Customizing C++ to Improve the Reusability of Class Libraries

Taizo Kojima

Mitsubishi Electric Corporation, Central Research Laboratory
1-1, Tsukaguchi-honmachi 8-chome, Amagasaki, Hyogo, JAPAN
Tel: (06) 497-7144
Email: kojima@sys.crl.melco.co.jp

Akira Sugimoto

Mitsubishi Electric Corporation, Central Research Laboratory
1-1, Tsukaguchi-honmachi 8-chome, Amagasaki, Hyogo, JAPAN
Tel: (06) 497-7144
Email: sugimoto@sys.crl.melco.co.jp

Abstract

OPTEC is a language extension tool for customizing C++ language to improve the usability of specific class libraries. Using OPTEC, a system specific language for non-expert programmers can be built easily by extending C++. The system specific language supports selection of the appropriate class from the class libraries, simplifies the use of the class, and makes it easy to define a subclass from an abstract class. The system-specific language uses libraries as its run-time libraries, however, this language can hide the detailed structure of the libraries from programmers. This article describes how extension and customization of the C++ language is effective for increasing the reusability of a library. It also describes an outline of OPTEC system.

Keywords: class library, reuse, language extension, translator generator, C++

Workshop Goals: Learning; exchange views on practical experiences.

Working Groups: Reuse and OO methods, Tools and environments.

1 Background

Recently, software architectures of large-scale systems have become complicated, and the productivity of the application programmers is decreasing. Customers want systems which are as integrated, user-friendly, reliable and scalable as they can be. As a result, large, complicated software libraries are required to develop large-scale systems. Also, the libraries of large-scale systems are not optimal tools for reducing programming work. Each programmer should be able to understand the framework of the libraries and use them appropriately, so that many programmers can cooperatively implement the high-level facilities of the system.

General purpose object-oriented programming languages are very effective for developing a system library. Using the advanced features of the existing object-oriented languages, such as encapsulation, inheritance, and polymorphism, we can create a well organized library based on classes. However, providing only a class library is not enough to improve software productivity. To use a class library effectively, such as making a subclass from an abstract class, the programmer must have a detailed knowledge of the library source code. To develop a large-scale system requires many programmers, and unfortunately, all programmers may not have adequate knowledge of the library.

2 Position

Although object-oriented languages provide an abstraction mechanism, enhancing the languages is an effective means of simplifying application programs for a specific system. An example is Argus[1]. Argus language was developed by enhancing CLU[2] to design a fault-tolerant distributed system. C++[3] has also been extended by many researchers to solve specific problem domains or create new paradigms. For example, Concurrent C++[4] was developed for concurrent processing, and RTC++[5] for real time systems.

2.1 System-Specific Extension of Object-Oriented Language

System-specific extension of language is effective for increasing the reusability of a library. Because system-specific language uses libraries as its run-time libraries, however, this language can hide the detailed structure of the libraries from programmers. Followings are examples of system-specific extensions of language, which improve the usablity of a library.

- Simplifying the use of class

  To simplify the use of a class library, it is important to minimize the necessity of knowing the usage of the classes. This can be done by introducing new syntax for the use of specific classes.

- Selection of appropriate class

  Selecting an appropriate class from libraries is another difficulty when a lot of classes are provided. Therefore, it would be effective if a language system could support the selection.

- Context sensitive transformation

  Context sensitive transformation is useful for simplifying the programs.

- Overloading the control statement

  In C++, overloading is allowed only for *functions* and *operators*. However, overloading the control statement is useful for simplifying a program[6, 7].

- Optimization using knowledge of the library

  In C++, operator overloading is resolved by looking only at the arguments of one operator. As a result, sometimes the execution efficiency may be lost. The execution efficiency will be improved by using appropriate functions in regard to the pattern.

- Customization of class definition

  In an object-oriented language such as C++, sometimes an abstract class is defined, to be used for the inheritance. When defining a class using inheritance, programmers must have sufficient knowledge of an inherited class. To simplify class definition, introducing a new construct is an effective way of defining the subclasses of a specific class.

2.2 A Language Translator Generator for Extending C++

Frequently, conventional extensions for a strongly typed, object-oriented language have been done by directly modifying the base language system. One drawback of this method, however, is that at the start of a system-specific language design, the specifications of language are not usually clear, and therefore, must be decided experimentally. Furthermore, in a practical language extension, it is necessary to handle several libraries provided for supporting various aspects of a system. Consequently, it is necessary to design a simple method of extending a language.

OPTEC is a language translator generator for extending C++. The purpose of OPTEC is to facilitate the construction of a system-specific language for each system architecture and associated class libraries. OPTEC automatically generates a language translator from a specification of the extension. To simplify specification of the translation, a tree rewriting method is adopted, in which a syntax tree for the source code is converted by transformation rules. The language translator converts a source code, using extended C++, into a non-extended C++ code. The reason for choosing C++ as a base language is that many programmers are familiar with C language; from which C++ was derived.

Figure 1 illustrates the software organization of the OPTEC system. Language extension definitions are described separately with 2 source files. One is for syntax extension definition, and the other is for transformation rules. A translator is constructed as follows: First, the parser generator creates a YACC source with a syntax extension definition and a YACC template file, in which C++ syntax is defined. The parser generator also creates a parser for rule descriptions. Then, transformation rules are processed with default rules, and the C++ program for the tree transformer is generated. During this process, the template pattern and the semantic predicates of the rules are preprocessed by the pattern compiler for making later rule selection efficient.

3 Comparison

The C++ language provides a semantic extension mechanism called overloading, however, overloading of C++ is restricted to *operators* and *functions*. Therefore, an effective way for extending a programming language would be to generalize the overloading. In order to generalize overloading, a feature for handling attributes of the tree, such as adding new attributes to a tree and controlling a scope, should be included in the extension mechanism. Although there have been many macro expansion systems developed[8], most macro expansion methods which use syntactical pattern only, have difficulty in handling semantics in the extensions.

In the language extension prototype system TXL by Cordy[9], variable and type declaration are

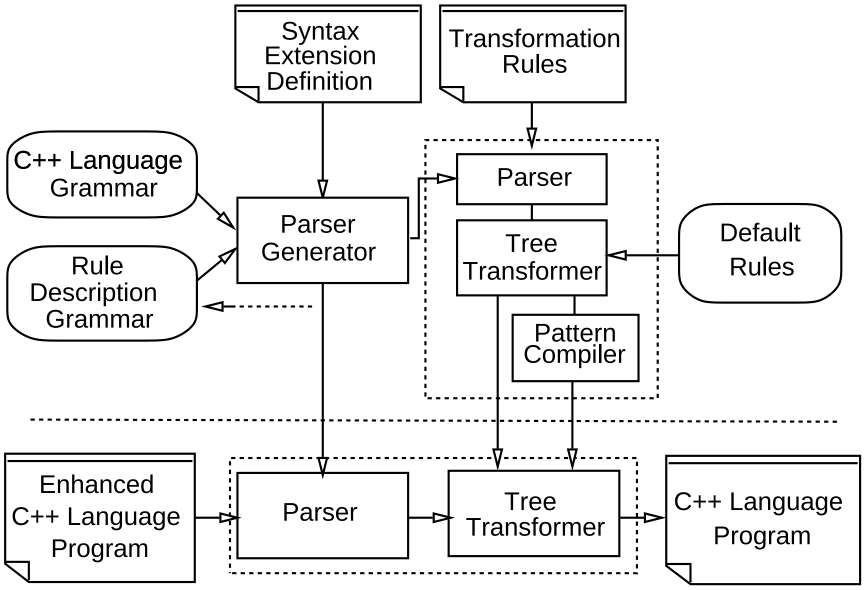

Figure 1: Organization of OPTEC system

handled by the dynamic rule creation, that is, rules that correspond each variable name are created. Although this method is very effective, a complicated rule description is needed and there are problems in the execution efficiency.

Cheatham et al[10] introduced data type to tree rewriting, which is derived from semantic analysis, and used data type as one of the elements that compose the tree pattern. They used this method to refine an abstract program into a concrete program[6]. Cheatham showed that the modularity of rules are improved by specifying the transformation with data types in the examples for the optimization of code and overloading of control statements. However, they did not consider the extension of variable and type declarations. Their system used a semantic analizer for their base language, and once the replacement of the tree occurs, the semantic analysis is performed for that tree again. Therefore, when the variable or the type declaration is inserted, it is necessary to perform the semantic analysis over the entire scope in which the declaration has effect.

On the other hand, the tree rewriting methods have also been studied for building a compiler code generator[11, 12]. Aho et al[11] introduced synthesized attributes to the tree rewriting method. The attributes represent the target machine instruction set, such as the kind of register.

The rewriting method used in OPTEC is basically an extension of Aho's method. In the compiler code generation, variable and type declarations are processed before the code generation phase. However, in the language extension, since the declaration will be inserted by the tree rewriting, it is necessary to do a semantic analysis for variables and types at the rewriting phase. Therefore, in OPTEC, features to handle synthesized and inherited attributes in the attribute grammar[13] are introduced to the transformation rules to permit flexible transformation. In the attribute grammar, variable declaration is propagated as an inherited attribute through the tree in the scope. In OPTEC, to perform transformation efficiently, attributes of variables, such as data types, are stored in a symbol table, and functions for manipulating a symbol table are provided. Furthermore, programming using inherited attributes enables complicated transformation, which is difficult in conventional tree rewriting methods.

References

[1] B. Liskov, "The Argus Language and System," in *Distributed Systems, Lecture Notes No.190*, Springer Verlag, 1984.

[2] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Comunications of ACM*, vol. 20, no. 8, pp. 564–576, 1977.

[3] M. Elis and B. Stroustrup, *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[4] N. Gehani and W. Roome, *The Concurrent C Programming Language.* Silicon Press, 1989.

[5] Y. Ishikawa, H. Tokuda, and C. Mercer, "Real-Time Object-Oriented Language Design: Constructors for Timing Constraints," Tech. Rep. CMU-CS-90-111, CMU, 1990.

[6] T. Cheatham, "Reusability Through Program Transformations," *IEEE Trans. Software Engineering*, vol. SE-10, no. 5, pp. 589–594, 1984.

[7] J. Katzenelson, "Introduction to Enhanced C(EC)," *Softw. Pract. Exper.*, vol. 13, no. 7, pp. 551–576, 1983.

[8] P. Layzell, "The history of macro processors in programming language extensibility," *Computer Jornal*, vol. 28, no. 1, pp. 29–33, 1985.

[9] J. Cordy and E. Promislow, "Specification and Automatic Prototype Implementation of Polymorphic Objects in TURING Using the TXL Dialect Processor," in *Proceedings of IEEE 1990 ICCL*, pp. 280–285, 1990.

[10] T. Cheatham, G. Holloway, and J. Townley, "Program Refinement By Transformation," in *Proceedings of 5th IEEE International Conference on SoftwareEngineering*, pp. 430–437, 1981.

[11] A. Aho and M. Ganapathi, "Efficient Tree Pattern Matching an Aid to Code Generation," in *Proceedings of 12th ACM POPL*, pp. 334–340, 1984.

[12] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1986.

[13] T. Reps, *Generating Language-Based Environments.* The MIT Press, 1984.

Biography

Taizo Kojima has been a researcher at Mitsubishi Electric Corporation's Central Research Laboratory since 1982. His current intrests include object-oriented systems, distributed systems, database systems, and application-oriented programming environment He developed several application systems for practical use. Kojima received a BME from University of Tokyo in 1982.

Akira Sugimoto has been a researcher Mitsubishi Electric Corporation's Central Research Laboratory since 1979. He leads research on object-oriented software engineering, visual programming, and user interface systems. He received a BS and a MS degrees from Kyoto university in 1977 and 1979, and Dr degree from University of Tokyo in 1989.