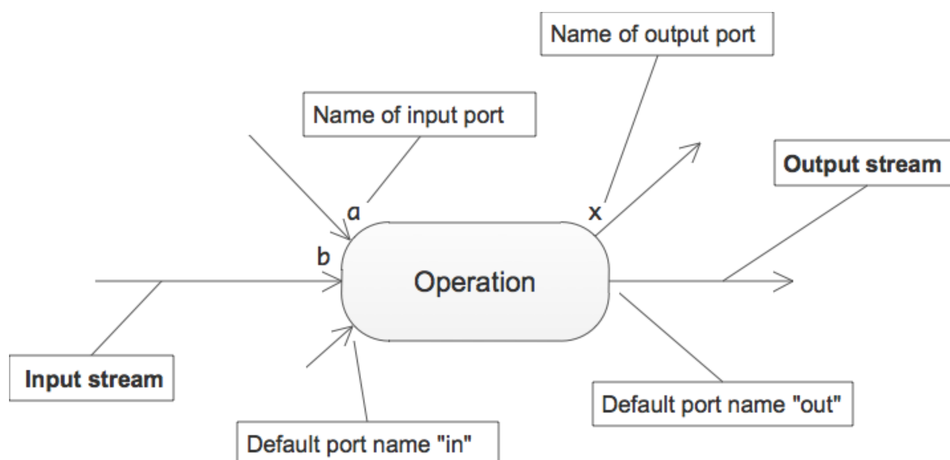
PantaRhei – An Execution Engine for Data Flow Networks I

PantaRhei (PR) helps to describe software as a process working on data flowing through a network of operations.

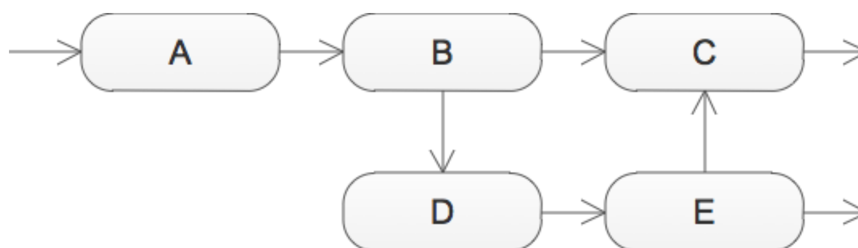
Basic terminology

Data flows into so called operations, which work on it, produce output, and possibly causes some side effect. Each operation can accept input through several streams and can produce output on several streams.

Streams enter or leave operations at ports. Each such port has a name unique to the operation.[1]



Several operations can be connected to form a network, a data flow network or flow for short.

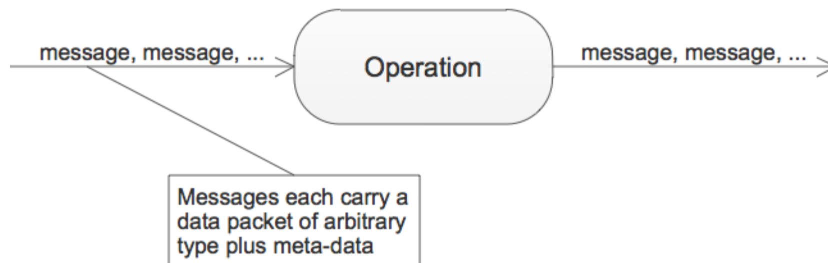


Flows are synchronously executing functional units. Only one operation at a time is running. All data is processed sequentially.

What flows through streams are messages. Each message contains a data packet – possibly accompanied by meta-data.

[1] If no explicit name is given to a port it is called „In“ for an input port and „Out“ for an output port.

As far as streams are concerned, the data packets have no specific type. But operations can of course expect data packets of certain types to arrive over certain streams. Operation ports are thus typed.



Mapping operations to code

Operations and streams are elements of Flow-Design. They are conceptual building blocks for software. To be executed they need to be mapped to 3GL code.

An operation can be described in several ways.

Functions as operations

Any function with a single parameter can be viewed as an operation:

```
OutputType Transform(InputType input) {…}
```

The input parameter is the input port and is named "In", the result is the output port and is named "Out".

Procedures as operations

Procedures of the following form can be viewed as operations:

```
void Consume(InputType input) {…}

void Transform(InputType input, out OutputType output) {…}

void Transform(InputType input,
               Action<OutputType> outputContinuation) {…}
```

The first procedure just consumes input data through. The parameter is the input port named "In". No output is produced.

The second procedure also produces a single output via the out parameter which is the output port named "Out".

The third procedure is able to produce several output messages on port "Out" per input message by calling a continuation procedure.[2]

[2] Note the type of the continuation; it´s a delegate type predefined by C#. Delegates are typed function pointers in .NET. If other languages are lacking function pointers they

Classes as operations

When classes define operations, they do so by using methods as input ports and events (delegates) as output ports.

```
class Transformator {
    public void DoThis(InputType0 input) {…}
    public void DoThat(InputType1 input) {…}
    …
    public event Action<OutputType0> Output0;
    public event Action<OutputType1> Output1;
    …
}
```

The names of the ports are the procedure and event field names.

Classes are the only way to define operations with several input ports.

Normalizing operations

In general operations with any number of input and output ports can be described by this delegate:

```
delegate void Operation(IMessage input,
                        Action<IMessage> outputContinuation);

interface IMessage {
  string Portname {get;}
  object Data {get;}
}
```

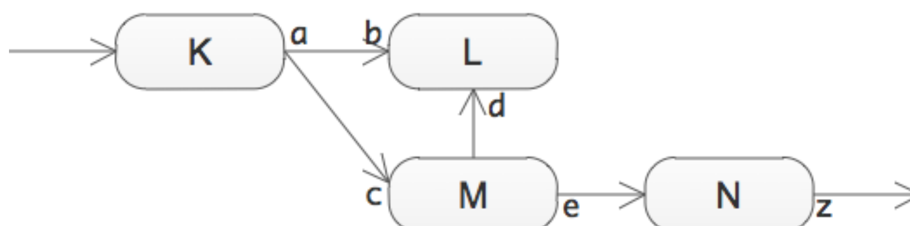
Concrete operations can be fit to this form using an adapter.

Please note: Whereas concrete operations are strongly typed, the data in messages is not.

A textual description for data flow networks

Data flow networks can easily be described by enumerating their streams, i.e. the connections between output and input ports.

Here´s a sample flow:



---

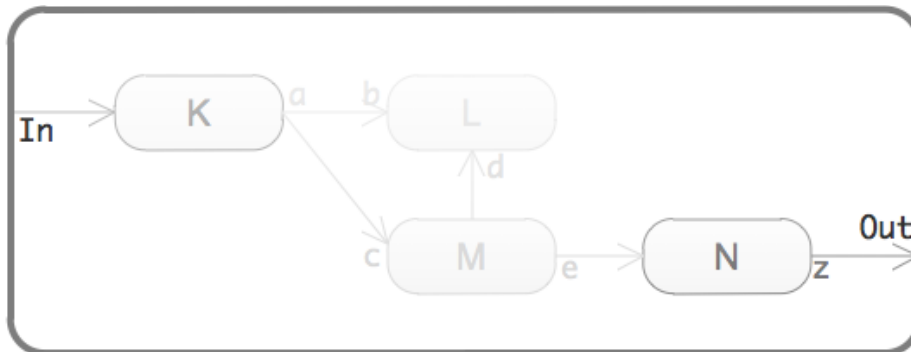
need to compensate this by using other means (e.g. interfaces) or cannot provide certain forms of operations.

And here´s the stream list of this flow:

.In	K.In
K.a	L.b
K.a	M.c
M.Out	L.d
M.e	N.In
N.z	.Out

Some ports have explicit names, some ports are implicitly named with the default names.

There are two ports, however, which are special: .In and .Out. They don´t belong to any operation within the flow.[3] Rather they are the start and end ports of the flow itself. This can be depicted like so:



A flow can be thought of as being nested inside a container, which looks like an operation itself with input/output ports[4] – but connected to operations inside of it.

Execution Engine API

The execution engine´s task is to accept input, get it processed by a data flow network, and produce output. It thus is an operation itself that can be described by the following interface:

```
interface IPantaRhei {
    void Process(IMessage input);
    event Action<IMessage> Result;

    void AddStream(IStream stream);
    void AddOperation(IOperation operation);
}
```

[3] That´s why they start with a dot instead of an operation name.
[4] Of course the ports oft he containing virtual „operation“ can be named arbitrarily as long as they are unique regarding the virtual „operation“.

```
interface IStream {
    string FromPortname {get;}
    string ToPortname {get;}
}

interface IOperation {
    string Name {get;}
    Operation Implementation {get;}
}
```

Usage then is straightforward:

1. Register streams by calling AddStream().
2. Register operations by calling AddOperation().
3. Register event-handler on Result for output from the flow.
4. Pass in input by calling Process().

The messages passed to Process() should refer to a port without an operation name. The messages produced by the flow as output from Result will be those with ports without an operation name.