A system to port unit generators between audio DSP systems

Karl MacMillan, Michael Droettboom, Ichiro Fujinaga

Peabody Institute of Johns Hopkins University
email: {karlmac, mdboom, ich}@peabody.jhu.edu

Abstract

Musicians currently have a wide variety of choices for audio processing programs. Most of these systems include a mechanism for users to create extensions in a low-level language such as C or C++. These extensions, which are conceptually unit generators and often called plugins, are similar in design but practically different. These practical differences prevent the interchange of unit generators between systems. In this paper, we present RATL, a system for automatically generating unit generators for different systems from the same C++ source code.

1 Introduction

The number and variety of computer applications for audio manipulation available to musicians is larger than ever before. Most of these systems, though powerful and flexible in their standard configuration, include some method for users to create extensions in a low-level language such as C or C++. These extensions, which are conceptually unit generators (UG) and often called plugins, are similar in design but practically different; though they accomplish similar tasks in a consistent manner, the specifics of the application programming interfaces (API) are different.

It is certainly possible for programmers to target multiple audio systems, called hosts, but this is a daunting task. A non-exhaustive list of audio applications that accept UGs includes Csound, RTCmix, Microsoft DirectShow-compatible hosts, jMax, Max/MSP, PureData, SuperCollider, and Steinberg VST-compatible hosts. The time and knowledge required to adapt UGs to all of these hosts makes it impractical to do so, but this lack of interoperability limits users and programmers alike. Additionally, the task of creating a new UG host is complicated by the need to produce a large body of UGs to make the program generally useful.

This paper introduces RATL, a system to automatically port UGs between audio DSP systems. By using a variety of techniques, RATL assists programmers in creating high-performance C++ UGs that can be automatically adapted to a variety of audio systems. Additionally, RATL is an open system that can be extended to support other audio systems. This paper will present an overview of UGs, describe the practical incompatibilities between UGs, present the RATL system, and briefly compare the RATL approach to other UG conversion systems.

2 Unit Generator Overview

UGs are software modules that produce and accept audio and control signals. Max V. Mathews originally introduced them in the Music III language developed at Bell Telephone Laboratories in 1960 (Roads 1999, 89). The UG paradigm provides considerable flexibility for the user; almost any UG can be connected to other UGs, allowing the creation of complex signal- and control-processing networks. Today UGs are often referred to as opcodes, externals, and plugins.

In practice, a UG contains four features: instance data, control input and output, an audio processing function, and host-specific data and functions.

Instance Data. The instance data holds all of the variables and data that are unique to a UG instance, allowing for multiple active instances in an audio system.

Control Input and Output. The control input and output provides a method to pass non-audio data in and out of a UG instance. Systems often accept multiple types of control data and have a corresponding mechanism for type checking.

Audio Processing. The audio processing function is where all of the audio data is accepted, processed, and output.

Host-Specific Functions and Data. The host-specific functions and data register the UG in the host system, describe to the system the types and number of inputs and outputs, and provide a mechanism to initialize UG instances.

3 Unit Generator Incompatibilities

The similarities in UGs are mostly conceptual, while the differences are largely practical. Beyond the obvious host-specific functions and data, the differences between UGs can be found mostly in control data types, the availability of control data output, and differences in audio processing.

Control Data Types. Most systems provide for either integer or floating-point numeric types as control data, with many systems allowing both. In addition, many systems allow other types of data to be passed between UGs.

Control Data Output. Some UG systems do not provide a mechanism for control data output. Generally, systems that are designed to allow the creation of large UG graphs, like Max-style languages, allow for control data outputs while systems that use UGs primarily for the replacement of traditional effects units, like those that use DirectShow plugins, do not (Microsoft 2001).

Audio Processing. UG audio processing functions differ in the sample representation, buffer layout, and buffer access. The different sample representations are important because floating-point and integer math must often be handled in different ways. Most current systems have standardized on single- or double-precision floating-point, though integer formats of varying precisions and compressed formats also exist. The buffer layout differs between single-sample processing, non-interleaved buffers, and interleaved buffers. Within systems that process buffers (blocks) of samples, audio systems either provide separate input and output buffers (out-of-place processing) or unified buffers (in-place processing). Furthermore, writing output to the buffer is done either through simple assignment or through addition with the existing buffer contents.

4 The RATL System

The RATL system is designed to solve the incompatibilities among different systems and provide a convenient method for developing new UGs.

4.1 Overview

The RATL system provides a framework for the creation of UGs with a common subset of the features found in most UG hosts. It simplifies both the process of creating UGs and provides portability between a wide range of UG systems by a combination of advanced C++ features and code generation.

From a programmers perspective, the creation of a RATL UG is as simple as writing a C++ class that follows certain conventions and contains special code comments. In addition to reducing the learning curve for writing RATL UGs, the objects that are created are normal C++ classes, and can be used independently of a UG host.

The subset of UG features in RATL is sufficient to implement a wide variety of UGs while reducing the complexity of the system to a manageable level. The intention is not to support all possible UGs, but to support enough features to provide for a useful interchange of UGs. More specifically, RATL UGs are limited to floating-point format for both control and audio data. Control data output is provided despite the fact that it is not supported in all audio systems. The features of RATL, which address the remaining incompatibilities between hosts, are:

- Conversion of single-sample to block processing.
- Buffer layout conversion.
- Buffer access methods (in- and out-of-place processing and simple and additive assignment).
- Generation of system specific functions, data, and compilation scripts from simple code comments.

4.2 Block Conversion

Many audio-processing systems require their UGs to operate on blocks of samples, however many algorithms are more easily expressed in terms of single samples. RATL automatically converts from single-sample to block processing. A straightforward approach would involve calling the single-sample processing function with within a block-processing loop. This, unfortunately, incurs a function-call overhead with the processing of every sample. To remove the function-call overhead, the C++ inline feature can be used.

To create a natural object-oriented design, we would like to have a base class containing a block-processing method that calls a single-sample function in a subclass. In, C++ the normal method for this would involve inheritance with virtual functions. Unfortunately, current C++ compilers cannot inline virtual functions, meaning that the performance of this solution is the same or worse than the straightforward approach using a normal function call.

At the cost of some flexibility, it is possible to have fully inlined functions in a subclass but still provide the clarity of the inheritance-based approach (Veldhuizen 2000) (see Figure 1). Preliminary testing shows this method to be 10–15% faster than the virtual function-based method.

```
template<class SubClass>
class FastBase {
public:
  void process_block(float* out, int n) {
    float* buf = out;
    for (int j = 0; j < n; j++, buf++)
      *buf = as_subclass()->tick(); }
  SubClass* as_subclass() {
    return static_cast<Sublass*>(this); }
};

class FastRand : public FastBase<FastRand> {
public:
  float tick() { return rand(); }
};
```

Figure 1. Fast block conversion.

The conversion process also allows for UGs with an arbitrary number of audio inputs and outputs. For algorithms that require explicit block processing, it is possible to circumvent the automatic block conversion.

4.3 Buffer Layout Conversion

Host systems provide buffers either as interleaved samples or separate buffers for each channel. A simple solution to this incompatibility is to copy data from the host-provided buffers into a standard buffer configuration and then back into the host format after the UG has processed the data. This is less than ideal, however, because of the extra memory and CPU overhead required. Instead, RATL changes how the data is accessed using the generic programming features of C++ and the Standard Template Library (STL) rather than changing the layout of the data itself.

The STL separates the details of the storage of sequences of data (containers) and the access to the elements of these sequences (iterators). This separation into containers and iterators allows the creation of generic algorithms that work on any container that provides standard iterators (Stroustrup 1997, 549–78). This type of generic programming is not new; what makes generic programming in C++ attractive for audio signal processing is the speed and similarity to existing practices. Figure 2 shows the transformation of the block conversion routine in Figure 1 to a generic algorithm. This example shows that iterators are an abstraction of pointers and programmers can treat them as pointers in most circumstances. This provides a familiar interface to programmers and facilitates the use of existing code within RATL (in fact, if the buffers are an array of floats, the iterators are C-style float pointers.) Features of C++ classes allow the creation of iterators that encapsulate a variety of conversion routines. RATL currently includes an iterator designed to access interleaved buffers. This iterator could be passed into a generic function, like the one in Figure 2, allowing an algorithm designed to work on non-interleaved buffers to work on interleaved buffers without layout conversion. In addition, because the specific type of iterators are known at compile-time, their methods can be fully inlined, making this solution as fast as hand coding the interleaved access. This method can be extended to allow a variety of conversions to the buffer layout or sample format.

```
template<class T>
void gen_process_block(T out, const T end) {
  while(out != end)
    *out++ = as_subclass()->tick();
}
```

Figure 2. Standard C function converted to the corresponding generic version.

4.4 Buffer Access Methods

In addition to different buffer layouts, different audio systems have different access methods to the data. In some systems, assignment to the buffers is done with simple assignment while in other systems the assigned value is added (mixed) to the existing contents of the buffer. In RATL, these issues are solved by having two automatically generated block processing functions like those in Figure 3. The `run_replacing` method does simple assignment while the `run_adding` method adds the output to what is in the input.

```
class UgenFilter {
public:
 template<class T>
 void run_replacing(T in, const T end, T out);
 template<class T>
 void run_adding(T in, const T end, T out);
};
```

Figure 3. Example block processing functions from a RATL Ugen.

Furthermore, some systems provide separate input and output buffers (out-of-place processing) while others provide only one buffer for both input and output (in-place processing). For each input and output, a separate iterator is provided as an argument (see Figure 3). Because the UGs are written without the assumption that these iterators refer to unique data, in-place processing can be performed by passing iterators to the same buffer, while out-of-place processing can be done by passing iterators to different buffers. This provides support for all four possible buffer access methods.

4.5 Code Generation

In writing UGs for a variety of systems, one of the most time-consuming tasks is writing the large amount of code for interfacing with the host systems and generating UG instances. This process can be expedited using RATL's automatic code-generation facilities. The code generation has three steps. First, information about the UG is gathered from special comments in the code. Second, this information is used to generate all of the necessary code for each target UG system. Finally, any makefiles or project files that are needed for compilation are created. All parts of this process use a preprocessor based on the Python programming language. Our preprocessor's design was inspired by Tobler (2001).

The RATL Preprocessor. The RATL preprocessor allows the execution of Python code embedded in the C++ source file. It supports three simple operations:

· Substitution – the result of any Python expression enclosed in '`@@`' is inserted into the text.
· Looped substitution – performs substitution over a section of text multiple times using standard Python loops.
· Inline Python – Python code can be written directly in the C++ source. Standard `print` statements will insert text into the output source file.

We have found this minimal set of operations more than adequate to generate complex code.

Unit Generator Parsing. RATL UGs use special comments to declare information about the UG that is then used to generate the system-specific code. Figure 4 shows a typical UG and the comments (in boldface) that provide the necessary information about the number and types of input and output. This method was chosen in favor of parsing the C++ code because it is substantially simpler than parsing the complex syntax of C++.

```
//@ Ugen("RandUgen")
class SomeUgen :public UgenFilter<SomeUgen> {
public:
  //@ inlet("input_one")
  void input_one(float x);
  //@ outlet("output_one")
  void output_one(float x);
  //@ tick(1, 1)
  sample tick(sample in);
};
```

Figure 4. A sample UG declaration.

Output. After the UG has been parsed, all of the target UG types are generated from boilerplate files provided by RATL (see Figure 5). By using the RATL preprocessor, it is simple to convert an existing UG for a specific system into a template from which other UGs for the same system can be generated.

```
//@inline
# a variable for later use
pd_type = ugen.name + "_t"
//@end

// struct to hold UG instance data
typedef struct _@@ugen.name@@ {
  // this is a pd requirement
  t_object x_obj;
  // the c++ object
  @@ugen.name@@* object;
  // the outlets
  //@for x in ugen.outlets:
  t_outlet* x_outlet_@@x@@
  //@end for
} @@pd_type@@;
```

Figure 5a. An example of the PureData RATL template.

4.5.4 Compilation Scripts - In addition to generating the code for each target UG type, makefiles, build scripts, or project files are created for each UG.

```
// struct to hold UG instance data
typedef struct _Rand {
  // this is a pd requirement
  t_object x_obj;
  // the c++ object
  Rand* object;
  // the outlets
  t_outlet* x_outlet_output_one;
} Rand_t;
```

Figure 5b. The result of sending Figure 5a through the RATL preprocessor.

5 Other Approaches

Other approaches exist both for porting UGs between systems and automatically generating UG code.

Run-time adaptors load UGs compiled for one system into another system. Adaptors exist for using VST plugins in DirectShow hosts (Spin Audio: www.spinaudio.com) and the use of VST plugins to Max/MSP objects (Pluggo: www.cycling74.com). These approaches can never have the performance of a fully compiled approach due to the conversion of the data formats and extra function-call overhead.

Code generators exist to help developers write UGs for particular host systems. For example, Nyquist includes a translator that generates a C UG from simple specifications (Dannenberg 2001, 69–75). There is also a DirectShow plugin "wizard" for Microsoft Visual Studio (Twelve Tone 1996). These approaches, while useful for their specific purpose, are not as flexible and portable as our more general approach.

6 Conclusion

Using the RATL system, we have successfully generated UGs for DirectShow, VST, PureData, and Max/MSP from a common code base. RATL has proven to be a convenient tool to maximize developer efficiency, run-time performance, and portability. We intend to extend the system to support the UG models of other host systems in the near future.

Reference

Dannenberg, R. B. 2001. Nyquist reference manual. Version 2.12. Pittsburg, PA: Carnegie Mellon University.

Microsoft. 2001. DirectX 8 Software Development Kit Documentation. Redmond WA: Microsoft Corporation.

Roads, C. 1999. The computer music tutorial. Cambridge, MA: MIT Press.

Steinberg. 1999. Steinberg Virtual Studio Technology (VST) Plug-in Specification. Hamburg: Steinberg Media Technologies AG.

Stroustrup, B. 1997. The C++ programming language. 3rd ed. Reading, MA: Addison-Wesley.

Tobler, R. F. 2001. PYM: A macro preprocessor based on Python. In *Proceedings of the Ninth International Python Conference,* 23–8.

Twelve Tone. 1996. Cakewalk/DirectX Plug-In App Wizard Documentation. Cambridge, MA: Twelve Tone Systems, Inc.

Twelve Tone. 2001. MFX and DXi: MIDI effects filters, DirectX instruments, DirectShow filters. Cambridge, MA: Twelve ToneSystems, Inc.