From LOTOS to C++, Issues and Tool Development

Yoonsik Cheon, Kangho Kim, Chulhong Kim and Youngbae Oh
ETRI Computer & Software Technology Laboratory
161 Kajung-dong, Yusong-ku, Taejon 305-350 Korea
cheon@etri.re.kr

Abstract

Code generation is a technique for automatically generating implementations from formal specifications. In this paper we describe automatic generation of programs in the object-oriented programming language C++ from formal specifications written in LOTOS. We present its theoretical foundation and report on our experience in implementing a support toolset. A key feature of our tool development is the use of idiomatic patterns of system organization and application of formal methods. The application of formal methods was attempted to be rigorous rather than completely formal.

Keywords: code generation, formal methods, LOTOS, software architecture, tool support

1 Introduction

It is understood that formal methods can not be effectively applied to large-scale industrial problems without appropriate tool support. Targeted to the industrial use, we have developed a validation-oriented support toolset for LOTOS, a formal specification language for distributed systems standardized by ISO [5]. A number of validation techniques exist for LOTOS, of which the most popular one is to obtain an executable specification by transforming the original specification, since in general specifications are not executable. For example, let \( L \) be the set of all LOTOS specifications and \( P \) the set of all C++ programs. We can define the semantics of LOTOS by a meaning function \( M : L \rightarrow 2^P \), where \( 2^P \) is the power set of \( P \).

\[
M(l) \overset{\text{def}}{=} \{ p \in P \mid p \text{ satisfies } l \} \quad \text{for all } l \in L
\]

That is, a LOTOS specification denotes the set of all C++ programs that "satisfies" the specification.\(^1\) Now we can view code generation as the process of choosing a C++ program \( c \in M(l) \) for a given specification \( l \). In practice however we should also consider non-functional issues such as performance and cost.

The main purpose of code generation is prototyping and validating specifications. By executing a prototype we can validate specifications in the very early stages of development, which is essential for formal development in practice. If code is generated in an implementation language, we can refine it into a product implementation. We may need to replace a portion of generated code or integrate it with a (more efficient) hand-written implementation. The system described in this paper provides facilities for that.

In this paper we describe a technique for automatically generating C++ programs from LOTOS data type specifications and the development of a support toolset [1]. We first present its theoretical background and then report on our experience in designing and implementing the support toolset. In particular, we discuss in detail such design issues as structure of generated programs, specifying implementation details, hand-coding and interfacing with generated programs. A key feature of our tool development is the use of idiomatic patterns of system organization and application of formal methods. Many of these patterns, sometimes called architectural styles, have been developed over years by researchers around the world. The application of formal methods was attempted to be rigorous rather than completely formal. The tool architecture and module interfaces were partially formalized and some of their properties have been proved. The formalization process increased our confidence on the design.

Several support tools have been developed for LOTOS. A couple of examples, most related to our work, are Caesar and TOPO [4, 6]. Caesar employs a pattern matching technique and heuristics, and is known to be very efficient in generating C programs [4]. In TOPO, specifications are not directly converted into C, but first translated into an intermediate code [6]. The use of intermediate code provides a separation of concerns and improves modularity and extensibility of the system. In fact, our toolset is based on TOPO, partially proving its extensibility.

The most distinguished feature of our toolset is support for C++ and Microsoft Windows ’95. At the time of writing this paper we have not heard of any LOTOS code generation tools that support C++ or Microsoft Windows platforms. We think that for formal methods to be effectively used by industrial practitioners they must speak in their languages on their platforms. In addition to its popularity in industry, C++ has a strong strength as a target language. Its abstraction mechanism allows a smooth and natural translation from LOTOS to C++. As a result, the generated programs are

\(^{1}\)Of course to make it a formal semantics, we have to formalize the notion of satisfies relations.
type Stack is Boolean, Natural
sorts stack
opns
  empty: -> stack
  push: stack, nat -> stack
  pop: stack -> stack
  top: stack -> nat
  isempty: stack -> bool
eqns
  forall s: stack, n: nat
  ofsort nat
  top(push(s,n)) = n;
  ofsort stack
  pop(push(s,n)) = s;
  pop(empty) = empty;
  ofsort bool
  isempty(empty) = true;
  isempty(push(s,n)) = false;
endtype

Figure 1: An example of LOTOS data type specification

modular, readable and comprehensible. The object-oriented
concepts such as inheritance provides an excellent mechanism for reusing, refining and tailoring automatically generated
programs.

2 Theoretical Background

LOTOS specifies a system by defining the temporal relation
among the interactions that constitute the externally observable behavior of the system [5]. When an interaction occurs,
data values can be exchanged as the result of the interaction.
An algebraic specification language ACT ONE is used to
describe the exchanged values [3]. In LOTOS, therefore, a
system is seen as a set of processes which interact and exchange data with each other and with their environment.

A LOTOS data type specification defines a set of data values,
called sorts and a set of total functions on them, called operations (see Figure 1). The meaning of operations is defined
by a set of equations on them. The initial semantics provides a standard interpretation of algebraic specifications
[3], but does not provide operational models. The most concise computing model is the so-called rewriting technique.
The idea is to transform the equations of specification into rewriting rules. The resulting rewrite system can be implemented in a variety of ways to provide an operational model for the specification. The rules can be statically compiled into executable code using pattern-matching techniques or can be used to dynamically rewrite expressions at runtime.

3 Design Issues

- Structure of C++ programs. What should the generated C++ code look like? Basically each sort is translated into a C++ class and an operation is translated into a member function of the corresponding class. For example, the specification in Figure 1 generates the following C++ declaration.

```cpp
cvref<stack> empty();
class Stack: public KSort {
  public:
    Stack(int oid = 0, KLink *p = NULL)
      : KSort(oid, p) {}
    cvref<bool> isempty();
    cvref<nat> top();
    cvref<stack> pop();
    cvref<stack> push(const cvref<nat>&);
};
```

The automatically generated class Stack is defined to be a subclass of class KSort, called a kernel class. The kernel class KSort defines properties that are common to all the generated classes. For example, it provides data structures to represent abstract values and functions to manipulate them. The template class CVRef implements smart pointers to manage C++ heap storage. It automates the management of dynamic memories in the generated code.

- From abstract to concrete. How to represent abstract values and operations in C++? Code generation — programming in general — is the process of refining abstractions into concrete executables. The refinement occurs in two aspects. We must represent abstract values as concrete values using C++ data structures and implement operations in C++ functions to manipulate these concrete values. In algebraic specification languages such as LOTOS, abstract values are denoted by operation applications, called value expressions or terms. For example, `push(push(empty,0),0)` is a value of sort stack. In the technical terms it denote an initial value [3]. To represent abstract values in C++, we use the abstract syntax tree representation [1].

- Specifying details — annotations. The generated code implements a rewrite system specifically tailored to a given specification. It provides an early prototype, but often it is inappropriate to become a product implementation (e.g., for performance reason). The lexical conventions and syntactic rules of C++ are different from those of LOTOS. A natural question that arise then is “how to specify these implementation details and decisions?” We examined several possibilities and adopted the most popular approach — annotations. Annotation is a directive to the code generation tool and is implemented as a special class of comments2. This means that it is transparent to other support tools. Annotations can be used to modify or extend the functionality and to improve the performance of the generated code.

2 It is the convention of LOTOS support tools that annotation has the form (* | *) though the kind of annotations varies from tools to tools.
About 20 kinds of annotations are provided for specifying from syntactic issues like lexical conventions to semantics such as partial implementations.

- **Hand-coding and interfacing with external programs.** It is often the case that a great benefit is obtained by implementing specifications by hand, say, to improve performance. For example, sometimes it would be a burden to write equations for operations, while it is rather simple to directly implement them in C++. In this case, it is necessary to coordinate the hand-coded part with the interfaces expected by the automatically generated part. For this, a kernel class USort is provided, specifying the protocol between hand-coded implementations and automatically generated programs. Each hand-coded class is expected to be a subclass of class USort. The granularity of hand coding can be a function, a class, or a piece of code. A hand-coded implementation can be provided as a separate external file or embedded in the specification. A modular mechanism is provided for change management of code, either automatically generated or hand-coded, and the generated code is not expected to be directly modified. The kernel library also provides a set of APIs through which the generated code can be interconnected with external programs.

4 Tool Development – A Formal Architectural Approach

A key feature of our tool development is the use of idiomatic patterns of system organization and application of formal methods. Many of these patterns, sometimes called architectural styles, have been developed over years. The application of formal methods was attempted to be rigorous rather than completely formal. The architecture and interfaces were partially formalized and some of their properties were proved. The application of formal methods increased our confidence on the design and its implementation.

4.1 Architectural Approach

The basic architecture of our toolset is shown in Figure 2. The system has a layered design that separates the basic functionality of code generation from the graphical user interfaces and other non-essential features. It is comprised of three layers. The lowest layer, called the core layer, performs basic functions required for code generation. It consists of a set of independent components, i.e., stand-alone executables. A lot of these components come from TOPO implementation without major modification [6]. It is possible to upgrade or re-implement a component in this layer without affecting the rest of system. The second later, called the manager layer, is a system coordinator and also provides a communication channel among the components of the core layer and the upper layer. It dynamically rearranges the components in the bottom layer to provide services requested by the upper layer. The highest layer, called the graphical user interface layer, provides a user-friendly graphical user interface and auxiliary functions such as file browsing, source code viewing, error reporting, etc. The layered architecture yields a substantial improvement in the system modularity and flexibility. In addition to the top-level layered architecture, we used a lot of other design patterns such as pipeline architecture, data abstraction and object-oriented organization style, event-based implicit invocation style, and repository style. For example, the components in the core layer are dynamically interconnected in a form of pipeline architecture.

4.2 Formal Approach

A partial formalization of the system architecture was performed on the important aspects of design specifications to increase our confidence that they are correct. In the architectural point of view we needed an assurance that the proposed architecture work. The formalization efforts focused on the behavioral aspect and the dynamic semantics of the architecture. To reuse existing modules and to interconnect existing and newly designed modules together, we needed to be precise about their external interfaces and externally observable behaviors. Thus we also formalized the interfaces of several program modules. Some modules were formally specified at the design stage before actual implementations began, others, in particular TOPO modules were specified afterward as formal documentation of the code. We used a combination of formal specification languages such as LOTOS, Z and Larch. There is no universal specification language best suited for specifying the architecture and behavioral semantics of the system. It seems that the best approach is through a composite approach which uses several techniques suggested so far.

- **Architectural specifications.** The dynamic semantics — for example, dynamic configurations and interactions among the components of the core layer, was formally specified in LOTOS. Some system properties were analyzed and proved rigorously. LOTOS is ideally suited to the specification of systems consisting of a number of distinct components operating concurrently and synchronizing on certain events.
• **Static semantics.** We used Z to specify static semantics and to formalize mathematical concepts. From the TOPO source code we reverse engineered the format of TOPO intermediate code, called LOTOS Data Machine (LDM). We specified both the LDM syntax and its semantics. To develop the LDM-to-C++ translation module, we must be precise and accurate about the LDM. The formalization process helped us a lot to deepen our understanding of LDM and as a side benefit, TOPO implementation itself.

• **Interface specifications.** Using Larch behavioral interface specification languages (BISL) [2], we specified the interface and behavior of some of newly designed and existing program modules. The formal specification made it easy for the engineers constructing and using the modules to communicate with each other, and reduced the effort required to integrate them.

Once formal specifications were composed, if necessary, important system properties could be formulated, rigorously analyzed, or proved formally. As we mentioned before, the application of formal methods was attempted to be rigorous rather than completely formal. The initial use of formal methods was to provide formal specifications for the system architecture, partly replacing and complementing some of the informal design documents. The fact that we formally specified the system deepened our understanding of system properties and increased our confidence that it work as expected. We achieved a large reduction of efforts at the integration stage of development life cycle, as the component interactions have been formally specified and verified within the design cycle.

4.3 Implementation

The system has been implemented in C++ using YACC and supports Microsoft Windows ’95. Most of the core layer modules are DOS mode executable files. The manager layer utilizes the `make` facility to coordinate and interconnect these modules together. The communication and data passing among the modules are performed through pipes and external temporary files. The system has been incorporated into the ForTIA toolset, an integrated support environment for LOTOS, being developed at ETRI. In addition to the code generation facility, the toolset provides a syntax-directed specification editor, a visual simulator and a simple form of specification verifier. Figure 3 shows a snapshot of ForTIA toolset highlighted with the code generation facility.

5 Conclusion

We presented the design issues of generating C++ programs from LOTOS data type specifications and reported on our experience in implementing a support tool. In particular, we discussed several important issues such as formats of generated programs, transitions from abstract to concrete types, annotations and hand coding. The most distinguished feature of our tool development is the use of design patterns and application of formal methods. A GUI-based support toolset has been implemented and currently being evaluated by practitioners in an industrial setting.

References


