AN INTERNAL DSL FOR BUILDING FASTFLOW SKELETONS USING A C++ FLUENT INTERFACE

Guy Tremblay and Marco Aldinucci

Juin 2015

Département d'informatique

Université du Québec à Montréal

Rapport de recherche Latece 2015-1

Laboratoire de recherche sur les technologies du commerce électronique

AN INTERNAL DSL FOR BUILDING FASTFLOW SKELETONS USING A C++ FLUENT INTERFACE

Guy Tremblay
Département d'informatique
UQAM
Montréal, Qc, Canada

Marco Aldinucci
Dipartimento di Informatica
Università degli Studi di Torino
Torino, Italia

Laboratoire de recherche sur les technologies du commerce électronique
Département d'informatique
Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal, QC, Canada
H3C 3P8
`http://www.latece.uqam.ca`

Juin 2015

Rapport de recherche Latece 2015-1

The work described in this report was performed while Prof. Tremblay was in sabbatical at the Università degli Studi di Torino, during Winter and Spring 2015. Prof. Tremblay thus wishes to acknowledge the warm welcome he received from Prof. Aldinucci and his students.

Summary

This report motivates and describes a new API for building FastFlow skeleton objects—i.e., pipelines, farms, DAGs, etc. It is targeted to FastFlow researchers, so the basic concepts of the FastFlow framework *are not explained*. For details on the FastFlow approach, see for instance M. Torquati's tutorial [8].

Contents

List of Figures

Chapter 1

Introduction

1.1 Aim and scope of this report

The goal of this report is to motivate, describe, and explain a new API for building FastFlow skeleton objects—i.e., pipelines, farms, DAGs, etc.

This report is targeted to FastFlow researchers, so the basic concepts of the FastFlow framework *are not explained*. For details on the FastFlow approach, see for instance M. Torquati's tutorial [8].

The report aims to provide an informal specification for an internal DSL, in C++, for the configuration of FastFlow skeleton objects. It could be used, for example, as a starting point for a student implementation project: the specification is described informally, inspired by a Ruby implementation done by the first author. However, this API is defined in such a way that a C++ implementation should be relatively straigthforward. In fact, many of the proposed elements have already been implemented in C++, although in an *ad hoc* manner.

1.2 The motivation: Building FastFlow skeleton objects is not easy

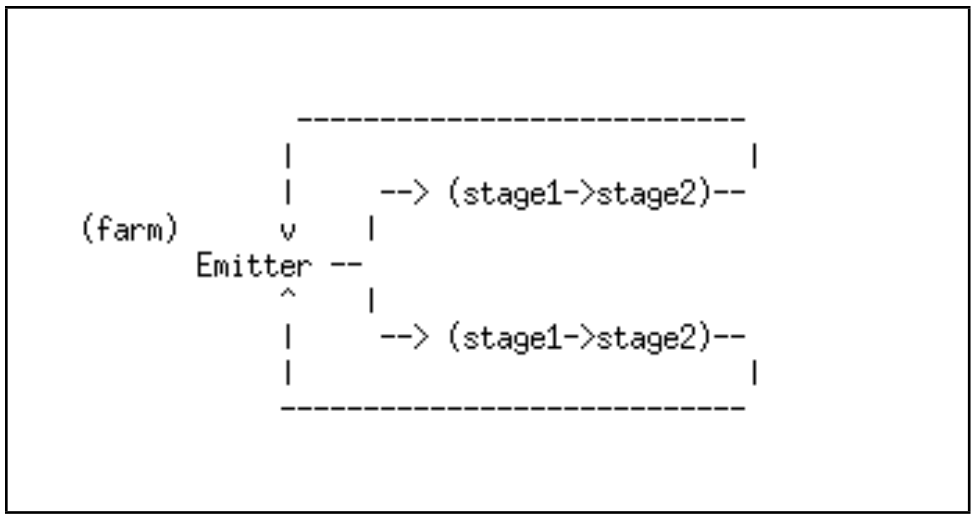

Figure 1.1: A farm composed of pipelined workers with feedback: Graphical representation.

In the current FastFlow implementation, building FastFlow skeleton objects—i.e., instantiating skeletons—is done in a largely imperative manner. Figure 1.1 shows a graphical representation of a farm with two workers and a feedback loop, where each worker in turn is a pipeline consisting of two stages (the figure is taken from one of the test file included in the fastflow source code distribution). Building such a farm is currently done using code similar to the one shown in FastFlow example 1 (p. 3).

FastFlow example 1 A farm composed of (two-stages) pipelined workers with feedback: Current C++ implementation.

```
class Emitter: public ff_node { ... }
class Stage1: public ff_node { ... }
class Stage2: public ff_node { ... }

{
  ...
  ff_farm<> farm;
  farm.add_emitter( new Emitter );

  std::vector<ff_node *> w;
  for( int i = 0; i < nworkers;++i ) {
    ff_pipeline* pipe = new ff_pipeline;
    pipe->add_stage( new Stage1 );
    pipe->add_stage( new Stage2 );
    w.push_back(pipe);
  }
  farm.add_workers(w);
  farm.wrap_around();

  farm.run();
  ...
}
```

Some remarks:

- The imperative approach makes it difficult to have a clear view of the structure of the farm with inner pipelines.
- Adding a sequence of workers is done by explicitly builing (and imperatively using a loop) a vector of workers (using `push_back`) and then passing this vector as argument to the appropriate farm operation (`add_workers`).
- Even for such a simple farm/pipeline definition, various auxiliary variables have to be declared and used.

1.3 An alternative approach: Constructors

One possible solution to the imperative-style is to use constructors. For example, here is an existing function definition for an `ff_farm` constructor taken from the FastFlow library:

```
ff_farm( std::vector<ff_node*>& W,
         ff_node *const Emitter = NULL,
         ff_node *const Collector = NULL,
         bool input_ch = false )
```

Using default values for optional arguments makes it possible to construct objects having various mandatory vs. optional properties. Furthermore, using function overloading, multiple constructors with different argument types can also be defined.

However, some drawbacks of such constructors are the followings:

- When there are many arguments/attributes, the positional order of the parameters is crucial, although not easy to remember, which may lead to errors. For example, since both the `Emitter` and `Collector` arguments are of the same types, passing them in the wrong order—`new ff_farm(workers, co, em)`—would clearly not produce the correct result, but would not even generate any compile-time error.

- Default values can be used for some optional arguments. However, simple default values may not be enough when there are many arguments with non-trivial combinations.

1.4 Another alternative approach: Skeleton builders and *fluent interface*

```
class Emitter: public ff_node { ... }
class Stage1: public ff_node { ... }
class Stage2: public ff_node { ... }

{
  ...
  Build.farm().
    emitter( new Emitter ).
    workers( []{ return Build.pipeline()
                         | new Stage1()
                         | new Stage2(); },
             nworkers ).
    wrap_around().
    done().
    run();
  ...
}
```

Fluent interface example 1: A farm composed of pipelined workers with feedback: A skeleton builder implementation using a C++ fluent interface.

Fluent interface example 1 shows an alternative way, using a more declarative style, of building the farm/pipeline from Figure 1.1. This solution uses an *expression builder*—more precisely, in this case, a *skeleton builder*—based on a C++ *fluent interface*. Such *builders* are used to construct complex objects, managing and encapsulating the required rules and constraints for those objects.

In the following sections, we present what exactly is meant by a "*fluent interface*". We also discuss why this approach is proposed for building FastFlow skeleton objects, instead of using, for example, an independent external DSL.

Chapter 2

Various kinds of DSLs: Pros and cons in the FastFlow context

2.1 What is a Domain-Specific Language

Let us briefly recall some key notions related with Domain-Specific Languages (DSLs).[1]

According to Fowler [4], a DSL is "A computer *programming language* of *limited expressiveness* focused on a *particular domain*." This implies the following:

- A DSL is a *programming language*, i.e., a language used by humans to program computers.
- A DSL may have limited expressiveness in the sense that it may not necessarily be *Turing-complete*—by contrast with a GPL (General Purpose Language) like C, Java, Ruby, etc.
- A DSL is tied to a specific application domain—a specific problem area—so it (generally) uses concepts from that application domain.

According to Voelter, DSLs are targeted to different uses, among which (*emphasis* is ours):

- Application domain DSLs:

  > "[These] DSLs describe the core business logic of an application system independent of its technical implementation. These DSLs are intended *to be used by domain experts, usually non-programmers*. This leads to more stringent requirements regarding notation, ease of use and tool support." [10]

- Utility DSLs:

[1] For more details and examples, see the following presentation: `http://www.labunix.uqam.ca/~tremblay/dsl.pdf`.

> "One use of DSLs is simply *as utilities for developers*. A developer, or a small team of developers, creates a small DSL that automates *a specific, usually well-bounded aspect of software development*. The overall development process is not based on DSLs, it's a few developers being creative and simplifying their own lives." [10]

Languages and tools such as SQL, HTML, make, ant, rake, rspec are examples of utility DSLs, whose users are strictly software developers, not "non-technical business" users. A DSL for FastFlow most definitely will fall into this category.

2.2 Internal DSL vs. External DSL

Another important distinction is the one between external and internal DSL—see also Figure 2.1:

- External DSL: The DSL is *distinct* from the language used to develop the application, so an independent parser/interpreter must be built—using tools such as lexer/parser generator (e.g., lex/yacc, flex/bison, antlr) or language workbenches (e.g., Eclipse's xText[2]).
- Internal DSL: The DSL is a "sub-language" of the language used to develop the application, i.e., it "uses" the infrastructure of an existing programming language (called the host language). In other words, it is implemented "*on top of*" an existing programming language, so it must obey the constraints (syntax, semantics) of the host language.

[2] `http://eclipse.org/xtext/`

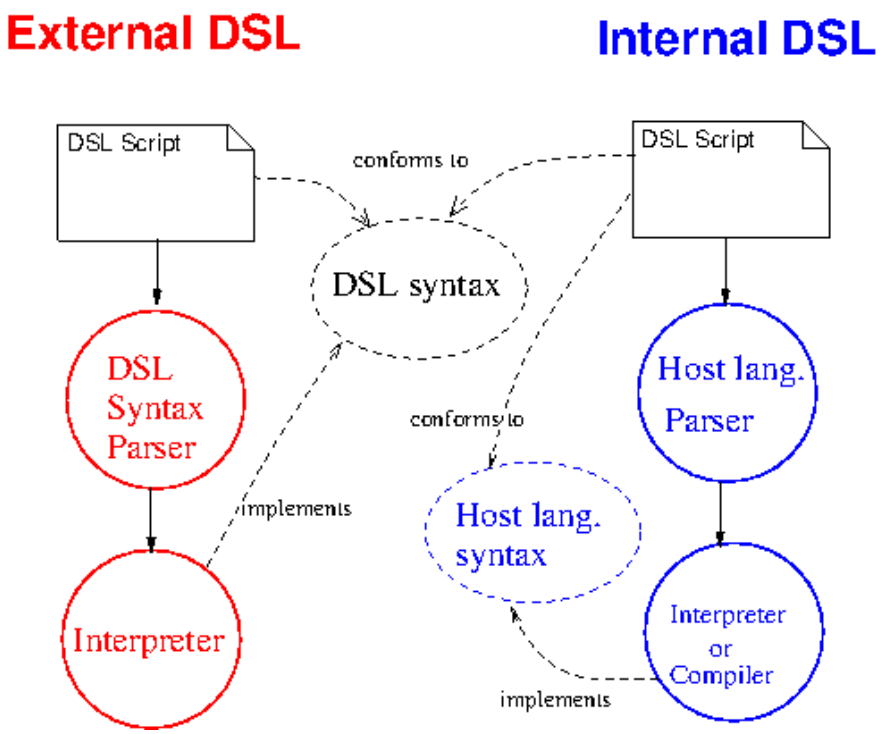

Figure 2.1: External vs. internal DSL.

2.3 Fluent interface API as internal DSL

As mentioned earlier, a utility DSL is used by programmers to address a specific aspect of software development. Such a DSL is thus, in essence, an API—an Application Programming Interface.

Fowler distinguishes between an internal DSL and a regular API as follows:

> "[By contrast with a regular API], an internal DSL should *have the feel of putting together whole sentences*, rather than a sequence of disconnected commands." [4]

For such an API, Fowler and Evans (quoted in Wikipedia) coined the term *fluent interface*:

> "A *fluent interface* is a way of implementing an object oriented API in a way that aims to provide for more readable code."[3]

Fowler further describes a fluent interface as follows:

> "APIs are usually designed to provide a set of self-standing methods on objects. Ideally, these methods can be understood individually. I call this style of API a command-query API [...]. DSLs require a different kind of API, what I

[3] `http://en.wikipedia.org/wiki/Fluent_interface`

> call a *fluent interface*, which is designed with the goal of readability of a whole expression. *Fluent interfaces lead to methods that make little sense individually, and often violate the rules for good command-query APIs.*" [4]

A key pattern described by Fowler for providing a fluent interface is the *Expression Builder*:[4]

> *Expression Builder* An object, or family of objects, that provides a fluent interface over a normal command-query API.

In many languages, including those with little support for meta-programming, a typical technique used to define such builders is *method chaining*:[5]

> *Method Chaining* Make modifier methods return the host object [(i.e., `self`, `this`, etc.], so that multiple modifiers can be invoked in a single expression.

2.4 Brief comparison of various approaches for FastFlow configuration DSL

From the discussions that took place during Winter and Spring 2015, two of the key requirements that were identified for FastFlow were the following:

- Performance is crucial.
- The use of C++ for task processing nodes is a must.

In this section, we briefly present and compare some approaches that we examined to define a DSL for skeleton configuration in FastFlow, keeping in mind those two requirements.

Internal Ruby DSL

The work that initially inspired the interest for a FastFlow DSL was, in part, the Ruby implementation of FastFlow developed by the first author.[6] Not all of FastFlow was implemented, although most of the core concepts were. Because of Ruby's expressiveness and flexible syntax, this led to simple and elegant FastFlow programs.

So, one of the first approach that we examined was to define an internal Ruby DSL that would express the configuration of FastFlow skeleton objects.

The key issue with this approach was finding a way to make two different worlds work together: the Ruby world—in which the configuration would be described—and the C++ world—in which the code for the effective computation would be written, compiled and executed.

[4] `http://martinfowler.com/dslCatalog/expressionBuilder.html`
[5] `http://martinfowler.com/dslCatalog/methodChaining.html`
[6] `http://www.labunix.uqam.ca/~tremblay/seminaire-ruby-ff.pdf`

In Ruby, it is possible to define "extensions," thus allowing Ruby to use external libraries written in other languages. For the C language, this is done using various header files provided by the Ruby environement—the original Ruby implementation, called MRI, is written in C. This can also be done using various gems (e.g., the `FFI` gem = Foreign Function Interface) that provide a high-level interface to external languages. One such gem, `Rice`, is in fact specifically targeted to C++.

However, a key constraint for these extensions is that the C/C++ library that needs to be called from within Ruby has to be provided as a *dynamic* library.[7] But the C++ elements that need to be integrated together to perform the real computation must be compiled by the C++ compiler.

So, having a Ruby DSL to configure the skeletons would have required a process somewhat along the following lines:

- The user writes the code for the computation (FastFlow task processing node) in C++ and compiles them.
- The user writes his skeleton configuration code in Ruby. He executes this configuration script, whose task is to create the skeleton objects using the appropriate Ruby/C++ library extensions.
- The user links all the object code together and executes the resulting program.

The last part could also have been executed from the Ruby script. However, dealing with such *heterogeneous multi-language levels* generally makes it quite difficult to identify the source of errors—compile-time or run-time errors work.

External DSL

A purely external DSL would have suffered from the same problems as an internal Ruby DSL mentioned previously. Furthermore, it would also have made it necessary to define and build a full-fledge parser *along with an interpreter*.

Internal C++ DSL

Given the two key requirements mentioned earlier, using an internal DSL in C++ seemed the best approach. Although meta-programming is possible in C++—using *template meta-programming* [1, 9]—, such an approach is far from trivial. A more straigthforward approach, easier to understand and implement, is the use of a *fluent interface*. Such an approach makes it possible to remain exclusively within the C++

[7] Although it might be possible to use a *static* library, Rice specifically discourages it.

realm, avoiding dealing with multiple languages—Babel tower problems.

2.5 An example illustrating a skeleton builder fluent interface: Use and implementation

Fluent interface example 1 is used to illustrate the concepts introduced in the previous sub-sections. Here is the key part of that example:

```
Build.farm().
  emitter( new Emitter ).
  workers( []{ return Build.pipeline()
                      | new Stage1()
                      | new Stage2(); },
           nworkers ).
  wrap_around().
  done().
  run();
```

The call to `Build.farm()` returns a `FarmBuilder` object. This object is used to incrementally construct a farm object, using methods such as `emitter()`, `workers()`, `wrap_around()`, etc. Once all the object properties have been specified, the farm can be effectively constructed using the `done()` method. Whereas the previous methods all returned the `FarmBuilder` object being built, this last method returns a "real" `Farm` object, which can then be `run()`.

Fluent interface example 2 presents a (short) excerpt of the Ruby implementation of the `FarmBuilder` class. Here are some remarks to explain how expression building with method chaining works:[8]

- The `FarmBuilder` object keeps track, through instance variables,[9] of the various elements or properties of the farm being built.
- A method such as `emitter()` "takes note" of a single property, in this case, the emitter to be used. This method is thus a *setter*, i.e., it is a command that modifies the internal state of the `FarmBuilder` object. In a typical command–query API,[10] that command would not return anything. Here, to be used through a fluent interface, `emitter()` instead returns `self`, the current `FarmBuilder` object. Returning `self` then allows subsequent calls to the object's methods *to be chained together*—thus the name "*method chaining*."

[8] In typical Ruby style, `return`s are generally omitted when not absolutely required. Here, `return`s have been used explicitly to ease the understanding of the code for C++readers. Similarly, in Ruby, empty parentheses are usually omitted, but in this case they have added as in C++.

[9] In Ruby, instance variables are prefixed with "`@`", to distinguish them from local variables.

[10] In a command–query API, a good practice suggested by various authors [7], a method is either a command—it modifies the state of an object *without returning anything*—or a query—it observes the state of an object, *without modifying it*, and returns some value.

```
class FarmBuilder
  def initialize
    @emitter = nil
    @workers = []
    @collector = nil
    @wrap_around = false
    @no_collector = false
  end

  def emitter( e )
    @emitter = e

    return self
  end

  def collector( c )
    @collector = c

    return self
  end

  ...

  def done()
    col =  (@no_collector && @wrap_around) ? :none : @collector
    f = Farm.create( @emitter, @workers, col )

    f = f.wrap_around() if @wrap_around

    return f
  end
end
```

Fluent interface example 2: Excerpt from Ruby `FarmBuilder` implementation.

- The `done()` method is the one that effectively builds the required object. It is this method that "*knows*" about the lower-level API used to allocate and construct objects—e.g., it knows the correct and appropriate order to be used for the "real" constructor.

- In the example above, the `done()` method has a simple and naïve implementation. However, a more realistic implementation might do various validations, e.g., ensure that the specified properties are valid when combined together, that all required properties have been specified, etc.

- Using a fluent interface with method chaining, it is even possible to impose constraints *on the order in which some methods have to be called*, e.g., method `bar` can be called only if immediately preceded by a call to method `foo`. All such rules can be implemented in the appropriate `Builder` object, by adding additional properties in the object.

Chapter 3

The domain of FastFlow skeletons: A (partial) meta-model

Before defining a DSL, it is necessary to better identify the domain concepts that need to be expressed by the DSL. In our case, we want to define configurations of FastFlow skeleton objects.

3.1 Feature models are not expressive enough

One possible approach to define the various alternatives that compose a software product is to use *feature models*. Such models indicate the elements that are mandatory vs. optional, or sets of optional elements that are inclusive (or) or mutually exclusive (xor).[1]

However, feature models aim at defining products with a *finite sets* of features—i.e., finite sets of configurations. Thus, feature models generally do not support—at least in their basic, better known form—recursive definitions. But, FastFlow skeleton objects can in fact be recursively defined composite objects: a pipeline could be composed of various stages, and a stage might well be a pipeline itself or a farm, and a farm's workers could themselves be pipelines, etc. Thus, the number of possible configurations *is not finite*, so feature models do not seem appropriate.

3.2 UML meta-model as abstract syntax

The UML diagram of Figure 3.1 is a simplified and partial *meta-model* for skeletons. Whereas a specific skeleton configuration can be *represented by* an UML object model (with specific instances), the overall structure and *family* of skeletons can be represented by a meta-model. A specific model (a specific skeleton instance) then *conforms to* the meta-model (the "syntax")—see Figure 3.2.

[1] `http://en.wikipedia.org/wiki/Feature_model`

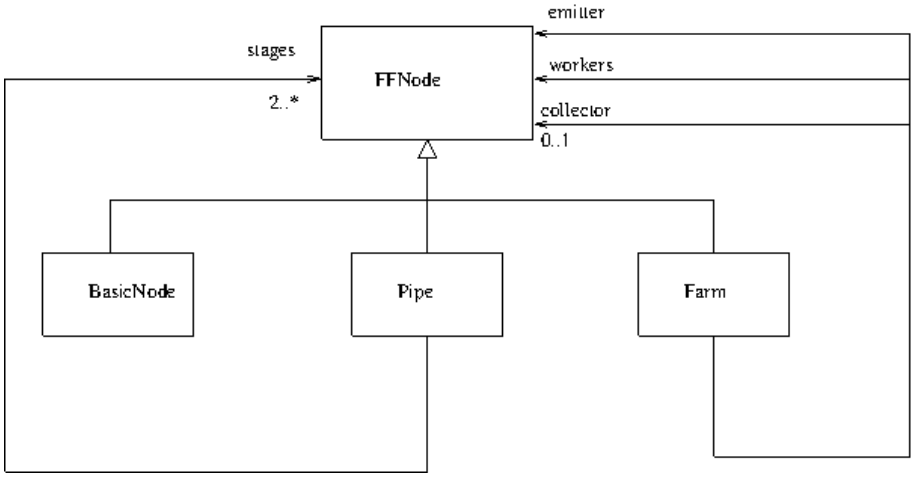

Figure 3.1: A simplified (partial) meta-model for FastFlow skeletons: Additional details showing the recursive nature of composite objects.

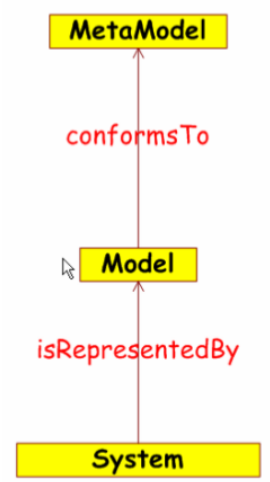

Figure 3.2: Model vs. Meta-model (taken from [2]).

More precisely, the metamodel specifies the *abstract syntax* of the configuration language. It is a representation of all the models—i.e., in our case, all the configurations—that can be expressed.

When defining a programming language, it is customary to define both its *abstract syntax* as well as its *concrete syntax*:

> *Abstract syntax* is used to give a high-level description of programs being compiled [...]. The abstract syntax specifies the [syntax] tree's structure specified in terms of categories such as "statement", "expression" and "identifier." This is independent of the *source syntax* (*concrete syntax*) of the language being compiled[...].
>
> `http://dictionary.reference.com/browse/abstract+syntax`

However, in what follows, we are aiming to describe a fluent interface for FastFlow in C++. The concrete (source) syntax is thus already well specified—already "cast in stone," since the concrete syntax is *C++'s syntax*! This is why, in the following section, we directly describe the concrete syntax of the proposed fluent interface withouth providing an abstract syntax.

Chapter 4

The proposed approach: A C++ skeleton builders fluent interface

We now present the proposed fluent interface for building FastFlow skeleton objects. As mentioned earlier, we present the *concrete syntax*. This concrete syntax is based on the *Ruby syntax* used to implement the core elements of FastFlow.[1]

The API is defined in such a way that it should be easily translatable to C++, using basic templates (without complex template meta-programming), function overloading, default values for arguments, etc. This should be possible because many Ruby features—e.g., dynamic dispatch based on types, keyword parameters, optional parentheses in function calls, etc.—have been avoided while defining this fluent interface.

[1] `http://www.labunix.uqam.ca/~tremblay/seminaire-ruby-ff.pdf`

4.1 The notation

Here are the key elements of the notation used in the following to define the concrete syntax of the C++ fluent interface:

- `'foo'`: A terminal symbol, i.e., a piece of text that must appear "as such," i.e., textually.
- *`foo`*: A non-terminal symbol, whose definition is given afterwards, usually by indicating the possible *types* allowed for such values—in italics, e.g., *`ff_node`* | *`lambda`*—or the allowed constant values—in upper case, e.g., `ALL` | `ONE`.
- [*`element`*]$^{?}$: The indicated element is optional, but if it appears it must necessarily appear at this specific position.
- {*`element`*}$^{?}$: The indicated element is optional, and it can appear *anywhere* in the sequences of method calls.
- [*`element`*]$^{+}$: The indicated element can occur one or more times, and must necessarily appear at this specific position.
- {*`element`*}$^{+}$: The indicated element can occur one or more times, and can appear anywhere in the sequences of method calls.
- [*`element`*]*: The indicated element can occur 0, 1 or more times, and must necessarily appear at this specific position.
- {*`element`*}*: The indicated element can occur 0, 1 or more times, and can appear anywhere in the sequences of method calls.
- *`elem1`* | *`elem2`*: One of *`elem1`* or *`elem2`* must appear.
- $\underline{\{}$*`element`*$\underline{\}}$: The indicated element must necessarily appear at this <u>final</u> position.

4.2 Pipeline

`PipelineBuilder`, whose result produces a `pipeline` object (an `ff_node`), obeys Protocol 1.

```
'PipelineBuilder'
  '.new()'
  { '.buffer(' size ')' }?
  { '.window(' size ')' }?
  { '.with(' stage ')'
    [ '.buffer(' size ')' ]?
    [ '.window(' size ')' ]?
  }*
  { '.with(' task_lambda [',' nb_instances ]? ')'
    [ '.buffer(' size ')' ]?
    [ '.window(' size ')' ]?
  }*
  { '.wrap_around()' }?
  { '.done()' | '|' stage }

stage: ff_node | task_lambda

size: int
nb_instances: int
```

Protocol 1: Protocol for `PipelineBuilder`.

Notes:

- Possible *aliases* for `'with'`: `'followed_by'`, `'stage'`, `'<<'`.
- A call to `buffer()` at the "top-level" is used to specify the default buffer size, i.e., for the connections that have no explicit buffer size specified. (A size of 0 denotes an *unbounded buffer*.)

  A call to `buffer()` immediately after a `with()` specifies the buffer size to be used between the preceding stage and the stage being added—thus, such a call *should not appear immediately* after the first stage.
- Idem for `window()`, except this is used to specify the window size accessible for reading.

  So, given a call to `window(`k`)` and a reference `node` representing the node that is connected to that buffer, the following elements will then be accessible:

  – `node->peek(0)`

    This is a synonym for the `task*` received as argument.

  - `node->peek(`j`)` for $j = 1, \ldots, k-1$.
    In this case, if the returned value is `NULL`, then it means no appropriate element has still been received in the buffer—i.e., `peek` is non-blocking.

  Possible *aliases* for `'peek'`: `'buffer'`, `'input'`, `'data'`.

- A `task_lambda` is a `lambda` that receives a `task*` *and* a `ff_node*` as arguments. It can then be used to instantiate a `ff_node`, whose `svc` method will be `task_lambda`.

- A call such as "`with(` *`a_lambda,`* k `)`" create a serie of parallel stages, i.e., a simple farm stage with k workers.

- The use of `done()` could be omitted if the resulting pipeline object is not used in another pipeline but is run directly. In other words, given `pb` a `PipelineBuilder`, "`pb.done().run()`" could be simplified as "`pb.run()`". This would simply require `pb.run()` to be implemented as "`done().run()`". A similar remark holds for the other types of builders.

4.3 Farm

`FarmBuilder`, whose result produces a `farm` object (an `ff_node`), obeys Protocol 2.

```
'FarmBuilder'
  '.new()'
  { '.emitter(' emitter ')' }?
  { '.collector(' collector ')' | '.no_collector()' }?
  { '.worker(' worker ')' }*
  { '.workers(' worker [',' worker]+ ')' }*
  { '.workers(' workers ')' }*
  { '.workers(' worker_generator ',' nb_instances ')' }*
  { '.wrap_around()' }?
  { '.done()' }

emitter: ff_node | lambda

collector: ff_node | lambda

worker: ff_node | task_lambda

worker_generator: lambda that creates a ff_node object (0 arg)
                | task_lambda (2 args)

workers: std::vector<ff_node*>

stage: ff_node | task_lambda

nb_instances: int
```

Protocol 2: Protocol for `FarmBuilder`.

4.4 DAG

`DAGBuilder`, whose result produces a `DAG` object (an `ff_node`), obeys Protocol 3.

```
'DAGBuilder'
  '.new()'
  { '.node(' node_name ',' node ')' }+
  { '.connect(' node_name ')'
      [ '.to(' node_name [',' node_name]+ ')'
        [ '.send_to(' policy ')' ]?
        [ '.buffer(' size ')' ]?
        [ '.window(' size ')' ]?
      ]+
      [ '.from(' node_name [',' node_name]+ ')'
        [ '.receive_from(' policy ')' ]?
        [ '.buffer(' size ')' ]?
        [ '.window(' size ')' ]?
      ]+
  }+
  { '.done()' }


policy: ALL | ANY
size: int
node_name: string
node: ff_node | task_lambda
```

Protocol 3: Protocol for `DAGBuilder`.

Notes:

- Given a node `foo` connected `to` nodes `f1` and `f2` then:[2]
  - A call to `send_to(ALL)` will ensure that every value returned by `foo` will be sent to both connected nodes.
  - A call to `send_to(ANY)` will ensure that a value returned by `foo` will be sent to a single one of the connected nodes, in a (pseudo)round-robin fashion.
- Given a node `foo` connected `from` nodes `f1` and `f2`, then:
  - A call to `receive_from(ALL)` will ensure that `foo` will be activated only when both connected nodes have sent a value.
  - A call to `receive_from(ANY)` will ensure that `foo` is activated as soon as one of the connected nodes has sent a value.

[2]This is explained using two nodes. Of course, this should be generalized to multiple nodes.

More complex activation policy could also be specified by using overloaded functions that receive as argument a lambda expression (with appropriate arguments) that performs the appropriate input/output port selection policy.

Chapter 5

Conclusion

Much work remains to be done to implement, in C++, the proposed fluent interface. Although a "proof of concept" has been done, through an implementation in Ruby and a (very!) partial implementation in C++, not all features of FastFlow have been handled. Some of these omitted aspects are, for instance, the load balancer in the emitter, the eos notification, etc.

If/when this new Application Programming Interface is implemented, it would be interesting to examine the possibility of making `ff_node` a purely *opaque* type. In other words, at the skeleton configuration-level, the user (i.e., the application programmer) should not have to deal sometimes with `ff_node` objects, sometimes with `ff_node*`, depending on the context. This would incur a small additional overhead—always having to deal with a pointer for such objects—, but this overhead would be rather small and would only be incurred at configuration-time.

Bibliography

[1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] J. Bézevin, "Model engineering for software modernization," in *The 11th IEEE Working Conf. on Reverse Engineering*, (Delft, NE), Nov. 2004.

[3] M. Fowler, "A pedagogical framework for domain-specific languages," *IEEE Software*, vol. 26, pp. 13–14, July 2009.

[4] M. Fowler, *Domain-Specific Languages*. Addison-Wesley, 2011.

[5] D. Ghosh, *DSLs in Action*. Manning, 2011.

[6] M. Mernik, J. Heering, and A. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.

[7] B. Meyer, *Object-Oriented Software Construction (Second edition)*. Prentice-Hall, 1997.

[8] M. Torquati, "Parallel programming using FastFlow," tech. rep., Computer Science Department, University of Pisa, August 2014.

[9] T. Veldhuizen, "C++ gems," ch. Using C++ Template Metaprograms, pp. 459–473, SIGS Publications, Inc., 1996.

[10] M. Voelter, *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. Createspace, 2013. `dslbook.org`.