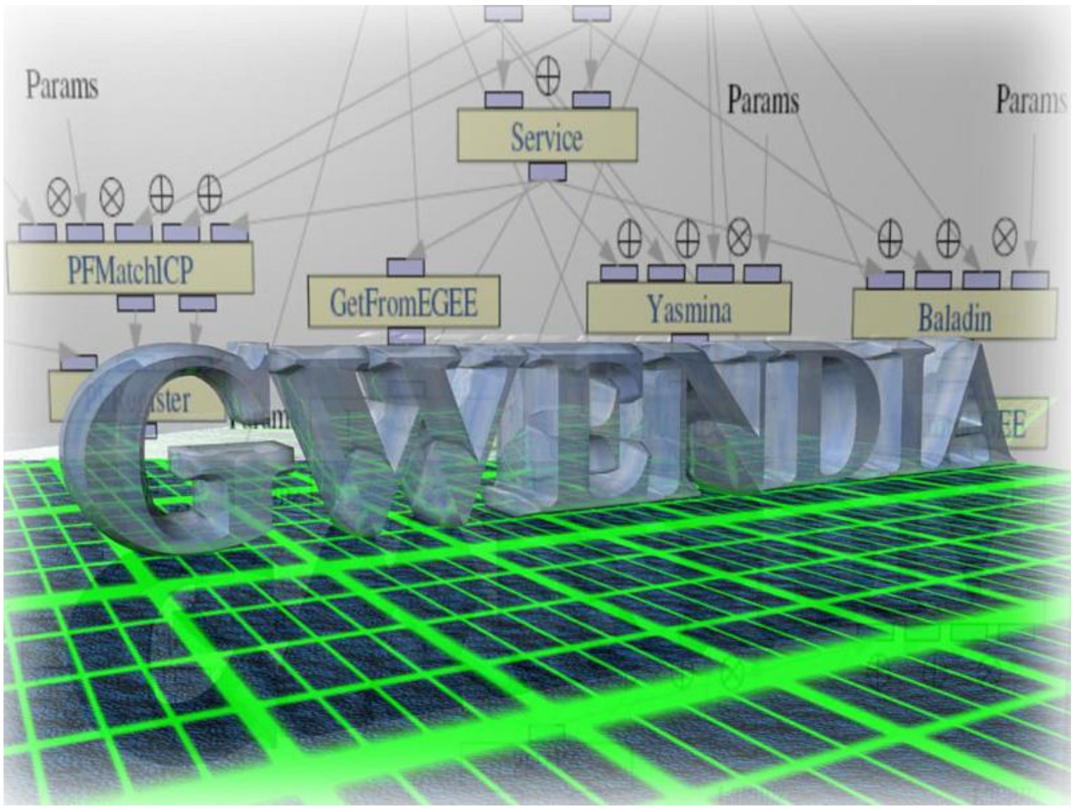
Gwendia ANR-06-MDCA-009
Grid Workflow Efficient Enactment for Data Intensive Applications

Financé par ANR

Workflow language proposal



Johan Montagnat	MODALIS (I3S)	johan@i3s.unice.fr
Benjamin Isnard	GRAAL (LIP)	benjamin.isnard@ens-lyon.fr
Tristan Glatard	CREATIS-LRMN	glatard@creatis.insa-lyon.fr
Mireille Blay Fornarino	MODALIS (I3S)	blay@polytech.unice.fr
Ketan Maheshwari	MODALIS (I3S)	ketan@polytech.unice.fr

Abstract

Workflow language proposal for the Gwendia project. Emphasize on control structures.

1 Motivation and goals

<to be completed>.

1.1 Data-driven parallel execution

We adopt a data-driven parallel execution model that ease parallel processes description from a user point of view and make graphical representation possible (both compact and easy to represent graphically the application logic graph).

1.2 Array programming

Arrays, also known as lists (especially in the functional programming context) or vectors (especially in the parallel computing community), are first class objects in the language. The manipulation of nested arrays is common. Arrays are manipulated through:

- array nesting level consideration: an array may be manipulated as a single data item or as many individual data items (each of them possibly being a nested array); and
- multiple arrays are combined through various iteration strategies (also known as array operators).

1.3 Motivations for a new scientific workflow language

The languages targets the coherent integration of:

1. the data-driven approach;
2. arrays manipulation;
3. control structures; and
4. maximum asynchronous execution (the operators should be implemented so as to introduce as little synchronization barriers as possible).

2 Language structures

The language enables the description of data to be manipulated, processing activities (interconnected through data dependencies) and control structures.

2.1 Data

The data manipulated in the language is composed from scalar typed data items. Data with homogeneous types may be grouped in arrays. Arrays may contain nested arrays. As will be shown below, nested arrays may have heterogeneous sizes (two inner arrays of different sizes may appear in a same container array).

2.1.1 Scalars

Scalar values are typed. The basic types considered are `integer`, `double`, `string` and `file` (*i.e.* string identifiers referencing files, that are not interpreted by the workflow manager). The special scalar value $\emptyset$ represents the absence of data. It is to be noted that the special value $\emptyset$ has an instantiation in each scalar type: it may denote a non-existing integer $\emptyset_{\mathbf{integer}}$, a non-existing string $\emptyset_{\mathbf{string}}$, etc. For simplification, we will use the non-subscripted notation $\emptyset$ in all cases in the rest of this document. Formally, a scalar type ($\mathbf{s}$) is defined as follows:

$$\mathbf{s} ::= \mathbf{integer} \,|\, \mathbf{double} \,|\, \mathbf{string} \,|\, \mathbf{file}$$

2.1.2 Data structures and arrays

A data structure is a fixed size heterogeneous collection of data items. The type of a structure is inferred from the types of its composing data items and the cross operator: given n items typed τ_i with $i \in [1..n]$, the type of the corresponding structure is $\tau_1 \times \ldots \times \tau_n$.

An array is an ordered collection of data items with the same type. A simple array is a collection of scalars (*e.g.* $\mathbf{a} = [2, -3, 1]$ is an array of integers). A one-dimension index designates each of its data item ($\mathbf{a}_0$ designs the integer 2). Whe denote as $A(\tau)$ the type of an array of type τ items. An array may be empty. An array may contain other arrays at any nesting level. An array of arrays is further referenced as a *2 nesting levels* array. Each additional nesting level increases the nesting level counter by 1. For convenience, we will denote the nested array type $A(A(\ldots A(\tau)\ldots))$ with n nesting levels as $A^n(\tau)$. Note that a scalar s and a singleton $\{s\}$ are different data entities, with different types $\mathbf{s}$ and $A(\mathbf{s})$ respectively. A scalar data item corresponds to a *0 nesting level* array: $\mathbf{s} = A^0(\mathbf{s})$.

Any type is defined by:

$$\tau ::= \mathbf{s} \,|\, A(\tau) \,|\, \tau \times \tau \,|\, \mathbf{1}$$

where the cross operator designates the type of a pair of data items. It can be used to represent either a data structure or the output of a processor with two output ports. $\emptyset$ also has an additional instantiation to describe the particular case of a workflow with no output. Its type is $\mathbf{1}$.

As usual in language syntax definition, we define a context Γ as a list of typed variables $x_1, \ldots x_n$:

$$\Gamma ::= x_1 : \tau_1, \ldots, x_n : \tau_n$$

where τ_i is the type of x_i.

2.1.3 Loose typing option

For prototyping, developers may want to adopt a losy type checking system. In that case, all scalar data items may be considered as string: $\mathbf{s} ::= \mathbf{string}$. The rest of this document remains valid with or without the loose type checking assumption.

2.2 Workflows

Workflows are described as a graph of activities interconnected through data dependency links. Each activity correspond to the execution of some application code. Activities are fired as soon as input data become available for processing. Activities may be fired an arbitrary number of times, or never fired at all, depending on the data flowing in the workflow. The special data item Ø causes no firing of an activity. It is transferred to the subsequent activities without causing user code invocation.

As described in [1], workflows with input context Γ and output type τ are formally represented as sequents of the form:

$$\Gamma \vdash W : \tau$$

2.2.1 Activities

A workflow activity is an atomic process containing a piece of application code that is bound to an arbitrary number of *input* and *output ports*. The ports represent data buffers where data items to process are received (input ports) or produced data items are stored after firing the activity (output ports). Input and output ports are typed (only data matching the input port types can be processed and data produced matches the output port types). The output port types define the activity type. Formally, a workflow activity **a** which output type is τ is an axiom:

$$\Gamma \vdash \mathbf{a} : \tau$$

Upon firing, an activity may either execute successfully, thus producing an output of type τ (possibly the special value Ø), or encounter an error an throw an exception. An exception cause the workflow engine to be notified of the error (user reporting) and produce the special value Ø as the result of the execution. An activity which receives Ø as input does not fire and just pass the value on to the subsequent activity(ies). This execution semantics guarantees that the process continues execution as much as possible, processing other data items that did not raise exception conditions.

Workflow inputs are special activities, with no input port and a single output port, that fire without pre-requesite when the workflow is started. Valid workflow inputs are (i) data *sources*, containing user defined data, (ii) *constants*, containing a single value (scalar or singleton), or (iii) user defined activities with no input ports that will fire only once, when the workflow is started. For instance:

$$\vdash \mathbf{source_0} : \tau$$

Workflow outputs are special activities, with no output port and a single input port, that perform no processing and collect results received from other activities in the workflow. We consider that the type of an output is the type of its input ports (*i.e.* the type of the data received on its input ports):

$$x : \tau \vdash \mathbf{output_0} : \tau$$

Regular activities (also called *processors*) are user defined activities . They usually have at least one input and one output port, although some user-defined processors may have no input port (user-defined input) or not output port (workflow dead-end without result collection). A processor with more than one output port as a product type. For instance, if **p** is a processor with two output ports of type τ and σ respectively, then the processor type is:

$$\Gamma \vdash \mathbf{p} : \tau \times \sigma$$

Similarly, two workflows with no junction links produce a product type of their respective outputs.

2.2.2 Activity port depths

The depth of processor input ports define the nesting level of arrays that this processor will consider atomically (and therefore it impacts the number of firing of this processor). Similarly, the depth of processor output ports, in conjunction with the nesting level of data item received, defines the nesting level of arrays that this processor will produce. Let us denote with exponent n the nesting level of an array: $A^n(\mathbf{s})$, with $A^0(\mathbf{s}) = \mathbf{s}$ and $A^1(\mathbf{s}) = A(\mathbf{s})$. A processor $\mathbf{p}$ with a single input port of depth i (type $\mathbf{s}$) and a single output port of depth o (type $\mathbf{t}$) fires once for each nested data item with i nesting levels received, and produces a depth o output for each of these invocations. Therefore, if $n \geq i$:

$$\Gamma, x : A^n(\mathbf{s}) \vdash \mathbf{p} : A^{n+o-i}(\mathbf{t})$$

If $n < i$ the processor invocation raises an exception.

As a consequence, input and output port types are always defined as a scalar type $\mathbf{s}$, complemented with a depth. The array nesting level of input data items, n, varies independently of the processor definition (the only constraint is that $n \geq i$, consequently there is no constraint for depth 0 input ports which consider individual scalar items). Similarly, the nesting level of data item produced, $n + o - i$ depends on n and therefore the type of data item received.

An important property of activities invocation in an asynchronous execution is that multiple invocations of an activity on array items preserve the array indexing scheme. The data indices are preserved during processing: the j^{th} data item in the output port will correspond to the processing of the j^{th} data item in the input port. This property is reflected in the operational semantic rule:

$$\frac{P \Downarrow \mathbf{u} \qquad \{Q[u_i/x] \Downarrow v_i\}\big|_{i=1..|\mathbf{v}|}}{\text{let } x \leftarrow P \text{ in } Q \Downarrow \mathbf{v}}$$

where $\mathbf{u} = [u_1, \ldots, u_m]$ is a n nested levels array and $\mathbf{v} = [v_1, \ldots, v_m]$ is a $n + o - i$ nested levels array.

2.2.3 Data links

A data link interconnects one activity output port with one activity input port. It defines a data dependency between two activities. A link is formally represented as a composition rule. If the output ports of an activity $\mathbf{a_1}$ are linked to the input ports of an activity $\mathbf{a_2}$ and if the port types match, the connection is represented by:

$$\frac{\Gamma \vdash \mathbf{a_1} : \tau_1 \qquad \Gamma, x : \tau_1 \vdash \mathbf{a_2} : \tau_2}{\Gamma \vdash \text{let } x \leftarrow \mathbf{a_1} \text{ in } \mathbf{a_2} : \tau_2}$$

This definition prevents multiple links to be linked to a single input port. Note however that different outputs of $\mathbf{a_1}$ may be connected to different activities, which is formally described through projection rules to separate $\mathbf{a_1}$'s outputs prior to applying the composition rule:

$$\frac{\Gamma \vdash \mathbf{a_1} : \sigma \times \tau}{\Gamma \vdash \text{fst}(\mathbf{a_1}) : \sigma} \qquad \frac{\Gamma \vdash \mathbf{a_1} : \sigma \times \tau}{\Gamma \vdash \text{snd}(\mathbf{a_1}) : \tau}$$

With projection any workflow graph can be assembled, and the composition rule can be rewritten to apply to any workflows W and X with W's outputs connected to X's inputs:

$$\frac{\Gamma \vdash W : \sigma \qquad \Gamma, x : \sigma \vdash X : \tau}{\Gamma \vdash \text{let } x \leftarrow W \text{ in } X : \tau}$$

2.2.4 Control links

In some cases, there is no data dependencies explicitly defined between two processors but an order of execution should be preserved nevertheless. A control link interconnecting these processors may then be defined. A control link interconnecting a source to a target processor emit a signals only once the source processor completed all its executions. The target processor will start firing only once it has received the control link signal, processing the data items buffered in its input ports or to be received, as usual. In case several control links are connected to a target processor, it only starts firing when all control signals have been received. In an asynchronous execution environment, a control link thus introduces a complete data synchronization barrier.

2.2.5 Iteration strategies

Iteration strategies define how input data items received on several input ports of a same processor are combined together for processing. They therefore define how many times a processor fires and what is its exact input data sequence for each invocation. Iteration strategies are also responsible for defining an indexing scheme that describes how items from multiple input nested arrays are sorted in an output nested array.

dot product. The dot product matches data items with exactly the same index in an arbitrary number of input ports. The dot product formal syntax is:

$$\frac{\Gamma \vdash P_1 : A(\sigma_1) \qquad \Gamma \vdash P_2 : A(\sigma_2) \qquad \Gamma, x_1 : \sigma_1, x_2 : \sigma_2 \vdash Q : \tau}{\Gamma \vdash \text{let } x_1 \odot x_2 \leftarrow P_1 \odot P_2 \text{ in } Q : A(\tau)}$$

where (let $x_1 \odot x_2 \leftarrow P_1 \odot P_2$ in Q) is a primitive conforming to the dot product semantics. The processor fire once for each common index, and produces an output indexed with the common index. The nesting level of input data items, as received and transformed after port depth considerations, in all ports of a dot product should be identical. The number of items in all input arrays should be the same. Hence:

$$\frac{P_1 \Downarrow \mathbf{u} \qquad P_2 \Downarrow \mathbf{v} \qquad \{Q[u_i/x_1, v_i/x_2] \Downarrow w_i\}\Big|_{i=1..|\mathbf{u}|} \qquad |\mathbf{u}| = |\mathbf{v}|}{\text{let } x_1 \odot x_2 \leftarrow P_1 \odot P2 \text{ in } Q \Downarrow \mathbf{w}}$$

The ports of a dot product are associative and commutative. A $\emptyset$ value received on a dot product port matches with the data item(s) with the same index(ices) received on the other port(s) and produces a $\emptyset$ output without firing the activity.

cross product. The cross product matches all possible data items combinaisons in an arbitrary number of input ports. The processor fires once for each possible combinaison, and produces an output indexed such that all indices of all inputs are concatenated into a multi-dimensionnal array (data items $\mathbf{a}_i$ and $\mathbf{b}_j$ received on two input ports produce a data item $\mathbf{c}_{ij}$).

$$\frac{\Gamma \vdash P_1 : A(\sigma_1) \qquad \Gamma \vdash P_2 : A(\sigma_2) \qquad \Gamma, x_1 : \sigma_1, x_2 : \sigma_2 \vdash Q : \tau}{\Gamma \vdash \text{let } x_1 \otimes x_2 \leftarrow P_1 \otimes P_2 \text{ in } Q : A^2(\tau)}$$

where

$$\text{let } x_1 \otimes x_2 \leftarrow P_1 \otimes P_2 \text{ in } Q \equiv \text{let } x_1 \leftarrow P_1 \text{ in } (\text{let } x_2 \leftarrow P_2 \text{ in } Q)$$

and
$$\frac{P_1 \Downarrow \mathbf{u} \qquad P_2 \Downarrow \mathbf{v} \qquad \{\{Q[u_i/x_1, v_j/x_2] \Downarrow w_{ij}\}\}\Big|_{\substack{i=1..n\\ j=1..m}}}{\text{let } x_1 \otimes x_2 \leftarrow P_1 \otimes P2 \text{ in } Q \Downarrow [[w_{11} \dots w_{1m}] \dots [w_{n1} \dots w_{nm}]]}$$

The ports of a cross product are associative but not commutative. A $\emptyset$ value received on a cross product port matches with all possible combinaisons of other data items received in other ports and produces a $\emptyset$ output without firing the activity. The cross product formal syntax is:

flat cross product. The flat cross-product matches inputs identically to a regular cross product:
$$\frac{\Gamma \vdash P_1 : A(\sigma_1) \qquad \Gamma \vdash P_2 : A(\sigma_2) \qquad \Gamma, x_1 : \sigma_1, x_2 : \sigma_2 \vdash Q : \tau}{\Gamma \vdash \text{let } x_1 \ominus x_2 \leftarrow P_1 \ominus P_2 \text{ in } Q : A(\tau)}$$

The difference is in the indexing scheme of the data items produced: it is computed as a unique index value by flattening the nested-array structure of regular cross produces ($\mathbf{a}_i$ and $\mathbf{b}_j$ received on two input ports produce a data item $\mathbf{c}_k$ with index $k = i \times m + j$ where m is the size of array $\mathbf{b}$):

$$\frac{P_1 \Downarrow \mathbf{u} \qquad P_2 \Downarrow \mathbf{v} \qquad \{\{Q[u_i/x_1, v_j/x_2] \Downarrow w_{i \times m+j}\}\}\Big|_{\substack{i=1..n\\ j=1..m}}}{\text{let } x_1 \ominus x_2 \leftarrow P_1 \ominus P2 \text{ in } Q \Downarrow [w_1 \dots w_{nm}]}$$

As a consequence, the flat cross product may be partially synchronous. As long as the input array dimension are not known, some indices cannot be computed. Similarly as the cross product, the ports of a flat cross product are associative but not commutative. A $\emptyset$ value received on a flat cross product port behaves as in the case of a regular cross product. The formal syntax of the flat cross product is the same as the cross product's one.

match product. The match product matches data items carrying one or more identical user-defined tags, independently of their indexing scheme.

$$\frac{\Gamma \vdash P_1 : A(\sigma_1) \qquad \Gamma \vdash P_2 : A(\sigma_2) \qquad \Gamma, x_1 : \sigma_1, x_2 : \sigma_2 \vdash Q : \tau}{\Gamma \vdash \text{let } x_1 \oplus x_2 \leftarrow P_1 \oplus P_2 \text{ in } Q : A^2(\tau)}$$

where (let $x_1 \oplus x_2 \leftarrow P_1 \oplus P_2$ in Q) is a primitive conforming to the match product semantics. Similarly to a cross product, the output of a match is indexed in a multiple nesting levels array item which index is the concatenation of the input indices. A match product implicitly defines a boolean valued function match$(\mathbf{u}_i, \mathbf{v}_j)$ which evaluates to true when tags assigned to $\mathbf{u}_i$ and $\mathbf{v}_j$ match (*i.e.* the specified tags values are equal). The output array has a value at index i, j if match$(\mathbf{u}_i, \mathbf{v}_j)$ is true. It is completed with $\emptyset$ values: if match$(\mathbf{u}_i, \mathbf{v}_j)$ is false then $\mathbf{w}_{ij} = \emptyset$.

$$\frac{P_1 \Downarrow \mathbf{u} \qquad P_2 \Downarrow \mathbf{v} \qquad \{\{Q[u_i/x_1, v_j/x_2] \Downarrow w_{ij}\}\}\Big|_{\substack{i=1..n\\ j=1..m\\ \text{match}(\mathbf{u}_i,\mathbf{v}_j)}} \qquad \{\{w_{ij} = \emptyset\}\}\Big|_{\substack{i=1..n\\ j=1..m\\ \neg\text{match}(\mathbf{u}_i,\mathbf{v}_j)}}}{\text{let } x_1 \oplus x_2 \leftarrow P_1 \oplus P2 \text{ in } Q \Downarrow [[w_{11} \dots w_{1m}] \dots [w_{n1} \dots w_{nm}]]}$$

The ports of a match product are thus associative but not commutative. A $\emptyset$ value received on a match product input does not match any other data item and does not cause processor firing.

2.3 Control structures

The data-driven and graph-based approached adopted in the Gwendia language makes parallelism expression straight forward for the end users:

- Data parallelism is completely hidden through the use of arrays. Advanced data composition operators are available through activity port depth definitions and iteration strategies. Complex data parallelisation patterns and data synchronization can therefore be expressed without additional control structures. `foreach` kind of structures that are usually used for explicit data parallelization is no needed.
- Code parallelism is implicit in the description of the workflow graph. `fork` and `join` kind of structures are not needed either.

The only control structures considered for the Gwendia language are therefore conditionals and loops. The semantics of contional and loops operating over array types need to be precisely defined. To our knowledge, existing array-based languages do not define such a semantic and the programmer needs to define the conditional and loop expressions on scalar values (consequently using `foreach` kind of structures to iterate on the content of arrays).

Special activities are defined to express conditionals and loops. These activities have a constrained format and input/output ports for enforcing the semantics defined in this document.

2.3.1 Beanshell processors

Conditional and loop expression computations are better understood by analogy to *beanshell* processors. A beanshell is a fully customizable processor (no restrictions on input/output ports) embarking user-defined java code to be interpreted each time the processor is fired. The data received on the input ports of a beanshell processor is mapped to java variables (basic types or java ArrayLists depending on the input port depths) and, similarly, values stored in java variables are mapped to output ports after computation. Beanshells can be used to evaluate expressions such as the one needed for conditionals or loop stop conditions.

2.3.2 Conditionals

A conditional activity represents an array-compliant *if then else* kind of structure. A conditional has:

1. an arbitrary number of input ports (possibly operating iteration strategies);
2. a test expression to evaluate for each data received from the input ports; and
3. an arbitrary number of special paired output ports. Each pair corresponds to a single output with the first pair element linking to the *then* branch and the second pair element linking to the *else* branch.

Let $\mathrm{cond}(x)$ represent the test expression evaluated on value x:

$$\frac{\Gamma \vdash P : A(\sigma) \qquad \Gamma, x : \sigma \vdash \mathrm{cond}(x) : A(\mathbf{bool}) \qquad \Gamma, y : A(\sigma) \vdash Q : A(\tau) \qquad \Gamma, y : A(\sigma) \vdash R : A(\tau)}{\Gamma \vdash \mathrm{let}\ x \leftarrow P\ \mathrm{in}\ \mathbf{if}\ \mathrm{cond}(x)\ \mathbf{then}\ (\mathrm{let}\ y \leftarrow P \mathrm{in}\ Q)\ \mathbf{else}\ (\mathrm{let}\ y \leftarrow P\ \mathrm{in}\ R) : A(\tau) \times A(\tau)}$$

The test expression is evaluated each time the conditional processor fires. A user-defined result is assigned to the *then*, and optionally to the *else*, output port for each data sequence evaluated. An empty result (∅) is assigned to the opposite port automatically. Consequently, the *then* and the *else* output ports receive a nested array with the same structure and size, as defined by the input nesting levels, ports depths and iteration strategies used, but complementary in the sense that if a value is available in one of the ouput, the corresponding item is empty in the other output and vice versa. The indexing scheme used is coherent with the usual indices computed by iteration strategies. The empty results are therefore indexed coherently.

$$\frac{P \Downarrow \mathbf{u} \quad \{Q[u_i/y] \Downarrow v_i\}\Big|_{\substack{i=1..n \\ \text{cond}(\mathbf{u}_i)}} \quad \{v_i = \emptyset\}\Big|_{\substack{i=1..n \\ \neg\text{cond}(\mathbf{u}_i)}} \quad \{R[u_i/y] \Downarrow w_i\}\Big|_{\substack{i=1..n \\ \neg\text{cond}(\mathbf{u}_i)}} \quad \{w_i = \emptyset\}\Big|_{\substack{i=1..n \\ \text{cond}(\mathbf{u}_i)}}}{\text{let } x \leftarrow P \text{ in } \mathbf{if} \text{ cond}(x) \ \mathbf{then} \ (\text{let } y \leftarrow P \text{in } Q) \ \mathbf{else} \ (\text{let } y \leftarrow P \text{ in } R) \Downarrow \mathbf{v} \times \mathbf{w}}$$

In case the *else* assignment is omitted by the user, a ∅ output value is produced on the *else* outputs each time the condition is invoked. A ∅ value received on the input ports cause the conditional processor to produce two ∅ values in all its *then* and *else* outputs without evaluating the conditional.

Figure 1 shows example of conditional activity and the result of the enactment over multiple-nesting level arrays. The left side example is a simple conditional without definition of the *else* condition. The output list contains one empty item for the value that did not pass the condition in the *then* branch and the *else*branch only receives ∅ values. The center example is a complete conditional with both *then* and *else* branches. The two output arrays are complementary. This example also shows the use of two inputs. The 1 nesting level input arrays are transformed in 2 nesting levels output arrays by the iteration strategy applied between the inputs. The right side example is a complex example with the mixed use of multiple port depth values, iteration strategy and multiple output ports.

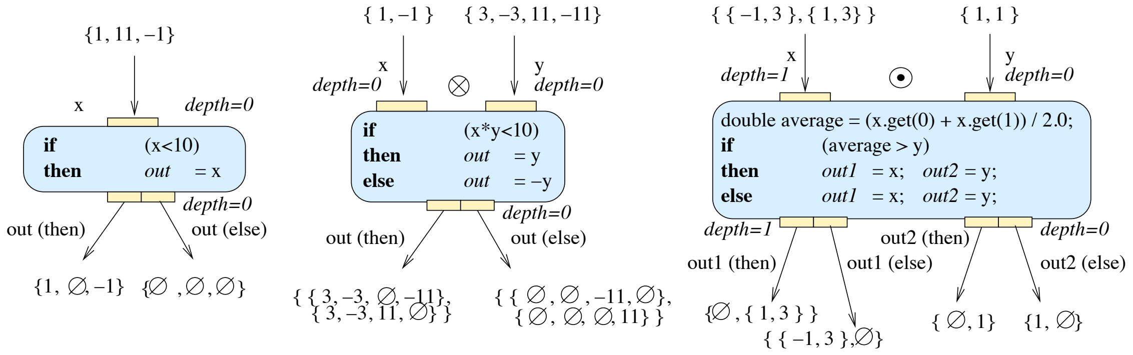


Figure 1: Three conditional examples.

With partial (and complementary) arrays produced by conditionals, two additional list manipulation activities become useful as exemplified in figure 2:

- The *filter* activity is a single input / single output ports activity that filters a nested array structure such that all empty items are removed from the array. This activity is useful to discard all results that have not passed the condition, if the indexing of resulting items does not need to be preserved. As a consequence, the items in the structure will be re-indexed. It is to be noted that this activity introduces a partial synchronization barrier: an item in an array cannot be re-indexed until all preceding items have been computed and determined as empty or not. The filtering operation can create unbalanced lists in terms of size.

- The *merge* activity is a two input ports / one output port activity that merges the content of two complementary lists with the same structure into a single list. It can be used to merge the lists resulting from the *then* and the *else* branch of the conditional for instance. If the list structures differ or the lists are not complementary (an item at a given index is non empty in both lists) the merge activity raises an exception.

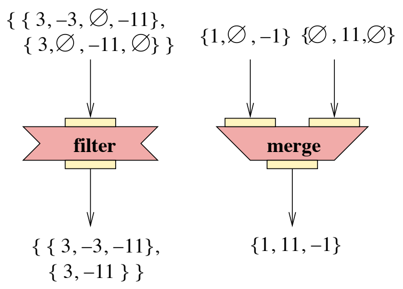


Figure 2: Filtering and merging lists with empty items.

2.3.3 Loops

A loop represents an array-compliant *while* kind of structure. A loop is composed by:

- An expression used as stop condition.
- One or more input ports. Loop input ports have a particular dual structure: they are composed of an outer part, receiving the loop initialization value from the outer part of the workflow, and an inner part, receiving the values that loop back to the activity after one or more iteration of the loop.
- As many output ports as there are input ports. Each output port is bound to one input port as it will receive the values sent to the corresponding input port. Each output port also has a dual structure: the outer part will only receive a value when the loop condition become false (hence the loop stops iterating) while the inner part will receive iteratively all values received either on the initialization (outer) or the looping (inner) part of the corresponding input port.

The inner input port from a loop can only receive a link that is connecting from the inner output port (*i.e.* a loop has to exist). In addition, a loop activity as a specific indexing scheme on its inner port which increases the nesting level of input arrays by one: for each initialization value causing the activity to fire, a sub-array is created that will hold all the values generated by this initialization while the loop iterates. A ∅ value received on the input ports cause a ∅ value to be produced on the corresponding outer port without evaluation of the condition.

Figure 3 illustrates a simple loop and the data flowing through each port. This loops receives an array with two values (1 and 2) as initialization. As the condition passes for the first value, it is transferred to the inner part of the output port, causing a 2 nesting levels array to be created. The second initialization value also passes the condition and is transferred to a second 2 nesting levels sub-array. The first value will cause 2 iterations of the loop before the stop condition is met while the second value will only cause 1 iteration. As a consequence, the inner array as two sub-arrays with different lengths. The outer part of the output port only receives the stop condition values. The array transferred on the output port has the same nesting level as the input since both input and output ports have depth 0.

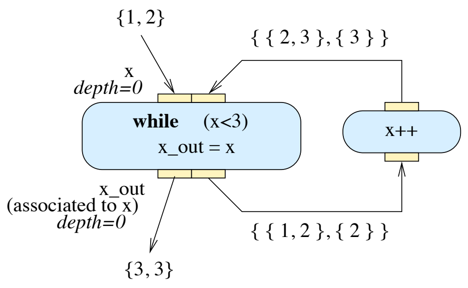


Figure 3: Simple loop example.

2.3.4 Iterations

A *for* kind of control structure has exactly the same structure as the loop structure. In the *for* case, the number of iterations is the same for all initialization value. Consequently, the inner sub-arrays will be of equal length. Figure 4 illustrates a simple *for* loop examples:

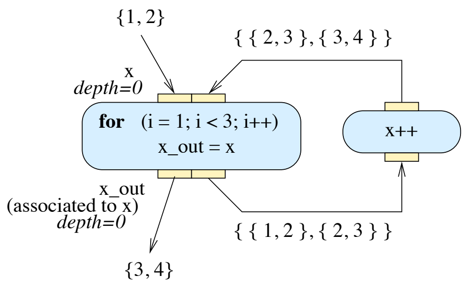


Figure 4: Simple iteration example.

2.3.5 Complete example

Figure 5 illustrates a complete example including a loop, a conditional and a merge activity.

3 Thoughts on iteration strategies extensibility

Many specific iteration strategies could be added to the language. For example, a symmetric cross-product would make sense when processing a pair of input data (a_i, b_j) produces the same result as processing the opposite pair (a_j, b_i). In that case, the iteration strategy

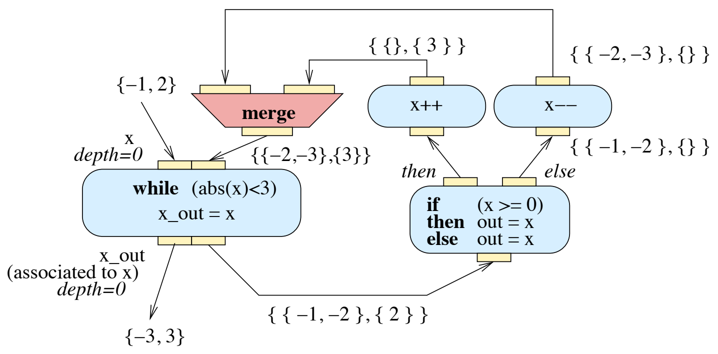


Figure 5: Complete example with loop, conditional, and array merging activities.

should only match once and fire the activity once for both pairs. It is difficult to anticipate and cover any application need. Therefore, enabling user-defined iteration strategies will be required in the long term.

A new iteration strategy can almost be implemented through a specific beanshell processor: a beanshell with two inputs connected through a cross product, and two ouputs, will fire for each possible pair of input data. If the beanshell code filters out some of the pairs, returning a pair only if it matches according to the semantics defined by the customized iteration strategy, and returning ∅ otherwise, the beanshell indeed defines an iteration strategy. Its outputs can be connected to a subsequent processor with a neutral iteration strategy (*i.e.* a dot product firing for all input pairs received).

However, implementing an iteration strategy, such as the symmetric cross-product, asynchronously is only possible if the processor is able to manipulate the indices of the data items manipulated: the processor needs to be aware of the indices i and j of the pair (a_i, b_j) and to compute an index k as a function of i and j for the resulting data item. Therefore, an iteration strategy processors needs to be an extended kind of beanshell with the ability to access the input and output indices of the data items manipulated (there are normally hidden to the workflow engine).

In addition, iteration strategies defined as individual processor may be cumbersome if they appear many time in one or several workflow: the processor needs to be repeated each time it is used. Consequently, a repository of iteration strategy specific-processor is needed to define these strategies only once and reuse them as much as needed. To be complete, a workflow file should contain the code for any customized iteration strategy it uses.

Instead of beanshells, the iteration strategies could be implemented as java classes dynamically loaded into the workflow engine code. In that case however, including these strategies in the workflow files is more difficult.

References

[1] Daniele Turi, Paolo Missier, Carole Goble, David de Roure, and Tom Oinn. Taverna Workflows: Syntax and Semantics. In *IEEE International Conference on e-Science and Grid Computing (eScience'07)*, pages 441–448, Bangalore, India, December 2007.