SimPEL

- Variables assignment (foo = bar) has a by-copy semantic (like in BPEL) and not by-reference (like in most other languages).

Optional target namespace declaration. For process elements it defaults to ODE target namespace. Namespaces are optional. When no namespace is provided for the process, a default one is assumed.

```
namespace foo = "urn:/example.com"
process foo::request {
}
# Or
use namespace foo
process request {
}
```

Importing documents definitions. After import, elements can be referenced relatively.

```
foo = import "xsd|wsdl|..."
foo.portTypes.bar
```

Variable declaration is optional and mostly used to add some specific behavior to them. Modifiers in a variable declaration are its type, unique and external:

```
var foo string unique, bar external(baz), baz int
```

Subsequently variable assignment is fairly classic. Javascript code is supported natively anytime an expression can be passed (rvalues) or at some specific places of the process declaration (like before the process definition).

```
foo = bar * (bar - 1)
```

XML is a native type and can directly be included in-line, just like in E4X (including inner expressions). So creating an XML literal fragment is as easy as:

```
foos = <foos><foo>bar</foo><foo>baz</foo></foos>
```

Process definition

```
process foo {
  ...
}
```

Code that's surrounded by curly brackets is always assumed to be sequential, statements will be executed in the order they're defined.

Pick syntax, receive is the same thing without the pick wrapper

```
pick {
  receive(p1, o1) { |msg|
    # More BPEL code
  }
  receive(p2, o2) { |msg|
  }
  someVar = receive(p3, o3)
  timeout(val) {
  }
}
```

Flow, each block is a sequence. The signal and join instructions are here to model links,

```
parrallel {
   ...
   signal(link1, "expr")
   ...
} and {
   ...
   join(link1, link2, link3, "($link1 and $link2) or $link3")
   ...
} and {
}
```

If, else if and else:

```
if (expr) {
} else if (expr) {
} else {
}
```

While:

```
while(i<10) {
  i = i + 1
}
```

Repeat until:

```
do {
  i = i+1
} until(i<10)
```

Sequential foreach and parrallel foreach are two different constructs. Allow break?

```
for(m = 0; m < 10; m = m + 2) {
}
forall(m = 0 to 10) {
}
```

Invoke syntax - toParts? fromParts?

```
foo = invoke(pl, operation, msg)
```

Throw, faultVar is optional:

```
throw faultName, faultVar
```

Wait, support now()+duration expression

```
wait 2m
```

Partner links must be defined before being used (but no explicit role definition is needed) and are associated with the closest surrounding scope. How they are bound to an endpoint or interface (service/port, Java class, ...) is outside of the scope of SimPEL and is meant to be specified by a deployment descriptor.

```
partnerLink foo, bar, baz
```

Correlation is achieved by declaring a function that will extract a value from a message and match it against another value. So there's no real correlation set per se, just a set of functions used to extract data from messages. We also introduce a special variable declarator that allows for some variables to be unique (and indexed).

```
function oidFromOrder(msg) {
    ....
}
process {
    var oid unique
    msg = receive(pl, op1)
    oid = oidFromOrder(msg)
    ....
    receive(pl, op2, {oidFromOrder: oid}) { |msg2|
        ....
    }
}
```

We've separated the fault handling role of scopes (try...catch) from the eventing role (*Handlers).

```
try {
    scope {
        ....
    } event (pl, op) { |var|
       ...
    } alarm {
    } compensation {
    }
} catch (Fault f1) {
} catch (f2) {
}
```

For all variables that are passed inside blocks (mostly receive and event), a context can be provided as well. The is meant to introduce security context manipulations:

```
receive(pl, op) { |var, ctx|
   ...
}
```

Examples

Hello world example

```
process Helloworld {
  receive(client, hello) (msg) {
    msg = msg + " World"
    reply msg
  }
}
```

External counter example

```
process ExternalCounter {
  receive(my_pl, start_op) (msg_in) {
    resp = <root><count>0</count></root>
    while(resp < 10) {
      invoke(partner_pl, partner_start_op) (msg_in)
      resp = receive(partner_pl, partner_reply_op)
    }
    reply resp
  }
}
```

Function definition in Javascript that returns an xpath function, plain javascript inside and allows to "hook" other expression languages. Those functions are also used for correlation (see later).

```
function foo(msg) {
  xpath("foo/bar");
}
something.other = foo
something.other = xpath('.....')
```