Project no. 644235

REPHRASE

Research & Innovation Action (RIA)
REFACTORING PARALLEL HETEROGENEOUS RESOURCE-AWARE APPLICATIONS – A SOFTWARE ENGINEERING APPROACH

Advanced patterns
D2.5

Due date of deliverable: M18

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP2

Responsible institution: UNIPI
Editor and editor's address: M. Danelutto, UNIPI

Version 1.0

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

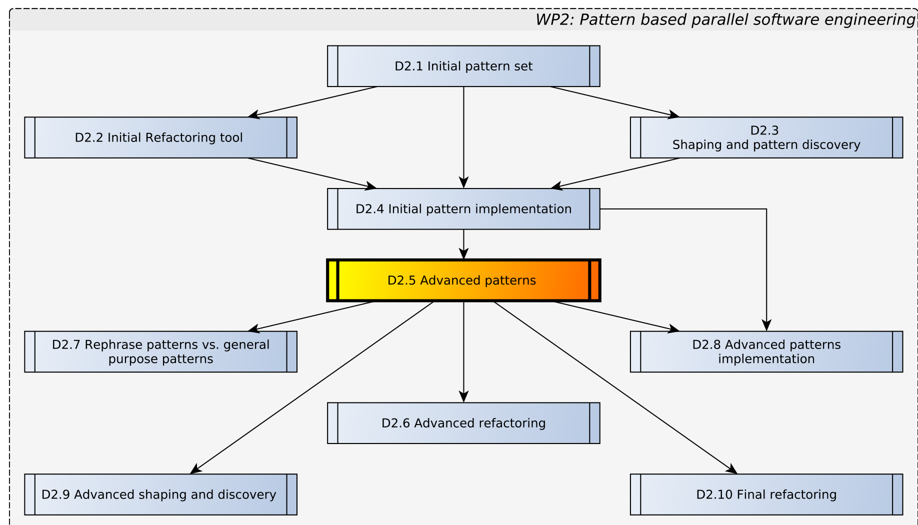
Rev.	Date	Who	Site	What
1	10/10/16	Marco Danelutto	UNIPI	Initial version
2	13/10/16	Michael Rossbory	SCCH	Syntax fixes
3	17/10/16	Christopher Brown	USTAN	Syntax fixes
4	20/02/17	Marco Danelutto	UNIPI	Updated changelog

Executive Summary

This deliverable is the fifth WP2 (“Pattern-Based Parallel Software Engineering”) deliverable. It comes at mid project to host all those patterns not initially included in the first deliverable (D2.1 “Report on the initial pattern set”, produced at M6 and including some well know, general purpose patterns) but some how “needed” to support the project use cases as well as some different applications currently developed by the project beneficiaries.

The deliverable therefore includes some new patterns divided into two main categories: “high level patterns”, that may be used to model the entire (or a large part of the) parallel structure of an application and “building block” patterns, that may be used in composition with other patterns to model and support more complex parallel application structures.

The main contributions to this deliverable may be summarized as follows: building block patterns and windowed patterns (UNIPI), pool pattern variants (UNIPI and SCCH), data intensive patterns (UNITO), image convolution pattern (EVO-PRO, UNIPI, UNITO).



Contents

Executive Summary 2

1 Introduction **4**

2 High level patterns **5**
- 2.1 Pool pattern 5
  - 2.1.1 Pattern description 6
- 2.2 Image convolution pattern 7
  - 2.2.1 Pattern description 7
- 2.3 Stream iteration pattern with multiple outputs 8
  - 2.3.1 Pattern description 9
- 2.4 DASP patterns 10
  - 2.4.1 Windowed stream farm 10
  - 2.4.2 Keyed stream farm 11

3 Building block patterns **14**
- 3.1 Inline stream generation pattern 14
  - 3.1.1 Pattern description 15
- 3.2 Stream merger 15
  - 3.2.1 Pattern description 15
- 3.3 Stream tupler 16
  - 3.3.1 Pattern description 16
- 3.4 Stream splitter 17
  - 3.4.1 Pattern description 17
- 3.5 Stream de-tupler 18
  - 3.5.1 Pattern description 18

4 *RePhrase* pattern set **19**

Introduction

Parallel design patterns have been recognized since a long time viable tools to overcome the problem of rapid and efficient development of parallel applications [2,5] and the *RePhrase* project has been designed to exploit the potential of parallel pattern in the data intensive application scenario.

At the beginning of the project, an initial set of patterns has been individuated that support the development of a full range of parallel applications, including some kind of data intensive parallel applications. The original set, described in deliverable D2.1 [4] included in fact general purpose patterns, such as pipeline and farms–for the stream parallel computations–and map, reduce and stencil patterns–for the data parallel computations–as well as more specific patterns suitable to support the development of data intensive parallel applications such as the divide&conquer or the Google mapreduce patterns.

As soon as the structure of the use cases individuated in *RePhrase* WP6 became more and more evident, over patterns have been individuated suitable to support different kind of data intensive applications (phases). These patterns have been investigated, in particular to verify that they are actual patterns, that is form of orchestration of parallel applications that may fit a number of different applications from different domains, and eventually distilled into a small set of patterns to be added to the ones already included in D2.1.

To this set of "high level" patterns modelling large parts of an existing application, we added a further set of more limited patterns, aimed at supporting other kind of data intensive parallel computations. These second kind of patterns mimic the typical operations supported by other, non patterned data intensive parallel programming frameworks such as Storm [7] and represent typical "building blocks" to be used in proper composition to support data intensive computations (parts) not modelled by the other, high level *RePhrase* patterns.

The deliverable first illustrates (Chap. 2) the new high level patterns included in the *RePhrase* pattern set. Then illustrates the new building block patterns (Chap. 3) and eventually briefly discuss the *RePhrase* pattern set as a whole.

High level patterns

This part of the deliverable hosts new "high level" patterns that where not included in the original D2.1 pattern set but that have been considered worth being introduced in the *RePhrase* pattern set due to their possible exploitation in use cases (WP6) and "internal" applications used to assess the *RePhrase* achievements.

Three different "high level patterns" will be included in the *RePhrase* pattern set:

- A *pool* pattern, derived from the one originally developed in the ParaPhrase project [1, 6], that models evolution of a population according to the principles typical of evolutionary computing
- An *image convolution* pattern, modelling typical computations appearing in a number of different applications, which may be actually considered a specialization of the *stencil* pattern already included in D2.1 pattern set.
- A set of data stream processing patterns, including a *stream iteration pattern* and a couple of *DASP* [3] patterns, modelling windowed processing of stream data, which are typically used in financial applications and may be considered a (kind of) specialization of the *stream accumulator* pattern in the D2.1 pattern set.

2.1 Pool pattern

The *pool* pattern models the evolution of a population of individuals. Iteratively, selected individuals are subject to evolution steps. The resulting new individuals are inserted in the population or discarded according to their "fitness" score. The process is iterated up to a given number of iterations (or up to a given computation time) or up to the point an individual with a given fitness is inserted in the population. Low fitness individuals may be removed from the population to keep the population size constant at each iteration.

Two versions of the pattern will be considered:

- a *synchronous* version, where iterations behave as barriers, that is all the evolution plus fitness computations are completed before considering the next iteration on the new population, and

- a *asynchronous* version, where individuals result of the evolution process are immediately evaluated and, in case, inserted in the population, asynchronously with respect to the processing of other individuals/evolutions.

2.1.1 Pattern description

Problem solved The pattern models the evolution of a population. In the pattern, a "candidate selection" function (s) selects a subset of objects belonging to an unstructured object pool (P). The selected objects are processed by means of an "evolution" function (e). The evolution function may produce any number of new/modified objects out of the input one. The set of objects computed by the evolution function on the selected object are filtered through a "filter" function (f) and eventually inserted into the object pool. At any insertion/extraction into/from the object pool a "termination" function (t) is evaluated on the object pool to determine whether the evolution process should be stopped or continued for further iterations.

Functional parameters The computation takes as an input a population $x_0, \ldots, x_k$ of individuals and computes in parallel the following function:

function POOL(P)
  while $not(t(P))$ **do**
    $N = e(s(P))$
    $P = (P \setminus s(P)) \cup f(N, P)$x
  end while
end function

As such, being α the type of the individuals, the following functional parameters are needed to define the pool pattern:

- t : α **collection** $\rightarrow$ **bool**, the function computing the termination flag
- s : α **collection** $\rightarrow$ α **collection**, the function selecting the individuals subject to evolution
- e : α **collection** $\rightarrow$ α **collection**, the evolution function
- f : α **collection** $\rightarrow$ α **collection**, the function filtering the population individual for next iteration of the pool pattern.

Non functional parameters Different non functional parameters may be provided to the pattern including (at least) the parallelism degree and a Boolean parameter telling whether the iterations should be blocking or not, or, in other words, specifying synchronous or asynchronous iterations. In the former case (synchronous iterations), the pattern computes exactly as stated by the algorithm above. First

compute all new individuals out of the selection of individuals identified as candidates for evolution, then select the better "evolved" individuals and eventually update population. Iterations happens only when both phases have been completed, in order. In the latter case, the process is continuous. This means that individuals are continuously selected for evolution, evolved and (if selected by the filter function f) put back into the original population. The process of any new individual is managed independently of the process relative to the rest of the new individuals.

Implementation The implementation of the pattern may follow a data parallel schema or a stream parallel schema. Asynchronous iteration mode naturally best fits the stream parallel schema, while synchronous mode naturally fits the data parallel schema where partitions of the selected population are evolved independently before submitting new individuals to some kind of coordinated filtering process.

Functional semantics The functional semantics naturally can be derived from the algorithm (function POOL(P)) described above.

Parallel semantics In principle, selection and evolution of individuals may be computed in parallel, while filtering is necessarily a coordinated (possibly centralized) activity when synchronous mode is adopted. In asynchronous mode, the selection, evolution and filtering of a single individual may be computed in parallel to selection, evolution and filtering of other individuals.

Component pre-conditions All parameter functions must be pure functions.

2.2 Image convolution pattern

The image convolution pattern computes image convolution according to some input "kernel" parameter. A kernel parameter is an $N \times N$ matrix (usually 3x3 or 5x5) of integer values. The image convolution is obtained from the source image processing each pixel at position i, j by taking the $N \times N$ values centered at i, j, multiplying each of the values by the corresponding value of the kernel and summing up all the results to get the new i, j pixel of the resulting matrix. Image convolution may be used to obtain different effects with different kernels, ranging from image blurring to image enhance, emboss, sharpen etc.

The image convolution pattern is clearly a specialization of the stencil pattern. It will be eventually provided as a *RePhrase* full pattern to capture and model common image and video processing parallel structures.

2.2.1 Pattern description

Problem solved The pattern transforms an image A into an image B computing each sub-pixel of B as a function of the corresponding pixel of A, of a small square

matrix of integers and of the pixels surrounding the original pixel (as many as the elements in the square matrix, centered at the original pixel).

Functional parameters The pattern takes a parameter specifying the "kernel" matrix K.

Non functional parameters Non functional parameter include the parallelism degree and the kind of decomposition policy used to partition the input image for parallel processing.

Implementation Input image is decomposed into submatrixes and each submatrix is computed in parallel. Different decomposition strategies may be used according to the correspondent non functional parameter.

Functional semantics The pattern transforms an $M \times N$ image A into an $M \times N$ image B using a $D \times D$ matrix K as follows:

- for each pixel $A_{i,j}$ the $K \times K$ matrix K' made of the K^2 items of A centered at $A_{i,j}$ is build
- $p = \Sigma_{\forall i,j \in \{0,K-1\}} K'_{i,j} K_{i,j}$ is computed
- $B_{i,j}$ is assigned the value p.

Parallel semantics All the pixels in the result matrix may be computed in parallel, in principle. In practice the input matrix will be partitioned according to some proper partitioning strategy and the partitions will be computed in parallel.

Component pre-conditions None

2.3 Stream iteration pattern with multiple outputs

This is a variant of the stream iteration pattern of D2.1. That pattern where assuming that a computation was repeatedly applied to a single stream item up to the point a given condition holded true. At the end, the result of the computation was delivered onto the output stream.

This pattern, instead, assumes that at each iteration a value may be output onto the output stream–depending on the value of a "output guard" function–such that the cardinality of the stream may vary.

2.3.1 Pattern description

Problem solved Generation of multiple items (zero, one or more than one) out of a single input stream item. All produced items should have the same type. The special input item EOS (end of stream) may itself produce multiple items (one or more than one) provided the last item produced is the EOS itself.

Functional parameters The functional parameters include those defined for the stream iteration pattern defined in D2.1:

- a function $f : \alpha \to \alpha$ computing a new value out of an input task, and
- a Boolean predicate $\mathcal{P} : \alpha \to \{true, false\}$ telling whether the result of the computation of f should be processed again or it may be delivered to the output stream

as well as new parameters driving the output process over the output stream:

- a Boolean predicate $\mathcal{O} : \alpha \to \{true, false\}$ telling whether the item computed by f should also be output on the output stream in case $\mathcal{P}$ tells it has to be processed again
- (possibly) a function $g : \alpha \to \beta$ to be used to derive the actual value to be output on the output stream in case the result of f is processed again[1]

Non functional parameters Non functional parameter include parallelism degree.

Implementation

Functional semantics The function computed by the pattern may be expressed through the following pseudo code:

while not(eos) **do**
  x_i = next input stream item
  if $\mathcal{P}(f(x_i))$ **then**
    feedback $f(x_i)$ as a new input item
  end if
  if $\mathcal{O}(f(x_i))$ **then**
    output $g(f(x_i))$ on the output stream
  end if
end while

[1] Ideally, this should be $g : \alpha \to \alpha$ or another function $g' : \alpha \to \beta$ (possibly same as g) has to be applied to the value computed by f when the value is output, to keep types corrects.

Parallel semantics Computations relative to different input items may be executed in parallel. Precedence should be given to the input items "feed back" from the pattern over the items coming from the input stream.

Component pre-conditions $\mathcal{P}, \mathcal{O}, f$ and g should be pure functions.

2.4 DASP patterns

2.4.1 Windowed stream farm

Problem solved The problem solved is to compute functions on "windows" of stream item values. In particular, this pattern implements a computation that outputs items on the output stream corresponding to the evaluation of a given function over successive, consecutive windows of items appearing on the input stream.

The "windows" have a length (number of items to be listed in the window) and an overlap factor (number of items in window w_i also appearing in window w_{i+1}). The number of items in a window may be defined either as an actual number (*count-based windows*) or as a time interval, that is as the items appearing onto the input stream within the given interval of time (*time-based windows*).

Functional parameters The functional parameters include:

- the window length w, expressed in terms of the number of items or the time interval to be used to fill the window with the items appearing onto the input stream;
- the window overlap size ov, that is the amount of overlap (possibly zero) between consecutive windows;
- the state to be kept (optional) between the processing of consecutive windows along with some kind of initial or "zero" state value;
- the function $f_w : \alpha^* \to \beta$ to compute over the window items to produce the result to be transmitted onto the output stream.
- the function $s_w : \alpha^* \times \gamma \to \gamma$ computing the new state value out of the current window and current state value.

Functional semantics The functional semantics of the pattern is described by the following pseudo code:

while not(eos) **do**
build window w_i out of w_{i-1} and further (size$(w_i) - ov$) items from input stream
$y = f_w(w_i, s_i)$
output y

$s_{i+1} = s_w(w_i, s_i)$
end while

Parallel semantics In case enough items are available to build several consecutive windows $w_i, w_{i+1}, \ldots, w_{i+n-1}$ the computation of the output and new state relative to window i may be started immediately after the state s_{i-1} is available. In case there is no state to be maintained, the computation of the n output values relative to the n consecutive windows available may be performed in parallel.

Implementation The pattern can be implemented as a variation of the standard farm pattern, with an emitter, a set of workers and a collector functionality. Several implementation strategies can be adopted. In the basic case, the emitter is in charge of handling the window management logic, i.e. it is responsible to buffer the input items, detect when a window is triggered and schedule it (with its content) to an available worker which is completely agnostic of the window management. Otherwise, more sophisticated implementations can be designed, in which the workers are active in the window management. Now, the emitter receives input items, and for each item determines the set of windows to which it belongs to. Windows are pre-assigned to workers (e.g., using round-robin assignment or other policies) and each tuple is scheduled to multiple workers that fill the corresponding windows and apply the business logic on them when triggered. A collector is used for interfacing purposes only, i.e. to gather the results of the windows and transmit them onto the output stream out of the pattern (possibly by re-ordering them if needed).

2.4.2 Keyed stream farm

Problem solved The problem solved is to compute functions on "windows" of stream item values. Each input item belongs to a unique class called *key*, i.e. in other words the physical stream can be viewed as a multiplexing a several logical streams, each one conveys the items with the same key value. This pattern implements a computation that outputs items on the output stream corresponding to the evaluation of a given function over successive, consecutive windows of items appearing on the same logical input stream.

The "windows" have a length (number of items to be listed in the window) and an overlap factor (number of items in window w_i also appearing in window w_{i+1}). The number of items in a window may be defined either as an actual number (*count-based windows*) or as a time interval, that is as the items appearing onto the input stream within the given interval of time (*time-based windows*).

Functional parameters The functional parameters include:

- the window length w, expressed in terms of the number of items or the time interval to be used to fill the window with the items appearing onto the input stream;

- the window overlap size ov, that is the amount of overlap (possibly zero) between consecutive windows;
- the state to be kept (optional) between the processing of consecutive windows of the same logical stream along with some kind of initial or "zero" state value;
- the function $g : \alpha \to \mathcal{K}$ that maps each input item to a unique identifier of the corresponding key (let $\mathcal{K}$ be the key domain);
- the function $f_w : \alpha^* \to \beta$ to compute over the window items to produce the result to be transmitted onto the output stream.
- the function $s_w : \alpha^* \times \gamma \to \gamma$ computing the new state value out of the current window and current state value.

Functional semantics The functional semantics of the pattern is described by the following pseudo code:

while not(eos) **do**
  build window w_i^k out of w_{i-1}^k and further (size$(w_i^k) - ov$) items from input stream
  $y = f_w(w_i^k, s_i^k)$
  output y
  $s_{i+1}^k = s_w(w_i, s_i^k)$
end while

Parallel semantics In case enough items are available to build several windows of different logical streams $w_i^k, w_j^{k'}, \ldots$ the computation of the output and new state relative to window i of key k may be started immediately after the state s_{i-1}^k is available. Both in case of an internal state or not, the computation on windows of different logical streams can be executed in parallel while windows of the same logical stream are always executed serially (both with an internal state or not).

Implementation The pattern can be implemented as a variation of the standard farm pattern, with an emitter, a set of workers and a collector functionality. The emitter is in charge of distributing input items to the workers according to any given internal policy such that all the input items of the same logical stream are always distributed to the same worker. Each worker builds and maintains consecutive windows of the assigned logical streams, and execute the business logic on each triggered window by producing the corresponding results to the collector. The collector in turn can simply forward the results over the output stream (results of the same logical stream are already ordered as they have been processed sequentially by the same worker).

Component pre-conditions Functions computing output and next state values are pure functions.

Building block patterns

The intent of the patterns listed in this part of the deliverable is to support the possibility to implement generic data stream processing applications "*à la* Storm/Spark". These patterns complement the ones already described in the Deliverable D2.1 and overall contributed to complete the scenario summarized in Table 4.1. We call these patterns *building block*s as each of them, alone, would not capture the whole parallel structure of a parallel application. Rather, they should be used in proper compositions–possibly including also some of the patterns originally listed in Deliverable D2.1–to model the parallel structure of an application. As we referred to the patterns of Chap. 2 as "high level" patterns, we will also call the patterns listed below "low level" patterns.

The patterns listed in this Section complement the patterns listed in D2.1 under the "Data collection" patterns. They provide possibilities to operate on different streams, merging, splitting, joining the streams with minimal or no modification at all of the items appearing on the input stream(s). In a sense, these pattern represent operators suitable to build complex data stream patterns out of the "computation" patterns, the patterns actually transforming input stream items to output stream results.

In most cases, these patterns will only support a sequential implementation, but these sequential implementation will support anyway the construction of complex *parallel* pattern expressions. As a consequence, the description of these patterns is more concise and does not include all the sections used to describe high level patterns or the patterns already listed in D2.1.

3.1 Inline stream generation pattern

This is regular Stream generator pattern, where the optional input from the input stream is actually mandatory. Therefore the inline stream generation pattern may be used in any computation to increase the amount of items on a stream. Each of the original stream items may be used to generate a number of new items that are output onto the output stream.

As an example, the pattern supports generation of $\langle word, 1\rangle$ $\langle key, value\rangle$ pairs in the Google mapreduce computing word count of a set of files. Supposing to have a stage producing a stream of filenames, the inline stream generator pattern will

receive the filename, open the file and output a $\langle word, 1\rangle$ for all the words read from the file.

3.1.1 Pattern description

Problem solved Produce multiple stream items (of type β) out of a single input stream item (of type α).

Functional parameters Functional parameters include the function $f : \alpha \rightarrow \beta$ **collection** producing zero, one or more β items out of a single α input.

Non functional parameters Non functional parameters include the parallelism degree and a Boolean ordering flag. If the flag is true, all items relative to the input item x_i must be output before any other item relative to inputs x_{i+k}.

Functional semantics Given a stream of input items $x_n, \ldots, x_2, x_1, x_0$, the pattern computes the stream resulting from the "flattening" of $f(x_n), \ldots, f(x_1), f(x_0)$.

Parallel semantics The generation of output items relative to different input items may be performed in parallel, possibly relying on some "re-ordering" stage working if the ordering flag is set true.

Component pre-conditions None.

3.2 Stream merger

The pattern accepts a number of input streams and produces an output stream hosting all the items appearing on the input streams, ordered according to a parameter policy. Default policies are provided that produce:

- time ordered merge (output items in the same order (perceived) of the arrival to the merger)
- user defined ordered (output items appearing at the same time at the merger inputs as sorted by a user supplied "compare" function)
- random merge (any ordering possible)

Optional parameters may be specified $f_1, \ldots, f_s$ such that f_i is applied to stream items x_{i_j} coming from input stream i and the result is delivered–according to the policy–rather that the plain x_{i_j}.

3.2.1 Pattern description

Problem solved Merge multiple streams into a single stream, possibly transforming the input items before delivering them to the output stream.

Functional parameters Functional parameters include the merge policy–from a finite set of predefined policies–and a set of functions $f_0, \dots, f_{k-1}$ such that input item x appearing on input stream i ($i \in [0, k-1]$) is delivered as $f_i(x)$ onto the output stream when stated by the current merge policy.

Non functional parameters Non functional parameters include parallelism degree.

Functional semantics The merge functional semantics is completely defined by the policy specified for the merge.

Parallel semantics Stream merge may be implemented with a tree structured parallel activity network, such that subsequent items of the input streams are actually merged in parallel.

Component pre-conditions None

3.3 Stream tupler

The pattern accepts a number of input streams and produces an output stream with tuples such that each tuple hosts items from the different input streams, built according to a parameter policy. Default policies include:

- *gatherall*: wait one item per input stream and deliver the tuple made by these items
- *time window*: wait for an amount of time and deliver the tuple made by the first item received (if any) on each of the input streams in the time interval.

3.3.1 Pattern description

Problem solved Merge streams into a single stream of tuples.

Functional parameters Functional parameters include the tuple build policy, from a finite set of pre defined build policies including the gatherall and the time window and–in case the policy is *time window*–the time amount to be used to gather the input stream items to be included in the tuple. Possibly, a set of functions '$f_0, \dots, f_{k-1}$ may be passed as parameter such that item x appearing on input stream i is included in the proper tuple as $f_i(x)$.

Non functional parameters

Functional semantics Functional semantics is completely defined by the tuple build policy:

- if the policy is *gatherall*, the pattern awaits for one item from each one of the input streams and output the tuple made of the k items from the k input streams, possibly transformed using the proper f_i functions.
- if the policy is *time window*, the pattern awaits for a given time amount and outputs the tuple hosting all the items appeared in the k input streams during the time amount, possibly transformed using the proper f_i functions. Items are sorted in the tuple per stream and FIFO.

Parallel semantics None

Component pre-conditions None

3.4 Stream splitter

The pattern generates a number of output streams out of a single input stream, according to a parameter policy. Default policies are provided that:

- *RoundRobin*: distribute round robin input stream items to the output streams
- *UserDef*: send input item x_i to output stream $h(x_i)$
- *Random*: randomly direct input items to output streams

3.4.1 Pattern description

Problem solved Split single input stream into multiple output streams.

Functional parameters Functional parameters include the splitting policy, from a set of predefined policies, and a function $h : \alpha \to \textbf{int}$ if the output stream target policy is a user defined one.

Non functional parameters

Functional semantics The functional semantics is completely defined by the parameter policy:

RR item x_i of the input stream is directed to the output stream $i \% n_s$ (being n_s the total number of output streams)

User item x of the input stream is directed to stream $h(x)$ (begin h a user defined function)

Rand item x_i is directed to stream r_i (being r_i the $i - th$ item in a pseudo number sequence R)

Parallel semantics None.

Component pre-conditions None.

3.5 Stream de-tupler

The pattern processes an input stream of tuples. Each tuple is used to generate items on different output streams according to a parameter policy. Default policies are provided including:

- scatter (tuple components to different output streams, in order)
- unicast (tuple components to the same output stream, one after the other, the stream is identified through a user supplied function)

3.5.1 Pattern description

Problem solved Split tuples appearing onto an input stream and deliver components to a set of output streams.

Functional parameters Functional parameters include the policy used to direct tuple items to output streams and, in case of unicast policy, a function $h : \alpha \rightarrow \textbf{int}$ determining the index of the output stream to be targeted out of the tuple component. Possibly, a set of functions $f_0, \ldots, f_{k-1}$ is included, such that items x directed to output stream i are output as $f_i(x)$ rather than as x.

Non functional parameters

Functional semantics The functional semantics is completely determined by the policy and by functions f_i (if present):

scatter item x_i in the tuple $\langle x_0, \ldots, x_m$ are directed to stream $i\%k$ (being k the number of output streams

unicast item i_i in the tuple is directed to stream $h(x_i)$

Parallel semantics None.

Component pre-conditions None.

RePhrase pattern set

We briefly recall in this Chapter the the different patterns considered in the *RePhrase*, including the ones introduced in this deliverable and those actually introduced in the first prepackage deliverable, that is in D2.1. The *RePhrase* patterns are summarized in Table 4.1, where they are classified according to different criteria:

HL/BB High level or Building block patterns, the former being more suitable to model full (parts of an) application, the latter more suitable to be used in compositions

SP/DP Stream parallel or Data parallel

P/S Parallel or Sequential implementation. Sequential implementation patterns are mostly Building block patterns used in composition to "glue together" other patterns

N/M Cardinality of the input and output streams, differentiating again "computation" (High level) from "building block" patterns.

It is worth pointing out how the new "building block" patterns, along with the patterns already included in the D2.1 *RePhrase* pattern set, support the creation of almost completely free data stream processing parallel applications. As an example, consider the application structure sketched on the Storm home page at `http://storm.apache.org/` (see Fig. 4.2 (top)). An application with this structure may be build using the *RePhrase* patterns as outlined in Fig. 4.2 (bottom).

Name	Where defined	Stream/Data parallel	High level/Building block	Parallel/Sequential	Stream-in/Stream-out
Pipeline	D2.1	SP	HL	P	1/1
Farm	D2.1	SP	HL	P	1/1
Stream accumulator	D2.1	SP	HL	P	1/1
Stream iteration	D2.1	SP	HL	P	1/1
Windowed stream map	D2.5	SP	HL	P	1/1
Stream iterator with multiple outputs	D2.5	SP	HL	P	1/1
Map	D2.1	DP	HL	P	1/1
Reduce	D2.1	DP	HL	P	1/1
Stencil	D2.1	DP	HL	P	1/1
Divide&Conquer	D2.1	DP	HL	P	1/1
Mapreduce	D2.1	DP	HL	P	1/1
Pool	D2.5	DP	HL	P	1/1
Convolution	D2.5	DP	HL	P	1/1
DASP: windowed stream farm	D2.5	SP	HL	P	1/1
DASP: keyed stream farm	D2.5	SP	HL	P	1/1
Stream generator	D2.1	SP	BB	S	0/1
Stream collapser	D2.1	SP	BB	S	1/0
Stream filter	D2.1	SP	BB	P	1/1
Inline stream generation	D2.5	SP	BB	S	1/1
Stream merger	D2.5	SP	BB	P	N/1
Stream tupler	D2.5	SP	BB	S	N/1
Stream splitter	D2.5	SP	BB	S	1/N
Stream de-tupler	D2.5	SP	BB	S	1/N
Data Splitter (formerly "Splitter")	D2.1	DP	BB	S	1/N
Data Merger (formerly "Merger")	D2.1	DP	BB	S	N/1

Figure 4.1: *RePhrase* pattern summary

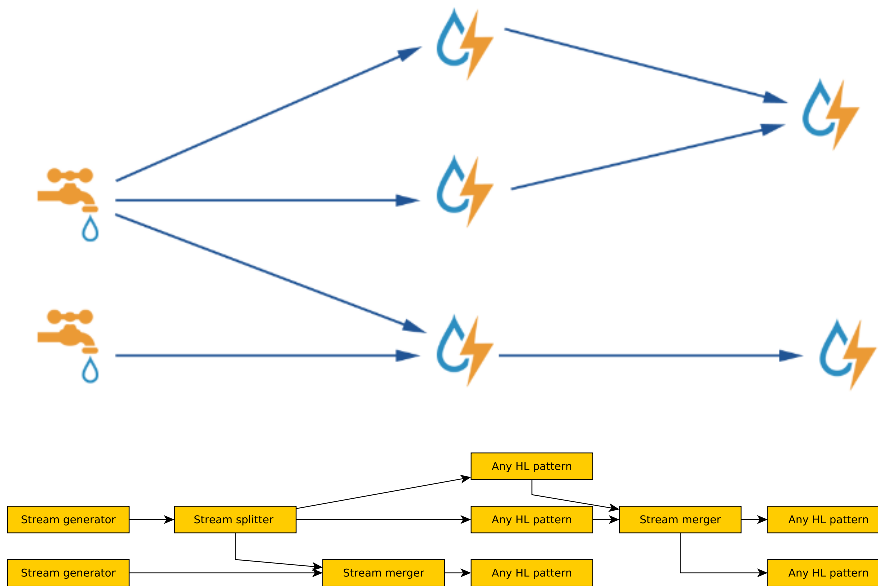


Figure 4.2: Typical Storm application picture/schema (top) and corresponding building *RePhrase* block pattern composition (bottom).

Bibliography

[1] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Pool evolution: A parallel pattern for evolutionary and symbolic computing. *Int. J. Parallel Program.*, 44(3):531–551, June 2016.

[2] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[3] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[4] *Rephrase* WP2. Report on the initial pattern set, 2015. available at http://rephrase-ict.eu/deliverables.html.

[5] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[6] ParaPhrase home page, 2015. http://www.paraphrase-ict.eu.

[7] Storm home page, 2016. http://storm.apache.org/.