaning of the program source, whereas the syntax tree only conveys the textual structure of the program. For example, a syntax tree represents several kinds of declarations using the same type of node, a *Declaration* node. In contrast, a semantic tree represents a variable declaration with a *VariableDecl* node, a function declaration with a *FunctionDecl* node, and a *typedef* declaration with a *TypedefDecl* node. By using different node types to represent different kinds of declarations, a semantic tree is able to record additional information that is specific to the particular kind of declaration being processed. Moreover, the semantic tree uses a uniform (canonical) representation for each kind of declaration, which makes it is easier to analyze and manipulate the nodes for a declaration.

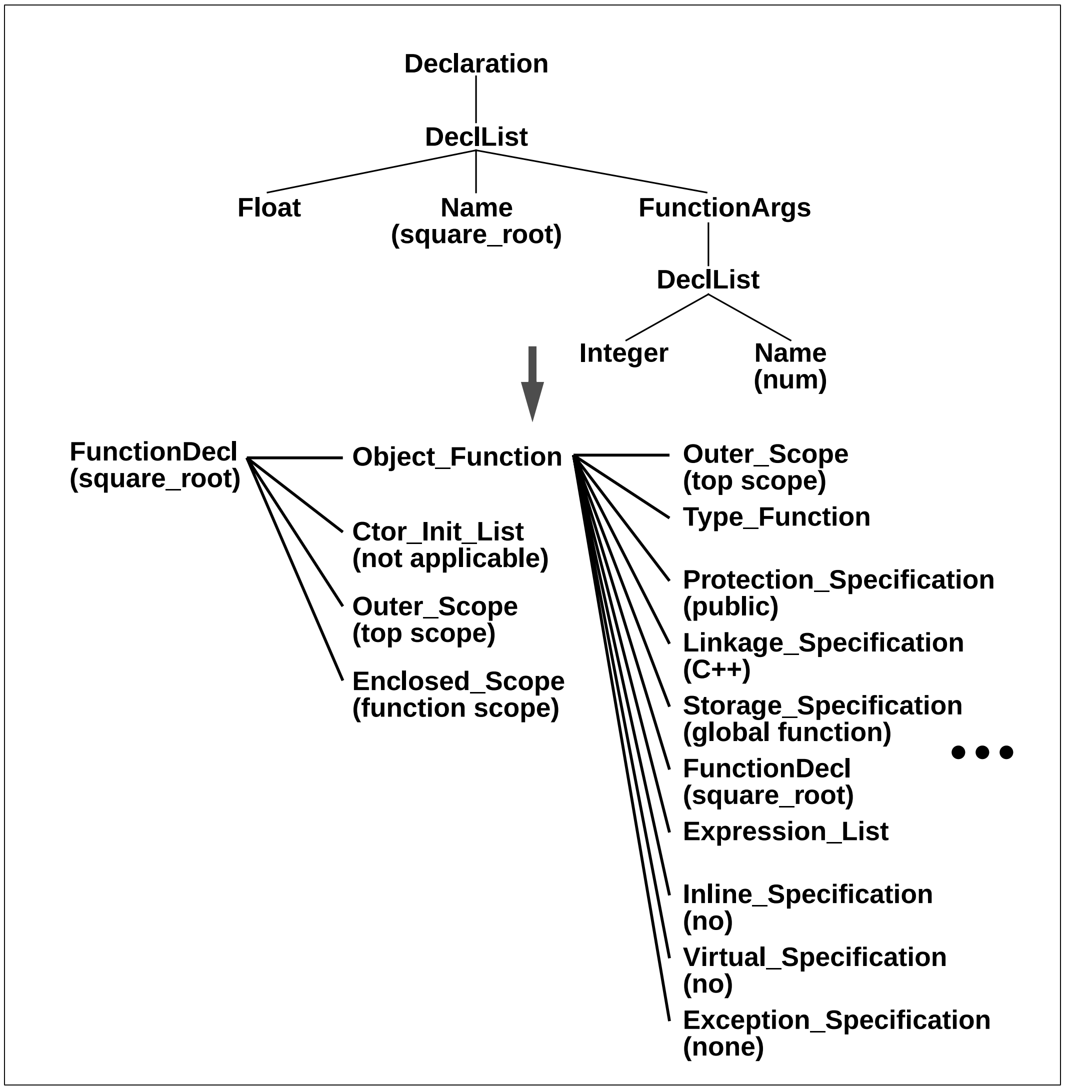

Declaration
DeclList
Float
Name (square_root)
FunctionArgs
DeclList
Integer
Name (num)
FunctionDecl (square_root)
Object_Function
Ctor_Init_List (not applicable)
Outer_Scope (top scope)
Enclosed_Scope (function scope)
Outer_Scope (top scope)
Type_Function
Protection_Specification (public)
Linkage_Specification (C++)
Storage_Specification (global function)
FunctionDecl (square_root)
Expression_List
Inline_Specification (no)
Virtual_Specification (no)
Exception_Specification (none)
• • •

Figure 4.2 Translation of a syntax tree into a semantic tree.

Figure 4.2 shows a fragment of the semantic tree which corresponds to the syntax tree of Figure 4.1. As this example demonstrates, the semantic tree offers a more detailed representation of a code fragment. From a *FunctionDecl* node, we can easily determine a function's scope, return type, or inline status. Because the semantic tree organizes all this information in a consistent format, the semantic tree is a convenient representation for reasoning about a program. In contrast, the syntax tree is unable to convey this information because its node types just capture the textual organization of the function declaration.

While constructing the semantic tree, the compiler performs several code simplification tasks too. For example, if the source code contains an expression involving only constant values, the compiler replaces the corresponding portion of the semantic tree with a subtree containing the equivalent computed value.

To support P++, we added another code simplification task to the compiler: the expansion of parameterized declarations. We found that certain P++ constructs cannot be translated directly to C++. In particular, parameterized realms and components cannot be expressed as C++ templates, because C++ instantiates templates in a different manner than P++ expands realms and components (e.g., a P++ component may include forward declarations or implicit parameters). To solve this problem, the P++ compiler finds all expressions involving parameterized declarations. It replaces the corresponding portions of the semantic tree with equivalent subtrees that represent expanded versions of the declarations. In this way, P++ provides the mechanism necessary to instantiate parameterized declarations. (An example of this procedure is given in the next section.)

The final task involved C++ code generation. The original CPPP system produced no output; instead, the semantic tree was the only result of parsing. In the P++ compiler, we added code to scan the semantic tree and output the corresponding C++ program. The compiler recursively traverses the tree and generates a C++ representation of each node. Because each node's type identifies the kind of code element to be output, this code generation module was relatively easy to implement. Figure 4.3 depicts the intermediate compilation steps which transform P++ code into C++ code.

To summarize, the P++ compiler produces C++ code after making three passes of the input program. The first pass transforms the input source code into a

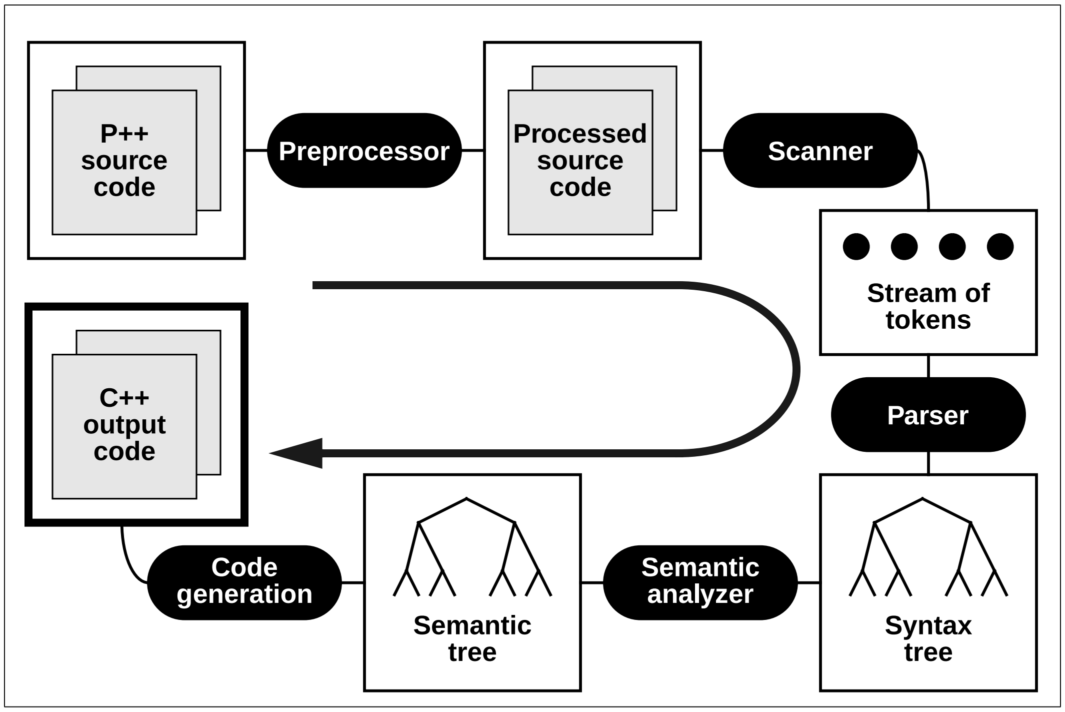


Figure 4.3 Compilation steps of the P++ compiler.

syntax tree. The second pass scans the syntax tree and produces a semantic tree; this process also translates all P++ constructs into C++, simplifies expressions involving only constant values, and converts parameterized declarations into unparameterized declarations. The third pass scans the semantic tree and outputs the correspond C++ code.

As a performance optimization, it would be possible to precompile source files which are frequently encountered. For example, if P++ were able to save in an efficient file format the syntax trees for realms and components, then these trees could be reloaded by P++ during subsequent compilations. This optimization would reduce compilation time because it would avoid the cost of repeatedly scanning and parsing the source files. Ultimately, we might need to implement this feature so that P++ could scale to efficiently compile large systems.

4.3 Translating P++ into C++

When the compiler creates the semantic tree, it eliminates all P++ code from the syntax tree and replaces it with equivalent C++ code. This procedure is noteworthy because the compiler must often perform extensive analysis and code modification to accomplish this translation. We believe that the study of this process will help to explain the relationship between P++ and C++.

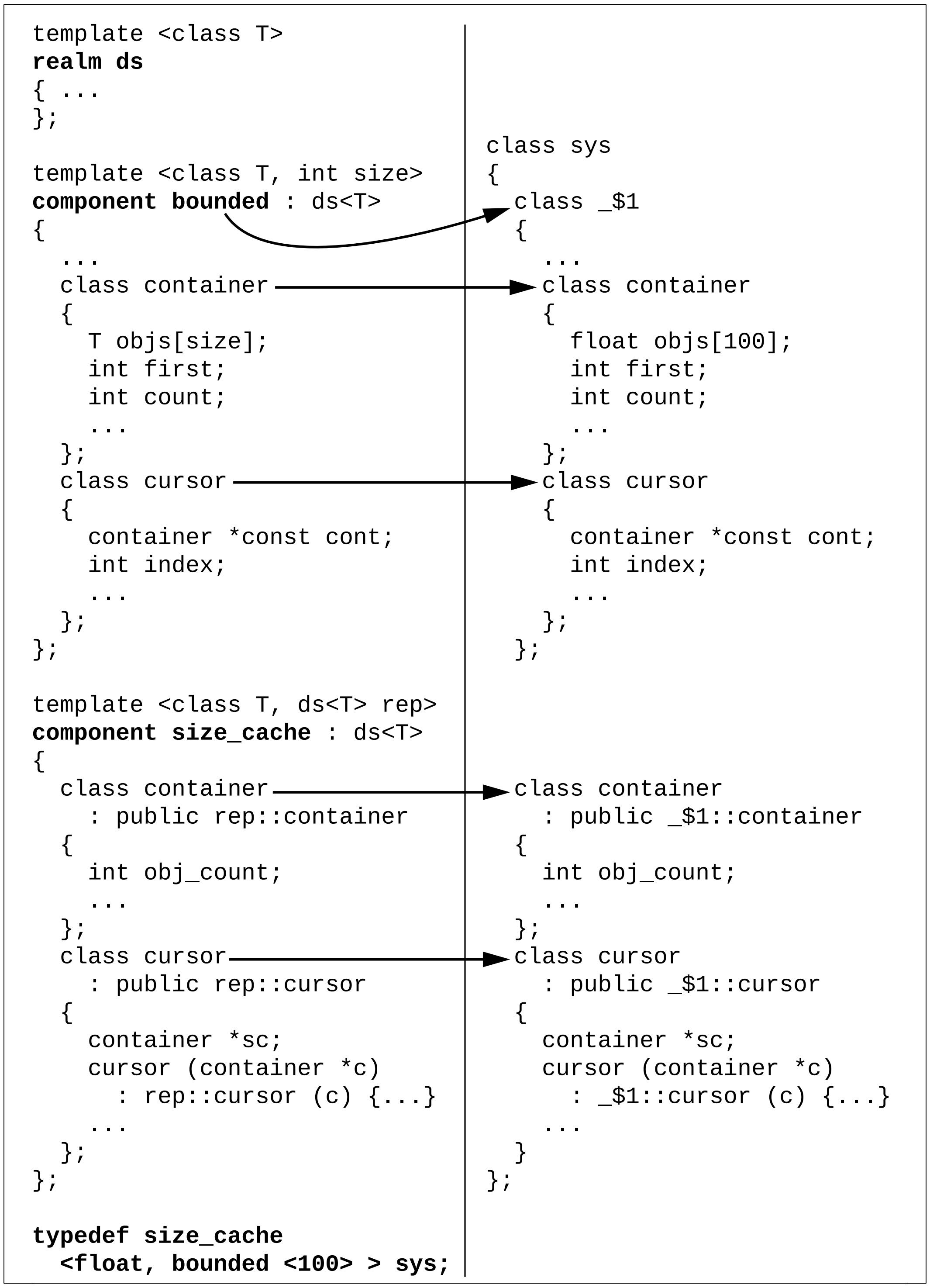
A realm declaration describes only the interface of a component. It corresponds to no executable code; therefore, the P++ compiler generates no C++ output for this construct. Nevertheless, a realm declaration still plays an important role in compiler error checking. In subsequent component declarations, the compiler can check whether a component actually implements all functions, classes, and methods declared by its realm. In addition, if a program declares a realm parameter, the compiler can verify that the parameter accesses only the interface described by the realm. Thus, a realm serves as a formal interface description which is used extensively during compilation.

The P++ compiler processes a parameterized realm declaration in a different manner than an unparameterized declaration. When it first encounters a parameterized realm, the P++ compiler simply records the syntax tree for the declaration in an internal symbol table. If the parameterized realm is subsequently instantiated, the compiler recalls the original declaration and performs several steps to create an unparameterized replica of the realm declaration. First, the compiler creates a copy of the syntax tree that corresponds to the parameterized realm. Next, it replaces within the copy all instances of parameter names with the corresponding parameter values. Finally, the compiler creates a new name for the copied realm declaration; this name uniquely identifies this particular instantiation of the realm with these parameter values. The result of all this work is a new realm declaration which can be processed in the same manner as any other unparameterized realm. To summarize, the P++ compiler instantiates a parameterized realm by creating a new syntax tree which corresponds to the equivalent unparameterized realm.

A component construct supplies the implementation details of a component. Perhaps surprisingly, the compiler generates no C++ output when this construct is seen. Instead, the compiler simply records the definition of the component within its symbol table. If the component is subsequently used within a system implementation, only then will the component's definition be recalled and translated into C++.

How is a P++ system implementation converted to C++? A system implementation consists of the composition of one or more components. Simultaneously instantiating several components to generate C++ code would be difficult; instead, the P++ compiler handles a parameterized component similar to the way it handles a parameterized realm. Recall that P++ converts a parameterized realm into an equivalent unparameterized realm. This allows the compiler to process both kinds of realms in the same manner. Likewise, P++ converts a parameterized component into an unparameterized component. If a component has only type and constant parameters, then the compiler simply creates a copy of the component's syntax tree and substitutes parameter names with parameter values in the new tree. However, if the component has a realm parameter, then the compiler creates a nested class declaration and it instantiates the component value for that parameter within the nested class.

For a concrete example of this translation process, consider the code of Figure 4.4. The left column shows the declarations for a parameterized realm, two parameterized components, and a component composition; the right column lists the equivalent C++ code. The P++ compiler would perform the following steps to generate this code:



```
template <class T>
realm ds
{ ...
};

template <class T, int size>
component bounded : ds<T>
{
  ...
  class container
  {
    T objs[size];
    int first;
    int count;
    ...
  };
  class cursor
  {
    container *const cont;
    int index;
    ...
  };
};

template <class T, ds<T> rep>
component size_cache : ds<T>
{
  class container
    : public rep::container
  {
    int obj_count;
    ...
  };
  class cursor
    : public rep::cursor
  {
    container *sc;
    cursor (container *c)
      : rep::cursor (c) {...}
    ...
  };
};

typedef size_cache
  <float, bounded <100> > sys;
```

```
class sys
{
  class _$1
  {
    ...
    class container
    {
      float objs[100];
      int first;
      int count;
      ...
    };
    class cursor
    {
      container *const cont;
      int index;
      ...
    };
  };

  class container
    : public _$1::container
  {
    int obj_count;
    ...
  };
  class cursor
    : public _$1::cursor
  {
    container *sc;
    cursor (container *c)
      : _$1::cursor (c) {...}
    ...
  }
};
```

Figure 4.4 Translation of P++ code (left column) to C++ code (right column).

1. The compiler parses the entire program and constructs a syntax tree. In this case, the syntax tree contains the declarations for *ds*, *bounded*, *size_cache*, and *sys*.
2. The syntax tree is converted into a semantic tree by performing a depth-first traversal and recursively processing each subtree. When the compiler traverses the subtrees for the *ds*, *bounded*, or *size_cache* declarations, it copies the declaration into the symbol table (but outputs no code).

   When the subtree for *sys* is traversed, the compiler converts each parameterized declaration of the composition into an equivalent unparameterized declaration. Thus, the next step is to generate an unparameterized version of *size_cache*. In doing so, the compiler discovers that *size_cache* actually belongs to the parameterized realm *ds*; consequently, the compiler first instantiates *ds* with the parameter value *float*. The compiler then evaluates all of the parameter values of *size_cache*. One of these values is actually the parameterized component *bounded*, which must be instantiated next. In turn, *bounded* belongs to a parameterized realm (*ds*), which must be instantiated too. The compiler is then able to generate an unparameterized replica of *bounded*, which it calls *_$1* (which is just a globally unique identifier name).

   Finally, the compiler generates an unparameterized replica of *size_cache*, which it calls *sys*. Subtrees for these replicas are inserted into the semantic tree in place of the component composition *sys*. (This entire process is summarized in Figure 4.5.)

   In Figure 4.4, note that the replica for *bounded* is nested within the replica for *size_cache*; this nesting reflects the fact *bounded* is a component value for one of *size_cache*'s parameters. Therefore, if the components within a system implementation are nested *n* levels deep, then the corresponding generated code will contain classes that are nested *n* levels deep.

3. The compiler performs a depth-first traversal of the semantic tree, outputting the C++ representation of each node it encounters. At this point, the declarations for the *_$1* and *sys* classes are output.

- Process *size_cache<float, bounded<100> >*
  - Lookup *size_cache* ... it belongs to a parameterized realm
  - Instantiate *ds<float>*
  - Examine parameters for *size_cache* ... one is a component parameter
  - Process *bounded<100>*
    - Lookup *bounded* ... it belongs to a parameterized realm
    - Instantiate *ds<float>*
    - Examine parameters for *bounded* ... all are constants or types
    - Instantiate *bounded<100>* and call it *_$1*
  - Instantiate *size_cache<float, _$1>* and call it *sys*

Figure 4.5 The instantiation steps for the system *sys*.

In summary, the compiler replaces each P++ construct with equivalent C++ code when it converts the syntax tree into the semantic tree. If a parameterized declaration or definition is seen, the compiler replaces it with an equivalent unparameterized version. If the declaration / definition has constant or type parameters, the corresponding value is substituted in place of each parameter name. If it has component parameters, they are recursively instantiated and transformed into nested class declarations.

Chapter 5

Results

To evaluate the benefits provided by P++ in implementing scalable domain-specific component libraries, we created a library of components for the domain of data structures and compared these components with an existing data structures class library, the Booch C++ Components.

The Booch library is a meaningful basis of comparison because it is a widely-used library whose design and organization has been refined over time. Originally an Ada library consisting of 150,000 lines of code, it evolved into a 30,000 line C++ library. In [Big94], Biggerstaff suggests that the Booch library serves as "challenge problem" for measuring the benefits offered by programming languages in writing scalable software libraries:

> "I believe that Booch's library is a benchmark that approximates the limits to which today's languages can be pushed in creating reuse libraries that emphasize reduced library growth while providing an acceptable performance level. The degree to which new, proposed language constructs improve on this benchmark will indicate their contribution to the solution of the scaling problem."

This chapter reviews our experiment's design and explain the results.

5.1 The Booch C++ Component Library

Our experiment compared two implementations of a component library, one written in C++ and the other written in P++. The two versions of the library implemented identical algorithms and interfaces; the differences arose from the particular way that each library organizes and combines its components. Our goal was to quantify the differences in code size and performance between the two libraries.

We selected the Booch C++ Components[1] as the basis for comparison. The Booch library is unique because it is specifically designed to implement many different versions of algorithms for certain data structures. In contrast, libraries like the C++ Standard Template Library [Pla95] and the GNU libg++ library [Lea88] only provide a few implementations of a broad variety of data structures. We chose the Booch C++ Components because it uses inheritance and parameterized types to reduce code replication and reuse code. This library offered an excellent opportunity to study issues like scalability and inheritance in library design.

The data structures domain consists of algorithms and data types which implement containers of objects — examples include bags, sets, lists, and queues. Why choose this domain? Data structures is a simple, easy-to-understand problem domain. The basic ideas and algorithms are already familiar to most programmers, making data structures an ideal framework for testing and explaining our results. This domain consists of many enumerable data structure “systems”. Moreover, in our experience with this domain, we found that data structures pose the same tech-

1. We used version 1.47 of the Booch C++ Component Library for our experiments. We also had access to a “beta” copy of version 2 of this library, but because it was still undergoing development and testing, it was not suitable for our use. Nevertheless, based on a preliminary examination of this library, we believe that our overall results would be the same; even though the new library uses different algorithms, it still possesses the same limitations that we will describe in this chapter.

nical challenges and operate in the same manner as more complicated domains like databases and network protocols.

The primary goal of the Booch library "is to provide a carefully designed collection of useful data structures" [Boo90]. To do so, the library offers a wide selection of data structure classes in the hope that these classes will be relevant to many user applications. The library is organized as a suite of data structure families, where a family corresponds to a basic algorithm like bag or set (see Figure 5.1 for a complete list). A family implements several variations of an algorithm; variants may have different performance characteristics, memory requirements, or concurrency control mechanisms.

The organization of the Booch library is the result of a careful object-oriented analysis and design effort [Boo87]. Techniques such as parameterization and inheritance have been employed to minimize code replication between the library's classes. For example, the Booch library defines a hierarchy of classes to implement several varieties of bags (see Figure 5.2). The superclass of this hierarchy (i.e. *Bag*) is responsible for declaring the interface which is common to all bag data structures. The superclass also implements some simple bag methods, which the other classes in the hierarchy inherit. The subclasses of the hierarchy correspond to bags which implement various permutations of data structure features. Thus, the *Guarded_Bounded_Bag* class combines algorithms for three distinct features: the bag algorithm implements an unordered container of elements where duplicates are permitted; the guarded algorithm provides lock / unlock operations so that the user can explicitly implement a concurrency control policy; and the bounded algorithm preallocates a fixed number of elements in the container. The *Item* and *Size* parameters of *Guarded_Bounded_Bag* are used to customize the behavior of this data structure. The *Item* parameter specifies the type of element stored by the con-

Data structure families in the Booch library	
Bag	unordered collection of objects, which may contain duplicates
Deque	ordered sequence of objects, where objects may be inserted or removed at either end
List*	a linked list of objects, which are automatically garbage collected
Map*	a tabular data structure which associates instances of one kind of object with instances of some other kind of object
Queue	ordered sequence of objects, with “first-in first-out” semantics
Ring list	ordered sequence of objects, organized in a loop
Set	unordered collection of objects, which contains no duplicates
Stack	ordered sequence of objects, with “last-in first-out” semantics
String	ordered sequence of objects, where any object may be accessed directly
Tree*	a binary tree of objects, which are automatically garbage collected
Data structure features which are available to every family	
Bounded	static memory allocation algorithm (upper bound on total number of objects)
Unbounded	dynamic memory allocation algorithm (no upper bound on total number of objects)
Managed	freed objects are stored on a list for subsequent reuse
Controlled	a version of managed which operates correctly ina multi-threaded environment
Sequential	assumes a single-threaded environment
Guarded	assumes a multi-threaded environment, where mutual exclusion is explicitly performed by the user
Concurrent	assumes a multi-threaded environment, where the object ensures that all read and write accesses are serialized
Multiple	assumes a multi-threaded environment, where the object permits multiple simultaneous read accesses, but serializes write accesses
Data structure features which are available to deques and queues only	
Balking	objects may be removed from the middle of a sequence
Priority	objects are sorted based on some priority function

Figure 5.1 A glossary of Booch data structure terminology.

* We did not reimplement this data structure family in the P++ Data Structures library.

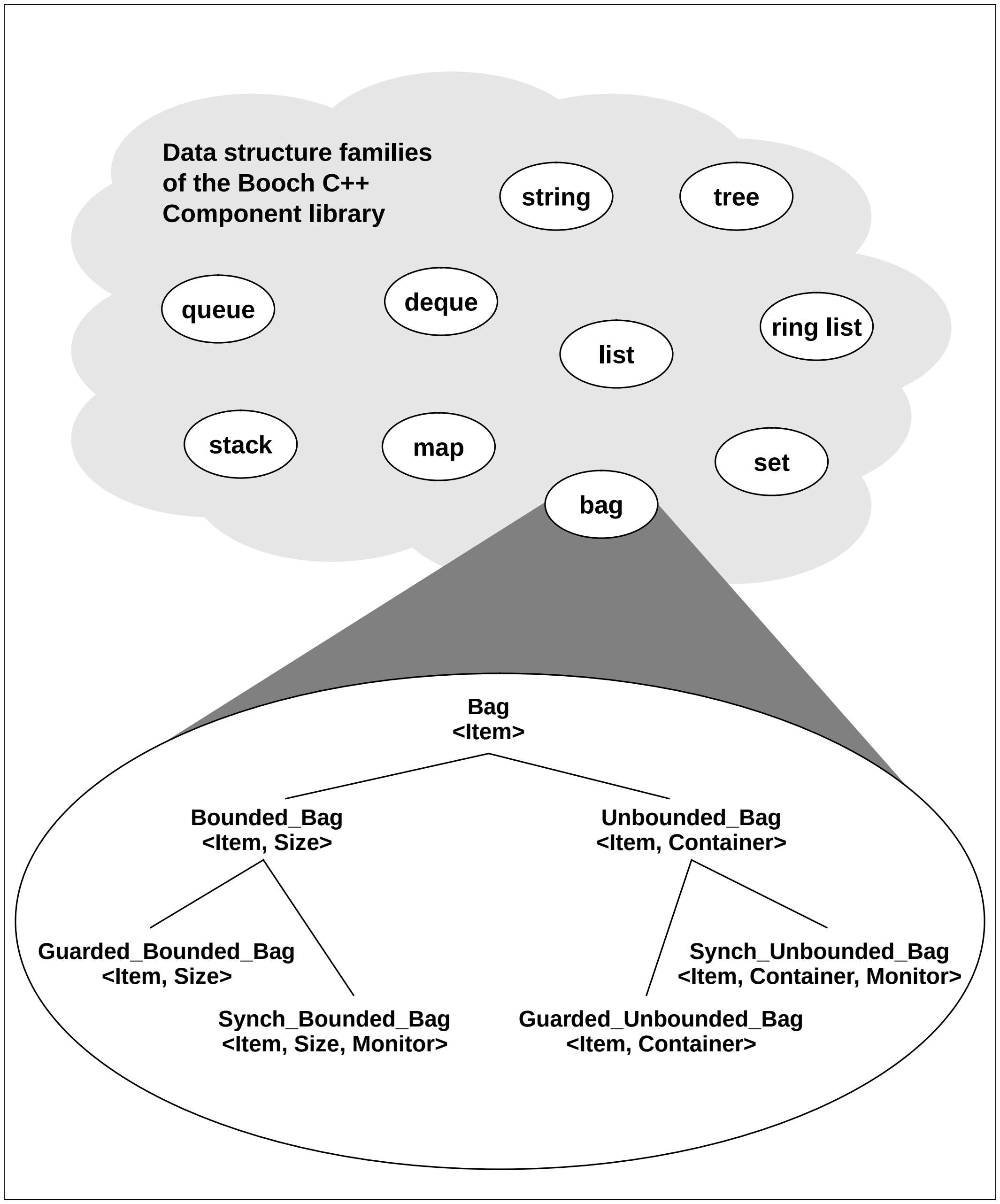


Figure 5.2 Organization of the Booch library.

tainer, and the *Size* parameter tells the bounded algorithm how many elements it should preallocate.

Parameterization can also be used to customize the behavior of generic data structure algorithms. The example of Figure 5.2 shows that one parameter of *Synchronized_Bounded_Bag* class is *Monitor.* Depending on the value specified for this parameter, this class can generate a “concurrent bounded bag” (which has exclusive-lock semantics) or a “multiple bounded bag” (which has read-lock / write-lock semantics). This example demonstrates that a single parameterized class may correspond to several permutations of features. Consequently, a library which uses parameterization in this manner can avoid exhaustively enumerating all permutations of similar data structure features.

5.2 The P++ Data Structures library

Our experiment was to create a library of P++ realms and components for the data structures domain, and to compare the size and performance of this library with the Booch library. To measure performance, we devised several benchmark programs which heavily exercised the operations provided by the generated container data structures. These programs verified that both libraries implemented functionally identical components and that the generated systems had similar runtime speeds.

5.2.1 Library organization

Figure 5.3 lists the realms and components which comprise the P++ library. Some of these realms directly correspond to abstract classes from the Booch library. In other cases, a single P++ realm replaces several Booch abstract classes. One of the challenges in designing the P++ data structures library was identifying opportunities for consolidating similar interfaces into a single realm. Even more difficult was recognizing the need for new realms to represent funda-

Realm	***Components***
ds<T>	*bounded<T, int>* *concurrent<T, ds>* *guarded<T, ds>* *multiple<T, ds>* *size_cache<T, ds>* *unbounded<T, memory>* *unbounded_double<T, memory>*
memory<T>	*controlled<T, memory>* *heap<T>* *managed<T, memory>*
collection<T>	*bag<T, ds>* *bag_noref<T, ds>* *cset<T, ds>*
ring<T>	*ring_list<T, ds>*
balking_sequence<T>	*balking_queue<T, ds>* *priority_balking_queue<T, ds>*
nonbalking_deque_sequence<T>	*deque<T, balking_sequence>*
balking_queue_sequence<T>	*balking_queue<T, balking_sequence>*
nonbalking_queue_sequence<T>	*nonbalking_queue<T,balking_sequence>*
balking_stack_sequence<T>	*balking_stack<T, balking_sequence>*
nonbalking_stack_sequence<T>	*nonbalking_stack<T, balking_sequence>*
variable_string<T>	*string_list<T, ds>*

Figure 5.3 Organization of the P++ data structures library.

mental concepts that are not explicitly represented as components in the Booch library. The basic strategy behind our domain decomposition was to identify and represent the fundamental abstractions that occur in the Booch data structures domain.

Notice that the organization of features in the Booch library is very different from the organization of realms in the P++ library. The features of the Booch library are loosely grouped according to their semantics (i.e. bags, deques, and sets are all data structures which store collections of elements, even though their interfaces are substantially different). In the P++ library, however, components are grouped into realms because of their interfaces (i.e. the components in a realm share the same interface). This means that there is not necessarily a one-to-one correspondence between Booch features and P++ realms.

The key insight in our design of the P++ library was to recognize how to factor out several algorithms that repeatedly occurred in numerous Booch data structures. These algorithms provided generic data structure capabilities such as concurrency control and memory allocation, yet they were available in data structures as varied as strings and priority queues. After much trial and error, we finally settled on the interface for *ds*, a realm which represents generic data structure algorithms. The design of *ds* was partly driven by the lessons learned from the Genesis generator, which demonstrated how to model database algorithms as type transformations [Bat88]. By applying similar techniques to the Booch data structures, we were able to define this realm and its components, to implement all of the common algorithms of the Booch library.

The following overview lists the realms in the P++ library. We start with the simplest abstractions of the library and progress to more complex abstractions which build upon the earlier ones. For each realm, we indicate the corresponding Booch abstract classes.

- *ds* — This realm describes the interface of a primitive data structure collection. It offers operations and data types to manipulate a collection of objects. The *ds* realm specifies few details about the semantics of this container. Rather, the purpose of *ds* is to permit the representation of generic data structure algorithms which are not concerned about specific order rules, memory allocation schemes, or concurrency control mechanisms. There is no Booch abstract class which is analogous to this realm.

- *memory* — This realm describes an abstraction which does not directly correspond to any Booch abstract class. The purpose of this realm is to capture the various memory allocation algorithms that occur in the Booch library; *memory* declares classes which perform memory management.
- *collection* — This realm provides operations and data types for manipulating an unordered container of objects. *collection* does not specify the behavior of the container if a duplicate object is inserted. By omitting this aspect of the container's operational semantics, *collection* is functionally equivalent to two Booch abstract classes, *Bag* and *Set*. A *Bag* represents containers which allow duplicate copies of objects, and a *Set* represents containers which prohibit duplicates (i.e. an exception is thrown if duplicate objects are inserted).
- *ring* — This realm describes a container data structure which stores its elements in a circularly linked list. One element is marked as the "start" of the ring. Element insertions and deletions typically occur at this marker position. This realm corresponds to the *Ring* abstract class.
- *sequence* — The P++ library contains realms which describes several kinds of sequences. Each realm corresponds to a different combination of insertion semantics and access semantics. The *stack* realms have first-in last-out semantics, whereas the *queue* realms have first-in first-out semantics, and the *deque* realms permit insertions and removals at either end of the sequence. The *balking* realms permit the removal of intermediate elements in the sequence, while the *nonbalking* realms only allow removal of the end elements of the sequence. The result is six varieties of *sequence* realms, each

with slightly different interfaces and semantics. These realms correspond to several abstract classes from the Booch library, such as *Balking_Deque*, *Nonbalking_Queue*, etc.

- *variable_string* — This realm represents a container data structure which organizes its elements linearly. The semantics are equivalent to those of a resizable array: elements may be accessed by an index value, and new elements can be prepended or appended to the array. This realm corresponds to the *String* abstract class.

For each realm in the P++ library (see Figure 5.3), there are one or more components which provide a concrete implementation of that realm's interface. The *ds* realm has the most components, because this realm corresponds to general-purpose algorithms that are relevant to many data structures. For example, the *size_cache* component defines data types and operations which count the number of elements in a container; the *bounded* component preallocates a fixed-size memory region to store the elements of a container; and the *concurrent* component implements a concurrency-control monitor to serialize all read and write operations in a multi-threaded environment.

Within a given realm, most components generally have the same realm parameters. This observation is not surprising, because a component can be modelled as a layered software abstraction. Figure 5.4 shows the hierarchy of abstractions in the P++ component library. A node corresponds to a realm, and an edge indicates how components in a realm are typically parameterized (components corresponding to the realm at the source node of an edge usually possess a realm parameter which corresponds to the destination node of the edge).

In conclusion, using the *ds* realm as the cornerstone for the P++ library, it was fairly straightforward to design the remaining realms and components. All Booch data structure families implemented different variations of a simple con-

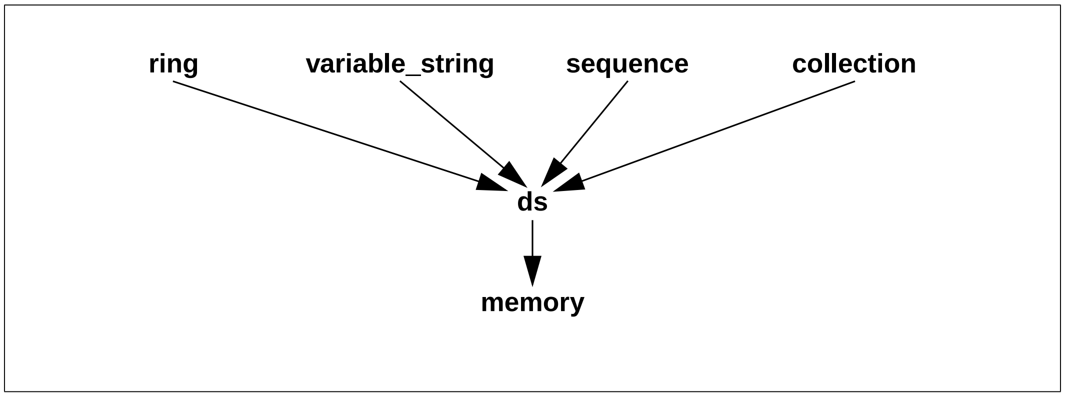


Figure 5.4 Hierarchy of realm abstractions.

tainer abstraction. That is, realms like *collection*, *ring*, and *variable_string* bear a strong resemblance to *ds*, except that methods and classes have been renamed. Consequently, the components belonging to these realms were quite simple; they just translated operations from one interface into *ds* operations.

5.2.2 Comparing library sizes

One basic goal of our experiment was to compare the sizes of the P++ and Booch libraries. The size of a library is proportional to the time and effort required to develop and maintain the library — it is therefore advantageous to make libraries smaller. Because there is no single metric for determining a library's size, we evaluated both libraries on several criteria: the number of abstract interfaces, the number of concrete components, and the number of lines of code.

The overall Booch library defines a total of ten data structure families; for this research project, we selected seven families to reimplement in P++ (see the top portion of Figure 5.1). We omitted three of the Booch families because we found them to be very similar to data structure families already represented by the P++ library. The goal of our work was not to simply reimplement the Booch library, but to explore issues of scalable library design; the seven families that we selected were sufficient for this task. Nevertheless, we still performed a careful analysis of the remaining three families to understand what realms and components would be needed, without actually implementing them. We combined these results with measurements of the size of the P++ library, to predict the overall size of a complete reimplementation of the Booch C++ Components. Because of this analysis, our partial reimplementation of the Booch library was almost as instructive as a full reimplementation would have been.

Figures 5.5 and 5.6 summarize the size differences of the two libraries. The first figure shows the measurements obtained from the seven Booch data structure families and from our current P++ library implementation. The second figure shows results for all ten Booch data structure families and extrapolated results for a complete P++ library implementation.

Our first measurement was to compare the number of abstractions defined by both libraries. It was no surprise to learn that there are a few more realms in the P++ library than abstract classes in the Booch library: as we described before, the P++ library introduces new interfaces to represent primitive abstractions which do not explicitly appear in the Booch library. Specifically, in the first figure we see that the Booch library has 7 abstract classes (corresponding to 7 data structure families), whereas the P++ library has 11 realms (which were listed in Figure 5.3).

The next measurement was to compare the number of Booch concrete classes with the number of P++ components. The first figure shows that the Booch library was four times larger than the P++ library (82 concrete classes versus 21 components). In the second figure, we predict that the entire Booch library would be three times larger than a complete implementation of the P++ library (100 concrete classes versus 29 components). In either case, the P++ library is substantially smaller.

	Booch C++ Components	**P++ Data Structures Library**
No. of abstract classes / realms	7	11
No. of concrete classes / components	82	22
Lines of code	11067	2760
No. of generated data structures	169	208

Figure 5.5 A comparison of the sizes of the Booch and P++ libraries (for seven data structure families).

	Booch C++ Components	**P++ Data Structures Library**
No. of abstract classes / realms	10	18
No. of concrete classes / components	100	30
No. of generated data structures	206	308

Figure 5.6 A comparison of the sizes of the Booch and P++ libraries (for all ten data structure families).

What accounts for this difference? Looking back to Figure 5.2, we observe that several data structure algorithms repeatedly occur in the Booch class hierarchy. For example, the guarded algorithm is separately reimplemented in the *Guarded_Bounded_Bag* and *Guarded_Unbounded_Bag* components. Similarly, the synchronized algorithm appears in *Synch_Bounded_Bag* and *Synch_Unbounded_Bag*. The Booch library contains numerous examples of this replication, whereas the P++ library avoids it.

The final measurement compared the number of lines of code in each library (ignoring blank lines and comments). For this measurement, we could obtain data only for the current P++ library, which emulates seven data structure families. This data shows that the Booch library is again four times larger than the P++ library (11067 versus 2760 lines of code). Because it is impossible to accurately predict the size of a P++ library which reimplements all ten Booch data structure families, we have omitted that data from Figure 5.6. Nevertheless, we believe that a full reimplementation of the Booch library using P++ would show a similar reduction in code size (which would be consistent with the other data reported in the figure).

5.2.3 Scalability and generative power

As we discussed in Chapter 2, scalability is an important factor in evaluating the quality and utility of a reuse library. A library which is not scalable will be difficult to extend — adding a new primitive feature to the library could entail doubling the number of components. Clearly, this would be a maintenance nightmare. Conversely, if a library is scalable, then the library can easily accommodate new features. Moreover, a useful library will have generative power: the library's components can be used to generate a large number of unique systems. If a scalable library has generative power, then the small effort required to add a new component to a scalable library will eventually yield a tremendous productivity payoff, because the library will now be able to generate an even larger number of systems.

The data of Figure 5.5 indicate that the P++ library is more scalable than the Booch library, because there is less replication of algorithms in the P++ library. Most classes of the Booch library were written using C++ templates. By instantiating those templates with different values, different data structures can be generated. Our calculations show that the 82 classes belonging to the seven Booch data structure families could generate 169 unique data structure systems. In comparison, the 21 components of the P++ library can be combined to produce 208 system implementations.

Figure 5.6 reveals similar results for the full Booch library and the corresponding P++ library. The 100 Booch classes can generate 206 systems, whereas the 30 P++ components would generate 308 systems. In this case, we found that adding a few (8) more components to the P++ library ultimately yields many (100) more systems that can be generated.[2] This result is a clear example of generative power in action — adding components to the P++ library had a multiplicative effect on the number of generated systems.

Why can the P++ library generate more systems? A careful examination of the Booch library reveals that it fails to implement certain combinations of data structure features. For example, it includes balking deques, balking queues, priority deques, and priority queues, but it omits balking stacks and priority stacks. (The LEAPS production system compiler actually uses balking stacks extensively [Mir90].) All six data structures are semantically valid, but because it is so tedious to exhaustively enumerate all permutations of data structure features, the Booch library only includes those data structures judged to be most common. In contrast, P++ can easily enumerate almost any data structure implementation, by selecting and combining the appropriate components from its library. (The non-reference counting bounded bag example to be discussed in Section 5.2.4 demonstrates the scalability of the P++ library.)

5.2.4 Performance

Though the P++ library is substantially smaller than the Booch library, this benefit has little value if the P++ library performs poorly. Our next measurements compared the execution times of simple benchmark programs which used the P++ and

2. The eight components (along with the corresponding realms) are: *directed_graph* (*dgraph*), *undirected_graph* (*ugraph*), *single_list* (*slist*), *double_list* (*dlist*), *binary_tree* (*btree*), *arbitrary_tree* (*atree*), *hash_map* (*map*), and *shared* (*ds*).

Booch libraries. The goal of these programs was to exercise the operations provided by various generated data structures. Some of these benchmarks performed redundant operations or computed results in a simplistic manner. Our intent was not to find the fastest absolute solutions for these computational problems; rather, we just wanted to compare the performance of corresponding data structures from each library.

We implemented two versions of each benchmark program, one using the Booch library and the other using the P++ library. Because both libraries provided almost identical programming interfaces, both versions of the benchmark programs were very similar. The only differences involved renamed header files, class names, and method names. The close similarity of the two libraries' interfaces permitted us to verify that these libraries were functionally equivalent. That is, by substituting one library for another in a benchmark program, we could verify that the operations implemented by the two libraries were equivalent; if not, the benchmark program would behave differently.

Our first benchmark computed prime numbers using a version of the Sieve of Eratosthenes algorithm. This program first created a bag or set container and inserted numbers from 2 to *n* (where *n* = 5000, 10000, or 12000). Next, the program searched the container to find the smallest number, which always would be prime. It created a second container, which contained all multiples of the prime number just printed that are less than *n*. Finally, the program computed the difference of the first container and the second, thereby removing all multiples of the prime number from the first container. The program repeated this process of removing multiples of the smallest prime in the container until the container was empty.

Figure 5.7 and Figure 5.8 show the source code for both implementations of this benchmark using bounded bags. Notice that the programs are identical

```
#include “bag.c”
#include “svector.c”
#include “bag_b.c”

#ifndef MAX_NUMBER
#define MAX_NUMBER 10000
#endif

typedef Bounded_Bag<int, MAX_NUMBER> Number_Collection;
typedef Bag_Active_Iterator<int> Number_Cursor;

#include <stream.h>

int get_smallest (Number_Collection &coll)
{
  int smallest = MAX_NUMBER;
  for (Number_Cursor c (coll); !c.is_done (); c.next ())
    if (*c.item () < smallest)
      smallest = *c.item ();
  return smallest;
}

main ()
{
  Number_Collection numbers;

  // Populate bag with numbers from 2 to n
  for (int i = 2; i < MAX_NUMBER; ++i)
    numbers.add (i);

  // Print primes and remove its multiples from bag
  while (!numbers.is_empty ())
  {
    int prime = get_smallest (numbers);
    cout << prime << ‘ ‘;

    Number_Collection multiples;
    for (int i = prime; i < MAX_NUMBER; i += prime)
      multiples.add (i);
    numbers.difference (multiples);
  }
  cout << endl;
}
```

Figure 5.7 Source code for Sieve of Eratosthenes using Booch bounded bags.

```
#include “memory-heap.hh”
#include “ds-bounded.hh”
#include “collection-bag.hh”

#ifndef MAX_NUMBER
#define MAX_NUMBER 10000
#endif

typedef bag<int, bounded<MAX_NUMBER> > system;
typedef system::collection Number_Collection;
typedef system::iterator Number_Cursor;

#include <stream.h>

int get_smallest (Number_Collection &coll)
{
  int smallest = MAX_NUMBER;
  for (Number_Cursor c (coll); !c.is_done (); c.next ())
    if (*c.item () < smallest)
      smallest = *c.item ();
  return smallest;
}

main ()
{
  Number_Collection numbers;

  // Populate bag with numbers from 2 to n
  for (int i = 2; i < MAX_NUMBER; ++i)
    numbers.add (i);

  // Print primes and remove its multiples from bag
  while (!numbers.is_empty ())
  {
    int prime = get_smallest (numbers);
    cout << prime << ‘ ‘;

    Number_Collection multiples;
    for (int i = prime; i < MAX_NUMBER; i += prime)
      multiples.add (i);
    numbers.difference (multiples);
  }
  cout << endl;
}
```

Figure 5.8 Source code for Sieve of Eratosthenes using P++ bounded bags.

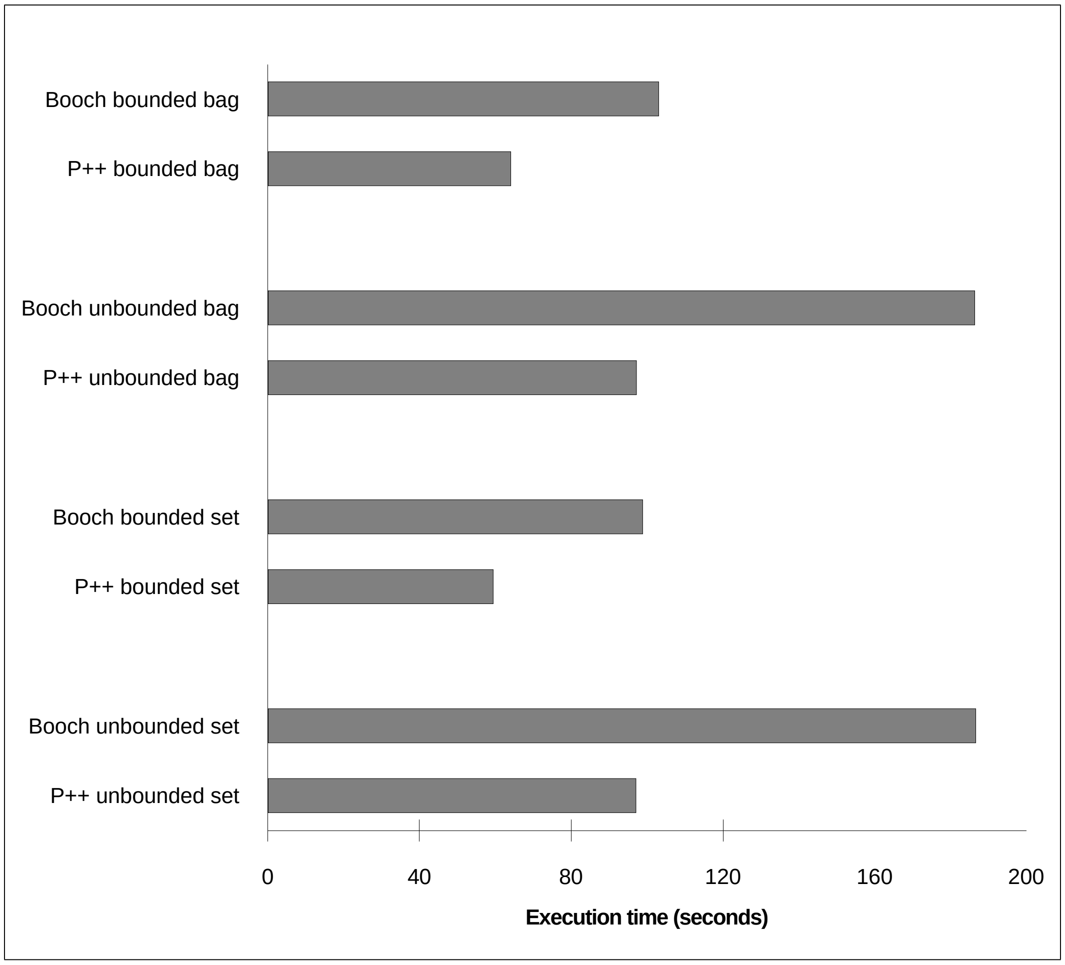


Figure 5.9 The prime number benchmark, for *n* = 10000.

except for the initial declarations of the *Number_Collection* and *Number_Iterator* classes. The other benchmarks were also implemented in this manner: the Booch and P++ data structure algorithms had very similar interfaces and semantics, so switching from one library to another was trivial. Figure 5.9 shows the execution times of the Sieve of Eratosthenes benchmark for *n* = 10000. This graph indicates that the P++ versions consistently ran faster than the corresponding Booch versions. (We omitted the graphs for *n* = 5000 and *n* = 12000 because they revealed the same performance characteristics.)

What accounts for this difference? As we described earlier, the Booch library makes extensive use of inheritance when it implements the classes of a data structure family. In C++, this often entails the use of virtual methods. A virtual method invocation requires a runtime lookup in a dispatch table and an indirect function call. We found that most methods in the Booch library were virtual, but not in the P++ library. Consequently, the Booch versions of the benchmark spent a lot of time performing virtual method lookups, whereas the P++ versions avoided that cost altogether. In addition, because a virtual method call involves a computed jump to a function, most compilers cannot perform inline code optimization of virtual methods. Therefore, by avoiding virtual methods, the P++ library increases the opportunities for the compiler to perform inlining and code rescheduling optimizations, which in turn leads to better performance [Pla95].

The next benchmark program also implemented the Sieve of Eratosthenes algorithm, but this time using ring list containers. The program first inserted numbers from 2 to *n* (where *n* = 5000, 8000, or 10000) into the ring. Next, the program removed the first element in the ring (which always would be the smallest prime) and printed it. It then proceeded to scan the list and remove all multiples of the smallest prime, until it reached the beginning of the list. The process was repeated until the ring was empty. In the bounded ring benchmark, the P++ version was slightly slower than the Booch version. However, in the unbounded ring benchmark, the P++ version was substantially faster.

To understand these performance results (shown in Figure 5.10), we carefully examined the various ring list data structures. We found slight differences in the implementations of these data structures which had significant performance ramifications. In the case of the bounded ring list, we found that the P++ version performed some redundant error checking that the Booch version avoided. The P++ *ring_list* component is parameterized by the *ds* realm; consequently, *ring_list*

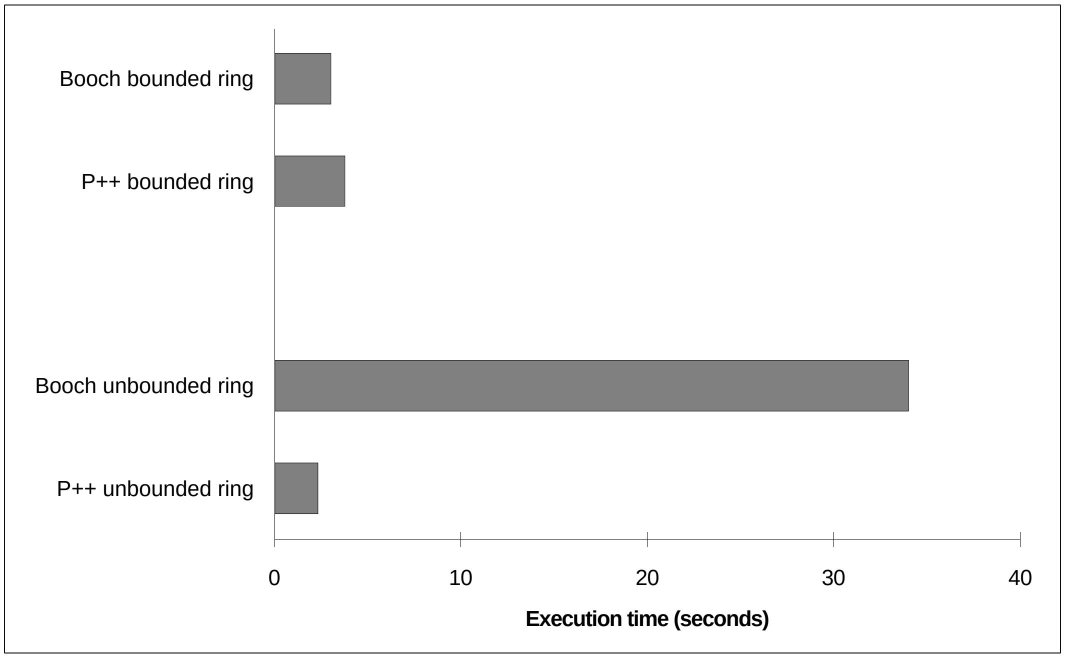


Figure 5.10 The ring list benchmark, for $n = 8000$.

imports the operations of *bounded* to implement the bounded ring list algorithm. Buried within the operations of *bounded* was boundary-case error checking code which necessary only when bounded was used in certain component combinations. Consequently, the P++ bounded ring list suffered a slight performance penalty because the *bounded* component included error checking which was not actually needed when it was combined with the *ring_list* component.

This minor performance discrepancy does not reveal a flaw in P++; rather, it illustrates how crucial it is to design the semantics of realms and components carefully. In this case, the error checking semantics of an operation were overly strict, which resulted in redundant computations. If the operations of *ds* and *ring* had initially been designed slightly differently to anticipate this problem, it might have been avoided.

The Booch version of the unbounded ring list has a different problem which made it much slower than the P++ version. Because a ring list provides

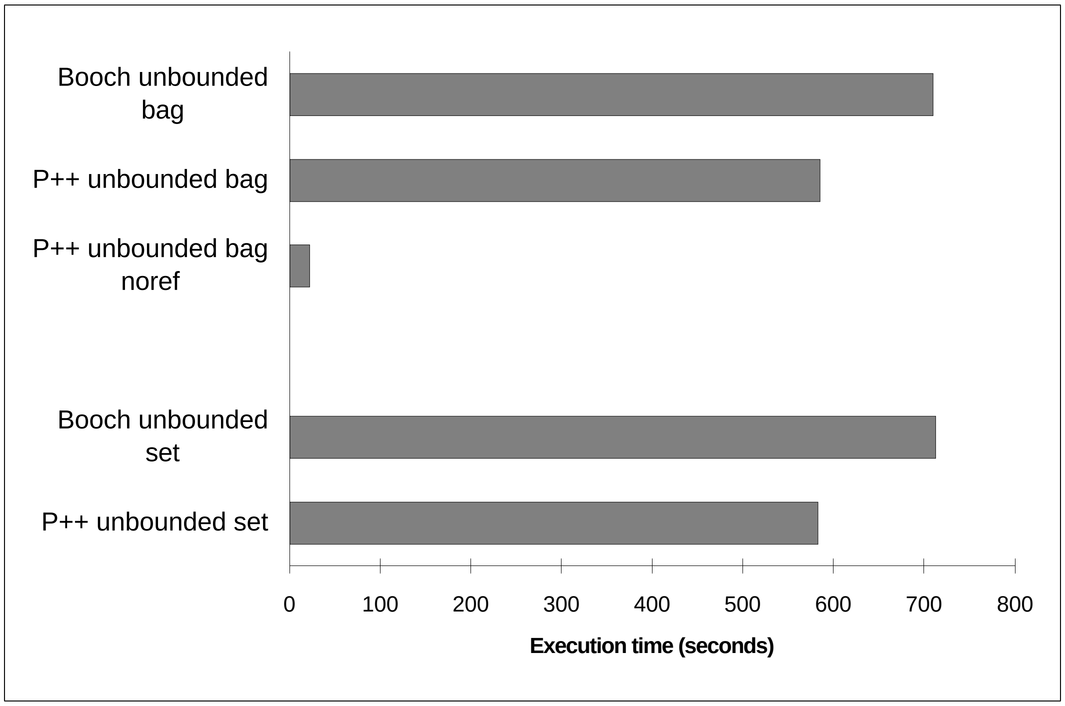


Figure 5.11 The dictionary benchmark, on U.S. Constitution document.

operations to traverse its elements in both directions, this data structure must be able to quickly find the previous and next neighbors of any element. However, the Booch library implements the unbounded algorithm using a singly linked list. As a result, the Booch unbounded ring list is able to quickly move to the next element in the ring, but it is very slow in moving to the previous element. The *unbounded* component of the P++ library uses doubly linked lists to avoid this problem, thus making it much faster.

The final benchmark was a spelling checker program. This program created two bag containers; it loaded one container with 25000 dictionary words and the other container with words from the document to be checked. The program computed the difference of the two container to identify the misspelled words; that is, the program found all words that appeared in the document container but not the dictionary container. The results of Figure 5.11 correspond to spell checking the

U.S. Constitution document; we omitted the graphs for some other documents because those results were essentially identical.

By avoiding virtual dispatch overhead, the graph shows that the P++ bounded bag and unbounded bag were slightly faster than the corresponding Booch data structures. We obtained an interesting result when we tested an extra P++ data structure, called non-reference counting bounded bag. In the normal implementation of bags, both the P++ and Booch libraries use reference counting to eliminate duplicates from the bag container. Thus, when an element is inserted into one of these bags, the bag must check if that element already exists; if so, the reference count is incremented, otherwise the element is added to the bag. This algorithm is obviously very inefficient, especially when there are few duplicate elements in a bag. Consequently, we devised a different version of bag which does not perform reference counting. This version is far more efficient for some operations (like insert), but less efficient for other operations (like difference).

The graph shows the tremendous speedup gained from this optimized version of bag. Implementing this feature was easy, because only one component needed to be added to the P++ library. In contrast, implementing this feature for the Booch library would entail writing many new components. Thus, this benchmark demonstrates the enormous benefit of a scalable library design.

5.3 P++ makes SSG development easier

In existing GenVoca generators (such as Genesis or Avoca), the machinery for representing and combining components constituted a critical portion of the generator's implementation. Because no tool was available to automatically build this infrastructure, each generator had to separately reimplement it. Often, the resulting generator would possess functional limitations because various design compromises were made to expedite the implementation. For example, some generators had an awkward notation to describe a component's implementation; others produced inefficient code for component compositions. These problems would have been avoided if the infrastructure had been automatically created by P++, taking advantage of its succinct notation and automated code generation.

Consider some of the limitations of Genesis. Because Genesis was written in C, there was no built-in language support for GenVoca concepts like realms, components, or parameterization. Instead, each of these features had to be manually emulated in C. The resulting implementation of Genesis was complex to understand and cumbersome to modify. The code for emulating realms, components, etc. was intertwined with the implementations of various database algorithms. Adding a component was tricky, because it involved comprehending and modifying the GenVoca emulation infrastructure. Adding a realm was almost impossible, because the abstractions Genesis used for component interfaces were embedded in the design of the overall system; changing them would be tantamount to rewriting Genesis. If P++ had been used, the design and implementation of Genesis would have been considerably simpler, more extensible, and more maintainable.

Genesis faced other obstacles too. It has a simple but unoptimized mechanism for combining components. It recorded interconnections between components as entries in a dispatch table; the entries of this table consist of pointers to the functions exported by each component. Thus, when a component calls a function defined by some imported component, Genesis translates that function call into a dispatch table lookup and indirect function invocation. The drawback of this approach is that it imposes a runtime overhead cost whenever two components communicate. P++ avoids this overhead by replacing each abstract function call to an imported component with a concrete function invocation, thereby avoiding the cost of a table lookup.

Because P++ offers linguistic support for components, type transformations, and forward type declarations, P++ provides a powerful infrastructure for developing generators. It is difficult to precisely quantify this contribution: the most convincing experiment would be to reimplement several existing generators using P++, to compare the size and performance of the different versions. However, because a typical generator represents tens of thousands of lines of code, such an experiment would be beyond the scope of our work. Instead, we selected an alternate experiment which involved less effort but still provides a solid framework for evaluating the contributions of P++.

5.4 Conclusion

The results presented in this chapter demonstrate that a well-designed software library (which is scalable and has generative power), coupled with P++ language constructs, yields a compact yet versatile library. Not only is the volume of code much smaller in the P++ library, but that library can generate far more systems that the Booch library. At the same time, the systems generated by a properly designed P++ library need not sacrifice performance, as compared to hand-coded C++ data structures.

The conclusion is that P++ successfully responds to the challenge problem posed by Biggerstaff. Language support for GenVoca concepts is a good idea, because it facilitates scalable library implementation and substantially simplifies the task of writing a GenVoca-style software system generator.

Chapter 6

Related work

The P++ language was not only motivated by experiences of building software system generators for various problem domains, but it was also driven by developments in the object-oriented research community. P++ combines ideas from diverse areas of software engineering and programming language research. In this chapter, we discuss several noteworthy areas of research which are relevant to P++. *Frameworks* is an object-oriented software development technique which strives to increase the efficiency of system construction by increasing the scale of encapsulation to subsystems. *Parameterized programming languages* offer special features for writing reusable software modules. Parameterized programming libraries strive to improve the versatility of a software module by using parameters to represent imported algorithms and classes. *Software architectures* addresses the problems inherent in building large systems from components. *Transformation systems* convert high-level specifications into executable programs.

6.1 Frameworks

Frameworks are a popular technique for capturing recurring object-oriented designs in software systems [Deu89, Joh88]. A framework consists of a set of interrelated abstract classes and concrete implementations of those classes (see

Figure 6.1). The abstract classes $A_1, ..., A_n$ define the framework's interface — they are essentially equivalent to a GenVoca realm. For each abstract class, there are one or more concrete classes which inherit from it. To use a framework, a system designer must select a concrete implementation of each abstract class; for example, one possible implementation of the framework of Figure 6.1 would be $C_{11}, C_{12}, ..., C_{1n}$. In GenVoca terms, a component corresponds to a semantically valid tuple of concrete classes from the framework.

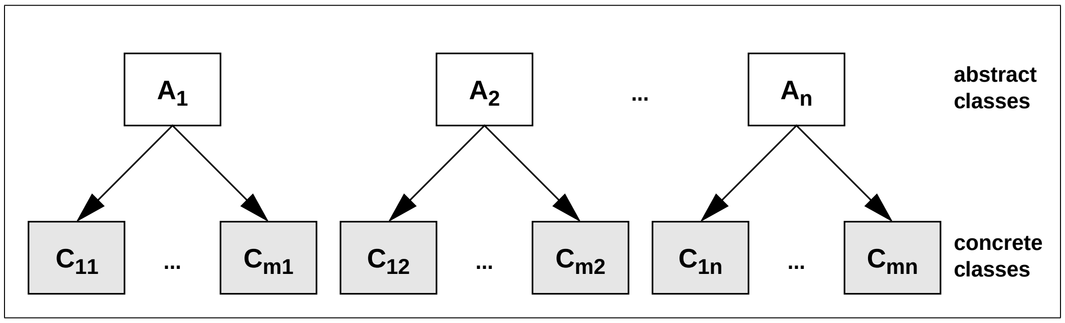


Figure 6.1 Organization of the classes in a framework.

Frameworks are concerned with the reuse of software designs. The concrete classes of a framework form the basic architecture of an application. To complete the application, the system designer provides customized subclasses which implement the desired functionality. There are two benefits to using a framework. First, because the framework captures architectural design decisions for a particular problem domain, the system designer can simply reuse that design and concentrate on the specifics of the application. Second, if frameworks are well-documented and omit implementation-specific details, applications which use a framework may be easy to understand and maintain [Gam95].

Despite the similarities between frameworks and GenVoca realms and components, there are significant differences too. A framework (or a collection of frameworks) does not form a complete system implementation. Instead, the system designer must integrate the framework(s) together with hand-written code to implement the full system. Programming with frameworks still entails comprehending low-level coding details; consequently, the primary benefit of frameworks is the reuse of designs, not the reuse of code. In contrast, GenVoca generators provide large-scale programming constructs which can be easily combined to produce complete system implementations; thus, generators facilitate both design reuse and code reuse.

Frameworks do not provide a modelling notation for specifying which combinations of concrete classes are semantically valid. Instead, the system designer must consult the documentation accompanying the frameworks to determine which permutations of concrete classes are legal. In contrast, a GenVoca component encapsulates a tuple of classes which are designed to work together.

There is no standard mechanism for combining several frameworks in a single application. It is left to the system designer to determine whether the frameworks are compatible, and to supply the “glue” code (if needed) to interconnect the classes of the frameworks. Component combination is much easier in GenVoca because realm parameters provide clear and concise composition model. Frameworks lack the formal structure of realm parameters, which means that combining frameworks is a manual process. (The Choices system is a notable exception; this software system generator represents components as composable frameworks, i.e. frameworks which are designed to be combined. Operating systems can be generated out of framework combinations, with little hand-coding [Cam92]. Unfortunately, frameworks do not offer guidance in the construction of software system generators; instead, typical frameworks focus on building subsystems, not full system implementations.)

Our conclusion is that frameworks impose less structure and formality on the design process. Although a domain-specific framework would certainly aid in the construction of certain software systems, a system designer who chooses to deviate from the anticipated usage of a framework will be forced to write significant portions of the system by hand. We therefore believe that frameworks are impractical for rapidly building large system implementations, for two reasons. First, a typical framework is not intended to provide a full system implementation but only the architectural skeleton, which means that the remainder of the system must be written by hand. Second, without a formal mechanism for combining frameworks, large systems (which might embody several frameworks) will be difficult to develop and maintain. Both limitations imply that frameworks is not presently a design technique which is scalable to large systems.

6.2 Parameterized programming

Parameterized programming is an emerging style of design for writing generic software modules. Some researchers have proposed specialized programming languages which include advanced features to declare and combine parameterized software modules. Other researchers continue to use standard object-oriented languages, but advocate a certain layered software design style for generic modules.

6.2.1 Parameterized programming languages

Parameterized programming languages offer sophisticated module composition features to promote the reuse of designs, specifications, and code. These features can appear as enhancements to conventional languages (as in LIL, the library interconnection language [Gog86]) or object-oriented languages (such as FOOPS [Gog93]). Not only do these languages use the concepts of horizontal and vertical parameterization to facilitate module composition, they also employ special techniques for remapping module interfaces to permit the interconnection of incompatible modules [Tra93].

In the terminology of parameterized programming, a *theory* is a module interface. It places syntactic and semantic restrictions on the values supplied for a module's parameters. A *view* corresponds to the parametric instantiation of a theory — that is, it specifies the binding of values to a module's parameters. In order for a view to be valid, the values must satisfy the restrictions imposed by the theory. A *module expression* represents the composition of several modules; the result may correspond to an executable system (because the parameters of all modules have been bound to legal values) or a parameterized system (which has been partially instantiated but still requires additional values to complete the system).

For example, the GLISP system can produce a specialized version of a generic numerical algorithm by compiling it relative to a particular view [Nov95]. The view specifies the mapping between the abstract data types and parameters used by the algorithm and the concrete data types and values which are available as input.

There are numerous similarities between GenVoca concepts and parameterized programming. In both models, the interfaces of modules are explicitly declared; the building blocks of systems are modules; and vertical and horizontal parameterization is used to support module combination and customization, respectively.

However, the two models differ in significant ways too. The primary distinction arises from the manner in which the language facilities of the two programming models are used. In the case of P++, not only does the language support component abstraction, encapsulation, and parameterization, but it is also accompanied by a GenVoca design methodology [Bat93], which advocates the use of domain-specific, interchangeable, and scalable components. In contrast, parameterized programming languages do not offer guidance in the use of their language constructs, which means that the application developer is wholly responsible for the design and structure of module libraries. For example, suppose the modules in a library do not have standardized interfaces (theories). In that case, the developer will be responsible for remapping module interfaces so that each view satisfies the restrictions of the corresponding theory. Connecting modules in this way can be time-consuming and inefficient

6.2.2 Parameterized programming libraries

The Standard Template Library (STL) is a C++ library which offers several implementations of common container data structures and iterators, such as vectors, deques, lists, sets, maps, and stacks [Mus96]. Also provided are a suite of utility functions which operate upon containers; these functions can be used to compare, search, sort, permute, and merge containers.

What distinguishes STL from other data structures libraries is the application of parameterized programming techniques in the design of the library. Just as type parameters can be used to make a C++ class type independent, parameterized programming makes careful use of template parameters so that a class or function is algorithm independent. Essentially, STL is designed in terms of layered algorithms — each layer is parameterized in terms of its lower layer, and the choice of lower layer determines the behavior and performance characteristics of the upper layer. For example, STL provides sorting functions which are not dependent upon a container's implementation. Instead, the sorting function simply imposes certain restrictions on the interface provided by the container; the overall execution time of the sort function is determined by the kind of container selected and the sort algorithm.

The designers of STL claim that parameterized programming yields several benefits. First, the run-time performance of data structure algorithms is not compromised, even though the algorithms are represented in a generic fashion. Because C++ template parameters are resolved at compile-time, there is no run-time penalty when parameter values are accessed. Second, parameterized programming does not preclude library extensions. The type hierarchy of STL is designed to allow library users to easily integrate their own container classes, iterator classes, and utility functions. User-supplied container classes can be used with existing utility functions, and user-supplied functions can operate upon existing STL containers. Third, the syntactic validity of compositions can be verified at compile-time, using the normal C++ type-checking of templates.

Although parameterized programming offers promise as an effective technique for implementing reusable algorithms with C++ templates, by itself this technique is not a panacea. The authors of STL still had to carefully design the library in order to achieve the previously listed benefits. Ultimately, the success of the library was the result of a good design, not just the application of parameterized programming.

For example, it is difficult to write a fully generic implementation of a sort function. In designing the function, great care must be given to the specific operations that will be used from the container class. A comparison operation and a function for swapping container elements must be selected: these routines must work for all element types, regardless of whether they are user-defined or built-in types. In addition, because some varieties of containers support just a few iterator operations, the sort algorithm must be carefully written to use only the commonly available operations. Further complicating the design task is the lack of an explicit specification syntax for the interfaces imported by a generic function. That is, C++ provides no mechanism for a generic function to declare the interface or semantics that it expects from the imported container class or element type. Consequently, when new container classes or element types are written, there is no specification which indicates the routines that must be available for the sort function to work with those classes or types. In GenVoca, these problems are avoided because a realm parameter explicitly indicates the interface that a component imports; incompatible interfaces are immediately detected, instead of being reported as obscure compile errors.

6.3 Software architectures

Research on *software architectures* focuses on the problems of building large systems from prefabricated components [Gar95, Sha96]. Issues that must be addressed include the overall organization of the constructed system, intercomponent communication protocols, synchronization policies, module composition techniques, and scalability. The approach taken by this research is to identify common software structure paradigms. This facilitates the construction of new systems which reuse the same architecture or structure.

Software architectures typically are domain-independent. They are represented as a graph of interconnected nodes, where each node corresponds to a component and each edge corresponds to a *connector* (e.g. procedural invocation, event broadcast, pipe). The idea is to abstract the details of specific component algorithms and interconnection mechanisms, in order to recognize the underlying structure of a system (or class of systems).

In the GenVoca model and in software architectures, components are the building blocks of systems. However, no particular design methodology is advocated by software architectures, which means that components are not necessarily interchangeable nor interoperable. Instead, component interfaces are ad hoc, which makes it difficult to combine components and check the validity of system compositions.

Thus, the primary contribution of software architectures research is to provide a vocabulary and framework for understanding common software system organization patterns. By itself, this research does not aid in the rapid construction of new systems, because substantial effort is still necessary to interconnect a group of components into a working system.

6.4 Transformation systems

Transformation systems are tools for converting abstract program specifications into concrete program implementations by applying semantics-preserving substitutions [Bax94]. A variety of such systems exist; each has a different suite of available transforms, and a different representation for the input specification and the output implementation.

Draco [Nei89] is a noteworthy transformation system which employs a network of related languages to represent the successive transformations from abstract specification into concrete implementation. The Draco system employs three categories of languages: languages for representing concepts from the problem domain; languages for modelling relevant algorithms and program subsystems; and languages for writing executable programs. The purpose of all these languages is to permit concise specification of a system’s requirements using a notation which is convenient for the given problem. Starting from the problem specification, which is written in the application domain language, the system designer applies a series of transformations which successively converts the specification into a concrete implementation in an executable programming language.

What is the benefit of using Draco? The payoff occurs when the high-level abstractions of the application domain languages and modelling languages are reused [Lei94]. Two challenges must be overcome, however. First, a relevant language for a given application domain may not exist. In this case, the system designer must create the language before any specification can be written. Second, the transformation of programs from one language to another is only partly automated. The system designer must guide Draco in the choice of transformation.

Microsoft’s Intentional Programming (IP) system takes a different approach to transformations [Sim95]. Programs are represented in a canonical format called the “IP source tree”. Programs can be directly written in the IP format using a special IP editor; alternatively, a “parser” tool can be used to automatically translate code from conventional programming languages (like C++) or specialized domain-specific languages into the canonical IP format. “Unparser” tools perform the reverse transformation, converting the canonical representation into a language-specific view of the program.

The goal of IP is to avoid the limitations imposed by developing a system in any particular language. By permitting a developer to switch from one language to another as the system is written, the notational limitations of a language can be avoided; moreover, the developer can take advantage of a language’s relevance to the problem domain. The canonical representation captures the “intent” of a program, without constraining the representation language for the program.

P++ and the transformation systems described here employ a common approach: the use of program transformations to convert high level abstractions into concrete implementations. The difference is that P++ uses a fixed language for representing the specification and implementation, whereas Draco and IP use whatever language happens to be most convenient. The trade-off are the costs of designing domain-specific languages (in Draco), writing tools for converting domain-specific languages to/from the canonical format (in IP), and defining transformations which have domain-specific optimizations (in both).

Chapter 7

Conclusions and future directions

This chapter summarizes the primary contributions of our research and reviews the central results of our experimental work. We conclude by discussing a few areas of future research and enhancement to P++: design rule checking, subjective interfaces, dynamic compositions of components, and subsystem aliases.

7.1 Conclusions

Following are the primary contributions of P++ to the design of domain-specific software system generators:

Scalable libraries. Developers will create component libraries and software system generators only if these tools offer substantial leverage in increasing programmer productivity. If a component library is not designed with scalability in mind, then its utility will be limited because it will be capable of generating relatively few systems. Thus the keys to successful domain-specific component libraries is scalability and generative power. P++ offer language features such as realm interfaces and realm parameters, to encourage the use of standardized interfaces and layered abstractions. These in turn promote the development of scalable libraries with generative power.

Forward declarations. In GenVoca, the components of a system are modelled as layered abstractions. Each layer transforms a set of abstract types, functions, and methods into concrete implementations. What distinguishes P++ components from standard C++ parameterized classes is the order in which transformations occur: P++ transformations are both “top-down” and “bottom-up”, whereas C++ templates are only bottom-up. The P++ forward construct, used in conjunction with realm parameters, makes it possible to represent top-down transformations.

Realm parameters. For any software system generator to be successful, it must be easy to combine components. In P++, component combination is modelled as realm parameters, using a syntax similar to that of C++ templates. The result is a representation that is concise and easily understood.

Component encapsulation. The efficient development of large software systems depends on large units of program construction. Reuse libraries should not simply consist of functions or classes, but components which correspond to suites of functions and classes. The P++ component construct increases the scale of encapsulation to subsystems, where each component represents a domain-specific feature.

Experimental results. Our experimental work also yielded several results. The primary focus was to verify that P++ could indeed be used to develop a scalable software library with generative power. Using the challenge problem posed by Biggerstaff, we redesigned the Booch library with a different set of abstractions and components, which collectively implemented the same functionality as the original library. We then performed tests to verify the runtime execution speed of generated systems. We observed that the P++ library was substantially smaller (one-fourth the lines of code of the Booch library), yet actually possessed greater generative power (we could generate a larger family of data structures with one-fourth as many components) and faster (performance was generally better than the Booch library).

7.2 Lessons learned

We learned several valuable lessons in the course of our research. Although each of these ideas has been previously discussed in detail, it is still useful to summarize some of our insights and observations.

In the early stages of our research, it was not obvious that language support was an essential ingredient for the acceptance of the GenVoca model and the creation of GenVoca generators. Only after attempting to write such a generator in C++ did we truly appreciate the difficulty in developing GenVoca generators. Later, we tried to reverse-engineer a P++ component composition using C++. It took several attempts to correctly emulate the complex interactions of a P++ component composition which employed forward declarations. The resulting C++ code was difficult to comprehend and explain, which clearly demonstrated the benefit offered by the concise notation of P++.

The development of the P++ data structures library showed that language support is not sufficient to ensure that component libraries are scalable and efficient. A thorough knowledge of the problem domain was necessary to correctly design realms and components: a minor change to the *ds* realm interface had tremendous ramifications on the performance of generated systems. Even after *ds* and its components was satisfactorily designed, the remaining realms and components still had to be written carefully.

Finally, the implementation of P++ was substantially harder than expected because no usable C++ grammar was readily available. It took a substantial effort to locate and upgrade an existing grammar to make it a suitable foundation for the

P++ compiler. Moreover, because C++ is a large language with a complicated syntax and numerous features, it was a major effort to support all of the existing language features and still implement the P++ features too.

7.3 Future directions

7.3.1 Design rule checking

A basic challenge for component-based generator systems is ensuring the validity of a particular combination of components. Certain component combinations may be syntactically legal (i.e. the interface provided by one component is the same as the interface expected by another), but are semantically invalid. To remedy this problem, a generator should have a mechanism for verifying the semantic compatibility of all components in a generated system.

The semantics of a component determine the situations in which the component may be used. One representation for component semantics is to place logical constraints on the values supplied for component parameters, the presence or absence of other components in a system, or the overall ordering of components in the system. Based on these definitions, a system would be semantically invalid if two components had conflicting constraints.

In the GenVoca model, component compatibility is verified only at the syntactic level. That is, a generator only checks that each pair of components in a system import and export matching interfaces, without determining if those components make the same assumptions about one another. Consider an analogous situation from arithmetic: the division function takes two numeric arguments; if its imported values satisfy that condition, then the function invocation is syntactically legal. However, if the second value happens to be zero, then the function's semantic behavior is undefined. The semantics of the division function dictate that the second value should be non-zero, but this precondition is not conveyed by type restrictions alone.

The GenVoca model and P++ should be extended to expose the semantic properties of components too. *Design rule extensions* to GenVoca have already been proposed in [Bat96a]. In this extended model, components can impose preconditions and prerestrictions on their use. Preconditions are constraints on higher-level components in a system. Prerestrictions are constraints on lower-level components. Stated another way, preconditions specify the situations where a component can be used; prerestrictions define conditions under which component parameters can be legally instantiated.

P++ should be enhanced to permit the representation of conditions and restrictions. This would facilitate the task of *design rule checking* (DRC). DRC involves two phases: first, starting from the top of the component-hierarchy, postconditions are propagated to lower-level components and compared against component preconditions. Second, starting at the bottom of the component hierarchy, postrestrictions are propagated to higher-level components and compared against component parameter prerestrictions [Bat96a].

There are several challenges to supporting DRC in the P++ compiler. First, a representation for conditions and restrictions must be devised. Then, some constraints must be imposed on that representation so that arbitrary logical assertions cannot be written; otherwise, the task of resolving conditions and restrictions would be too difficult, and there would be no concise or consistent notation for representing component properties. To perform the actual DRC, efficient algorithms must be used to propagate conditions and restrictions. If there are too few constraints on the representation of logical properties, then DRC will take too long and it will not scale to large systems. Specific propagation algorithms must also be investigated.

7.3.2 Subjective interfaces

Standardized component interfaces are an important aspect of the GenVoca model. Such interfaces make it possible to define families of interchangeable components, which in turn increases the number of syntactically legal combinations of a software library's components. Thus, standardized interfaces are key to libraries with generative power.

However, despite the desire for interchangeable components, one often finds that some components of a library deviate slightly from the standard interfaces. In order to effectively represent a system feature, a component designer may extend a standard interface by introducing new methods, functions, or types. The purpose is to preserve the semantics of the standard interface, but also supply special functionality which cannot be easily represented by using just the standard interface.

The study of subjective interfaces is relevant to this problem: if a system is composed of several components and one or more of these components have extended interfaces, then what should be the interface of the resulting system? A component interface is *subjective* if it is dependent on the values supplied for its realm parameters.

The principles of subjective interfaces should be applied to GenVoca and P++. Ossher and Harrison [Oss92] have proposed formal semantics for determining the interface of a system which contains components with extended interfaces. They provide a notation for representing interface extensions, along with rules for computing the interface resulting from the combination of two or more interfaces.

Using these ideas, GenVoca and P++ could be enhanced to accommodate interface extensions in generated systems. The components of a system are modelled as layers, where each layer can introduce new methods, functions, and types. The extensions of a given layer are propagated to the layer above, such that the extensions appear as part of the upper layer's interface too. This technique is iteratively applied to all layers from bottom to top. The resulting system has an interface corresponding to the outermost component's original interface, along with the extensions contributed by the other components in the system [Bat96b].

7.3.3 Dynamic composition of components

A typical P++ program declares a static component composition which the P++ compiler converts into a generated system. Suppose that the system designer wants to defer the choice of components in the system until run-time. In that case, components would have to be composed *dynamically*; that is, type and function transformations would be computed when the system is instantiated, rather than when it is declared.

Support for this capability would require substantial changes P++. The basic approach would be generate an abstract class declaration for each realm declaration; furthermore, a concrete subclass of this abstract class would be generated for each component in the given realm. A realm parameter would be resolved at compile-time: the parameter would be represented by a pointer to the appropriate generated abstract class, and the parameter value would refer to an instance of the component's concrete subclass.

To instantiate a dynamic system, each component would be individually instantiated (in bottom-up order) and the resulting instance pointer would be supplied as the value for the upper layer's realm parameter. The most difficult problem would be to efficiently represent type transformations. Because the values of type parameters would not be known at compile-time, C++ templates could not be used in the generated code; instead, each component would represent object instances as generic (void) pointers. The primary drawbacks of this approach would be the loss of static type checking of parameter values and the cost of traversing extra pointer indirections.

7.3.4 Subsystem aliases

Several component compositions might repeatedly instantiate a group of components in particular way. For example, suppose the following declarations appeared in a P++ program:

```
typedef r <a <b <c> > > sys1;

typedef s <a <b <c> > > sys2;

typedef t <a <b <c> > > sys3;
```

All three compositions use a similar group of components. A useful extension to P++ would be to support the definition of *subsystem aliases*; that is, a mechanism for naming common groups of components in a system. Subsystem aliases would offer a convenient shorthand for referring to logical portions of a component composition. This feature might work like this:

```
typedef a <b <c> > > subsys;

typedef r <subsys> sys1;

typedef s <subsys> sys2;

typedef t <subsys> sys3;
```

Notice that the new syntax makes explicit the reuse of the *a*, *b*, and *c* components.

Supporting this enhancement would be fairly simple in P++. The caveat is that a subsystem alias captures the values supplied for all component parameters. Thus, if the only difference between two groups of components is the value of a single constant parameter, a subsystem alias could not be used to represent both groups of components.

Appendix: the P++ grammar

Following is a conflict-free BNF grammar for the P++ language (which is suitable for use with the “yacc” parser generator). This grammar is based on the C++ grammar that comes with the CPPP tool [Rei94]. It relies on the context-sensitive scanner that was described in Chapter 4. Most of the P++-specific enhancements appear near the end of the listing.

```
%token  <locval>        LX_ASM LX_AUTO LX_BREAK LX_CASE
%token  <locval>        LX_CATCH LX_CHAR LX_CLASS LX_COMPONENT
                        LX_CONST
%token  <locval>        LX_CONTINUE LX_DEFAULT LX_DELETE LX_DO
%token  <locval>        LX_DOUBLE LX_ENUM LX_EXTERN
%token  <locval>        LX_FLOAT LX_FOR LX_FORWARD LX_FRIEND
                        LX_GOTO
%token  <locval>        LX_INLINE LX_INT LX_LONG
%token  <locval>        LX_NEW LX_OPERATOR LX_PRIVATE LX_PROTECTED
%token  <locval>        LX_PUBLIC LX_REALM LX_REGISTER LX_RETURN
                        LX_SHORT
%token  <locval>        LX_SIGNED LX_STATIC LX_STRUCT
%token  <locval>        LX_SWITCH LX_TEMPLATE LX_THIS LX_THROW
%token  <locval>        LX_TRY LX_TYPEDEF LX_UNION LX_UNSIGNED
%token  <locval>        LX_VIRTUAL LX_VOID LX_VOLATILE LX_WHILE

%token  <locval>        LX_NEW0

%left   <locval>        ‘,’
%right  <locval>        ‘=’ LX_MUL_EQ LX_DIV_EQ LX_MOD_EQ
                        LX_ADD_EQ
                        LX_SUB_EQ LX_LSH_EQ LX_RSH_EQ LX_AND_EQ
                        LX_XOR_EQ LX_IOR_EQ
```

```
%right  <locval>        '?'
%left   <locval>        LX_OR_OR
%left   <locval>        LX_AND_AND
%left   <locval>        '|'
%left   <locval>        '^'
%left   <locval>        '&'
%left   <locval>        LX_EQL LX_NEQ
%left   <locval>        '<' '>' LX_LEQ LX_GEQ
%left   <locval>        LX_LSH LX_RSH
%left   <locval>        '+' '-'
%left   <locval>        '*' '/' '%'
%left   <locval>        LX_DOT_STAR LX_PTS_STAR
%left   <locval>        LX_INCR LX_DECR LX_SIZEOF PREC_UNARY
%left   <locval>        PREC_POSTFIX


%token  <locval>        LX_PTS
%token  <locval>        LX_COLON_COLON
%token  <locval>        LX_ELLIPSES

%token <astval>         LX_ID_TYPE_NAME LX_ID_ENUM_NAME
                        LX_ID_TEMPLATE_NAME
%token <astval>         LX_ID_REALM_NAME LX_ID_COMPONENT_NAME
                        LX_ID_CONSTRUCTOR
%token <strval>         LX_EXTERN_LINKAGE

%token <locval>         LX_END_TEMPLATE LX_QUAL_COLON
                        LX_ABST_LEFTP
%token <locval>         LX_INIT_LEFTP

%token                  IN_QUAL_TYPE IN_QUAL_CLASS IN_QUAL_PTR
                        IN_QUAL_ID
%token                  IN_DECLARATOR IN_CLASS_SPEC IN_ABST_DECL
%token                  IN_PLACEMENT IN_FCT_DEF

%token <astval>         LX_ID LX_INT_VAL LX_ZERO_VAL LX_FLT_VAL
                        LX_STRING_VAL
%token <astval>         LX_CHAR_VAL

%left   <locval>        LX_IF
%left   <locval>        LX_ELSE

%token  <astval>        LX_ID0
%token  <locval>        LX_CLASS0 LX_STRUCT0 LX_UNION0
                        LX_COMPONENT0

%start program
```

```
%type <astval> prog_decls
%type <astval> class_name enum_name template_name
%type <astval> opt_expression expression assignment_expression
%type <astval> conditional_expression
%type <astval> std_expr cast_expression unary_expression
               allocation_expression
%type <astval> new_operator opt_placement new_type_name
               opt_new_declarator
%type <astval> new_declarator new_declarator1 new_declarator2
%type <astval> opt_new_initializer new_initializer_list
               deallocation_expression
%type <astval> delete_operator postfix_expression
               opt_expression_list
%type <astval> expression_list
%type <astval> primary_expression expr_name name simple_name
               simple_expr_name
%type <astval> qualified_name qualified_expr_name literal
%type <astval> declaration decl_specifier opt_decl_specifiers
               decl_specifiers
%type <astval> storage_class_specifier fct_specifier
               type_specifier
%type <astval> simple_type_name elaborated_type_specifier
%type <astval> qualified_type_name qualified_type_name1
%type <astval> complete_class_name qualified_class_name
               qualified_class_name1
%type <astval> pointer_class_prefix
%type <astval> qualified_class_prefix enum_specifier
               opt_identifier
%type <astval> opt_enum_list enum_list enumerator
%type <astval> constant_expression linkage_specification
%type <astval> opt_link_declaration_list
%type <astval> link_declaration_list asm_declaration
%type <astval> opt_declarator_list declarator_list init_declarator
%type <astval> opt_initializer declarator declarator1
               opt_ptr_operator
%type <astval> ptr_operator opt_cv_qualifier_list cv_qualifier
%type <astval> opt_cvt_qualifier_list dname
%type <astval> type_name type_specifier_list
               opt_abstract_declarator
%type <astval> abstract_declarator opt_abst_declarator
               abst_declarator
%type <astval> abst_declarator1 abst_declarator3 abst_declarator2
%type <astval> argument_declaration_set argument_declaration_list
%type <astval> opt_arg_declaration_list arg_declaration_list
%type <astval> argument_declaration function_definition fct_body
%type <astval> initializer
```

```
%type <astval> initializer_list initializer_elt
%type <astval> class_specifier class_head opt_member_list
%type <astval> member_list member_list_elt member_declaration
%type <astval> opt_member_declarator_list
%type <astval> member_declarator_list member_declarator
%type <astval> opt_base_spec base_list base_specifier
%type <astval> virtual_specifier access_specifier
               conversion_function_name
%type <astval> conversion_type_name opt_ctor_initializer
               mem_initializer_list
%type <astval> mem_initializer operator_function_name operator
%type <astval> statement labeled_statement exprdecl_statement
               exprdecl
%type <astval> compound_statement opt_statement_list
               statement_list
%type <astval> selection_statement iteration_statement
               for_init_statement
%type <astval> jump_statement
%type <astval> template_declaration template_argument_list
               template_argument
%type <astval> template_class_name template_arg_list template_arg
%type <astval> try_block handler_list handler
               exception_declaration
%type <astval> throw_expression exception_specification
               opt_type_list
%type <astval> type_list
%type <astval> realm_declaration realm_head realm_name
               opt_realm_base_list
%type <astval> realm_base_list realm_base_specifier
               opt_realm_member_list
%type <astval> realm_member_list realm_member_elt
%type <astval> component_declaration component_head component_name
%type <astval> opt_component_base_list component_base_list
%type <astval> component_base_specifier opt_component_member_list
%type <astval> component_member_list component_member_elt
%type <astval> template_component_name
%type <astval> system_declaration system_decl_specifier
               system_decl
%type <astval> system_arg_list system_arg system_name

%type <locval> opt_comma

%token <locval>         ‘;’ ‘{‘ ‘}’ ‘(‘ ‘)’ ‘:’ ‘~’ ‘!’

%%

program :       prog_decls
```

```
        ;


prog_decls :    /* empty */
        |       prog_decls declaration
        |       prog_decls error
        ;



/*****************************************************************/
/* r.17.1 Keywords                                               */
/*****************************************************************/

class_name :    LX_ID_TYPE_NAME
        |       template_class_name
        ;


enum_name  :    LX_ID_ENUM_NAME
        ;


template_name : LX_ID_TEMPLATE_NAME
        ;



/*****************************************************************/
/* r.17.2 Expressions                                            */
/*****************************************************************/

opt_expression :
                /* empty */
        |       expression
        ;


expression :    assignment_expression
        |       expression ‘,’ assignment_expression
        ;


assignment_expression :
                conditional_expression
        |       conditional_expression ‘=’ assignment_expression
        |       conditional_expression LX_MUL_EQ
```

```
                    assignment_expression
        |       conditional_expression LX_DIV_EQ
                    assignment_expression
        |       conditional_expression LX_MOD_EQ
                    assignment_expression
        |       conditional_expression LX_ADD_EQ
                    assignment_expression
        |       conditional_expression LX_SUB_EQ
                    assignment_expression
        |       conditional_expression LX_RSH_EQ
                    assignment_expression
        |       conditional_expression LX_LSH_EQ
                    assignment_expression
        |       conditional_expression LX_AND_EQ
                    assignment_expression
        |       conditional_expression LX_XOR_EQ
                    assignment_expression
        |       conditional_expression LX_IOR_EQ
                    assignment_expression
        ;


conditional_expression :
                std_expr
        |       std_expr ‘?’ expression ‘:’ conditional_expression
        ;


std_expr :      std_expr LX_OR_OR std_expr
        |       std_expr LX_AND_AND std_expr
        |       std_expr ‘|’ std_expr
        |       std_expr ‘^’ std_expr
        |       std_expr ‘&’ std_expr
        |       std_expr LX_EQL std_expr
        |       std_expr LX_NEQ std_expr
        |       std_expr ‘<‘ std_expr
        |       std_expr ‘>’ std_expr
        |       std_expr LX_LEQ std_expr
        |       std_expr LX_GEQ std_expr
        |       std_expr LX_LSH std_expr
        |       std_expr LX_RSH std_expr
        |       std_expr ‘+’ std_expr
        |       std_expr ‘-’ std_expr
        |       std_expr ‘*’ std_expr
        |       std_expr ‘/’ std_expr
        |       std_expr ‘%’ std_expr
        |       std_expr LX_DOT_STAR std_expr
        |       std_expr LX_PTS_STAR std_expr
        |       cast_expression
        ;


cast_expression :
                unary_expression
        |       ‘(‘ type_name ‘)’ cast_expression
        ;


unary_expression :
                postfix_expression
        |       LX_INCR unary_expression
        |       LX_DECR unary_expression
        |       ‘*’ cast_expression %prec PREC_UNARY
        |       ‘&’ cast_expression %prec PREC_UNARY
        |       ‘+’ cast_expression %prec PREC_UNARY
        |       ‘-’ cast_expression %prec PREC_UNARY
        |       ‘!’ cast_expression %prec PREC_UNARY
        |       ‘~’ cast_expression %prec PREC_UNARY
        |       LX_SIZEOF unary_expression
        |       LX_SIZEOF ‘(‘ type_name ‘)’
        |       allocation_expression
        |       deallocation_expression
        |       throw_expression
        ;


allocation_expression :
                new_operator opt_placement new_type_name
                    opt_new_initializer %prec PREC_UNARY
        ;


new_operator :  LX_NEW
        |       LX_COLON_COLON LX_NEW
        ;


opt_placement : /* empty */
        |       IN_PLACEMENT ‘(‘ expression_list ‘)’
        ;


new_type_name : type_specifier_list opt_new_declarator
        |       ‘(‘ type_name ‘)’
        ;


opt_new_declarator :
                /* empty */ %prec PREC_UNARY
        |       new_declarator %prec PREC_UNARY
        ;


new_declarator :
                ‘*’ opt_cv_qualifier_list opt_new_declarator
        |       IN_QUAL_PTR pointer_class_prefix
                    opt_cv_qualifier_list opt_new_declarator
        |       new_declarator1
        ;


new_declarator1 :
                new_declarator2
        |       new_declarator1 ‘[‘ constant_expression ‘]’
        ;


new_declarator2 :
                ‘[‘ expression ‘]’
        ;


opt_new_initializer :
                /* empty */
        |       ‘(‘ ‘)’
        |       ‘(‘ new_initializer_list ‘)’
        ;


new_initializer_list :
                assignment_expression
        |       new_initializer_list ‘,’ assignment_expression
        |       new_initializer_list error
        ;



deallocation_expression :
                delete_operator cast_expression
                    %prec PREC_UNARY
        |       delete_operator ‘[‘ ‘]’ cast_expression
```

```
                    %prec PREC_UNARY
        ;


delete_operator :
                LX_DELETE
        |       LX_COLON_COLON LX_DELETE
        ;


postfix_expression :
                primary_expression
        |       postfix_expression ‘[‘ expression ‘]’
                    %prec PREC_POSTFIX
        |       postfix_expression ‘(‘ opt_expression_list ‘)’
                    %prec PREC_POSTFIX
        |       simple_type_name ‘(‘ opt_expression_list ‘)’
                    %prec PREC_POSTFIX
        |       postfix_expression ‘.’ expr_name
                    %prec PREC_POSTFIX
        |       postfix_expression LX_PTS expr_name
                    %prec PREC_POSTFIX
        |       postfix_expression LX_INCR
                    %prec PREC_POSTFIX
        |       postfix_expression LX_DECR
                    %prec PREC_POSTFIX
        ;


opt_expression_list :
                /* empty */
        |       expression_list
        ;


expression_list :
                assignment_expression
        |       expression_list ‘,’ assignment_expression
        ;


primary_expression :
                literal
        |       LX_THIS
        |       LX_COLON_COLON LX_ID
        |       LX_COLON_COLON operator_function_name
        |       LX_COLON_COLON qualified_expr_name
        |       ‘(‘ expression ‘)’
        |       expr_name
        ;


expr_name :     qualified_expr_name
        |       simple_expr_name
        ;


name    :       simple_name
        |       qualified_name
        ;


simple_name :   LX_ID
        |       operator_function_name
        |       conversion_function_name
        |       ‘~’ class_name
        |       LX_ID_CONSTRUCTOR
        ;


simple_expr_name :
                LX_ID
        |       operator_function_name
        ;


qualified_name :
                IN_QUAL_ID qualified_class_prefix simple_name
        ;


qualified_expr_name :
                IN_QUAL_ID qualified_class_prefix simple_expr_name
        |       IN_QUAL_ID qualified_class_prefix ‘~’ class_name
        ;


literal :       LX_INT_VAL
        |       LX_ZERO_VAL
        |       LX_CHAR_VAL
        |       LX_FLT_VAL
        |       LX_STRING_VAL
        ;
```

```
/****************************************************************/
/* r.17.3 Declarations                                          */
/****************************************************************/

declaration :   opt_decl_specifiers opt_declarator_list ‘;’
        |       asm_declaration
        |       function_definition
        |       template_declaration
        |       linkage_specification
        |       realm_declaration
        |       component_declaration
        |       system_declaration
        ;


decl_specifier :
                storage_class_specifier
        |       type_specifier
        |       fct_specifier
        |       LX_FRIEND
        |       LX_TYPEDEF
        ;


opt_decl_specifiers :
                /* empty */
        |       decl_specifiers
        ;


decl_specifiers :
                decl_specifier
        |       decl_specifiers decl_specifier
        ;


storage_class_specifier :
                LX_AUTO
        |       LX_REGISTER
        |       LX_STATIC
        |       LX_EXTERN
        ;


fct_specifier : LX_INLINE
```

```
        |       LX_VIRTUAL
        ;


type_specifier :
                simple_type_name
        |       class_specifier
        |       enum_specifier
        |       elaborated_type_specifier
        |       cv_qualifier
        ;


simple_type_name :
                complete_class_name
        |       qualified_type_name
        |       LX_CHAR
        |       LX_SHORT
        |       LX_INT
        |       LX_LONG
        |       LX_SIGNED
        |       LX_UNSIGNED
        |       LX_FLOAT
        |       LX_DOUBLE
        |       LX_VOID
        ;


elaborated_type_specifier :
                LX_CLASS LX_ID
        |       LX_CLASS class_name
        |       LX_STRUCT LX_ID
        |       LX_STRUCT class_name
        |       LX_UNION LX_ID
        |       LX_UNION class_name
        |       LX_ENUM enum_name
        ;


qualified_type_name :
                enum_name
        |       IN_QUAL_TYPE class_name LX_QUAL_COLON
                    qualified_type_name1
        ;


qualified_type_name1 :
```

```
                enum_name
        |       class_name LX_QUAL_COLON qualified_type_name1
        ;



complete_class_name :
                qualified_class_name
        |       LX_COLON_COLON qualified_class_name
        ;


qualified_class_name :
                class_name
        |       IN_QUAL_CLASS class_name LX_QUAL_COLON
                    qualified_class_name1
        |       IN_QUAL_CLASS LX_ID_COMPONENT_NAME LX_QUAL_COLON
                    qualified_class_name1
        |       IN_QUAL_CLASS template_component_name
                    LX_QUAL_COLON
                    qualified_class_name1
        |       LX_CLASS IN_QUAL_CLASS class_name LX_QUAL_COLON
                    qualified_class_name1
        |       LX_STRUCT IN_QUAL_CLASS class_name LX_QUAL_COLON
                    qualified_class_name1
        ;


qualified_class_name1 :
                class_name
        |       class_name LX_QUAL_COLON qualified_class_name1
        ;


pointer_class_prefix :
                IN_QUAL_PTR qualified_class_prefix '*'
        |       LX_COLON_COLON IN_QUAL_PTR qualified_class_prefix
                    '*'
        ;


qualified_class_prefix :
                class_name LX_QUAL_COLON
        |       LX_ID_COMPONENT_NAME LX_QUAL_COLON
        |       template_component_name LX_QUAL_COLON
        |       qualified_class_prefix class_name LX_QUAL_COLON
        ;
```

```
enum_specifier :
                LX_ENUM opt_identifier
        ;


opt_identifier :
                /* empty */
        |       LX_ID
        ;


opt_enum_list :
                /* empty */
        |       enum_list opt_comma
        ;


enum_list :     enumerator
        |       enum_list ‘,’ enumerator
        ;


enumerator :    LX_ID
        |       LX_ID ‘=’ constant_expression
        ;


constant_expression :
                conditional_expression
        ;


linkage_specification :
                LX_EXTERN_LINKAGE LX_STRING_VAL ‘{‘
                    opt_link_declaration_list ‘}’
        |       LX_EXTERN_LINKAGE LX_STRING_VAL declaration
        ;


opt_link_declaration_list :
                /* empty */
        |       link_declaration_list
        ;
```

```
link_declaration_list :
                declaration
        |       link_declaration_list declaration
        |       link_declaration_list error
        ;



asm_declaration :
                LX_ASM '(' LX_STRING_VAL ')' ';'
        ;



/*****************************************************************/
/* r.17.4 Declarators                                            */
/*****************************************************************/

opt_declarator_list :
                /* empty */
        |       IN_DECLARATOR declarator_list
        ;


declarator_list :
                init_declarator
        |       declarator_list ',' init_declarator
        ;


init_declarator :
                declarator opt_initializer
        ;


opt_initializer :
                /* empty */
        |       initializer
        ;


declarator :    '*' opt_cv_qualifier_list declarator
        |       '&' opt_cv_qualifier_list declarator
        |       pointer_class_prefix opt_cv_qualifier_list
                    declarator
        |       declarator1
        ;
```

```
declarator1 :   declarator1 argument_declaration_set
                    opt_cvt_qualifier_list
        |       declarator1 ‘[‘ constant_expression ‘]’
        |       declarator1 ‘[‘ ‘]’
        |       ‘(‘ declarator ‘)’
        |       dname
        ;


opt_cv_qualifier_list :
                /* empty */
        |       opt_cv_qualifier_list cv_qualifier
        ;


cv_qualifier :  LX_CONST
        |       LX_VOLATILE
        ;


opt_cvt_qualifier_list :
                /* empty */
        |       opt_cvt_qualifier_list cv_qualifier
        |       opt_cvt_qualifier_list exception_specification
        ;


dname   :       name
        |       qualified_class_name
        ;


type_name :     type_specifier_list opt_abstract_declarator
        ;


type_specifier_list :
                type_specifier
        |       type_specifier_list type_specifier
        ;


opt_abstract_declarator :
                /* empty */
        |       abstract_declarator
        ;


abstract_declarator :
                IN_ABST_DECL abst_declarator
        ;


opt_abst_declarator :
                /* empty */
        |       abst_declarator
        ;


abst_declarator :
                ‘*’ opt_cv_qualifier_list opt_abst_declarator
        |       ‘&’ opt_cv_qualifier_list opt_abst_declarator
        |       pointer_class_prefix opt_cv_qualifier_list
                    opt_abst_declarator
        |       abst_declarator1
        ;


abst_declarator1 :
                abst_declarator2 argument_declaration_set
                    opt_cv_qualifier_list
        |       abst_declarator3
        ;


abst_declarator3 :
                abst_declarator2 ‘[‘ constant_expression ‘]’
        |       abst_declarator2 ‘[‘ ‘]’
        |       abst_declarator3 ‘[‘ constant_expression ‘]’
        ;


abst_declarator2 :
                /* empty */
        |       LX_ABST_LEFTP abst_declarator ‘)’
        ;


argument_declaration_set :
                ‘(‘ argument_declaration_list ‘)’
        ;
```

```
argument_declaration_list :
                opt_arg_declaration_list
        |       opt_arg_declaration_list LX_ELLIPSES
        |       arg_declaration_list ‘,’ LX_ELLIPSES
        ;


opt_arg_declaration_list :
                /* empty */
        |       arg_declaration_list
        ;


arg_declaration_list :
                argument_declaration
        |       arg_declaration_list ‘,’ argument_declaration
        ;


argument_declaration :
                decl_specifiers IN_DECLARATOR declarator
        |       decl_specifiers IN_DECLARATOR declarator ‘=’
                    expression
        |       decl_specifiers opt_abstract_declarator
        |       decl_specifiers opt_abstract_declarator ‘=’
                    expression
        ;


function_definition :
                IN_FCT_DEF opt_decl_specifiers IN_DECLARATOR
                    declarator opt_ctor_initializer fct_body
        ;


fct_body :      ‘{‘ opt_statement_list ‘}’
        ;


initializer :   ‘=’ assignment_expression
        |       ‘=’ ‘{‘ initializer_list opt_comma ‘}’
        |       LX_INIT_LEFTP expression_list ‘)’
        ;


opt_comma :     /* empty */
```

```
        |       ','
        ;


initializer_list :
                initializer_elt
        |       initializer_list ',' initializer_elt
        |       initializer_list error
        ;


initializer_elt :
                assignment_expression
        |       '{' initializer_list opt_comma '}'
        ;



/****************************************************************/
/* r.17.5 Class Declarations                                    */
/****************************************************************/

class_specifier :
                IN_CLASS_SPEC class_head '{' opt_member_list '}'
        ;


class_head :    LX_CLASS opt_identifier opt_base_spec
        |       LX_CLASS class_name opt_base_spec
        |       LX_STRUCT opt_identifier opt_base_spec
        |       LX_STRUCT class_name opt_base_spec
        |       LX_UNION opt_identifier opt_base_spec
        |       LX_UNION class_name opt_base_spec
        ;


opt_member_list :
                /* empty */
        |       member_list
        |       error
        ;


member_list :   member_list_elt
        |       member_list member_list_elt
        |       member_list error
        ;
```

```
member_list_elt :
                member_declaration
        |       LX_PRIVATE ‘:’
        |       LX_PROTECTED ‘:’
        |       LX_PUBLIC ‘:’
        ;


member_declaration :
                opt_decl_specifiers opt_member_declarator_list ‘;’
        |       function_definition
        |       qualified_name ‘;’
        ;


opt_member_declarator_list :
                /* empty */
        |       IN_DECLARATOR member_declarator_list
        ;


member_declarator_list :
                member_declarator
        |       member_declarator_list ‘,’ member_declarator
        ;


member_declarator :
                declarator
        |       declarator ‘=’ LX_ZERO_VAL
        |       ‘:’ constant_expression
        |       LX_ID ‘:’ constant_expression
        ;


opt_base_spec :
                /* empty */
        |       ‘:’ base_list
        ;


base_list :     base_specifier
        |       base_list ‘,’ base_specifier
        ;
```

```
base_specifier :
                complete_class_name
        |       virtual_specifier complete_class_name
        |       virtual_specifier access_specifier
                    complete_class_name
        |       access_specifier complete_class_name
        |       access_specifier virtual_specifier
                    complete_class_name
        ;


virtual_specifier :
                 LX_VIRTUAL
        ;


access_specifier :
                LX_PRIVATE
        |       LX_PROTECTED
        |       LX_PUBLIC
        ;


conversion_function_name :
                LX_OPERATOR conversion_type_name
        ;


conversion_type_name :
                type_specifier_list opt_ptr_operator
        ;


opt_ptr_operator :
                /* empty */
        |       ptr_operator
        ;


ptr_operator :  ‘*’ opt_cv_qualifier_list
        |       ‘&’ opt_cv_qualifier_list
        |       pointer_class_prefix opt_cv_qualifier_list
        ;


opt_ctor_initializer :
```

```
                /* empty */
        |       ‘:’ mem_initializer_list
        |       ‘:’ error
        ;


mem_initializer_list :
                mem_initializer
        |       mem_initializer_list ‘,’ mem_initializer
        ;


mem_initializer :
                complete_class_name ‘(‘ opt_expression_list ‘)’
        |       LX_ID ‘(‘ opt_expression_list ‘)’
        ;


operator_function_name :
                LX_OPERATOR operator
        ;


operator :      LX_NEW0
        |       LX_DELETE
        |       ‘+’
        |       ‘-’
        |       ‘*’
        |       ‘/’
        |       ‘%’
        |       ‘^’
        |       ‘&’
        |       ‘|’
        |       ‘~’
        |       ‘!’
        |       ‘=’
        |       ‘<‘
        |       ‘>’
        |       LX_ADD_EQ
        |       LX_SUB_EQ
        |       LX_MUL_EQ
        |       LX_DIV_EQ
        |       LX_MOD_EQ
        |       LX_XOR_EQ
        |       LX_AND_EQ
        |       LX_IOR_EQ
        |       LX_LSH
        |       LX_RSH
        |       LX_LSH_EQ
        |       LX_RSH_EQ
        |       LX_EQL
        |       LX_NEQ
        |       LX_LEQ
        |       LX_GEQ
        |       LX_AND_AND
        |       LX_OR_OR
        |       LX_INCR
        |       LX_DECR
        |       ‘,’
        |       LX_PTS_STAR
        |       LX_PTS
        |       ‘(‘ ‘)’
        |       ‘[‘ ‘]’
        |       LX_DOT_STAR
        ;


/****************************************************************/
/* r.17.6 Statements                                            */
/****************************************************************/

statement :     labeled_statement
        |       exprdecl_statement
        |       compound_statement
        |       selection_statement
        |       iteration_statement
        |       jump_statement
        |       try_block
        |       error
        ;


labeled_statement :
                LX_ID ‘:’ statement
        |       LX_CASE constant_expression ‘:’ statement
        |       LX_DEFAULT ‘:’ statement
        ;


exprdecl_statement :
                exprdecl
        ;
```

```
exprdecl :      expression ‘;’
        |       declaration
        ;


compound_statement :
                ‘{‘ opt_statement_list ‘}’
        ;


opt_statement_list :
                /* empty */
        |       statement_list
        ;


statement_list :
                statement
        |       statement_list statement
        ;


selection_statement :
                LX_IF ‘(‘ expression ‘)’ statement
                    %prec LX_IF
        |       LX_IF ‘(‘ expression ‘)’ statement LX_ELSE
                    statement %prec LX_ELSE
        |       LX_SWITCH ‘(‘ expression ‘)’ statement
        ;


iteration_statement :
                LX_WHILE ‘(‘ expression ‘)’ statement
        |       LX_DO statement LX_WHILE ‘(‘ expression ‘)’ ‘;’
        |       LX_FOR ‘(‘ for_init_statement opt_expression ‘;’
                    opt_expression ‘)’ statement
        ;


for_init_statement :
                exprdecl_statement
        ;


jump_statement :
                LX_BREAK ‘;’
        |       LX_CONTINUE ‘;’
        |       LX_RETURN ‘;’
        |       LX_RETURN expression ‘;’
        |       LX_GOTO LX_ID ‘;’
        ;


/*****************************************************************/
/* r.17.8 Templates                                              */
/*****************************************************************/

template_declaration :
                LX_TEMPLATE ‘<‘ template_argument_list
                    LX_END_TEMPLATE declaration
        ;


template_argument_list :
                template_argument
        |       template_argument_list ‘,’ template_argument
        |       error
        ;


template_argument :
                argument_declaration
        |       LX_ID_REALM_NAME LX_ID
        |       LX_ID_REALM_NAME ‘<‘ template_arg_list
                    LX_END_TEMPLATE LX_ID
        ;


template_class_name :
                template_name ‘<‘ template_arg_list
                    LX_END_TEMPLATE
        ;


template_arg_list :
                template_arg
        |       template_arg_list ‘,’ template_arg
        ;


template_arg :  assignment_expression
        |       type_name
        |       LX_ID_COMPONENT_NAME
        |       LX_FORWARD LX_ID
        |       LX_FORWARD type_name
        ;



/****************************************************************/
/* r.17.9 Exception Handling                                    */
/****************************************************************/

try_block :     LX_TRY compound_statement handler_list
        ;


handler_list :  handler
        |       handler_list handler
        ;


handler :       LX_CATCH ‘(‘ exception_declaration ‘)’
                    compound_statement
        ;


exception_declaration :
                type_specifier_list IN_DECLARATOR declarator
        |       type_specifier_list abstract_declarator
        |       type_specifier_list
        |       LX_ELLIPSES
        ;


throw_expression :
                LX_THROW %prec PREC_UNARY
        |       LX_THROW cast_expression %prec PREC_UNARY
        ;


exception_specification :
                LX_THROW ‘(‘ opt_type_list ‘)’
        ;


opt_type_list : /* empty */
        |       type_list
        ;
```

```
type_list :     type_name
        |       type_list ',' type_name
        ;



/**************************************************************/
/* Realms                                                     */
/**************************************************************/

realm_declaration :
                realm_head '{' opt_realm_member_list '}' ';'
        ;

realm_head :    LX_REALM realm_name opt_realm_base_list
        ;

realm_name :    LX_ID
        |       LX_ID_REALM_NAME
                    /* invalid (just for error handling) */
        ;

opt_realm_base_list :
                /* empty */
        |       ':' realm_base_list
        ;

realm_base_list :
                realm_base_specifier
        |       realm_base_list ',' realm_base_specifier
        ;

realm_base_specifier :
                LX_ID_REALM_NAME
        |       LX_ID
                    /* invalid (just for error handling) */
        |       LX_ID_TYPE_NAME
                    /* invalid (just for error handling) */
        |       LX_ID_ENUM_NAME
                    /* invalid (just for error handling) */
        ;

opt_realm_member_list :
                /* empty */
        |       realm_member_list
        ;

realm_member_list :
                realm_member_elt
        |       realm_member_list realm_member_elt
        |       realm_member_list error
        ;

realm_member_elt :
                opt_decl_specifiers opt_member_declarator_list ‘;’
        ;



/*****************************************************************/
/* Components                                                    */
/*****************************************************************/

component_declaration :
                IN_CLASS_SPEC component_head
                ‘{‘ opt_component_member_list ‘}’ ‘;’
        ;

component_head :
                LX_COMPONENT component_name
                    opt_component_base_list
        ;

component_name :
                LX_ID
        |       LX_ID_COMPONENT_NAME
                    /* invalid (just for error handling) */
        ;

opt_component_base_list :
                /* empty */
                    /* invalid (just for error handling) */
        |       ‘:’ component_base_list
        ;

component_base_list :
                component_base_specifier
        |       component_base_list ‘,’ component_base_specifier
                    /* invalid */
        ;

component_base_specifier :
```

```
                LX_ID_REALM_NAME
        |       LX_ID_REALM_NAME ‘<‘ template_arg_list
                    LX_END_TEMPLATE
        |       LX_ID
                    /* invalid (just for error handling) */
        |       LX_ID_TYPE_NAME
                    /* invalid (just for error handling) */
        |       LX_ID_ENUM_NAME
                    /* invalid (just for error handling) */
        ;

opt_component_member_list :
                /* empty */
        |       component_member_list
        ;

component_member_list :
                component_member_elt
        |       component_member_list component_member_elt
        |       component_member_list error
        ;

component_member_elt :
                member_declaration
        |       LX_FORWARD member_declaration
        ;

template_component_name :
                LX_ID_COMPONENT_NAME ‘<‘ template_arg_list
                    LX_END_TEMPLATE
        ;



/*****************************************************************/
/* Systems                                                       */
/*****************************************************************/

system_declaration :
                system_decl_specifier system_name ‘;’
        ;

system_decl_specifier :
                LX_TYPEDEF system_decl
        ;

system_decl :   LX_ID_COMPONENT_NAME
```

```
        |       LX_ID_COMPONENT_NAME ‘<‘ system_arg_list
                    LX_END_TEMPLATE
        ;

system_arg_list :
                system_arg
        |       system_arg_list ‘,’ system_arg
        ;

system_arg :    assignment_expression
        |       type_name
        |       system_decl
        ;

system_name :   LX_ID
        ;
```

Bibliography

[Bar93] John Barnes. Introducting 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.

[Bat93] Don Batory and Lou Coglianese. Techniques for software system synthesis in ADAGE, Technical Report ADAGE-UT-93-05, IBM Owego, New York, August 1993.

[Bat96] Don Batory and Bart Geraci. Validating component compositions in software system generators. In *International Conference on Software Reuse,* 1996.

[Bax94] Ira Baxter. Design (not code!) maintenance. In *Proceedings of Brazilian Software Engineering Conference VIII*, October 1994.

[Big89] Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I*. ACM Press, 1989.

[Big94] Ted Biggerstaff. The library scaling problem and the limits of component reuse. In *Proceeding of the Third International Conference on Reuse*, November 1994.

[Boo87] Grady Booch. *Software components with Ada*, Benjamin / Cummings, Menlo Park, CA, 1987.

[Boo90] Grady Booch and Michael Vilot. The design of the C++ Booch Components. In *ECOOP / OOPSLA Conference Proceedings*, 1990.

[Cam92] R. H. Campbell, N. Islam, and P. Madany. Choices, frameworks and refinement. *Computing Systems*, 5(3), 1992.

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

[Deu89] L. Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 system. In Ted Biggerstaff and Alan Perls, editors, *Software Reusability, Volume II*. Addison-Wesley, Reading, MA, 1989.

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

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*, Addison-Wesley, Reading, MA, 1995.

[Gar95] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering, Volume I*, World Scientific Publishing, 1995.

[Gog86] Joseph Goguen. Reusing and interconnecting software components. *IEEE Computer*, February 1986.

[Gog93] Joseph Goguen and Adolfo Socorro. Module composition for the object paradigm. Technical report, Oxford University Computing Laboratory, August 1993.

[Gor90] Keith Gorlen, Sanford Orlow, and Perry Plexico. *Data Abstraction and Object-Oriented Programming in C++*, John Wiley, New York, 1990.

[Hei90] John S. Heidemann and Gerald J. Popek. An extensible, stackable method of file system development. Technical Report CSD-9000044, University of California, Los Angeles, December 1990.

[Hei91] John S. Heidemann and Gerald J. Popek. A layered approach to file system development. Technical Report CSD-910007, University of California, Los Angeles, March 1991.

[Hut91] Norman Hutchinson and Larry Peterson. The *x*-kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64-76, January 1991.

[Joh88] Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22-35, June 1988.

[Lea88] Doug Lea. libg++, the GNU C++ library. In *Proceedings of the USENIX C++ Conference*, 1988.

[Lei94] Julio Leite, Marcelo Sant'Anna, and Felip de Freitas. Draco-PUC: a technology assembly for domain oriented software development. In *Proceedings of the 3rd International Conference on Software Reuse*, Rio de Janeiro, November 1994,

[McI68] M. D. McIlroy. Mass produced software components. In P. Naur and B. Randell, editors. *Software Engineering: Report on a Conference by the NATO Science Committee*, October 1968.

[Mey88] Bertrand Meyer. *Object-oriented software construction*, Prentice-Hall, 1988.

[Mir90] D. Miranker, D. Brant, B. Lofaso, and D. Gadbois. On the performance of lazy matching in production systems. In *Proceedings of the National Conference on Artificial Intelligence*, 1990.

[Mus96] David Musser and Atul Saini. *STL Tutorial and Reference Guide*, Addison-Wesley, Reading, MA, 1996.

[Nau68] P. Naur and B. Randell, editors. *Software Engineering: Report on a Conference by the NATO Science Committee*, October 1968.

[Nei89] James Neighbors. Draco: a method for engineering reusable software systems. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I*. ACM Press, 1989.

[Nov95] Gordon Novak. Creation of views for reuse of software with different data representations. IEEE Transactions on Software Engineering, 21(12):993-1005, December 1995.

[OMa92] Sean O'Malley and Larry Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110-143, May 1992.

[Oss92] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *Proceedings of OOPSLA '92*, October 1992.

[Pla95] P. J. Plauger. *The Draft Standard C++ Library*, Prentice-Hall, 1995.

[Pri91] R. Prieto-Diaz and G. Arango. *Domain Analysis and Software System Modeling*, IEEE Computer Science Press, 1991.

[Rei94] Steven P Reiss. CPPP 1.61 (software package), Brown University, 1994. Available via anonymous ftp from *ftp.cs.brown.edu:/pub/cppp.tar.Z*.

[Sch90] Robert Scheifler and James Gettys. *X Window System*, Second Edition, Digital Press, 1990.

[Sha96] Mary Shaw and David Garlan. *Software Architecture*, Prentice-Hall, Upper Saddle River, NJ, 1996.

[Sim95] Charles Simonyi. The death of computer languages, the birth of intentional programming. In *Proceedings of the 28th Annual International Seminar on the Teach of Computer Science at the University Level*, University of Newcastle upon Tyne, September 1995.

[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.

[Sta94] Richard M. Stallman. *Using and Porting GCC*, Free Software Foundation, September 1994.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*, Addison-Wesley, Reading, MA, 1994.

[Tof90] Mads Tofte, Robin Milner, Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Tra93] W. Tracz. LILEANNA: a parameterized programming language. In *Proceedings of the 2nd International Workshop on Software Reuse*, IEEE Computer Science Press, March 1993.

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

Vita

Vivek Singhal was born in Detroit, Michigan, on October 26, 1968, the son of Dr. Ram Singhal and Vidyut Singhal. After completing his work at Wichita Collegiate School, Wichita, Kansas, in 1986, he entered the Massachusetts Institute of Technology in Cambridge, Massachusetts. He received the degree of Bachelors of Science from MIT in June 1990. In August 1990 he entered the Graduate School of The University of Texas.

A list of publications follows:

[Bat92a] 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.

[Sin92] Vivek Singhal, Sheetal Kakkad, and Paul Wilson. Texas: an efficient, portable persistent store. In *Persistent Object Systems: Proceedings of the Fifth International Workshop on Persistent Object Systems* (San Miniato, Italy), September 1992, pages 11-33.

[Bat92b] Don Batory, Vivek Singhal, and Jeff Thomas. Database challenge: single schema database management systems. Technical Report TR-92-

47, Department of Computer Sciences, University of Texas at Austin, December 1992.

[Sin93a] Vivek Singhal, Sheetal Kakkad, and Paul Wilson. Texas: good, fast, cheap persistence for C++. In *OOPSLA '92 Addendum to the Proceedings* (Vancouver), *OOPS Messenger,* 4(2):145-147, April 1993.

[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* (Baltimore, MD), May 1993, pages 437-446.

[Sin93b] Vivek Singhal and Don Batory. P++: a language for large-scale reusable software components. In *Proceedings of the 6th Annual Workshop on Software Reuse* (Owego, New York), November 1993.

[Tho93] Jeff Thomas, Don Batory, Vivek Singhal, and Marty Sirkin. A scalable approach to software libraries. In *Proceedings of the 6th Annual Workshop on Software Reuse* (Owego, New York), November 1993.

[Sin93c] Vivek Singhal and Don Batory. P++: a language for software system generators. Technical Report TR-93-16, Department of Computer Sciences, University of Texas at Austin, November 1993.

[Bat93] Don Batory, Vivek Singhal, Jeff Thomas, and Marty Sirkin. Scalable software libraries. In *Proceedings of the ACM SIGSOFT '93 Conference* (Los Angeles), December 1993.

Permanent address: 2827 Tallgrass, Wichita, Kansas 67226.

This dissertation was typed by the author.