The S# Manual

Using C# to write streaming applications

Kevin Mitchell

Measurement Research Laboratory

Agilent Technologies

T: +44 (0)131 452 0256 E: kevin_mitchell@agilent.com

Summary

The use of high-level languages such as C# in the development of mainstream applications is becoming ubiquitous. However, in situations where we need to maximise performance, for example stream/signal-processing pipelines, C/C++ is often still the language of choice. Ideally we would like the freedom to use C# in such cases as well.

As we move to a world where single-core performance has flattened out we will increasingly need to exploit multicores, FPGAs and GPUs. This is challenging if we want to target different platforms from the same code base, particularly when this is written in a low-level language such as C.

The goal of the S# project is to allow programmers to express their streaming algorithms in C#. The S# runtime will optimize this code for multiple target platforms. The aim is to produce results that are competitive with hand-written code for each platform. This document describes the main features of S#, and how to execute S# programs.

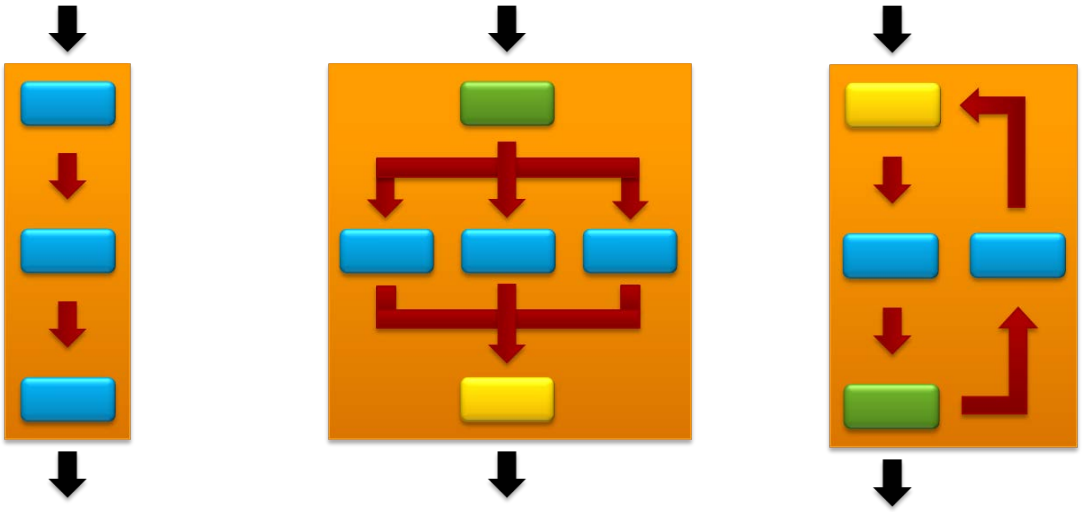
CONTENTS

Introduction

S# is a .NET library that allows a developer to implement high-speed stream processing applications within Microsoft's .NET framework. Code written using S# can be executed directly, or compiled to native code which is then automatically embedded within C#. The eventual goal is for the S# compiler to support multiple back-ends, for example, x86 multicore processors, FPGAs and GPUs, allowing the same application to adapt to different target platforms. The design of S# has been heavily influenced by the work on StreamIt at MIT, and so we start with a brief description of this system to provide some context.

StreamIt

StreamIt is a high-level stream programming language for implementing streaming algorithms, e.g. video, DSPs, networking, and encryption. It has been developed at MIT over the past decade, and is an example of a (mainly) Synchronous Dataflow language. By this we mean that the rate at which data is produced and consumed remains fixed during graph processing. Unlike many such languages StreamIt imposes a rigid hierachical structure on the permissible graphs. All StreamIt graphs are composed from the following components.



Pipeline Split/Join Feedback loop

The atomic elements of a StreamIt graph are filters. All computation takes place within these filters, and each filter has a single input stream and a single output stream. Note that each of the composite graph constructors also has this property. Many of the graph operations performed by the StreamIt compiler rely on the structured nature of the graphs to simplify the analysis.

As StreamIt is a synchronous dataflow language each filter has rate information associated with it. The computation is performed within a work method, and we must define how much data is consumed and produced each time a work method is executed. For example here is a simple up-sampler expressed in StreamIt.

```
float -> float filter UpSamp(int N) {
  work pop 1 push N {
    push(pop());
    for (int i = 0; i < N-1; i++) push(0);
  }
}
```

Another distinguishing feature of the language is its use of peeking. In addition to consuming data from the input port, and producing data for the output port, a filter can also peek ahead in the input. This can be beneficial as it allows us to express some filters in a stateless fashion, where the filters themselves maintain no internal state. One of the key optimizations performed by a StreamIt compiler is the rearrangement of the stream graph to increase the potential for parallel execution. Such rearrangements often rely on the ability to duplicate parts of the stream graph, and it is a lot easier to duplicate a stateless filter than one that maintains state.

StreamIt also supports an out-of-band communication mechanism called teleport messaging. As a motivating example, consider a radio application containing an in-band signal requesting a change in the listening frequency. This signal would happen infrequently, but it would be detected by a filter late in the application and require a change in a filter earlier in the application. Teleport messaging can be used to support such applications, allowing out-of-band data to flow upstream as well as downstream.

Optimizations

Whilst stream programs typically contain an abundance of parallelism, mapping this efficiently to a multicore machine is challenging. The overheads in data communication and synchronization can often overshadow any gains from parallel execution. These overheads can be reduced by performing various transformations. For example, the granularity of the stream graph can be coarsened via filter fusion, a transformation in which two neighboring filters are statically scheduled and inlined into a single filter. Neighboring stateless filters can be fused as much as possible so long as the resulting filter remains stateless, ensuring that it is still amenable to data parallelism. We can also duplicate stateless filters, increasing the amount of potential parallelism.

Domain-specific knowledge can be exploited to optimize the computation further. For example, it is possible to identify filters representing linear computations. We can abstract filters into their linear representations, combine these computations at this level, and then map them back into procedural code. The StreamIt compiler can also automatically translate suitable programs into the frequency domain, and back again, when this leads to improved performance.

Limitations

Why do we need another framework when we already have StreamIt? Despite being available for nearly a decade, there appears to have been little uptake of StreamIt in the commercial sector. In part this might be due to a lack of awareness amongst developers, or a reluctance to rely on an unsupported product. However, there are a number of deficiencies in the StreamIt approach that might have also contributed to the lack of use.

Developers are naturally reluctant to adopt a new language unless the advantages are seen to be over-whelming. The reluctance can be particularly acute for domain-specific languages. Where such a language is used as part of a bigger application, written in a general-purpose language such as C++, Java or C#, issues of interoperability arise. In some cases, such as the use of regular expression strings within a library call, these issues are easily addressed. However, when the domain-specific language plays a larger role in the overall application, for example where a major compute path is written in such a language, then interoperability becomes more of a challenge. The problems become even more acute when the DSL has parts that are similar to, but different from, the general-purpose language hosting it. Different operator precedence, scoping rules, and predefined functions are typical examples. Such differences can lead to a lot of frustration, confusion and errors.

It is hard to describe a problem at the StreamIt level of abstraction in a mainstream language and get a reasonable execution performance from the resulting code. A general-purpose compiler would not be able to

exploit data rates or stream-specific optimizations for example. So where high performance is required a developer must essentially encapsulate the compilation step and optimizations in the code they write.

StreamIt is typically presented as a standalone solution. But many applications will often need to combine a streaming component with a GUI, database access, network access or a myriad of other components. The problem is not just one of embedding a StreamIt component within such an application. We also need to interoperate with it, for example getting data in and out of graphs, and being able to call existing code from within these graphs. The existing StreamIt implementation does not address these issues particularly well, creating potential obstacles to its use.

The StreamIt compiler needs to be configured with knowledge of the target architecture we are compiling for, and command-line arguments are used to provide this information. In the case of a multicore machine such details includes the number of cores, and the compiler then attempts to reorganize the code to enable each of these cores to be kept busy. Whilst such a "closed-world" assumption might be appropriate for some applications, it is not ideal when the streaming execution only forms part of a larger application. In such cases a task-based approach seems preferable, where all components, including the StreamIt runtime, generate tasks that are scheduled by a global concurrency runtime component.

StreamIt requires the user to specify rate information for each filter. Such information allows a compiler to construct static schedules for executing filters, minimizing the buffering requirements, allowing overflow and underflow checks to be removed and so on. Of course not all streaming applications can be easily described using such static rates. The StreamIt developers have acknowledged this, and more recent releases of the compiler have allowed variable rates to be specified. However, support for these is currently incomplete.

One of the most severe restrictions of the current compiler is that it is limited to static graphs. A StreamIt program denotes a fixed graph which is optimized by the compiler prior to deployment. There is no possibility of changing the graph during execution, at startup, or even at deployment time. In a small number of cases we could potentially compile multiple graphs and then switch between them at runtime, but this is clearly not ideal. To increase the applicability of an approach like StreamIt we really require a more flexible solution.

S#

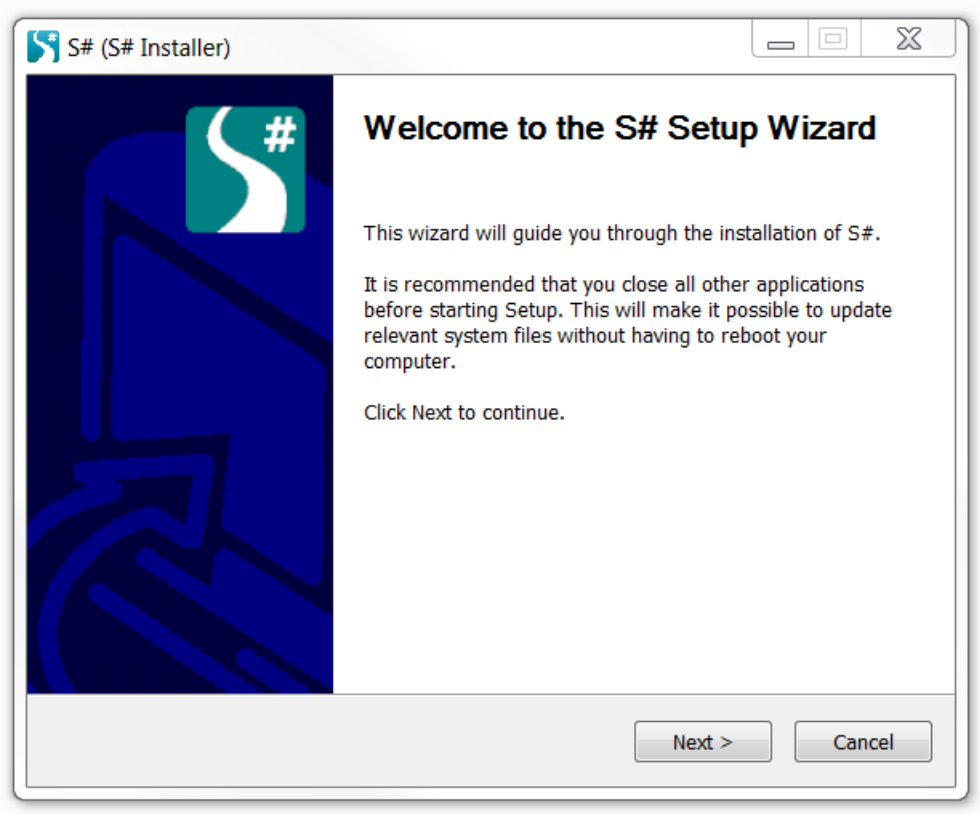
Rather than introducing a new domain-specific language we could encode our streaming programs within an existing language, and then use a customized compiler or post-processor to detect and handle the streaming components in a domain-specific fashion. We use the term S# to refer to a C# program that encodes a StreamIt graph. In addition to a set of stream-specific classes the S# system also provides a verifier and a compiler. To simplify the analysis and manipulation of filters and stream graphs the StreamIt formalism limits the expressiveness of code that can be used to define filters. If we allowed our S# filters to use the full power of C# it would be infeasible to do all the StreamIt analysis we require. However, we can identify a subset of C# that is roughly comparable in expressive power to StreamIt. The verifier ensures that the code inside all classes derived from streaming base classes is limited to this subset of the language, whilst leaving the remaining classes in an application unaffected.

The job of the compiler is to convert the stream graphs into code that can be efficiently executed on a variety of platforms. In principal most, if not all, StreamIt optimizations could be recast into the S# framework.

Installing S#

To install S# you first need to download an installer to your machine. The latest version of the library can always be found at http://mad.engineering.agilent.com/kevin/SSharp/Latest.zip.

Unpack the zip file and run the enclosed installer.



Click the "Next" button on each page until the installer has finished. This should deploy the library and support files to `C:\Program Files\S#`. The environment variable SS_DIR will also be set to this path. You may find it convenient to add `%SS_DIR%` to your `PATH` environment variable. For reasons that are currently unclear, occasionally the installer ends up corrupting the environment variables in the current session, but logging out and back in again fixes the problem.

The most important file in this directory is the assembly called `Agilent.Streaming.dll`.

You will need to add a reference to the `Agilent.Streaming.dll` assembly in any C# project that makes use of S#.

There is a second version of this assembly in the Debug subdirectory that has been compiled with debugging code. You may find it useful to reference this assembly initially as it does more runtime checking to detect errors. You can also find a copy of this document in the docs directory, together with a description of the public API for the library.

A Simple Example

Before delving into the details of the system we start with a simple example to give the reader a flavor of what an S# program looks like. Do not worry too much about the details at this stage. Consider the problem of defining an integer ratio up-sampler. Such an up-sampler mathematically has two steps (here, R is the up-sampling ratio):

1. Insert R-1 zeros between each sample of the input. The result is a signal that is almost all zeros, with spikes appearing with period R.
2. Apply a low-pass FIR filter with T taps, with T>>R. This smooths out the signal again.

In S# we can implement the first step using code like the following:

```
class UpSample : Filter<float, float> {
  private int _upsamplingRatio;

  public UpSample(int upsamplingRatio) : base(pop: 1, push: upsamplingRatio) {
    _upsamplingRatio = upsamplingRatio;
  }

  protected override void Work() {
    for (int i = 1; i < _upsamplingRatio; ++i)
      Push(0.0f);
    Push(Pop());
  }
}
```

Here is an S# FIR filter:

```
class FIRFilter : Filter<float, float> {
  readonly float[] _taps;

  public FIRFilter(float[] taps) : base(peek: taps.Length, pop: 1, push: 1) {
    _taps = taps;
  }

  protected override void Work() {
    float sum = 0.0f;
    for (int i = 0; i < _taps.Length; i++) sum += Peek(i) * _taps[i];
    Pop();
    Push(sum);
  }
}
```

Finally, we can combine these two filters into a pipeline that performs the up-sampling:

```
public class Upsampler : Pipeline<float, float> {
  public Upsampler(int upsamplingRatio, float[] taps) {
    Add(new UpSample(upsamplingRatio));
    Add(new FIRFilter(taps));
  }
}
```

This version of the up-sampler is not very efficient as we spend a lot of time multiplying by `0`. A better approach might be to split the input into a number of pipelines, apply a smaller FIR filter to each of these, and then combine the results. This has the advantage of eliminating the unnecessary multiplications, and also increases the potential parallelism of the program as each pipeline can be computed independently.

An integer ratio up-sampler that implements this strategy can be expressed in S# using the following code:

```
public class IntegerRatioResampler : SplitJoin<float, float> {
  public IntegerRatioResampler(int upsamplingRatio, float[] taps) {
    Contract.Assert(taps.Length % upsamplingRatio == 0);
    Splitter = new Duplicate<float>();
    for (int i = upsamplingRatio-1; i >= 0; --i) {
      float[] branchTaps = new float[taps.Length / upsamplingRatio];
      for (int j = 0; j < branchTaps.Length; ++j)
        branchTaps[j] = taps[i + j*upsamplingRatio];
      Add(new FIRFilter(branchTaps));
    }
    Joiner = new RoundRobin<float>();
  }
}
```

You may want to wait until you've read more of the manual before trying to run some S# code. But if the suspense is too much you will need to embed the up-sampler code in a context that supplies input values to the up-sampler and consumes the results it produces. The following code, whilst artificial, should illustrate the basic idea, and will give you something you can run. However, you will need to skip ahead to the section on S# verification to avoid the runtime verification exception that occurs when executing this program.

```
class Program {
  static void Main() {
    float[] taps = new float[] {
      0.0f, 0.1f, 0.2f, 0.3f, 0.4f, 0.5f, 0.6f, 0.7f, 0.8f, 0.9f, 0.10f, 0.11f
    };

    float v = 0;

    StreamProcessor sp = new StreamProcessor(
      StreamGraph.Pipeline(
        StreamGraph.Filter(() => v++),
        new IntegerRatioResampler(3, taps),
        StreamGraph.Filter((float f) => Console.WriteLine(f))
      )
    );

    sp.Execute();
    // Runs asynchronously by default, so pause main thread
    // so we get some output.
    System.Threading.Thread.Sleep(1000);
  }
}
```

Now you have seen what an S# program looks like it is time to dig into the details.

Graph Constructors

The S# library uses a *stream graph* to process a data stream. All stream graphs have a single input and a single output. Whilst this might, at first, seem rather limiting, we will see that S# provides a number of mechanisms for composing stream graphs that allow a rich variety of structured graphs to be built. In this section we describe how to build stream graphs started with the simplest, a filter.

Filters

The simplest stream graph is a filter. There are a number of different ways of defining filters in S#. One approach is to define a new class that inherits from the `Filter<InputT, OutputT>` class defined by the S# library.

```
using Agilent.Streaming;

public class Sqrt : Filter<int, double> {
  public Sqrt() : base(pop: 1, push: 1) { }

  protected override void Work() {
    Push(Math.Sqrt(Pop()));
  }
}
```

As with all S# graphs a filter reads data from a single input stream and produces results on a single output stream. In this example we have defined the input to the filter to be a stream of integer values and the output to be a stream of doubles.

A filter must do some work to process the values provided by the input stream. The `Work()` method defines how the filter performs this task, and every filter must override this method to define the desired behavior. Values may be read from the input stream using the `Pop` method, and results written to the output stream using the `Push` method. In our example a single value is read from the input, the square root calculated, and the result pushed on the output stream each time the `Work` method is called. At first glance such a filter would seem to be very inefficient; the cost of calling the `Work` method, and the methods to read and write data from the streams would surely be larger than the cost of performing the filter-specific processing. However, as we shall see later, the S# framework will typically compile a stream graph into native code, and as part of this process much of this overhead will be eliminated. It is therefore important that you do not worry too much about efficiency when constructing your stream graphs. Too much optimization at this stage may result in a very course-grained graph, preventing the S# compiler from optimizing the graph for a target platform.

S# is an example of a (mainly) synchronous dataflow language. The compiler uses rate information to compute a schedule for executing the graph, and the required buffer sizes to connect components. For simple examples a compiler could potentially infer the rates from the code. In our simple example it seems obvious that the filter will pop one value each time the work method is called, and produce one result. But in many cases, particularly when pops and pushes occur within conditional code or loops, it can be difficult to infer this information. The current S# system therefore requires the rates to be declared explicitly. In our example you can see this is done in the call to the base constructor. The default pop and push rates are zero, and so you only need to declare non-zero rates.

In addition to popping values from an input stream a filter can also peek ahead to access values without consuming them. The peek index is o-based, so `Peek(0)` returns the same value as `Pop()`. Here is a simple filter than produces an average of adjacent values:

```
public class Average : Filter<double, double> {
  public Average() : base(pop: 1, peek: 2, push: 1) { }

  protected override void Work() {
    Push((Peek(0) + Peek(1))/2.0);
    Pop();
  }
}
```

In this example the push and pop rates are identical to our previous filter, but we must now define a peek rate as well.

The `Average` filter is stateless; it maintains no state between calls to the `Work` method. Stateless filters are important when we start considering parallel execution as they can be duplicated to increase potential parallelism. An alternative formulation of this filter avoids peeking by maintaining some state.

```
public class AverageUsingState : Filter<double, double> {
  double previous = 0.0;

  public AverageUsingState() : base(pop: 1, push: 1) { }

  protected override void Work() {
    Push((previous + Peek(0)) / 2.0);
    previous = Pop();
  }
}
```

Whilst the behavior of this filter is similar to our stateless version, it is not quite identical. In the stateless version the first value we output will be the average of the first two values in the stream. In our alternative version we will output the average of the first value and the initial value of `previous`. We can modify our alternative to behave identically to the original by making use of another S# feature, `InitWork` methods. We want to read the first value from the input, use this to initialize the value of previous, and then proceed to execute the `Work` method repeatedly. We define the behavior we want executed in an `InitWork` method. This method, if defined, is executed exactly once prior to executing the Work method. Furthermore, it is allowed to declare different rates to the `Work` method.

```
public class AverageUsingState : Filter<double, double> {
  double previous = 0.0;

  public AverageUsingState()
  : base(/*init pop */ 1, /* pop */ 1, /* init push */ 0, /* push */ 1) { }

  protected override void InitWork() {
    previous = Pop();
  }

  protected override void Work() {
    Push((previous + Peek(0)) / 2.0);
    previous = Pop();
  }
}
```

Which version is best? In this particular example the stateless version using peeking is the most concise, and has other advantages from an optimization viewpoint. But in other cases state is an essential property of a filter, or it would be too expensive to keep recalculating it. So the general advice would be to prefer stateless filters where possible, but not at the expense of clarity.

```
public class LowPass : Filter<double, double> {
  private readonly double[] coeff;
  private readonly int taps;
  private readonly int decimation;

  public LowPass(double rate, double cutoff, int taps, int decimation)
    : base(peek: taps, pop: 1+decimation, push: 1) {
    coeff = new double[taps];
    for (int i = 0; i < taps; i++) {
      coeff[i] = ...;
    }
    this.taps = taps;
    this.decimation = decimation;
  }

  protected override void Work() {
    double sum = 0;
    for (int i = 0; i < taps; i++) sum += Peek(i) * coeff[i];
    Push(sum);
    for (int i = 0; i < decimation; i++) Pop();
    Pop();
  }
}
```

One of the goals of S# is to allow developers to write highly-parallel algorithms without having to worry about many of the issues normally associated with such programs, for example thread safety, deadlock, livelock, and race conditions. But if the data flowing through a stream graph was mutable, for example instances of C# classes, then we could not guarantee that the execution of a filter would not affect the execution of any of the other filters. For this reason stream types are limited to value types. At some point we might relax this restriction, allowing immutable reference types to be used as well.

Anonymous Filters

In some situations it can be tedious to define a new class just to implement a simple filter, particularly if it will only be used once. S# provides some convenient shortcuts for such cases. Returning to our `Sqrt` example, this filter takes a single value from the input, applies a transformation to it, and pushes the result on the output stream. We can define this transformation as a C# lambda expression, `(double d) => Math.Sqrt(d)`. Ideally we'd like to define an implicit conversion from such an expression to a filter instance, but this is not possible in the current version of C#. Instead, S# defines a static method `StreamGraph.Filter` that performs this task. Using this method we could replace an expression such as `new Sqrt()` by `StreamGraph.Filter((double d) => Math.Sqrt(d))`. Of course if we will use multiple instances of the filter, or the computation performed by the filter is non-trivial, then it may be preferable to define an explicit filter class for this purpose. But in "use-once" scenarios the use of the `Filter` static method leads to more concise code, particular as we can frequently drop the "`StreamGraph.`" prefix, and the filter type parameters can be inferred from the type of the lambda expression.

What if we want to consume more than one input value each time? We just define a lambda function that takes more than one argument. And what if we want to produce more than one result from the same input value? We define multiple lambdas; each one will be applied to the input value in turn, with the results pushed onto the output.

But what if we want to consume multiple values and produce multiple results. We could add more and more overloads to the `Filter` method, but the code would quickly become unreadable. For such cases we can define an anonymous filter by creating an instance of Filter rather than subclassing it. The constructor takes as arguments a work rate and a work delegate. This delegate is passed the filter that has been created, allowing the

body of the delegate to peek, pop and push values as required. Here is how we would define our `Sqrt` filter using this mechanism.

```
new Filter<double, double>(1, 1, (af) => { af.Push(Math.Sqrt(af.Pop())); })
```

The static methods defined in `StreamGraph` are often used in a context where the class prefix can be omitted. Even so, in this case our earlier versions of the filter would be preferable, and should be used where the additional flexibility of this approach is not required. A second constructor allows a pre-work rate and an init-work delegate to also be specified.

Sources and Sinks

In a stream graph data will typically enter the graph from an external source, or be generated in a filter. In the second case the filter will have a pop (and peek) rate of zero. Although such a filter will never read values from the input stream we still have to define an input type. The S# library defines a type called `VOID` for use in such situations.[1] It is an error for a filter to have `VOID` as the input type with a non-zero pop rate. To make the role of such filters more explicit the library defines a `SourceFilter` subclass that can also be used.

```
public class Ramp : SourceFilter<double> {
  private readonly double max;
  private double current;

  public Ramp(double max) : base(push: 1) {
    this.max = max;
    this.current = 0.0;
  }

  protected override void Work() {
    Push(current);
    if (current > max) {
      current = 0.0;
    } else {
      current++;
    }
  }
}
```

Similarly there will be some filters that act as sinks, consuming data from upstream but producing no data on the output stream. Such filters will have `VOID` as the output type, and the library also defines a convenience subclass called `SinkFilter` for this case as well.

```
public class Printer : SinkFilter<double> {
  public Printer() : base(pop: 1) { }

  protected override void Work() {
    System.Console.WriteLine("{0}", Pop());
  }
}
```

Variable Rates

Graphs where the components consume and produce data at fixed rates are amenable to a lot of optimizations. We rely on such optimizations when compiling graphs to efficient code. In particular, fixed rates allow us to

[1] .NET defines a `System.Void` type that would be ideal in this role, but this type cannot be used from C#.

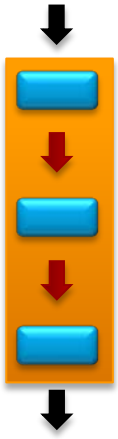
statically schedule the graph, and calculate buffer sizes for the links connecting the components. Unfortunately not all streaming applications can be expressed in this form. In some cases we can force an algorithm into the fixed-rate model by transmitting dummy values, but this is not ideal and creates its own inefficiencies. All our examples so far have used fixed rates, but S# also supports variable rates. For example, we might define

```
public Ramp(double max) : base(push: null) { ... }
```

This declares that the associated `Work` method will push a variable number of items each time it is invoked. You can also use `null` for the pop rate, indicating a variable number of items may be read each time. Peek rates must always be fixed, and if a peek rate is specified the pop rate must be fixed as well. Such variable rates severely limit the kinds of optimizations we can perform. Prior to execution the S# compiler rearranges the graph into fixed-rate islands connected by variable-rate links. The fixed-rate subgraphs can be statically scheduled and optimized. These graphs are then run as independent tasks, communicating with each other over variable-sized buffers. If you find that the majority of filters in an example require variable rates then there will be very little scope for optimization, and perhaps the example is not a good fit for something like S#. However, if the number of variable-rate filters is small, then their impact may be minimal, particularly when targeting a multicore machine or an FPGA.

Pipelines

A single filter is not very interesting from a stream processing perspective. We want to combine filters to produce a stream graph. The simplest way of combining graphs is to link them together in a pipeline. S# defines a composite graph class called `Pipeline`, and within the constructor for this class we can add sub-graphs to build the pipeline. Here is a simple example.



```
public class PipelineExample : Pipeline<VOID,VOID> {
  public PipelineExample() {
    Add(new Ramp(1000));
    Add(new Average());
    Add(new Printer());
  }
}
```

We can also build an anonymous pipeline by passing the list of sub-graphs to the pipeline constructor. So rather than constructing an instance of `PipelineExample` we could define the same pipeline using an expression such as

```
new Pipeline<VOID,VOID>(new Ramp(1000), new Average(), new Printer())
```

The library also supports a Unix-style pipe operator, as in

```
new Ramp(1000) | new Average() | new Printer()
```

However, note that due to limitations in the typing of operators in C#, the type of such an expression is simply `StreamGraph`, rather than the more accurate `Pipeline<VOID,VOID>`. This limits the usefulness of the pipe operator in contexts where a strongly-typed graph is expected, unless the result is cast to the appropriate type.

For a pipeline to be well-formed a number of constraints must be satisfied. The input type of the first graph in the pipeline must match the declared input type of the pipeline. The output type of the last sub-graph must also match the output type of the pipeline. Finally each adjacent pair of sub-graphs in the pipeline must agree on the link type connecting them.

In the general case we may want to vary the pipeline depending on the values of the constructor arguments. We might conditionally add a graph to the pipeline, or add multiple instances of a filter within a loop for example. For this reason we cannot always check that a graph is well-formed until runtime when any type mismatches will be detected and reported.

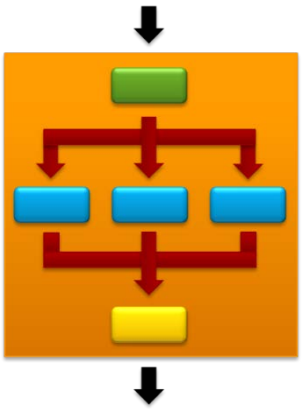
For simple graphs, where the structure of the graph is fixed at compile time, it would be more useful if ill-formed graphs could be identified at compile time as type-checking errors, rather than at runtime as thrown exceptions. The S# runtime provides a number of convenience methods to support this. They are defined as static methods in the `StreamGraph` class. As this class is the base class of all stream graph classes, such as `Filter` and `Pipeline`, the `StreamGraph.` prefix can be omitted when the methods are used within the context of a composite stream graph constructor. We could build our earlier pipeline using the expression

```
StreamGraph.Pipeline(new Ramp(1000), new Average(), new Printer())
```

However, the type inference mechanism in C# is easily confused, and so in some cases you may need to explicitly specify the type arguments for the function, as in `Pipeline<VOID,double,double, VOID>(…)`.

At first glance this expression looks similar to the version that uses the constructor. However, there are two important differences. Any mismatch in type between the pipeline components will be detected at compile time using the `Pipeline` method, and the input and output types of the pipeline are often inferred automatically. In the case of the constructor the user must explicitly specify the input and output types. Furthermore, the constructor takes a list of stream graphs and only checks they are compatible at runtime. Of course there is a price we pay for the convenience of static type-checking. The `Pipeline` method is overloaded, with one definition for a two element pipeline, another for three elements and so on. So we can only use this approach when the number of items in the pipeline is both fixed at compile time, and relatively small.

Split-Joins



Whilst pipelines allow us to build large graphs, they are clearly limited in the graph structures they can describe. What if we wanted to split a data stream into multiple flows, then process each of these separately in parallel, and combine the results in some fashion? Clearly such a requirement would be beyond the scope of a simple pipeline.

To express such behavior we introduce our second composite graph structure, a *split-join*. This component contains a *splitter*, some number of children that run in parallel, and a *joiner*.

Consider the task of building a band-pass filter. We could implement the core of it using a split-join combining two of our low-pass filters defined earlier.

```
public class BandPassCore : SplitJoin<double, double> {
  public BandPassCore(double rate, double low, double high, int taps) {
    Splitter = new Duplicate<double>();
    Add(new LowPass(rate, low, taps, 0));
    Add(new LowPass(rate, high, taps, 0));
    Joiner = new RoundRobin<double>();
  }
}
```

This split-join uses a duplicating splitter; each incoming item is sent to both children. This is not the only splitter type available in S#:

SPLITTER OPTIONS:

Duplicate()	Passes each incoming item to every child
RoundRobin()	Passes the first item to the first child, the second to the second, and so on in a round-robin fashion
RoundRobin(*n*)	Passes the first *n* items to the first child, the next *n* to the second, and so on. RoundRobin() is equivalent to RoundRobin(1)
RoundRobin(w_1, w_2, ..., w_n)	Passes the first w_1 items to the first child, the next w_2 items to the second child, and so on. If there are *n* weights specified there must be exactly *n* children. A weight can be `0`. In this case the corresponding child must have an input type of `VOID`

Our example uses a round-robin joiner, and so the output of the split-join will alternate between the outputs of each low-pass filter. Just as with the splitter there are multiple joiner types available to us.

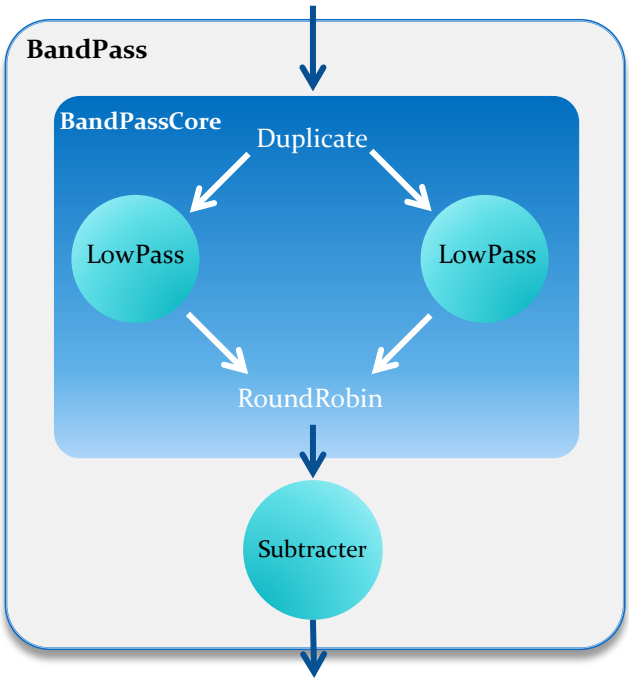
JOINER OPTIONS:

RoundRobin()	Passes the first item to the first child, the second to the second, and so on in a round-robin fashion
RoundRobin(*n*)	Passes the first *n* items to the first child, the next *n* to the second, and so on. RoundRobin() is equivalent to RoundRobin(1)
RoundRobin(w_1, w_2, ..., w_n)	Passes the first w_1 items to the first child, the next w_2 items to the second child, and so on. If there are *n* weights specified there must be exactly *n* children. A weight can be `0`. In this case the corresponding child must have an output type of `VOID`

To complete our band-pass filter we must combine the results from the split-join.

```
public class BandPass : Pipeline<double, double> {
  public BandPass(double rate, double low, double high, int taps) {
    Add(new BandPassCore(rate, low, high, taps));
    Add(Filter((double d1, double d2) => d2 - d1));
  }
}
```

Sometimes it can be helpful to visualize the resulting stream graph.



Just as with pipelines, the children of a split-join must have types that are compatible with the type of the split-join itself. Furthermore, in the general case type mismatches can only be detected at runtime, although the reason for this is perhaps less obvious than in the pipeline case. Consider the case of a component derived from `SplitJoin<int,double>` with a weighted round-robin for both splitter and joiner, and three children. We might think that each child would be constrained to instances of type `StreamGraph<int,double>`. But consider the case where the first weight in the splitter is `0`. In this case no data will flow to the first child, and so it would need to have `VOID` as the input type. Similarly if the last weight in the joiner was `0` then the last child would need `VOID` as the output type. The joiner would never consume data from this child, and we don't want data to be silently discarded. Finally, note that in general the weights passed to a round-robin might only be calculated at run-time, requiring us to delay checking for well-formed graphs until that point.

If a split-join class is used only once we can construct an instance of `SplitJoin` directly, rather than going to the expense of defining a subclass. Our `BandPassCore` class is such an example. We could redefine our `BandPass` class to use an anonymous split-join as follow.

```
public class BandPass : Pipeline<double, double> {
  public BandPass(double rate, double low, double high, int taps) {
    Add(new SplitJoin<double, double> {
          Splitter = new Duplicate<double>(),
          Branches = { new LowPass(rate, low, taps, 0),
                       new LowPass(rate, high, taps, 0) },
          Joiner = new RoundRobin<double>()
        });
    Add(Filter((double d1, double d2) => d2 - d1));
  }
}
```

Note how we are using the relatively new C# syntax to both create and initialize properties of a class instance in a single expression.

The majority of uses of split-join are rather simple, with a fixed number of children and non-zero weight splitters and joiners. For these cases we can use the static methods defined in `StreamGraph` to construct

anonymous instances of these components that can be type-checked at compile-time. This situation is similar to the approach followed for pipelines. For example, we could define

```
public class BandPass : Pipeline<double, double> {
  public BandPass(double rate, double low, double high, int taps) {
    Add(SplitJoin(
          new Duplicate<double>(),
          new LowPass(rate, low, taps, 0),
          new LowPass(rate, high, taps, 0),
          new RoundRobin<double>()
       ) );
    Add(Filter((double d1, double d2) => d2 - d1));
  }
}
```

Unlike in our previous versions, if we replaced one of the `LowPass` instances by a stream graph that wasn't an instance of `StreamGraph<double,double>` we would get a type-checking error.

Specialized Splitters and Joiners

Filters, splitters, and joiners are all examples of *workers*, classes that define a `Work` method together with information about the rates at which data is consumed and produced. We have seen how to define our own filter subclasses, but most of the time you will just use predefined splitters and joiners. We can, however, also create our own splitter and joiner subclasses in more advanced examples. This is more complex than in the filter case because we have to deal with multiple inputs and outputs, potentially with differing types.

Why would we want to define our own classes? Consider the case where we wish to implement a multiplexor. This might consist of a split/join with *N* branches calculating values and a selector branch determining which of these to choose. A joiner might pop a value from each branch and then use the value of the selector to pick one of these to push as the result. It is difficult to express such behavior using a `RoundRobin` joiner as *N* branches will have some type `T`, the output type of the split/join, but the selector branch will have an output type of `int`. We could write adapter filters to convert these types into a hybrid wrapper type, but this would be cumbersome. It would be neater to define a custom joiner class to handle this case.

A similar problem occurs when we want to construct a tuple of values as result, with each component of the tuple taken from a branch of the split/join. Again we could define wrapper filters to convert the types to a common type. This would be particularly tedious as the types are all value types, and so we cannot simply use `object` as the common type.

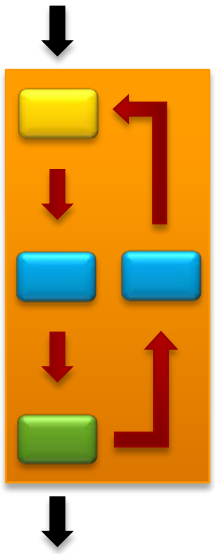
We might generalize these two examples by defining a lambda joiner that would take a lambda expression as argument, read values from each branch of the split/join, apply the lambda expression to these values, and then push the result. A `Tuple` joiner would be an instance of this where the lambda body simply constructed the tuple. Similarly the `Mux` joiner would just use a lambda expression consisting of a switch statement.

Another example of a custom joiner would be one that read values from the first input until some condition matched, then read values from the second branch until the condition matched, and so on in a round-robin fashion.

Defining such joiners is rather tedious, mainly because we have to handle multiple inputs, each potentially with a different input type and rate. The simplest approach to defining such joiners is probably to take one of the existing joiners, for example, `Reduce` or `JoinFromBranchUntil` defined in "%SS_DIR%\library\Joiners.cs", and use it as a starting point.

Feedback Loops

Using the graph constructors we have introduced so far we can only build acyclic graphs. This is obviously rather restrictive in some situations. The final composite graph structure provided by S# is a *feedback loop*. This is like an inverted split-join, with a joiner at the top and a splitter at the bottom. Unlike a split-join, a feedback loop always has two children, called the *body* and the *loop*. There is also a mechanism for initially pushing data onto the feedback path; we must enqueue enough values to prevent the joiner deadlocking. For example, suppose the splitter was a simple round-robin, and the filter at the head of the body stream graph had a pop rate of two. The S# runtime would need to pop a value off the input and a value off the feedback path before it could execute the body graph. But the feedback path requires output from the body before it can be executed, leading to deadlock. Pushing an initial value, or values, onto the feedback path can break this deadlock.



The definition of a feedback loop is best illustrated by an example. Consider an echo application where some part of the input needs to be fed back to the input each time the loop is invoked. We could capture this behavior using the following feedback loop.

```
public class Echo : FeedbackLoop<float, float> {
  public Echo(int n, float scale) {
    Joiner = new RoundRobin<float>(1, 1);
    Body = new FloatAdderBypass();
    Loop = Filter((float f) => f * scale);
    Splitter = new RoundRobin<float>();
    for (int i = 0; i < n; i++) Enqueue(0f);
  }

  class FloatAdderBypass : Filter<float, float> {
    public FloatAdderBypass() : base(pop: 2, push: 2) { }
    protected override void Work() {
      Push(Peek(0) + Peek(1));
      Push(Peek(0));
      Pop();
      Pop();
    }
  }
}
```

Where an anonymous feedback loop is desired we can instantiate a `FeedbackLoop` instance directly, or use a `StreamGraph` helper method, as illustrated below.

```
var loop1 = new FeedbackLoop<float, float> {
                  Joiner = new RoundRobin<float>(1, 1),
                  Body = new FloatAdderBypass(),
                  Loop = StreamGraph.Filter((float f) => f * scale),
                  Splitter = new RoundRobin<float>(),
                  Enqueued = Enumerable.Repeat(0.0f, n)
                };
```

```
var loop2 = StreamGraph.FeedbackLoop<float, float> (
                joiner: new RoundRobin<float>(1, 1),
                  body: new FloatAdderBypass(),
                  loop: StreamGraph.Filter((float f) => f * scale),
              splitter: new RoundRobin<float>(),
              enqueued: Enumerable.Repeat(0.0f, n)
            );
```

Verification

Having written your C# code to build S# stream graphs the next step is to run it. If you do so you will find the results are rather disappointing; the S# library raises an `InvalidOperation` exception with the message

> `The assembly` *<assembly>* `has not been processed by the S# verifier`

as soon as you attempt to create a filter.

The goal of S# is to compile streaming algorithms into high-performance parallel code. If we allowed our filters to use the full power of C# then this task would become intractable. It would be difficult to allocate objects on the heap within a `Work` method, for example, and then still expect to invoke this method millions of times per second. Compiler optimizations rely on the fact that filters can only communicate with the rest of the program using their input and output streams, but if they were allowed to contain potentially shared mutable state then this assumption might not be true. We would also like to automatically generate and execute C++ code from our graphs, and some C# features would be difficult to map to C++ without a substantial amount of analysis and translation.

To ensure that our S# code respects these constraints we must check it with the S# verifier that was installed along with the rest of the library. This takes a compiled DLL or .NET executable as input, and decompiles the code to find S# filters. It then checks these filters to ensure they are well-formed. If they are the assembly is tagged with a "verified" attribute, and otherwise an error message explaining the violation is generated.

We can automate the verification step by adding it as a post-build step to Visual Studio. Simply add the line

```
"%SS_DIR%\ssv.exe" $(TargetFileName)
```

to the project's post-build steps. This invokes the S# verifier, `ssv.exe`, passing it the name of the assembly to verify. If verification fails then the build process will also fail.

In addition to checking that our code meets the S# constraints the verifier also detects additional properties, such as when a filter is stateless, to help subsequent execution and compilation. Note that the verifier uses decompilation techniques to analyze a compiled assembly. Some extreme forms of mechanical obfuscation may make it difficult to perform this step, and in such cases it may be preferable to obfuscate the code after it has been verified.

Execution

Once our code has been accepted by the verifier we should be able to build a stream graph. But how do we use this graph to process a stream of data? The simplest approach would be to connect all the components with expandable buffers, assign each filter, splitter and joiner to a separate thread, and start all these threads in parallel. We'd need to provide some blocking mechanism on the buffers to stop producers getting too far ahead of consumers. But the main problem with such an approach would be its inefficiency. We aren't even exploiting the static rate information available to us.

To execute graphs in S# we use the `StreamProcessor` class. Each instance of this class is responsible for analyzing, optimizing and executing the stream graph passed to it in the constructor. Here is a simple example of its use.

```
StreamProcessor sp = new StreamProcessor(
  StreamGraph.Pipeline(new Ramp(1000), new Average(), new Printer()));
sp.Execute();
```

If you execute this code you might expect to see some output, but you will be disappointed.[2] The stream processor executes the graph in one or more tasks. But by default the call to `Execute` will not wait for these tasks to finish; indeed they might be expected to run for ever. When the call returns the main program will also terminate, assuming this is the only code it is executing. This will result in the termination of the graph execution, which is probably not what you wanted. There are a number of ways to solve this problem. Simply inserting a call to `System.Threading.Thread.Sleep(5000)` after the call to `Execute` is a crude solution that allows us to check we are producing the expected output. A better approach explicitly waits until the execution has been cancelled.

```
using (System.Threading.CancellationTokenSource tokenSource =
         new System.Threading.CancellationTokenSource()) {
  sp.Execute(tokenSource);
  tokenSource.Token.WaitHandle.WaitOne();
}
```

We can cancel execution from within a filter by calling the `ExitGraph()` method.

Now consider the case where the input stream and/or output stream for the graph should connect to the rest of the application. The `Average` filter is an example of such a graph.

```
StreamProcessor<double, double> sp =
  new StreamProcessor<double, double>(new Average());
```

You will find that the Execute method for this stream processor has various overloads that can be used to pass data into the graph and receive data from it. For example,

[2] If you run this example and get a `BadImageFormatException` make sure your platform is set to `Any CPU` in the project configuration.

```
double[] input = new double[] { 1.0, 2.0, 3.0, 4.0, 5.0 };
double[] output = new double[input.Length-1];

using (CancellationTokenSource tokenSource = new CancellationTokenSource()) {
  sp.Execute(input, output, tokenSource);
  tokenSource.Token.WaitHandle.WaitOne();
}
```

Note that as the definition of Average peeks ahead the size of the output array will be less than the input array. The `StreamProcessor` class defines a number of methods to help calculate the number of iterations required to process an input stream, and the size of the input and output buffers required to support some number of iterations. These are the results we would get from our current example.

Expression	Result
`sp.IterationsForInputOfSize(5)`	`4`
`sp.InputSizeRequiredFor(iterations: 4)`	`5`
`sp.OutputSizeRequiredFor(iterations: 4)`	`4`

If we were reusing the same buffers repeatedly then we might want to process an incomplete buffer. To handle this case the `Execute` method can also be passed the number of iterations to perform. If this is not specified the runtime uses the `IterationsForInputOfSize` method to calculate the number of iterations based on the buffer size. It assumes the output buffer is large enough to handle the output produced by this number of iterations.

Static Scheduling

One of the first optimizations performed by the S# runtime, as described earlier, is to partition the graph into synchronous sub-graphs. If a component has a fixed push rate and is connected to another component with fixed peek and pop rates then they will both be in the same synchronous sub-graph. As most S# examples just use fixed rates we will typically end up with a single graph after this step. However, in the rare case where we require variable rate connections the result will be two or more sub-graphs. Each of these will be executed asynchronously, with the components communicating with each other using thread-safe stream buffers.

Having constructed one or more synchronous graphs the runtime then produces a static schedule for each one. The idea behind a static schedule is simple. If the graph is well-formed there should be a sequence of executions of the `Work` functions for the filters making up the graph, as well as the splitters and joiners, such that the amount of data in the buffers at the end of this schedule is the same as at the start. This property allows us to execute such a schedule indefinitely without any of the buffers overflowing or underflowing. Not all graphs will have such a schedule, but in most cases this will be because of an error in the program. For example, consider a split-join with a duplicate splitter and an unweighted round-robin joiner. If the branches in the split-join have different push rates then data will gradually accumulate on the branch with the highest push rate, resulting in a buffer overflow. We would not be able to construct a static schedule for such a graph and it would be rejected by the S# runtime.

S# will be able to deduce a valid schedule for almost all synchronous graphs of interest. Furthermore, in many cases there will be multiple schedules that could be used, and there will be tradeoffs we need to consider. At one extreme we might try to minimize the size of the schedule, ideally only executing each work method the minimum number of times at a single point in the schedule. This will tend to require a lot of buffering of intermediate results, and result in a schedule with high latency. We could try to reduce the buffering

requirements, and the latency, by constructing a more complex schedule. But in this case the space saved by using smaller buffers is offset by the increased space required to store the schedule.

A `SchedulingOptions` instance can be passed to the `StreamProcessor` constructor. This can be used to alter the behavior of the scheduler, for example requesting the computed schedule be displayed prior to execution.

```
SchedulingOptions options = new SchedulingOptions { DisplaySchedule = true };
StreamProcessor<double, double> sp =
  new StreamProcessor<double, double>(..., options);
```

The current runtime implements both a minimum size scheduler and a minimum latency one. At present the minimum latency scheduler is used as the default, but the `SchedulingOptions` class will eventually allow other schedulers to be selected. The schedule discussed so far is a steady-state schedule. The scheduler also constructs an initial schedule which is executed once prior to the steady-state schedule. In some cases this schedule will be empty, but in other examples, particularly where peeking is involved, an initialization schedule is required to push data into some of the interior buffers.

Note that given a static schedule we can compute precisely how big each internal buffer will need to be to support this schedule. We can therefore allocate these buffers with fixed sizes. Furthermore, during the execution of the schedule we never have to worry about testing for buffer overflows or underflows. This is partly why executing a synchronous graph can be far more efficient than communicating between synchronous sub-graphs using asynchronous links.

Supporting Features

In this section we describe some of the support classes defined by the S# implementation.

Native Arrays

In a later section we will describe how some of our graphs can be compiled to C++ code for increased efficiency. We can either translate the whole graph, or selected sub-graphs. In a typical application we may therefore want to execute a graph where some components are still in .NET whilst others are native, and we need to pass data between them. If our buffers were .NET types then this would create problems. The garbage collector might asynchronously move a buffer whilst unmanaged code was executing, leading to a major error. We could pin the buffers to stop such movement, but this would have a serious impact on the garbage collector and runtime design. To avoid this problem S# uses a `NativeArray` type that consists of a .NET wrapper for data held in the unmanaged heap. It is defined in the `Agilent.Streaming.Support` namespace. When the .NET runtime garbage-collects such an array the unmanaged data on the heap is also released. But as long as the .NET code maintains a reference to this array we can safely let unmanaged code access the unmanaged data for the array without risk of it moving.

In principle designing such a type such is simple. However, we want it to be a generic type, and these do not interact well with the pointer types we need to reference the unmanaged heap. Internally the implementation must therefore generate private classes on the fly for each different native array type we use. Fortunately most of this complexity is hidden from the user, but it does prevent us using a simple constructor. Here is how we can create and use a native array.

```
NativeArray<int> array = NativeArray<int>.Create(10);
array[1] = array[0] * 2;
```

In most cases the system automatically creates the internal buffers that connect the graph components together, and so uses of this class are hidden from the users. However, suppose we want to pass data into a graph, just as we did earlier. In this case we need to consider the case where the component we are passing the data to is executing as unmanaged code. If we passed the data as a normal C# array then this data would need to be copied to a native array before it could be accessed by our code. The system will take care of such copying, but it is clearly inefficient. If we created our data in a native array then we could pass this array directly to our unmanaged code, avoiding the overhead of copying. Furthermore, as native arrays implement the `IList<T>`, `ICollection<T>`, and `IEnumerable<T>` interfaces most code should be able to use native arrays just as easily as .NET ones. The `StreamProcessor` class defines an overload of the `Execute` method that allows native arrays to be passed as arguments.

Tuples

When passing composite values between graph components it can be convenient in places to use a simple tuple type rather than go to the expense of defining a **`struct`** type that might be used in only one place. While .NET already provides a generic tuple type, this type is defined as a class which prevents us from using it as an S# type. The `Agilent.Streaming.Support` namespace defines a family of `StructTuple` generic structs that can be passed between S# filters.

Fixed-point Types

One of the goals for S# is to be able to generate code that can be fed to a tool such as AutoESL. As part of some exploratory work in this direction some examples were implemented in S# that required fixed-point arithmetic. Unfortunately .NET does not provide an implementation of such types, other than the Decimal type, and so a family of fixed-point types had to be created. They are defined as structs, allowing them to be passed between filters. Of course there are a large number of fixed point types we might want, depending on the number of bits required and the number of bits forming the integer component of the type. We clearly can't enumerate all possibilities, predefining a type for each one. The S# installer adds a file called fpg.exe to your S# directory. It takes three arguments, the bit width, the number of bits for the integer part, and a character indicating whether a signed or unsigned type is required. The result is a DLL that provides the definition of a fixed-point type with these properties that can be referenced by your project. For example,

```
"%SS_DIR%\fpg.exe" -s 5 2    generates FixedPoint5x2.dll
"%SS_DIR%\fpg.exe" -u 8 1    generates UFixedPoint8x1.dll
```

Note that as a result of these being struct types there is no inheritance structure defined on them, and no common "fixed-point" base type. There is also an `apg.exe` program that constructs arbitrary width integers.

Operators

Suppose we wanted to write a filter than popped two values, added them together, and pushed the result. Defining this for a particular type, such as `double`, would be trivial. But what if you wanted to define a generic version of this filter? C# does not support any form of `IArithmetic` interface that could be used, and attempting to add two values of some generic type T will fail. The `Operator` class can be used in such cases.

```
public class Add<T> : Filter<T, T> where T : struct {
  public Add() : base(pop: 2, push: 1) { }

  protected override void Work() {
    Push(Operator.Add(Peek(0), Peek(1)));
    Pop(); Pop();
  }
}
```

The class generates code at runtime for each type it is applied to, leading to reasonably efficient performance. Furthermore, when an instance of a class such as `Add` is compiled to native code the overhead of using the `Operator` class is completely eliminated.

Graph Properties

Another goal of S# is to be able to generate components that could be imported into something like a SystemVue model. SystemVue allows a model to have parameters that can alter the behavior of a model. At first glance we might think that all we'd need to do in S# is to allow a filter to have a mutable public property to achieve the same goal. However, allowing such parameters to change during the execution of a graph would introduce many of the problems associated with parallel execution we were trying to avoid. If we used such a parameter to define a rate then we would also need to ensure it couldn't change without recomputing the schedule. To further complicate matters, in many cases a graph will be compiled to native code, and so changing a parameter would require altering state within unmanaged code. If there are multiple instances of a filter in a graph then we must also pick which one we want to change.

The current S# implementation provides some experimental support for parameters. To make a field into an S# parameter you must annotate it with the [Parameter] attribute, as shown in this example.

```
public class Amplify : Filter<double, double> {
  [Parameter]
  double scale = 1.0;

  public Amplify(double scale) : base(pop: 1, push: 1) { this.scale = scale; }
  protected override void Work() { Push(Pop() * scale); }
}
```

When a stream processor is constructed it collects information about all the parameters that appear in the graph, including those in native code. The program can query this information and alter the parameter values, but only when the graph is not in use.

```
IParameter[] scales = sp.Parameters.FindParameter("scale");
IParameter<double> scale = (IParameter<double>)scales[0];
double d = scale.Value;
scale.SetValue(d * 2.0);
```

Compilation

Executing a stream graph, even in the case of a synchronous graph with a static schedule, will be relatively slow. The overheads of interpreting the schedule, the cost of passing data between filters, and the lack of cross-filter optimizations, make it difficult to compete with hand-written C code. Ideally we would like to automatically replace a graph, or selected subgraphs, by native code that performs the same task. We can use the same decompilation techniques as used by the verifier to disassemble the compiled graph, optimize it, and then generate native code for it. However, whilst the technology exists, all this takes time. Furthermore, we cannot assume we will always have a C++ compiler available at runtime. To address this issue we introduce *graph components* and *graph caching*.

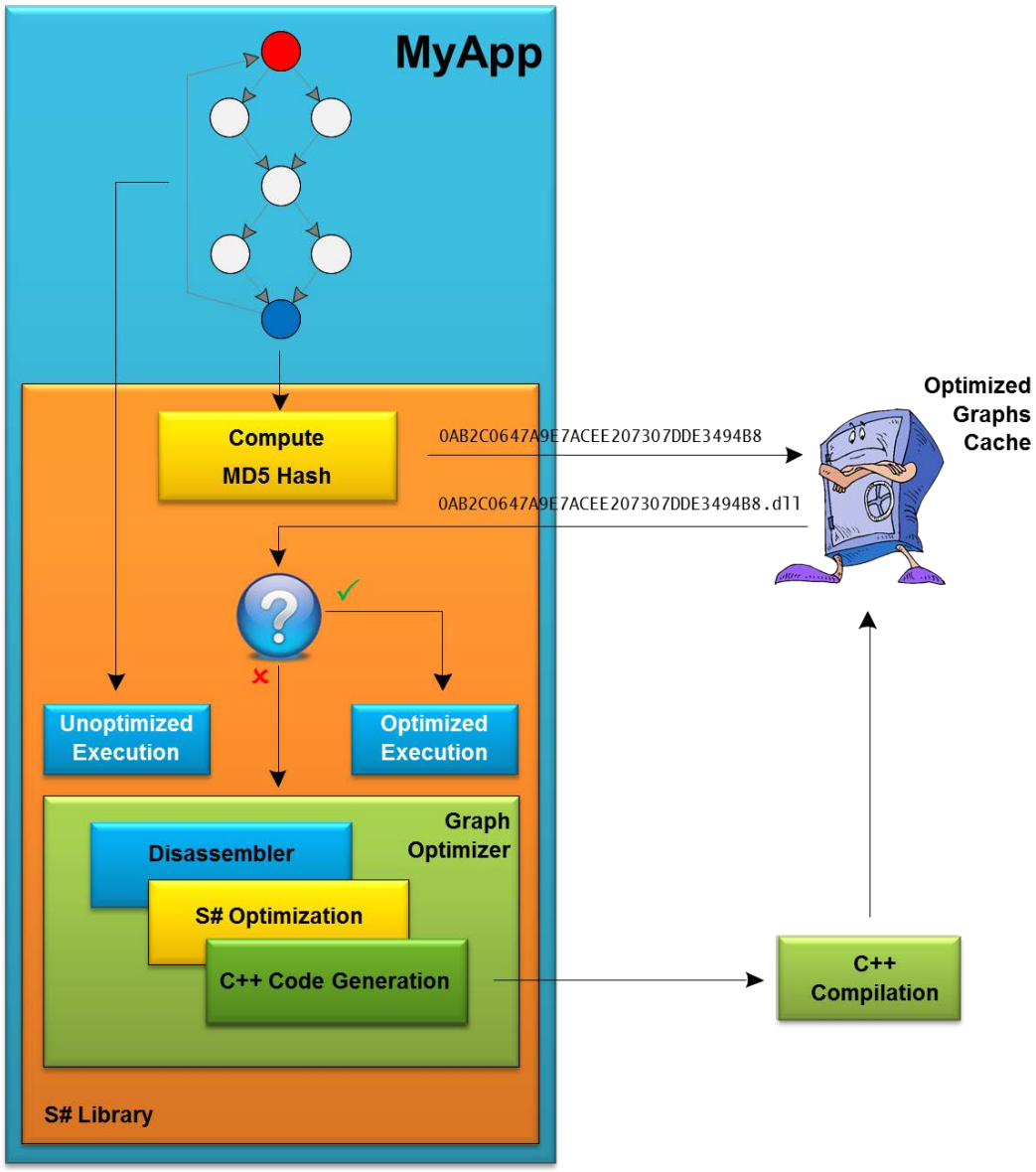
Even in an environment where each execution of the program produces a different graph, or where the graph structure is altered due to user interactions, we assume that there will be substantial portions of the graph that remain stable. In other words we view dynamic graphs as being built from a collection of stable graph building blocks. In many applications the entire graph may be viewed as a stable graph. If a graph is stable then it is worth investing computational resources to convert it to more efficient native code. But how do we identify which subgraphs are stable? It is difficult to do this without user assistance, although in the long term certain kinds of profiling might be able to provide this information. For now though we introduce a new class, `GraphComponent`, that can be used to identify stable graphs.

```
StreamGraph<double, double> p =
  StreamGraph.Pipeline(new Average(), new Add<double>(), new Amplify(2.0));

GraphComponent<double, double> gc = new GraphComponent<double, double>(p);
```

A graph component will typically contain a DLL encapsulating the native equivalent of the graph passed as argument. To use a graph component in a larger graph we must create a `GraphInstance` from it using the expression `gc.CreateInstance()`.

If we needed to generate a native DLL every time we created a graph component from a stream graph then the performance would be very slow, even if we could guarantee the availability of a C++ compiler. Fortunately the graph component uses a caching mechanism to avoid this overhead. When a graph is made into a graph component we first compute a hash for the graph. A graph cache is consulted to see if this graph has been seen previously. If it has then the DLL that was previously generated for this graph is simply loaded into the application, a fast operation. Of course a cache miss still requires a relatively expensive optimization and compilation step, but in many cases we can avoid this overhead completely at runtime by prepopulating the cache.



The details of how to use the S# compiler to generate x86 native code and FPGA code need to be written here.

Missing Features

Needs to be written.

Code is sequential at present, except for where variable-rate filters force the graph into a collection of loosely coupled subgraphs. We need to implement many of the StreamIt optimizations involved in altering a graph into a form where it can be efficiently executed in parallel.

We need to generate code for additional target platforms, e.g. GPUs.

One feature that is currently lacking is an analogue to StreamIt's teleport messaging mechanism.

Larger Examples

We have included various code fragments illustrating the use of S# throughout this manual. The installer also creates a library folder in the install directory containing some general-purpose filters and subgraphs you may find useful in your designs. Hopefully this collection will grow over time; feel free to send additions to the author. The library is also provided as a precompiled assembly you can reference in your applications. There is a strong connection between S# and StreamIt, and so the StreamIt benchmark suite is useful for illustrating this style of programming. It can be found here: http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.

We conclude this manual with a couple of larger examples.

Streaming FFT

The following code implements a streaming FFT. The outputs are in bit-reverse order.

```
abstract class Stage<T> : Filter<Array2<Complex<T>>, Array2<Complex<T>>> where T : struct {
  protected readonly int fftLength;
  protected readonly int stage;
  protected readonly int lrssp1; // fftLength >> (stage + 1)
  protected readonly int lrssp2; // fftLength >> (stage + 2)

  public Stage(int fftLength, int stage)
    : base(pop: 1, push: 1) {
    this.fftLength = fftLength;
    this.stage = stage;
    lrssp1 = fftLength >> (stage + 1);
    lrssp2 = fftLength >> (stage + 2);
  }
}


class FirstStage<T> : Stage<T> where T : struct {
  private readonly Complex<T>[] twiddles;
  private int modCount = 0;

  public FirstStage(int fftLength, Complex<T>[] twiddles)
    : base(fftLength, 0) {
    this.twiddles = twiddles;
  }

  protected override void Work() {
    Array2<Complex<T>> item = Pop();
    Complex<T> c0 = item.item0 + item.item1;
    Complex<T> c1 = item.item0 - item.item1;

    int angle = modCount & (lrssp1 - 1);

    Complex<T> tw =
      (angle > lrssp2)
      ? new Complex<T>(Operator.Negate(twiddles[lrssp1-angle].real), twiddles[lrssp1-angle].imag)
      : twiddles[angle];

    Push(new Array2<Complex<T>>(c0, c1 * tw));
    modCount++; if (modCount == fftLength) modCount = 0;
  }
}
```

```
class IntermediateStage<T> : Stage<T> where T : struct {
  private readonly Complex<T>[] twiddles;
  private readonly Complex<T>[] bufferD;
  private readonly Complex<T>[] bufferI;
  private int bufferPointer;
  private int modCount = 0;

  public IntermediateStage(int fftLength, int stage, Complex<T>[] twiddles)
    : base(fftLength, stage) {
    this.twiddles = twiddles;
    bufferD = new Complex<T>[lrssp1];
    bufferI = new Complex<T>[lrssp1];
  }

  protected override void Work() {
    Array2<Complex<T>> item = Pop();
    Complex<T> tmp = bufferI[bufferPointer];
    bufferI[bufferPointer] = item.item1;

    if ((modCount & lrssp1) != 0) {
      item.item1 = item.item0;
      item.item0 = tmp;
    } else {
      item.item1 = tmp;
    }

    tmp = bufferD[bufferPointer];
    bufferD[bufferPointer] = item.item0;

    Complex<T> c0 = tmp + item.item1;
    Complex<T> c1 = tmp - item.item1;

    int angle = modCount & (lrssp1 - 1);
    int cangle = (stage % 2 == 0) ? angle : angle << 1;
    int tc = (stage % 2 == 0) ? lrssp1 : lrssp1 << 1;

    Complex<T> tw =
      (angle > lrssp2)
      ? new Complex<T>(Operator.Negate(twiddles[tc - cangle].real), twiddles[tc - cangle].imag)
      : twiddles[cangle];

    Push(new Array2<Complex<T>>(c0, c1 * tw));

    modCount++; if (modCount == fftLength) modCount = 0;
    bufferPointer++; if (bufferPointer == bufferD.Length) bufferPointer = 0;
  }
}


class LastStage<T> : Stage<T> where T : struct {
  private Complex<T> bufferD;
  private Complex<T> bufferI;
  private bool odd = false;

  public LastStage(int fftLength)
    : base(fftLength, (int)(Math.Log(fftLength, 2)) - 1) {
  }

  protected override void Work() {
    Array2<Complex<T>> item = Pop();
```

```
    Complex<T> tmp = bufferI;
    bufferI = item.item1;

    if (odd) {
      item.item1 = item.item0;
      item.item0 = tmp;
    } else {
      item.item1 = tmp;
    }

    odd = !odd;
    tmp = bufferD;
    bufferD = item.item0;

    Complex<T> c0 = tmp + item.item1;
    Complex<T> c1 = tmp - item.item1;

    Push(new Array2<Complex<T>>(c0, c1));
  }
}


public class StreamingFFT<T> : Pipeline<Array2<Complex<T>>, Array2<Complex<T>>> where T : struct {
  private Complex<T>[] ComputeTwiddles(int stage, int fftLength) {
    int numTwid = fftLength >> (stage + 1);
    Complex<T>[] twiddles = new Complex<T>[1 + numTwid / 2];
    twiddles[0] = new Complex<T>(Operator.Convert<double, T>(1.0),
                                 Operator.Convert<double, T>(0.0));
    for (int i = 1; i < twiddles.Length; ++i) {
      double alpha = -i * Math.PI / numTwid;
      twiddles[i] = new Complex<T>(Operator.Convert<double, T>(Math.Cos(alpha)),
                                   Operator.Convert<double, T>(Math.Sin(alpha)));
    }
    return twiddles;
  }

  public StreamingFFT(int fftLength) {
    // Only works for a power of 2 and >= 4
    Contract.Assert((fftLength > 4) && ((fftLength & (~fftLength + 1)) == fftLength));

    int stages = (int)Math.Log(fftLength, 2);

    Complex<T>[] twiddles = ComputeTwiddles(0, fftLength);

    Add(new FirstStage<T>(fftLength, twiddles));
    for (int i = 1; i < stages - 1; ++i) {
      if (i % 2 == 0) twiddles = ComputeTwiddles(i, fftLength);
      Add(new IntermediateStage<T>(fftLength, i, twiddles));
    }
    Add(new LastStage<T>(fftLength));
  }
}
```

Short-read Gene Sequence Matching

The following example demonstrates how S# might be useful in more diverse areas than traditional DSP problems. The task is to match a collection of short DNA sequences against a long reference sequence. For each short-read we look for the best alignments of the sequence against the reference. Matching does not have to be exact; there is a cost for inserting a base-pair, removing one, or transforming one, and we are searching for those locations in the reference sequence where the cost of transforming the short-read to the reference at that location is below some threshold.

```
enum BasePair { A, C, G, T }


class Sequence : List<BasePair> { … }


struct State {
  public BasePair basePair;
  public MatchCost cost;
  public MatchCost previousCost;
  …
}


struct Match {
  public uint offset;
  public MatchCost cost;
  …
}


class PE : Filter<State, State> {
  private readonly uint shortReadIndex;
  private readonly uint shortReadLength;
  private readonly uint referenceLength;

  private uint position;
  private BasePair shortReadBasePair;
  private BasePair lastBasePair;
  private MatchCost lastCost;
  private MatchCost lastLastCost;

  // Penalty for deletion
  readonly static MatchCost Alpha = 1;

  // Penalty for insertion
  readonly static MatchCost Beta = 1;

  [Inline]
  // Penalty for a base-pair mismatch
  static MatchCost Gamma(BasePair s, BasePair r) {
    if (s == r) return 0; else return 2;
  }

  public PE(uint index, uint shortReadLength, uint referenceLength)
    : base(pop: 1, push: 1) {
    this.shortReadIndex = index;
    this.shortReadLength = shortReadLength;
```

```
    this.referenceLength = referenceLength;
    this.lastCost = Alpha;
    this.lastLastCost = lastCost + Alpha;
    this.position = index == 0 ? 0 : shortReadLength + referenceLength - index;
    this.lastBasePair = 0;
  }

  protected override void Work() {
    Push(new State(lastBasePair, lastCost, lastLastCost));
    State upstream = Pop();

    if (position >= shortReadLength) {
      MatchCost gamma = Gamma(shortReadBasePair, upstream.basePair);
      MatchCost diag = upstream.previousCost + gamma;
      MatchCost left = lastCost + Alpha;
      MatchCost up = upstream.cost + Beta;
      MatchCost cost = up < left ? (up < diag ? up : diag) : (left < diag ? left : diag);
      lastLastCost = lastCost;
      lastCost = cost;

    } else {
      // Receiving the next short sequence
      if (position == shortReadIndex) {
        shortReadBasePair = upstream.basePair;
        lastCost = (shortReadIndex + 1) * Alpha;
        lastLastCost = lastCost + Alpha;
      }
    }
    lastBasePair = upstream.basePair;
    var end = shortReadLength + referenceLength - 1;
    position = (position == end) ? 0 : position + 1;
  }
}


class Matcher : Pipeline<State, State> {
  public Matcher(uint shortLen, uint refLen) {
    for (uint i = 0; i < shortReadLen; ++i) {
      Add(new PE(i, shortLen, refLen));
    }
  }
}


class Threshold : Filter<State, Match> {
  private readonly MatchCost threshold;
  private readonly uint shortReadLength;
  private readonly uint totalLength;
  private uint position;
  private bool valid = false;

  public Threshold(MatchCost threshold, uint shortReadLength, uint referenceLength)
    : base(pop: 1, push: null) { // Variable output rate
    this.threshold = threshold;
    this.shortReadLength = shortReadLength;
    this.totalLength = shortReadLength + referenceLength;
    this.position = referenceLength;
```

```
  }

  protected override void Work() {
    State b = Pop();
    if (valid) {
      if (position == 0) {
        Push(new Match(uint.MaxValue, 0));
      } else if (position > shortReadLength) {
        var referencePos = position - shortReadLength;
        if (b.cost <= threshold &&  referencePos >= (shortReadLength - 1)) {
          var estimatedPos = referencePos - (shortReadLength - 1);
          Push(new Match(estimatedPos, b.cost));
        }
      }
    }
    if (position == totalLength - 1) {
      position = 0;
      valid = true;
    } else
      position++;
  }
}


class LocalMinimum : Filter<Match, Match> {
  private readonly uint gap;
  private readonly uint maxMatches;
  private uint lastOffset;
  private MatchCost localMinimum;
  private uint localMinimumOffset;
  private uint matches = UInt32.MaxValue;

  public LocalMinimum(uint gap, uint maxMatches)
    : base(pop: 1, push: null) {
    this.gap = gap;
    this.maxMatches = maxMatches;
  }

  protected override void Work() {
    Match m = Pop();
    var currentOffset = m.offset;
    var currentCost = m.cost;

    if (currentOffset == uint.MaxValue) { // Terminator
      if (matches == 0) // No match for the last read
        Push(new Match(uint.MaxValue, 0));
      else if (matches > 0 && matches <= maxMatches)
        Push(new Match(localMinimumOffset, localMinimum));
      matches = 0;

    } else {
      if (matches == 0) { // Pending terminator output
        Push(new Match(uint.MaxValue, 0));
        localMinimum = currentCost;
        localMinimumOffset = currentOffset;
        matches = 1;
```

```
      } else if ((currentOffset - lastOffset) > gap){
        if (matches <= maxMatches) {
          Push(new Match(localMinimumOffset, localMinimum));
          matches++;
        }
        localMinimum = currentCost;
        localMinimumOffset = currentOffset;

      } else if (m.cost < localMinimum) {
        localMinimum = currentCost;
        localMinimumOffset = currentOffset;
      }
      lastOffset = currentOffset;
    }
  }
}


class ShortReadMatcher : Pipeline<BasePair, Match> {
  public ShortReadMatcher(uint shortLen, uint refLen,
      MatchCost threshold, uint gap, uint maxMatches){
    Add(StreamGraph.Filter((BasePair bs) => new State(bs, 0, 0)));
    Add(new Matcher(shortLen, refLen));
    Add(new Threshold(threshold, shortLen, refLen));
    Add(new LocalMinimum(gap, maxMatches));
  }
}


class Prelude : Filter<BasePair, BasePair> {
  private readonly uint shortReadPreludeLength;
  private readonly uint shortReadStart;
  private readonly uint shortReadEnd;
  private readonly uint totalLength;
  private uint position = 0;

  public Prelude(uint branchIndex, uint numBranches, uint shortLen, uint refLen)
    : base(pop: 1, push: null) {
    shortReadStart = shortLen * branchIndex;
    shortReadEnd = shortReadStart + shortLen;
    shortReadPreludeLength = shortLen * numBranches;
    totalLength = shortReadPreludeLength + refLen;
  }

  protected override void Work() {
    var popped = Pop();
    if (position >= shortReadPreludeLength
    ||  (position >= shortReadStart && position < shortReadEnd)) {
      Push(popped);
    }
    position = (position == totalLength - 1) ? 0 : position + 1;
  }
}


class MatchJoiner : JoinFromBranchUntil<Match> {
  public MatchJoiner()
```

```
    : base((m) => m.offset == uint.MaxValue) {
  }
}


class ParallelMatcher : SplitJoin<BasePair, Match> {
  public ParallelMatcher(
      uint branches,
      uint maxShortReadLength, uint referenceLength,
      MatchCost threshold, uint gap, uint maxMatches) {

    Splitter = new DuplicateSplitter<BasePair>();

    for (uint i = 0; i < branches; ++i) {
      Add(new Pipeline<BasePair, Match>(
        new Prelude(i, branches, maxShortReadLength, referenceLength),
        new ShortReadMatcher(maxShortReadLength, referenceLength, threshold, gap, maxMatches)));
    }

    Joiner = new MatchJoiner;
  }
}
```