ON THE USE OF AN ALGEBRAIC LANGUAGE INTERFACE FOR WAVEFORM DEFINITION

Michael Dickens (Graduate Student, University of Notre Dame, IN : mdickens@nd.edu)
J. Nicholas Laneman (Associate Professor, University of Notre Dame, IN : jnl@nd.edu)

ABSTRACT

We discuss implementation aspects of a software-defined radio system that allows the user to define waveforms using an algebraic language interface, currently as an extension to C++. Current software-defined radio systems provide waveform definitions through a combination of a graphical interface, markup language, interpreted script, and compiled code. No matter which methods are used, the actual executed code generates each waveform via a series of graph-style connections: instantiating blocks and then explicitly connecting ports between blocks. We propose a system that allows the user to define waveforms using a novel text-based algebraic language interface similar to that found in MathWorks MATLAB or GNU Octave. Our system simplifies the waveform programming abstraction by using implicit graph-style connections; it makes extensive use of C++ templates and operator overloading to allow this high-level abstraction. Our interface is solely an abstraction layer providing an alternative means for coding waveforms when compared with current techniques, and hence has no more overhead than current techniques. Example code is provided for comparison and contrast of different methods of waveform definition.

1. INTRODUCTION AND MOTIVATION

Developing software-defined radio (SDR) technologies requires knowledge of both hardware and software, drawing from antenna physics to analog and digital signal processing techniques, from operating system (OS) kernel extensions to full graphical user interfaces (GUI) – not to mention knowledge of regulatory policies and the intellectual property scene. In the Open Systems Interconnection seven layer model [1], SDR technologies are involved at all layers – though certain layers are more strongly represented than others. Although SDR developers can specialize in a specific area, many end up spanning multiple disciplines in order to create a more integrated platform. The learning curve for developing and using SDR continues to be discussed [2][3], but the reality is that there are actual and perceived barriers to developing and using SDR technologies.

Current SDR frameworks – the collection of executables and libraries, header, resource, and data files for a given project – provide waveform definition through a combination of one or more of the following: GUIs, markup languages, interpreted scripts, and compiled code. No matter which methods are used, the actual executed code generates each waveform via a series of graph-style connections: instantiating signal processing blocks and then explicitly connecting ports between blocks.

This *block-centric* approach to stream-based signal processing differs from that used by industry standard applications such as MathWorks MATLAB [4] and GNU Octave [5]. Going back to the early 1980's, MATLAB (among related projects) has provided digital signal processing capabilities with a relatively simple learning curve. MATLAB script is written in a *buffer-centric* algebraic-like programming language, which is now being used by millions of end-users – many in academia but plenty in industry as well [6]. By "algebraic-like", we mean a mathematical expression in which only numbers, variables, and arithmetic operations are used. MATLAB scripts can currently work with scalars, arrays, and matrices of varying types, but not with streams beyond splitting stream data into arrays and doing array processing. One goal in this work is to show that MATLAB-style C++ code can be made to work with streams using data-flow techniques.

Block-centric (traditional SDR) and buffer-centric (industry standard) digital signal processing can be used to accomplish the same task, albeit using different language abstractions. For example, in typical Monte Carlo experiments for testing a channel coding model, vectors of random data are generated and then processed using operations in a set order that represent the encoder, channel, and decoder. With each new vector of random data, the simulation converges towards a result; each operation may keep state between random vector iterations. Such simulations are often written first in a MATLAB-style script because it generally provides the fastest development time. If the interpreted script is too slow, then it can be converted into a compiled language.

To perform such a Monte Carlo simulation using MATLAB-style script, an extended loop is used, within which the random data is created and processed; statistics

are kept using variables declared outside of the loop. Such a simulation can also be performed using SDR-style processing techniques, where the random data vectors become frames generated by a random source, and the ordered operations are SDR blocks joined to make a "waveform" of sorts, performing signal processing on those frames. Statistics can be handled either directly inside a decoder sink, or via a callback from the sink. The primary difference between these MATLAB-style and SDR-style scripts is the programming abstraction. One of our goals in this work is to start bridging the gap between these styles, with the hope of making SDR-style programming more accessible, and with a more-familiar learning curve, to programmers who learned signal processing using MATLAB-style programming.

Our *Surfer* SDR framework aims to enhance the user's experience by pushing complexity into the framework's programming [7]. Continuing this trend from the perspective of reducing the user's learning curve for creating script-based waveforms, we augmented *Surfer* to provide an alternative, algebraic-like language interface – using a buffer-centric approach similar to that provided by MATLAB and Octave.

The *Surfer* algebraic language interface (SALINE) is written in C++, as an independent layer that resides in its own C++ namespace, and provides an algebraic-like language interface for waveform definition. SALINE is split into the core user accessible classes, functions, and macros, and an interface to the underlying SDR framework. Although SALINE is designed with *Surfer* in mind, an interface into other SDR frameworks, e.g. GNU Radio [8], could be created relatively easily. SALINE is an abstraction layer that merely acts as an alternative means for coding waveforms when compared with current techniques; it has no more overhead that any other technique. The underlying SDR framework supplies the blocks / components and actual connections methods, and thus the framework is responsible for any heterogeneous processing or specialized instructions such as SSE, Neon, or Altivec.

Both *Surfer* and SALINE make extensive use of C++ templates; for example, all currently implemented *Surfer* blocks are written as templates and thus must be explicitly instantiated by the user's waveform application. By using template classes for all blocks, *Surfer* avoids code redundancy and related bug duplication issues, and also allows for easier debugging of code issues because the source is directly available as a header file. That said, explicit-typed SALINE functions and *Surfer* blocks can be created and utilized through the use of *Surfer*'s signal-processing flavors [7].

We provide relevant background information, in Section 2, on how SDR waveforms are defined in different abstractions. Section 3 presents important C++ concepts needed to understand the SALINE implementation. In

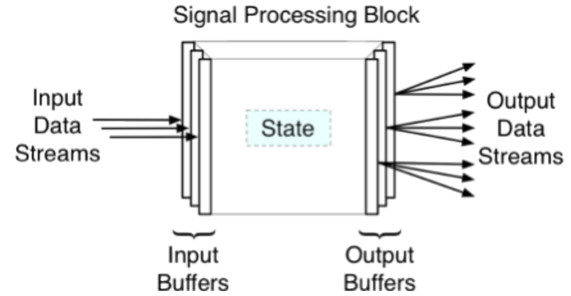

Figure 1 – General diagram of a SDR signal-processing block

Section 4, we describe the SALINE programming implementation, with emphasis on the types of operators required to create its algebraic-like interface as well as how C++ templates can be used to promote type propagation through the waveform graph. Example scripts and C++ code snippets are provided throughout, displayed in the **`Bold Courier`** font in order to help set them apart from the rest of the text. Conclusions, acknowledgements, and references are then provided in Sections 5, 6, and 7, respectively.

2. BACKGROUND

In this section, we discuss the programming abstractions used by current SDR frameworks to define waveforms, in as broad terms as possible. Our goal is to provide enough relevant information such that the implementation described in Section 4 can be compared with the current methods; we are not trying to fully describe how GNU Radio or JTRS SCA [9] defines waveforms, but rather to look at the way waveform definitions take place in a general sense.

Each SDR waveform can be described by an acyclic graph, whether performing packet or frame processing of data. Such a processing abstraction allows for a GUI to describe a given waveform – e.g., the GNU Radio Companion [10], MathWorks Simulink [11], National Instruments LabVIEW [12], and others [13-15] – as well as a text-based definition via a script or compiled program. Although we are considering expanding the SALINE concept into an interpretive script, for this work we are concentrating on compiled script-style programs. We firmly believe that both GUI and script interfaces have their benefits and usage. SALINE is not meant to replace any of the various GUIs; rather, it is meant to augment current scripting methods.

Figure 1 shows a generic signal-processing block, including input and output buffers (also called ports) and data streams, and the block state. Note that there must be exactly one input data stream per input buffer, while there can be more than one output data stream per output buffer. Source blocks provide only outputs, while sink blocks require only inputs. The waveform designer sets an initial state for each block requiring it. In the following

```
output = pp_down_N_block
  (input, N, options)
{
  // declare blocks first
  s2p = serial_to_parallel (N, options)
  for n = 1:N {
   filter[n] = fir_filter (options.ppf[n])
  }
  acc = sum (options)
  // connect blocks second
  connect ((input, 1), (s2p, 1))
  for n = 1:N {
    connect ((s2p, n), (filter[n], 1))
    connect ((filter[n], 1), (acc, n))
  }
  return (acc)
}
```

Listing 1 – Polyphase downsample-by-N written using block-centric programming

subsections, we provide scripts showing different programming abstractions for waveform description.

2.1. Block-centric abstraction

Current SDR frameworks, when defining the waveform graph via scripts or text-based programs, use a block-centric abstraction. In this abstraction, each block is created and then connections are formed between adjacent blocks' output and input ports. Data is manipulated by each block's signal-processing algorithm, taking data from input ports, performing processing, and the writing data into output ports. In a block-centric language, individual buffers are generally hidden from the user's script; the primary user-interface is each block: its instantiated object, input / output ports, and state.

As a simple example of block-centric programming, consider the polyphase downsample-by-N [16] implementation found in Listing 1, written in a script combining features of MATLAB and C++ in order to reduce code complexity while providing the features necessary for comparison and contrasting with other programming abstractions. This listing shows a function named **`pp_down_N_block`** that takes three arguments (**`input`**, **`N`**, and **`options`**) and returns one (**`output`**). The function **`serial_to_parallel`** takes a stream of items and parses them to **`N`** output streams, in order and without duplication. The function **`fir_filter`** creates a finite-impulse response filter using the provided vector as the filter taps.

For the function **`connect`**, the first argument pair always refers to a block and its output port, and the second argument always refers to a block and its input port; this function creates a graph connection between the provided

```
output = pp_down_N_buffer
  (input, N, options)
{
  s2p = serial_to_parallel
    (input, N, options)
  acc = 0
  for n = 1:N {
    // 'acc' reused
    acc += fir_filter (s2p[n],
                       options.ppf[n]))
  }
  return (acc)
}
```

Listing 2 – Polyphase downsample-by-N written using buffer-centric programming

pairs. The **`connect`** function is meant for demonstration purposes only, and can be assumed to be type-agnostic**.** For the sake of simplicity, we assume for this and related listings that array and port indices are 1-based (i.e., start numbering with 1, not 0), and that **`options`** contains the polyphase filter coefficients in the variable **`ppf[n]`** as well as anything else required for instantiation the blocks. For block-centric programming, the variables **`input`**, **`s2p`**, **`filter[n]`**, and **`acc`** refer to instantiated block objects.

In this code listing, blocks are created first and then connected together to form the waveform graph; one could combine the block creation and connection stages, but it does not change the underlying abstraction. Note that with this abstraction, the waveform graph can be connected in any order: from source to sink, sink to source, or from internal blocks towards both the source and sink. No matter the graph ordering, **`connect`** calls are required to form the graph.

2.2. Buffer-centric abstraction

MATLAB and Octave scripts are written in an algebraic-like language using a *buffer-centric* abstraction to define signal-processing algorithms. In this abstraction, data – in the form of scalars, arrays, or matrices – is manipulated by functions in the user's script via data buffers. In a buffer-centric language, objects are exposed in the user's script to the degree that the user and language allow.

Listing 2 provides a buffer-centric script for the polyphase downsample-by-N function. This listing shows a function named **`pp_down_N_buffer`** with the same function arguments and return as **`pp_down_N_block`**, and where the internally used functions have the same purpose. For buffer-centric programming, the variables **`input`**, **`s2p`**, and **`acc`** refer to output buffers from previous operators. The line **`acc = 0`** is shorthand for zeroing out the buffer before it is used; this code is written for clarity, not efficiency. Because connections are defined implicitly through the

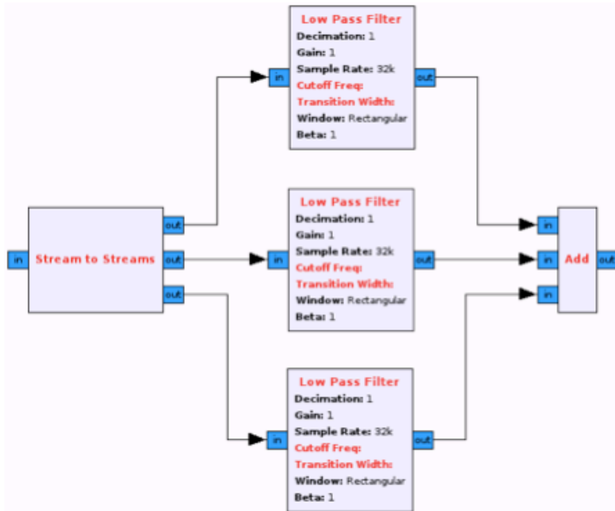

Figure 2 – Rough diagram of the polyphase downsample-by-N created using the GNU Radio Companion, for N = 3

manipulation of buffers, the underlying waveform graph is abstracted away from the user's script and hence no **`connect`** calls are required. The use of the **`+=`** operator provides a more intuitive language interface than that found when using block-centric programming, and also allows for certain runtime graph optimizations – both of which will be discussed further in Section 4.

2.3. Definition via GUI

GUI representations of waveforms are neither entirely buffer- nor block-centric, but rather a mix of the two – though more towards the latter because the user does not directly manipulate the buffers, but rather draws connections between blocks (ports). Each GUI-described waveform must contain both blocks as well as explicit connections between them. Figure 2 shows, approximately, the same polyphase downsample-by-N waveform drawn using GNU Radio Companion (GRC), for N=3. Please note that GRC provides a single block that performs this function; the goal here is to show the abstraction used for GUI waveform creation.

2.4. Other waveform definition methodologies

A variety of other script-oriented waveform or data-flow definition methodologies and languages have been proposed and/or are in use [17-24], the most notable of which in SDR circles is JTRS SCA. For the most part, these frameworks define waveforms (or the equivalent data-flow) through some combination of markup and compiled languages. All of the definition languages we have encountered, when reduced to their essence, use block-centric programming.

3. IMPORTANT C++ FEATURES

SALINE heavily relies on three standard features of C++ that are not available in C or other similar languages: templates, operator overloading, and runtime variable type comparison. These features are covered briefly in the following subsections.

3.1. Templates

Templates are part of the current C++ standard [25] and allow the definition of any instance of a class, method, variable, or function concisely, with minimal redundancy. A template function or class is designed to work on many different data types without being rewritten specifically for each one. For example, let us compare the C and C++ implementations for a **`max`** function, which takes two same-typed arguments and returns the maximum of them. In C the function **`max`** of two same-typed arguments for the types **`int`** and **`float`** could be written

```
int max_i (int a, int b)
{ return (a > b ? a : b); }
float max_f (float a, float b)
{ return (a > b ? a : b); }
```

Note that in C the function names must be unique, as must the input arguments and return type for each function. Clearly, there is significant code redundancy, and to expand these functions to include other types beyond those provided would require more similarly named functions.

C++ provides templates to reduce the code complexity for such functions. The **`max`** function taking two same-typed arguments, as above, can be written generically via a C++ template function as

```
template < typename T >
T max (T a, T b)
{ return (a > b ? a : b); }
```

This template function will be expanded by the compiler into an explicit-typed (non-template) function at compile-time. For example, if the code

```
float fm = max < float > (1, 2);
```

is issued, then the compiler will implicitly create the **`max`** function for type **`float`**. Templates are a powerful abstraction that can be used to greatly reduce written-code size and increase code-reuse.

That said, the current C++ standard does not allow for a variable number of template arguments (e.g., for a function taking those types as arguments); nor does C++ robustly handle a variable number of arguments to functions or

methods. The recently ratified standard, called C++11 [26], does allow for variadic templates, which in turn will allow for robust handling of a variable number of arguments to functions or methods.

Templates in and of themselves cannot be used to create a robust algebraic abstraction in C++. We still desire standard math operators (e.g., **+**, *****, **&**, **<**, and **%**) to be available; in C++, we can use operator overloading to fulfill this need.

3.2. Operator overloading

In the C language, the math operators **+**, *****, **&**, **<**, and **%** (and a handful of others similar to these) are defined solely for the built-in variable types, e.g., **int** and **long**; some are defined for **float**, but none can be made to work with user-defined classes. In C++, these operators can be overloaded to work with any class type; the function names for those listed above are **operator+**, **operator***, **operator&**, **operator<**, and **operator%**. This type of operator overloading provides a robust abstraction mechanism, allowing end-user programs to hide complexity. For example, suppose we define a template class that stores a type value, as

```
template < typename T > class foo {
public:
  T d_value;
  foo (T value = 0) : d_value (value) {}
  ~foo () {}
  T value () { return (d_value); }
}
```

Given this class, we want to be able to manipulate the stored value through operator overloading. One could concisely write the code for the **+** operator for same-typed arguments as

```
template < typename T > foo < T > operator+
(foo < T > lhs, foo < T > rhs) {
  return (foo < T > (lhs.value () +
                     rhs.value ()))
}
```

such that the **+** operator for identical **foo** template types returns another **foo** with the internal value of the sum of the provided arguments internal values. The above code can be used to produce the algebraic-like code segment

```
foo < int > a, b, c;
a = 1;
b = 2;
c = a + b;
```

Given the availability of templates and operator overloading, one can almost construct a C++ extension providing an algebraic-like abstraction. The missing key is for operator overloading in cases when the argument types are not identical. In this case, in order for the C++ code to compile and function correctly, the arguments' types must be able to be compared. The standard library **type_info** class provides this utility.

3.3. type_info and typeid

The **typeid** facility provided by the **type_info** class allows for type comparison of almost any two active variables, as well as a means of retrieving the actual variable type as a string. The **typeid** facility is not limited to built-in C++ types, but is also available for user-created types. Continuing from the previous examples, suppose we wanted to add two potentially different **foo** class types. Then, using **typeid**, one way to implement this functionality is

```
template < typename lhs_t,
           typename rhs_t >
foo < lhs_t > operator+
(foo < lhs_t > lhs, foo < rhs_t > rhs)
{
  lhs_t rhs_to_use = 0;
  if (typeid (lhs) == typeid (rhs)) {
    rhs_to_use = rhs.value ();
  } else {
    rhs_to_use = lhs_t (rhs.value ());
  }
  return (foo < lhs_t > (lhs.value () +
                         rhs_to_use));
}
```

Admittedly, for basic types such as **int** and **float**, the above function could be written with less complexity because the C++ compiler will do any type conversion implicitly. That said, the above code could also be used with *any* **lhs_t** and **rhs_t** types, so long as the type cast from **rhs_t** to **lhs_t** is valid and **operator+** is defined for the **lhs_t** type. The above code can now be used to produce the algebraic-like code segment

```
foo < int > a, c;
foo < float > b;
a = 1;
b = 2;
c = a + b;
```

where the addition is between the types **int** and **float**, and the result stored as an **int**. Similarly, we can also overload the **operator=** method (an in-class function), to allow code such as

```
foo < int > a;
foo < short > b;
foo < long > c;
a = 1;
b = 2;
c = a + b;
```

With the above three C++ concepts in mind, we now describe the C++ extension allowing for algebraic-like language waveform definition.

4. ALGEBRAIC ABSTRACTION

SALINE is written as an independent user-interface layer that resides in its own C++ namespace, and provides an algebraic-like language interface for waveform definition. This section describes the basic classes and concepts required to implement SALINE, including the types of variable classes and operators, how templates are used to perform type propagation through the waveform graph as it is being defined, and its runtime operation. We then briefly describe the interface to the underlying SDR framework.

4.1. Types of variables

Algebraic expressions are combinations of variables and operators on those variables. A variable might represent static data in the form of a scalar, vector, matrix, or constant stream, dynamic data being generated by from a source, or processed data generated by an operator. In order to represent these algebraic expressions using SALINE, three basic variable-oriented classes are required. Examples of the latter two classes are provided after their definition.

1. A base stream class from which all other stream-oriented variable classes are derived.
2. An *operator* class that can be either directly or indirectly instantiated, which represents the output buffer(s) resulting from some specific operator. When multiple output buffers are available, they are obtained using array indexing via overloading the C++ **`operator[]`** method.
3. An *enclosure* variable class that contains a reference to an operator variable. Enclosure variables are either explicitly declared in the waveform script or program, or created as temporary placeholders when multiple operators are executed before the **`operator=`** method is issued. When as the latter, a new object is created and knowledge of this memory allocation is retained by SALINE for later deletion.

The first two classes' prototypes are defined as

```
namespace saline {
  template < typename item_t >
  class stream_base;
  template < typename item_t >
  class enclosure :
    public stream_base < item_t >;
}
```

such that the **`enclosure`** class inherits from the **`stream_base`** class, and both are defined within the **`saline`** C++ namespace. Prototypes for operators such as **`fft`** and **`serial_to_parallel`** can be defined similarly to that of the **`enclosure`** class. Note that the **`stream_base`** template type defines the *output* buffer data type, as is required by buffer-centric implementations. For SALINE, types are defined explicitly at compile type; when using runtime-compiled kernels such as those available in OpenCL [27], types can be defined at compile-time or runtime as needed.

In the case where a single template type is used in the class definition, then at least one input or output stream must be of that type. It is possible to use explicitly typed streams when defining operators, but this definition technique is discouraged. In order to use input and output streams of different types, multiple template parameters can be provided and used. Thus, for example, to create an **`fft`** operation taking in, processing, and returning streams of different types, one could use the class prototype

```
namespace saline {
  template < typename in_t,
             typename proc_t,
             typename out_t >
  class fft_i_p_o :
    public stream_base < out_t >;
}
```

Note that only the output buffer type of the new **`fft`** class is provided to the base stream class.

4.2. Types of operators

There are six primary C++ compatible types of operators needed to form SALINE. Each is described briefly, with example functions. For all operator types below, **`op`** refers to an operator function, **`options`** to user-supplied initialization parameters, **`stream`** to a single class inheriting from **`saline::stream_base`**, and **`streams`** to two or more **`stream`** arguments separated by commas.

1. **`op (options)`** : Type 1 operators take no data streams as arguments, only options. Examples include stream sources, such as reading from a file, generating random data, or providing a constant value.

2. **op (stream, options)** : Type 2 operators take a single data stream and options. Examples include many common math and signal-processing functions, such as **sin**, and **fft**, and **serial_to_parallel**.
3. **op (streams, options)** : Type 3 operators take a variable number of input streams, followed by options. This operator type is currently implemented as (**options, streams)** because the current C++ specification does not robustly handle a variable number of arguments to a function or method [28]. The next C++ specification should provide the necessary functionality via variadic templates, which will allow for the desired argument ordering. Examples include **parallel_to_serial** and **analysis_filterbank**, which in this case both take multiple input streams and return a single stream. Certain native-C++ mathematical operators, such as **sum** and **+**, are implemented as both this type as well as the next.
4. **stream op stream** : Type 4 operators are those that overload built-in C++ math functions. Examples include command math functions such as **+**, *****, **&**, **<**, and **%**, but, instead of operating on scalars (or vectors, as provided by some libraries) these operators are overloaded to handle streams. C++ handles just a single instance of this operator at a time: the command **A + B + C** is interpreted by C++ to be **(A + B) + C** where the parentheses denote operator ordering. SALINE internally splits this equation into two separate equations: **tmp = A + B** and **tmp + C**, where **tmp** is a temporary enclosure variable allocated as a placeholder for the first sum. Under some circumstances, multiple type 4 operators can be combined together, as a sort of runtime optimization; this technique is described later in this section.
5. **stream = stream** : Type 5 operators overload the built-in C++ function **operator=**. The left-hand side (LHS) argument must be an enclosure variable; this property is checked for during runtime only.
6. **stream op= stream** : Type 6 operations overload a built-in C++ math function as well as set an enclosure variable. Examples of this operator include **+=** and **%=**, though some operators are specific to certain types (e.g., integers only). The LHS argument must again be an enclosure variable; this property is checked for during runtime only. Defining the LHS stream as **A**, and the right-hand side (RHS) stream as **B**, this operation is internally expanded into either **A = tmp_A op B** when the operator referenced by **A** is not identical in name and all types to than that being requested; **tmp_A** is an temporary enclosure variable that holds the value of **A** when this expression is issued. When the operator referenced by **A** is identical in name and type to that being requested, the variable **A** is augmented with **B** as another input. This operator type allows for internal graph optimization beyond what any C++ compiler can provide. For example, the accumulator used in Listing 2 can be reduced from N-1 **+** operators into a single N-way sum, as shown in Figure 3. Such runtime optimizations are currently limited to identical operators using identical stream types.

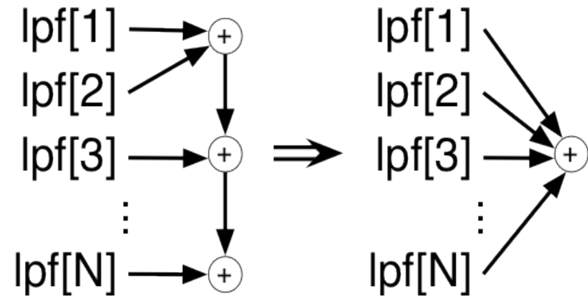

Figure 3 – Runtime optimization of N-1 2-way adders into a single N-way summation.

Using the above operator types, we can now implement the functionality to use templates for type propagation through the waveform graph as it is created.

4.3. Type Propagation via Templates

As described in Section 3, the C++ **typeid** facility is used to provide internal type-conversion, and templates are used for both operator classes and their methods to allow these operator variables to take and return the same or different arugment types. This robust type handling means that the user's script is not required to explicitly declare all function types. As an example using Listing 2 – which uses just type 2 operators – the **serial_to_parallel** function prototype could be defined in C++ as

```
namespace saline {
  template < typename arg_t >
  stream_base < arg_t >&
  serial_to_parallel
  (stream_base < arg_t >& arg,
   int num_outputs,
   options_t& options);
}
```

where the C++ template type, **arg_t**, is used to define both the input and output stream types. Given the **input** argument's type, the C++ compiler will choose the correct template expansion of the **serial_to_parallel** function. Similarly, the type of the chosen **fir_filter** function can be defined implicitly via the type of its input – in this case the variable **s2p[1]** or **s2p[n]**. In this manner, variable types are implicitly used to determine function template expansions, and these types are propagated through the waveform program. This form of type propagation from

input to output requires the sole constraint that any prior streams must already be define and available for use before the current operator is instantiated. Thus, unlike block-centric programming – which allows the waveform graph to be defined in any ordering – the user's program must define the waveform graph from source to sink.

4.4. Runtime operation checks

Certain features of SALINE can be determined during runtime only; hence the user's program will be checked for correctness / validity during runtime as well as compile time. The three primary areas where runtime checking is performed are discussed below. Another area where runtime checks could be performed – variable use that creates a graph cycle – is currently handled by the underlying SDR framework.

1. Variable overwriting : Consider the code

```
saline::enclosure < int > A;
A = 5;
A = 10;
```

The last line overwrites the expression from the prior line, effectively hiding the prior setting of the variable **A**. C++ will compile the above code, and it will execute as directed no matter the variable overwriting. During runtime, SALINE will print out a warning that the LHS variable is being overwritten; internally, the code is reinterpreted as

```
A = 5;
tmp_A = A;
A = 10;
```

where **tmp_A** is a temporary enclosure variable. This reinterpretation allows for a variable to be set for some more-legitimate purpose than the above example, and then overwritten once that purpose is no longer in scope. Variable overwriting can be performed any number of times, using any of the operators from Section 4.2, with each overwrite reinterpreted as a uniquely named temporary enclosure variable as above but for the given operation.

2. Implicit type changes : Consider the code

```
saline::enclosure < int > A;
saline::enclosure < float > B;
A = 5;
B = A;
```

where the variable **A** is set to a **constant_source** with the integer value **5**, and then the variable **B** is set to be the same value as **A** – except as the type **float** instead of **int**. Internally, the last line of the code is reinterpreted as

```
tmp_A = saline::type_converter
          < int, float > (A);
B = tmp_A;
```

where **tmp_A** is a temporary enclosure variable holding the type-converted version of the variable **A**. When the user's code is augmented in this fashion, SALINE will print a warning about the implicit type conversion, allowing the code to execute but letting the user know about the potential issue.

3. Variable declaration order : Consider the code

```
saline::enclosure < int > A, B;
A = B;
```

where the variable **A** is set to **B** before **B** is set to anything. This code will compile in C++ without warnings, but does not make algebraic sense because the variable **B** has not been set before it is used. SALINE will throw a runtime error when executing this code, stating that the RHS variable is being used before being set.

4.5. Interface to the Underlying SDR Framework

As each operator is created in SALINE, its corresponding block is instantiated by the SALINE SDR interface layer using the underlying SDR framework. This layer is lightweight, primarily responsible for instantiating and connecting blocks. It also provides the glue to manipulate the runtime status of the SDR framework – for example starting, pausing, locking, unlocking, and stopping both individual blocks and the framework. *Surfer* can generally handle the insertion and removal of blocks, during runtime without having to stop and restart processing. Further, this layer when using *Surfer* automatically starts framework-level processing before the user's waveform program is executed, and stops it when the program is finished. That said, we recognize that some SDR frameworks do require this functionality, and also that some waveforms might require it for proper processing.

4.6. Putting it all together

Listing 3 provides a SALINE-based version of the polyphase downsample-by-N function. This listing shows a function named **pp_down_N_SALINE** with the same function arguments and return as **pp_down_N_buffer**, and where the internally used functions have the same purpose. For buffer-centric programming, the variables **input**, **s2p**, and **acc** refer to output buffers from previous operators.

```
namespace saline {
  template < typename arg_t >
  stream_base < arg_t >
  pp_down_N_SALINE
  (stream_base < arg_t >& input,
   size_t N,
   options_t& options)
  {
    enclosure < arg_t > s2p, acc;
    s2p = serial_to_parallel
      (input, N, options);
    acc = fir_filter
      (s2p[1], options.ppf[1]))
    for (size_t n = 2; n < N; n++) {
      // 'acc' reused
      acc += fir_filter
       (s2p[n], options.ppf[n]))
    }
    return (acc);
  }
}
```

Listing 3 – Polyphase downsample-by-N written in C++ using SALINE

When compared with Listing 2, the line `acc = 0` is not used because it is interpreted to mean a `constant_source` of value `0`, which at least for the used operator (`+`) is unnecessary. This code is written for efficiency, and does not include error checking on the inputs as would be typical of such a function. Also, this code includes all of the required C++ glue for compiling, while Listing 2 is meant as an example of an interpreted script.

This listing shows a number of the properties mentioned above: implicit operator type selection, type propagation from input to output, operator overloading for greater code clarity, and the use of the `+=` operator for potential runtime optimization. Compared with Listing 1, the core programming (not the C++ glue) reads similarly to MATLAB or Octave script – but for data streams instead of scalars, vectors, or matrices. In more complicated examples – e.g., OFDM modulation or demodulation – SALINE programming will reduce the chances of incorrect connections as well as the overall program length. Providing a scripting experience similar to that of MATLAB should make the transition from general-purpose signal processing to SDR easier for many users.

5. CONCLUSIONS

We have developed an extension in C++ that provides an algebraic-like programming language interface as a novel means for creating SDR waveforms. This extension, called SALINE, currently works with our *Surfer* SDR framework but has been designed independent of *Surfer* and hence could be ported to other SDR projects. We accomplished this task by leveraging standard C++ properties and classes to define the variable types and operators, and creating a form of variable type propagation through the use of appropriate template functions and classes. SALINE reduces complexity compared with current SDR text-based programming interfaces by using a MATLAB-style buffer-based implementation that provides implicit graph-style connections. For many potential users, who wish to just use an SDR framework, the availability of a MATLAB-style interface should reduce the SDR learning curve.

6. ACKNOWLEDGEMENTS

This work has been supported in part by NIJ Grant 2006-IJ-CX-K034 and an NVIDIA Professor Partnership Award.

7. REFERENCES

[1] H. Zimmermann, "OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection", IEEE Transactions on Communications, vol. 28, no. 4, pp. 425 – 432, April 1980.

[2] L. J. Williams (ITT; Technical Director, JTRS Business Area), "Software Defined Radios: Are They Really That Hard?", presented at the IPFW Wireless Summer School, June 17, 2009.

[3] GNU Radio Discussion Email List, thread on "Why Isn't GNU Radio Used More", accessed June 2011: https://lists.gnu.org/archive/html/discuss-gnuradio/2011-05/msg00173.html

[4] MathWorks MATLAB Website, accessed October 2011: http://www.mathworks.com/products/matlab

[5] GNU Octave Website, accessed October 2011: https://www.gnu.org/software/octave

[6] MathWorks Company Facts Sheet, accessed October 2011: http://www.mathworks.com/company/factsheet.pdf

[7] M.L. Dickens, J.N. Laneman, and B.P. Dunn, "Seamless Dynamic Runtime Reconfiguration in a Software Defined Radio", *Proceedings of SDR'11 – WInnComm – Europe*, Brussels, Belgium, June 2011.

[8] GNU Radio Website, accessed June 2011: http://gnuradio.org/

[9] Software Communications Architecture Website, accessed October 2011: http://sca.jpeojtrs.mil

[10] GNU Radio Companion Website, accessed October 2011: http://www.joshknows.com/grc

[11] MathWorks Simulink Website, accessed October 2011: http://www.mathworks.com/products/simulink

[12] National Instruments Corporation, LabVIEW Website, accessed October 2011: http://www.ni.com/labview

[13] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming Heterogeneity – the Ptolemy Approach", Proceedings of the IEEE, v. 91, No. 1, pp. 127-144, January 2003.

[14] Agilent VEE Website, accessed October 2011: http://www.agilent.co/find/vee

[15] Mitov Software OpenWire Website, accessed October 2011: http://www.mitov.com/products/openwire

[16] P. Schniter, “Polyphase Decimation Filter”, Connexions Website, accessed October 2011: http://cnx.org/content/m10433/2.12

[17] J. P. Morrison, *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*, self-published: EAN-13 978-1451542325; https://www.createspace.com/3439170, 2010.

[18] C.-J. Hsu, I. Corretjer, M.-Y. Ko, W. Plishker, S. S. Bhattacharyya, “Dataflow Interchange Format”, Technical Report UMIACS-TR-2007-32, Institute for Advanced Computer Studies, University of Maryland at College Park, June 2007.

[19] SystemC Website, accessed October 2011: http://www.systemc.org

[20] C. Grimm, ed., *Languages for System Specification*, Kluwer Academic Publishers, Boston, MA, USA, 2004.

[21] NoFlo (Flow-based programming for Node.js) Website, accessed October 2011: https://github.com/bergie/noflo

[22] A. V. Berka, “Interlanguages and Synchronic Models of Computation”, published 25 May 2010 on the Isynchronise Ltd. Website, accessed October 2011: http://arxiv.org/pdf/1005.5183

[23] ParC Website, accessed October 2011: http://parallel.cc/

[24] W. Thies, M. Karczmarek, and S. Amarasinghe, “StreamIt: A Language for Streaming Applications”, in the Proceedings of the International Conference on Compiler Construction, Grenoble, France, 2002.

[25] C++ Working Group Documents Website, accessed October 2011: http://www.open-std.org/JTC1/SC22/WG21

[26] C++11 Wikipedia Entry, accessed October 2011: http://en.wikipedia.org/wiki/C%2B%2B11

[27] The Khronos Group, OpenCL Website, accessed October 2011: http://www.khronos.org/opencl

[28] B. Stroustrup, *The C++ Programming Language: Special Edition*, Addison-Wesley, ISBN 0201700735, 2000.