International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Analysing Multiple QoS Attributes in Parallel Design Patterns-based Applications

Antonio Brogi · Marco Danelutto · Daniele De Sensi · Ahmad Ibrahim · Jacopo Soldani · Massimo Torquati

Received: date / Accepted: date

Abstract Parallel design patterns can be fruitfully combined to develop parallel software applications. Different combinations of patterns can feature different QoS while being functionally equivalent. To support application developers in selecting the best combinations of patterns to develop their applications, we hereby propose a probabilistic approach that permits analysing, at design time, multiple QoS attributes of parallel design patterns-based application. We also present a proof-of-concept implementation of our approach, together with some experimental results.

Keywords Parallel design patterns · QoS · Probabilistic analysis · Design time

1 Introduction

Parallel design patterns enable the development of parallel software applications by composing well-known, widely accepted programming abstractions [21]. When developing a parallel design patterns-based application, different functionally equivalent compositions of parallel patterns can be considerably different from a non-functional perspective.

The availability of a support for estimating — at design time — the QoS of a given composition of patterns would allow application developers to compose and select the parallel design patterns yielding the desired QoS.

The approach of experimentally measuring an application's energy consumption and completion time (for all of its possible configurations) may be unfeasible, since it may require too much time and resources [10]. Various approaches focus on optimising the QoS of a parallel design patterns-based application at compile time (e.g., [1],

Department of Computer Science, University of Pisa
Largo B. Pontecorvo 3, 56127 Pisa, Italy
E-mail: *{brogi,marcod,desensi,ahmad,soldani,torquati}@di.unipi.it*
This is a pre-print. Final version of this manuscript can be downloaded at http://link.springer.com/article/10.1007%2Fs10766-016-0476-8

[4], and [13]), or permit dynamically changing the configuration of a parallel design patterns-based application at runtime, if given QoS requirements are violated (e.g., [2], [7], [9], [19], and [20]). However, none of these approaches truly supports parallel application developers at design time, as in the former cases QoS is hidden to the developer, while in the latter cases QoS is only available at runtime.

In this paper we present a technique to probabilistically predict the QoS of parallel design patterns-based applications. We illustrate a probabilistic approach that permits analysing multiple QoS attributes in stream parallel applications. Our approach relies on two simple ideas. First, to permit analysing whatever configuration of parallel design patterns, we reduce the control flow of such configurations to an expression combining two basic cost compositors. Second, to deal with non-determinism (e.g., whether a processed item is fed back or not to the input stream for further processing), we perform our analyses by exploiting Monte Carlo simulations [11].

To show the feasibility of our approach, we present PASA (Probabilistic Analyser of Skeleton-based Applications), an open-source proof-of-concept implementation of our technique for analysing multiple QoS attributes. As we will discuss in Sect. 5, our first experimental results confirm that PASA is a promising support to qualitatively compare — at design time — arbitrarily complex combinations of parallel design patterns and to select the ones most probably yielding the higher degree of QoS.

The rest of the paper is organised as follows. Sect. 2 provides an example motivating the need for predicting an application's QoS at design time. Sect. 3 illustrates the technique we propose for analysing multiple QoS attributes of parallel design patterns-based applications. Sects. 4 and 5 illustrate a proof-of-concept implementation of our methodology and some experimental results, respectively. Finally, Sects. 6 and 7 discuss related work and draw some concluding remarks.

2 Motivating Example

Consider the (toy) image blurring application in Fig. 1. Such application processes a video sequence (i) by reading each of its image frames, (ii) by applying two subsequent blurring filters *Blur* and *Blur2* to each image frame, and (iii) by writing each filtered image frame back to the disk. If the blurring of an image frame is not satisfactory, the frame has to be blurred again and is returned (through a feedback loop) to the input stream of the first blurring filter.

A parallel implementation of such application can be developed by exploiting different configurations of parallel design patterns [21], e.g., via a pipeline of the four steps, via a farm of pipelines, and so on. Among all possible configurations, we

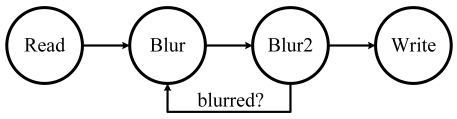


Fig. 1 An image blurring application (with a feedback loop).

Step	*Heavy*	*Light*
Read	1.84 mJ	1.84 mJ
Blur	14.44 mJ	4.99 mJ
Blur2	13.45 mJ	4.41 mJ
Write	1.95 mJ	1.95 mJ

(a) Energy consumption

Step	*Heavy*	*Light*
Read	0.144 msec	0.144 msec
Blur	3.991 msec	0.377 msec
Blur2	3.523 msec	0.335 msec
Write	0.151 msec	0.151 msec

(b) Completion time

Table 1 Resources required by each application step for processing a given image frame.

would like to select the ones most probably yielding the best QoS. For instance, we might be interested in minimising energy consumption and completion time.

Suppose, for instance, that we need to process two different types of image frames (e.g., *heavy* and *light*), whose processing consumes different amounts of millijoules, and which require different time intervals for being completed by each step (Table 1). Suppose also that our application is in a steady state where input frames arrive one right after the other (with negligible delay), that 10% of the image frames are *heavy*, and that the probability of a frame to be returned to the input stream is 0.3.

A natural question we would like to answer is the following: *What are the energy consumption and completion time of our application, if we implement it with a given composition of parallel design patterns?*

3 Estimating the QoS of Parallel Design Patterns-based Applications

Predicting the QoS of parallel design patterns-based applications is challenging (i) since we need to take into account the compositional nature of patterns, and (ii) because of the non-determinism characterising the input stream and feedback loops.

At design time, we do not know precisely which inputs our application is going to process, neither which of such inputs will be routed back by a feedback loop. An application developer can however distinguish different types of inputs of her application based upon the heaviness of their processing, and provide a probability distribution of input types (e.g., in our motivating example, we expect 10% of the input stream to be *heavy*, and the remaining 90% to be *light* — Sect. 2). She can also specify the probability of a feedback condition to get satisfied by an input stream item, thus characterising the probability of such item to be fed back to the input stream[1].

We hereby illustrate a probabilistic technique that, given a distribution of the input types and the probabilities of feedback conditions to get satisfied, permits predicting the QoS of parallel applications. Our approach relies on two simple ideas:

(i) To deal with configurations of parallel design patterns of whatever complexity, we reduce the control flow of a parallel design patterns-based application to a composition of two operators, called `Both` and `Delay` (Sect. 3.2).
(ii) To deal with the non-determinism characterising the input stream and feedback loops, we exploit Monte Carlo simulations [11]. Such simulations will be based

[1] Probability distributions can typically be derived from a requirement analysis or from monitoring data of previous implementations of an application.

upon a user-specified distribution of input types and on user-specified probabilities of feedback conditions to get satisfied by an input item.

3.1 Abstract syntax for parallel design patterns-based applications

We shall model parallel applications as compositions of the following core set of stream parallel design patterns: `Node`, `Comp`, `Pipe`, `Farm`, and `Feedback`.

- A `Node` is a basic activity, which processes the stream of data items arriving on the input channel and delivers the results on the output channel.
- `Comp` is a pattern that permits sequencing activities to be executed one after the other (e.g., in `Comp(A,B)`, `B` can start processing a given input only when `A` has completed processing such input).
- `Pipe` is a pattern to represent pipelines of parallel activities. Similarly to `Comp`, activities are executed in order (i.e., once an activity has completed processing a given input, such input can start being processed by the subsequent activity). It differs from `Comp` since an activity can start processing a new input once it has completed processing a given input, and while such given input is being processed by the subsequent activities.
- `Farm` is a pattern that permits running multiple instances (called *workers*) of the same activity to parallelly process different inputs.
- Finally, `Feedback` is a pattern that can be used to selectively route back results to the input stream (i.e., once a given input has been processed, it is either returned to the output stream or routed back to the input stream, if condition `cond` holds).

The above listed core patterns define the following abstract syntax[2]:

```
type QoS = float * float
type Activity =
  | Node of (InputType->QoS)
  | Comp of Activity list
  | Pipe of Activity list
  | Farm of Activity * int
  | Feedback of Activity * string * float
```

where `QoS` is used to represent multiple QoS attributes (viz., energy consumption and completion time in this paper) of an activity.

Example 1 Consider the application in our motivating example (Fig. 1). The (QoS of the) `Blur` node can be defined as follows:

```
let evaluateBlurQoS (inputType) : QoS =
  match inputType with
  | Heavy -> (14.44,3.991)
  | Light -> (4.99,0.377)
let Blur = Node(evaluateBlurQoS)
```

By defining also `Read`, `Blur2`, and `Write`, and by assuming 0.3 as the probability for the condition `"blurred?"` to get satisfied by an input item, we can define the whole application as follows:

[2] We employ F# [22] pseudo-code for all snippets in this paper, as our proof-of-concept implementation (Sect. 4) is developed in F#.

```
let motivatingExample =
  Pipe(Read,Feedback(Pipe(Blur,Blur2),"blurred?",0.3),Write)
```

Then, `motivatingExample` term can be given input to our probabilistic analysis, to estimate its QoS (as we will discuss next). □

3.2 Cost compositors for QoS attributes

To estimate the QoS of parallel design patterns-based applications, we will define (in Sect. 3.3) a structurally recursive function that associate, in a compositional way, each parallel design pattern with a cost structure. Please note that such cost structure is general, and it can be instantiated to define different QoS attributes, e.g., the time needed to complete an activity or the energy associated with its execution.

We reduce each parallel design pattern to a composition of two cost operators, called `Both` and `Delay`. `Both(A,B)` permits defining the cost of independently executing two activities whose costs are `A` and `B`, respectively. `Delay(A,B)` permits defining the cost of executing an activity whose cost is `A` and which can start only when another activity (whose cost is `B`) has been completed. The compositor `Both` is commutative and associative:

```
Both(A,B)=Both(B,A)
Both(A,Both(B,C))=Both(Both(A,B),C)
```

The compositor `Delay` is associative and right-distributive over `Both`:

```
Delay(A,Delay(B,C))=Delay(Delay(A,B),C)
Delay(Both(A,B),C)=Both(Delay(A,C),Delay(B,C))
```

We also explicitly name a neutral element `Zero` for `Both` and `Delay`:

```
Both(A,Zero)=Both(Zero,A)=A
Delay(A,Zero)=A
```

With `Both` and `Delay` we can naturally model the completion time and energy consumption of two activities. Let us denote with `aTime` and `bTime` their values of completion time. The completion time for running both activities in parallel is given by the maximum between `aTime` and `bTime`[3]. Instead, the completion time for delaying one activity after the other is obtained by summing `aTime` and `bTime`, as the delayed activity can start only after the first activity is completed.

```
let Both(aTime,bTime) = Max(aTime,bTime)
let Delay(aTime, bTime) = aTime + bTime
```

Let us denote with `aEnergy` and `bEnergy` the energy consumption of two activities. The energy consumed for independently executing `Both` activities can be approximated with the sum of their energy consumption. On the other hand, the energy consumed to `Delay` one activity after the other can be approximated by the energy consumption of the delayed activity (since delaying an activity does not increase the energy it consumes)[4].

[3] By taking the `Max` between `aTime` and `bTime`, we obtain an optimistic estimation of the completion time for executing `A` and `B` in parallel. Notice that different definitions of `Both` and `Delay` can be employed if needed. For instance, a pessimistic approach can estimate the completion time for executing both `A` and `B` in parallel by summing `aTime` and `bTime`.

[4] For the sake of simplicity, we employ a quite simplistic model for energy that abstracts from any overhead (e.g., idle times) due to concurrent execution of activities.

```
let Both(aEnergy,bEnergy) = aEnergy + bEnergy
let Delay(aEnergy, bEnergy) = aEnergy
```

3.3 Estimating the QoS of a composition of parallel design patterns

We hereby introduce a recursive evaluation function `exec` that estimates the QoS of a parallel application defined as a combination of the core parallel design patterns `Node`, `Comp`, `Farm`, `Pipe` and `Feedback` of Sect. 3.1. Essentially, `exec` (i) reduces a parallel design patterns-based application to a term combining the cost operators `Both` and `Delay`, and (ii) determines the `QoS` to be returned by evaluating the obtained term according to the cost composition rules described in Sect.3.2.

```
let rec exec (a:Activity, startIndex:int, endIndex:int):QoS=
  match a with
    | Node(evaluateQoS) ->  ...
    | Comp(aList)-> ...
    | Pipe(aList) -> ...
    | Farm (a,n) -> ...
    | Feedback(a,cond,prob) -> ...
```

The `exec` function inputs an `Activity a` and two indexes (`startIndex` and `endIndex`) identifying the portion of the input stream to be processed.

In the following we will assume the availability of a (global) list `inputStream` of `InputTypes`, which represents the input stream to be processed. Different categories of input items can be distinguished, each requiring different values of energy consumption and completion time. To set up a Monte Carlo simulation, the list modelling the `inputStream` is generated by exploiting a sampling function [11] whose behaviour strictly depends on the input stream to be modelled.

Example 2 Consider the input stream in our motivating example (Sect. 2), where 10% of the input image frames are *heavy*, and the remaining 90% are *light*. Suppose also that we want to build a small stream containing of 10000 image frames. The F# code to create a corresponding `inputStream` is the following:

```
type InputType = Heavy | Light
let rand = new System.Random()
let samplingFunction i =
    if rand.NextDouble() <= 0.1 then Heavy
    else Light
let inputStream = Array.init 10000 samplingFunction
```

Namely, we declare the `InputTypes` (`Heavy` and `Light`) and we define a sampling function, which returns `Heavy` with probability 0.1 and `Light` with probability 0.9. Then, we create an `inputStream` whose size is `10000` and whose content is generated according to the specified `samplingFunction`. □

3.3.1 Evaluating `Node` activities

The evaluation of a `Node` activity consists of estimating the QoS of such node for processing the portion of the `inputStream` identified by `startIndex` and `endIndex`:

```
| Node(evaluateQoS) ->
  let mutable nodeQoS = Zero
  for i=startIndex to endIndex do
    let iQoS = evaluateQoS(inputStream.[i])
    nodeQoS <- Both(iQoS,Delay(nodeQoS,iQoS))
  nodeQoS
```

Namely, we initially set the estimated `nodeQoS` to `Zero`. Then, for each item `i` from `startIndex` to `endIndex`, we evaluate the QoS `iQoS` of the current `Node` for processing item `i`, and we update `nodeQoS` with the cost of executing the current item `i` delayed after the previous items. Once all items from `startIndex` to `endIndex` have been processed, `nodeQoS` is returned.

3.3.2 Evaluating `Comp` activities

A `Comp` activity inputs a list of activities (`aList`). Its QoS can be estimated (i) by simulating the processing of each item of the given portion of the input stream by such a sequence of activities, and (ii) by ensuring that an item `i` starts being processed by the first activity in `aList` only when the last activity in `aList` has completed processing the preceding input `i-1`.

```
| Comp(aList)->
  let mutable compQoS = Zero
  for i = startIndex to endIndex do
    let mutable aListQoS = Zero
    for a in aList do
      let aQoS = exec (a,i,i)
      aListQoS <- Both(aListQoS,Delay(aQoS,aListQoS))
    compQoS <- Both(compQoS, Delay(aListQoS,compQoS))
  compQoS
```

More precisely, the estimated `compQoS` is initially set to `Zero`. Then, we compute the QoS (`aListQoS`) for processing each item `i` in the portion of the `inputStream` identified by `startIndex` and `endIndex`. Such `aListQoS` is simply obtained by executing each activity `a` in `aList` on the given input stream item `i`, and by delaying the cost of each activity after the preceding one. Then, `compQoS` is updated with the cost `aListQoS` for the current item `i` delayed after the cost for processing the previous items of the input stream. As soon as all items from `startIndex` to `endIndex` have been processed, `compQoS` is returned.

3.3.3 Evaluating `Pipe` activities

A `Pipe` activity inputs a list of activities. Its QoS can be estimated (i) by determining the cost of each of its stages for processing all items of the input stream from `startIndex` to `endIndex`, and (ii) by composing such costs with `Both`.

```
| Pipe(aList) ->
  let mutable pipeQoS = Zero
  for a in aList do
    let aQoS = exec(a,startIndex,endIndex)
    pipeQoS <- Both(pipeQoS,aQoS)
  pipeQoS
```

Namely, we estimate the cost (`pipeQoS`) of a `Pipe` by computing, for each activity `a` in `aList`, the cost `aQoS` for executing `a` on all items in the given portion of the

input stream (from `startIndex` to `endIndex`). Then, we compose all computed `aQoS` with `Both`.

The reason why we can compose all activities' costs with `Both` is that the cost of a `Pipe` can be approximated by the cost of independently executing all its stages over all the items in the given portion of the input stream. Namely:

- Since `Both` sums the values of energy consumption, we estimate the energy consumed by a `Pipe` as the sum of all costs for processing each item in the given portion of the input stream by each activity in `aList`.
- `Both` takes the maximum among the values of completion times, and this means that we estimate the completion time of a `Pipe` maximum among the completion times of its stages. Since completion times corresponds to multiplying the stages' service times by the size of the stream to be processed, we are actually estimating the completion time of a `Pipe` as the maximum among service times multiplied by the size of the stream to be processed.

3.3.4 Evaluating `Farm` activities

To estimate the QoS of a `Farm` (with a round robin scheduling policy) with `n` workers[5], (i) we partition the input stream among its workers, and (ii) we compute the cost for concurrently executing such workers, each processing the portion of the input stream it has been assigned to.

```
| Farm (a,n) ->
  let mutable farmQoS = Zero
  let workerStreamSize = (endIndex - startIndex + 1) / n
  for w=0 to n-1 do
    let wStartIndex = startIndex + w*workerStreamSize
    let wEndIndex = wStartIndex + workerStreamSize - 1
    let mutable wQoS = Zero
    if (wEndIndex  < inputStream.Length) then
      wQoS <- exec(a,wStartIndex,wEndIndex )
    else
      wQoS <- exec(a,wStartIndex,inputStream.Length)
    farmQoS <- Both(farmQoS,wQoS)
  farmQoS
```

Namely, we compute the size (`workerStreamSize`) of the portion of the input stream to be assigned to each worker. Then, for each worker `w`, we evaluate its QoS by estimating the cost required by the activity `a` (which is running on `w`) for processing the portion of the input stream assigned to `w`. Since all workers run in parallel, the cost (`farmQoS`) of the considered `Farm` is obtained by composing with `Both` all the costs `wQoS` required by the workers.

One may note that, strictly speaking, the above snippet does not implement a round robin policy, as the input stream is split into contiguous segments assigned to different workers. It is worth noting that, as we are setting up a Monte Carlo simulation (Sect. 3.3.6), the `inputStream` will be sampled according to a given probabilistic distribution. Moreover, we are interested on predicting the QoS based on the distribution of the input types in the input stream, rather than on the "identity" of

[5] For simplicity, when evaluating a farm with n workers, we assume the availability of n processing cores. In this way, we abstract from any delay due to scheduling n workers over $m < n$ processing cores.

each input stream item. This, along with the facts that the size of the stream is usually much bigger than the amount of workers, and that each simulation is going to be repeated a huge number of times [11], justifies evaluating a `Farm` by simply splitting the input stream among the workers.

3.3.5 Evaluating `Feedback` activities

A `Feedback` activity inputs the `Activity a` to be executed, and a condition `cond` that causes a processed input item to be routed back to the input stream with a certain probability `prob`. The actual satisfaction of a condition `cond` is unknown a priori (as it depends on the actual data items to be processed), and this introduces non-determinism when analysing a `Feedback` activity. To deal with such non-determinism, we rely on the probability `prob` to determine how many items are probably routed back to the input stream and to repeat the analysis over such items.

```
| Feedback(a,cond,prob) ->
  let mutable feedbackQoS = exec(a,startIndex,endIndex)
  let mutable f = 0
  for i=startIndex to endIndex do
    if(sampleCondition(cond,prob)) then f <- f+1
  if f > 0 then
    let fQoS = exec(Feedback(a,cond,prob),endIndex-f+1,endIndex)
    feedbackQoS <- Both(feedbackQoS,Delay(fQoS,feedbackQoS))
  feedbackQoS
```

More precisely, we initially set the cost `feedbackQoS` of the evaluated `Feedback` activity to the cost required by the inner activity `a` for processing the given portion of the input stream. We then compute `f` as the amount of items of the input stream that have to be routed back to the input stream. Then, to simulate the analysis of the `f` feedbacked items, we re-evaluate the current `Feedback` over the last `f` items of the input stream[6]. Such analysis results in a cost `fQoS` that is added to `feedbackQoS` (i.e., the cost of the feedbacked `f` items is delayed after the previously computed cost `feedbackQoS`) to compute the overall cost of the current `Feedback` activity.

Notice that, to compute the amount `f` of items to be fed back to the input stream, we exploit another sampling function, which simulates the non-deterministic behaviour of the feedback condition. Such function returns `true` with probability `prob` (i.e., condition `cond` is satisfied), `false` otherwise.

```
let rand = new System.Random()
let sampleCondition(cond:string, prob:float) =
    if rand.NextDouble() <= prob then true
    else false
```

3.3.6 Setting up the Monte Carlo simulations

In the previous sections we have shown how to generate the `inputStream` and how to evaluate the QoS of an `Activity a` (i.e., a given composition of parallel design

[6] As we noted in Sect. 3.3.4, we are interested on predicting the QoS based on the distribution of the input types in the input stream, rather than on the "identity" of each input stream item. Hence, it is enough to repeat our analysis on `f` items (most probably) satisfying such a probability distribution, such as the last `f` items of the given portion of the input stream.

patterns) for processing a given portion of such `inputStream`. We now need to put the pieces altogether for setting up a Monte Carlo simulation that permits predicting QoS of parallel design patterns-based applications.

```
let predictQoS(a:Activity,size:int,samplingFunction:int->InputType) =
  let inputStream = Array.init size samplingFunction
  let rec exec (a:Activity, startIndex:int, endIndex:int):QoS = ...
  exec(a,0,size-1)
```

The function `predictQoS` inputs an `Activity a` whose QoS has to be predicted, and the `size` of the `inputStream` to be processed. It assumes the availability of a global `samplingFunction` that permits populating the input stream according to a given distribution of input types. After defining function `exec`, `predictQoS` simply invokes such function to estimate the QoS of `a` for processing the whole `inputStream`.

Obviously, one invocation of `predictQoS` simulates a single execution of `a`. According to the Monte Carlo simulation theory [11], a reliable prediction can be obtained if we repeat a simulation a huge amount of times (each of which re-samples both the input stream and the feedback conditions) and properly aggregate the computed results. As one can expect, the higher the amount of iterations, the better the accuracy of the performed Monte Carlo simulation [11].

Example 3 Consider again our motivating example (Fig. 1). As we have shown in Example 1, we can define our application as follows:

```
let motivatingExample =
  Pipe(Read,Feedback(Pipe(Blur,Blur2),"blurred?",0.3),Write)
```

Suppose that, for instance, we now want to predict its energy consumption and completion time over an input stream built as shown in Example 2. To do it, we can iterate a huge amount of times the invocation of `predictQoS` and then average all estimated values of energy consumption and completion time.

```
let iterations = 1000000
let mutable avgEnergy = 0.0
let mutable avgTime = 0.0
for i = 0 to iterations - 1 do
  let iQoS = predictQoS(motivatingExample,10000,samplingFunction)
  avgEnergy <- avgEnergy + (fst iQoS)
  avgTime <- avgTime + (snd iQoS)
avgEnergy <- avgEnergy / (float iterations)
avgTime <- avgTime / (float iterations)
```

The above F# snippet shows how to compute the expected energy consumption (`avgEnergy`) and completion time (`avgTime`) for our motivating application for processing an input stream that contains `10000` images, and whose desired input type distribution is obtained by exploiting the given `samplingFunction`. □

4 Proof-of-Concept Implementation: PASA

To illustrate the feasibility of the approach described in Sect. 3, we developed a proof-of-concept implementation of it. PASA (Probabilistic Analyser of Skeleton-based

Applications — Fig. 2) is an open-source F# [22] application that permits predicting the QoS in parallel design patterns-based applications[7].

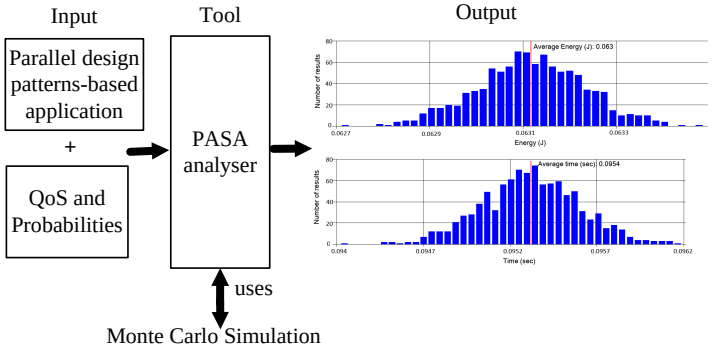


Fig. 2 Bird-eye view of the input-output behaviour of the PASA analyser.

PASA inputs (i) the description of a parallel application defined as a combination of basic activities (i.e., `Nodes`) and of the core stream parallel design patterns `Comp`, `Pipe`, `Farm`, `Feedback`, (ii) the size and the optional classification of the input stream (e.g., in our motivating example, we categorise the data items coming from the input stream as *heavy* or *light*, to distinguish the energy consumption and completion time they require for being processed), (iii) the QoS required by each node to process each type of input, and (iv) the probabilities of a given input type to occur and of a given `Feedback` conditions to get satisfied.

Given the above input, PASA predicts the QoS of the specified application for processing the input stream. More precisely, the input is passed to a back-end, which implements the recursive function `exec` and the Monte Carlo algorithm. The latter permits simulating multiple execution of the given composition of parallel design patterns on the given input stream. PASA generates `n` samples[8] (where `n` is exploited by users to indicate the total number of Monte Carlo iterations to be performed), each denoting an execution trace of the given composition. By applying `exec` to each sample, PASA computes the `QoS` (i.e., energy consumption and completion time) of each execution trace. The `QoS` computed for all traces are then aggregated to derive the desired prediction of QoS. For instance, by simply averaging all `QoS` values, it is possible to estimate the average energy consumption and the average completion time required by the specified application for processing the given input stream.

PASA also permits displaying the results of the performed analysis in different formats. For instance, it is possible to display such results in the form of histograms summarising the probability distribution of the estimated QoS (Fig. 2).

[7] The source code of PASA is publicly available at `https://github.com/ahmad1245/PASA`.

[8] The samples are generated with sampling functions following the probability distributions of input types and of `Feedback` conditions' satisfaction.

5 Experimental results

Consider again the image processing application in our motivating example (Fig. 1), which has to process a stream containing 10000 image frames of two different types (viz., *heavy* and *light*), each requiring different QoS to be processed. Suppose that we want to parallelise our application, and that we wish to qualitatively compare different compositions of parallel design patterns to select the one most probably yielding the higher degree of QoS.

First, we measured the energy and time required by each application step of our application to process an image frame of a given type[9]. All measurements were performed in isolation on a sequential version of the image processing application. In particular, energy consumption was measured by inspecting RAPL counters to measure the power consumption of the CPU [14]. The measured values of energy consumption and completion time are reported in Tables 1.(a) and 1.(b), respectively.

We defined a first parallel version of our application in PASA as follows:

```
let motivatingExample =
  Pipe(Read,Feedback(Pipe(Blur,Blur2),"blurred?",probF),Write)
```

where `probF` is the probability of the `"blurred?"` condition to get satisfied by a processed item. We compared the values of QoS predicted by PASA[10] (for different values of `probF`) with respect to those directly measured on a synthetic implementation of the same application in FastFlow [3]. As shown in Table 2, the predicted value for completion time is quite accurate[11].

Fig. 3 shows that, if we fix the amount of expected *heavy* input stream items (either to 10% or to 20%), and if we vary the probability of the feedback condition `"blurred?"` to get satisfied by a given item, PASA effectively identifies the trend of the multiple QoS attributes it analyses. Since this holds independently of the magnitude of the relative error, PASA can be exploited by parallel application developers who aims at qualitatively analysing multiple dimensions of the QoS of different parallel design patterns compositions.

To show an example of a qualitative analysis that we can perform with PASA, suppose that we want to assess the advantages of parallelising our motivating example application. Suppose that we have to process 1000 image frames (45% of which are *heavy*), and that we can exploit at most 16 processing units. Obviously, we can implement our application as (`C`) a composition of `Comp` and `Feedback`, or as (`P`) a composition of `Pipe` and `Feedback`:

```
let C = Comp(Read,Feedback(Comp(Blur,Blur2),"blurred?",0.2),Write)
let P = Pipe(Read,Feedback(Pipe(Blur,Blur2),"blurred?",0.2),Write)
```

We can also decide to build a `Farm`, each of whose workers is a composition of `Comp` and `Feedback`. In this case, since we can exploit at most 16 processing units, since

[9] Such measurements, as well as all the other experiments reported in this section, have been performed on an Intel Xeon Ivy Bridge (running at 2.40GHz with 12 cores 2-way Hyper-Threading), equipped with a Linux OS (version 3.14.49 x86 64 shipped with CentOS 7.1).

[10] Such values were obtained by performing 1000 iterations of the Monte Carlo simulation.

[11] Similar results were achieved when predicting the energy consumption of our application, even with an average relative error higher with respect to that affecting the predicted completion time (due to the extremely simplistic model we employed for estimating energy consumption).

Perc. of heavy items	Prob. of feedback	Predicted time (msec)	Measured time (msec)	Relative error
10%	0.1	8201.0	8344.4	1.718%
10%	0.2	9214.5	9301.5	0.935%
10%	0.3	10536.4	11256.9	6.400%
10%	0.4	12275.5	12528.9	2.023%
20%	0.1	12236.8	12395.5	1.280%
20%	0.2	13738.1	14324.5	4.093%
20%	0.3	15703.9	16457.9	4.582%
20%	0.4	18329.2	18657.8	1.761%
30%	0.1	16238.4	16606.4	2.216%
30%	0.2	18250.3	18464.0	1.157%
30%	0.3	20851.6	20822.8	0.138%
30%	0.4	24341.3	24280.2	0.252%
40%	0.1	20259.8	20404.9	0.711%
40%	0.2	22775.6	23109.0	1.443%
40%	0.3	26069.7	26023.0	0.180%
40%	0.4	30391.6	30342.9	0.160%

Min. err.	0.138%
Avg. err.	1.816%
Max. err.	6.400%

Table 2 Completion times for a pipelined implementation of our motivating example application.

each worker is a `Comp` (thus requiring a single processing unit), and since we have to spend two processing units for the `Farm`'s emitter and collector, we can instantiate from 2 to 14 workers:

```
let F2C = Farm(Comp(Read,Feedback(Comp(Blur,Blur2),"blurred?",0.2),
              Write),2)
...
let F14C = Farm(Comp(Read,Feedback(Comp(Blur,Blur2),"blurred?",0.2),
               Write),14)
```

Similarly, we can build a `Farm` whose workers are compositions of `Pipe` and `Feedback`. In this case, since each worker requires 4 processing units (one for each stage of the `Pipe`), and since we have to spend two processing units for the `Farm`'s emitter and collector, we can only instantiate either 2 or 3 workers:

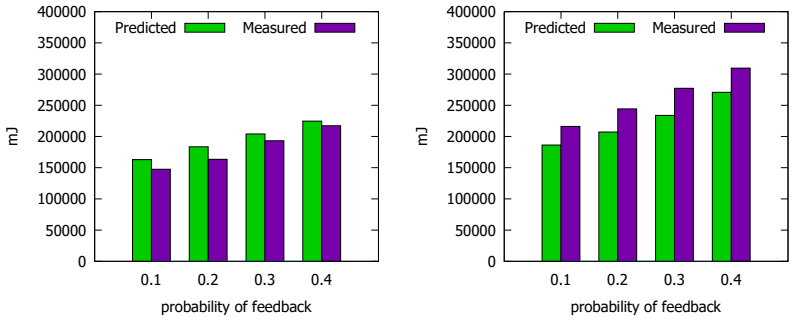
```
let F2P = Farm(Pipe(Read,Feedback(Pipe(Blur,Blur2),"blurred?",0.2),
              Write),2)
let F3P = Farm(Pipe(Read,Feedback(Pipe(Blur,Blur2),"blurred?",0.2),
              Write),3)
```

Another possible combination of patterns is a `Pipe` whose first and last stages are `Read` and `Write`, and with an intermediate stage that is a `Farm` whose workers are a `Comp` of `Blur` and `Blur2`. Due to the same restriction on processing units as above, we can instantiate from 2 to 12 workers in the intermediate `Farm`:

```
let PF2C = Pipe(Read,Farm(Feedback(Comp(Blur,Blur2),"blurred?",0.2),
               2),Write)
...
let PF12C = Pipe(Read,Farm(Feedback(Comp(Blur,Blur2),"blurred?",0.2),
                12),Write)
```

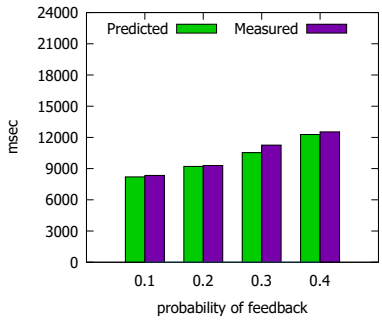
Finally, we can derive a different combination of patterns by simply exploiting `Pipe`s instead of `Comp`s to implement the workers in the intermediate `Farm` of the above `Pipe`. Since we increase the amount of processing units required by each worker, we need to decrease the maximum amount of workers that we can instantiate to 6:

```
let PF2P = Pipe(Read,Farm(Feedback(Pipe(Blur,Blur2),"blurred?",0.2),
```

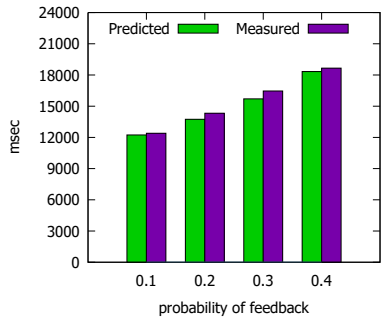


(a) Energy consumption (10% *heavy*)

(b) Energy consumption (20% *heavy*)



(c) Completion time (10% *heavy*)



(d) Completion time (20% *heavy*)

Fig. 3 Growth trends of the considered multiple QoS attributes (with fixed percentage of *heavy* items and varying probability of the feedback condition `"blurred?"` to get satisfied by an input item).

```
                2),Write)
...
let PF6P = Pipe(Read,Farm(Feedback(Pipe(Blur,Blur2),"blurred?",0.2),
                6),Write)
```

By running PASA, we can estimate the expected completion time for all identified combinations of parallel design patterns. We can then compare such combinations in the bidimensional space in Fig. 4 (where the x axis denotes the predicted completion time, while the y axis denotes the amount of processing units needed to run a given pattern combination) to assess the effect of increasing the amount of employed computing resources.

Furthermore, by including additional QoS properties (e.g., energy consumption) in the analysis (thus transforming the bidimensional space in Fig. 4 in an n-dimensional space) we can gain further information to decide whether to use one pattern combination or another.

6 Related Work

The importance of estimating the QoS of parallel applications is a widely recognised issue in the parallel design pattern community.

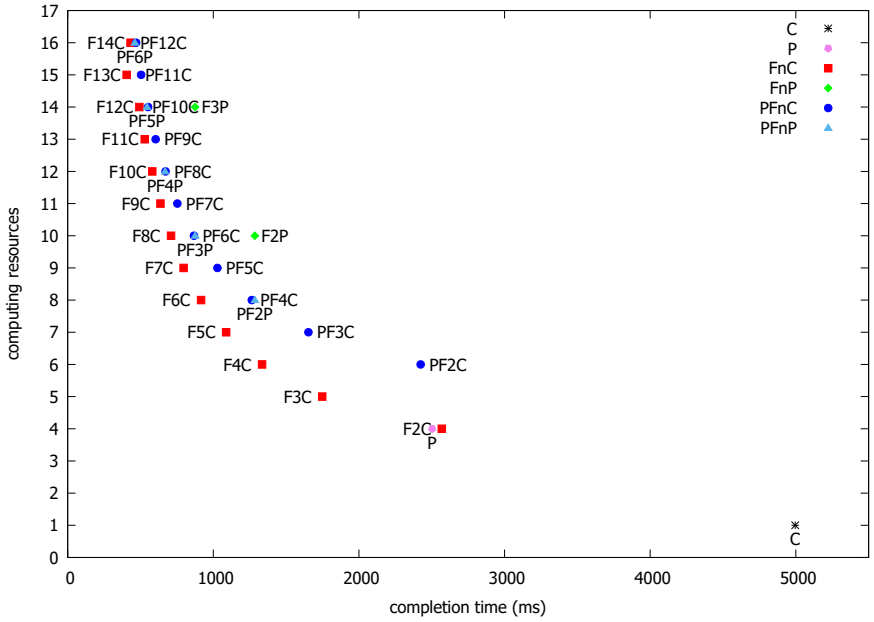


Fig. 4 An example of comparison for different composition of patterns.

[18] and [15] were among the earliest works to estimate the QoS of parallel applications. Given an application defined as a composition of data parallel patterns, [18] exploits a differential equation solver composition to generate alternative compositions, and to estimate the QoS, thus enabling their comparison. [15] instead translates a data parallel application into a program written in an intermediate language, and it abstractly interprets the obtained program to estimate the QoS of the original application. Both [18] and [15] differ from our approach since they target data parallel patterns and since they do not cope with non-determinism.

Other early work (e.g., [13], [1], [4], and [12]) done in this area focused on proposing transformation rules for different parallel design patterns to optimise the QoS of their compositions. [13] proposed two transformation rules to cover the data parallel patterns *scan* and *reduce*. By applying such rules to a network topology example, [13] shows that the resulting transformation yields a better QoS with respect to the originally specified composition of patterns. [1] extends the idea of [13] from data parallel patterns to stream parallel patterns (i.e., *pipe* and *farm*). Namely, [1] proposes transformation rules to convert a composition of *pipe*s and *farm*s to an equivalent normal form, which is essentially a *farm* of *comp*s. Experimental results demonstrate that the obtained normal forms provide better QoS with respect to original compositions. [4] is another example of a transformational framework that permits improving the (time) performances of parallel design patterns-based application by transforming a given composition of patterns into a functionally equivalent, but more efficient composition. [12] also proposed transformation rules for programs containing *shift*, *zip* and *map* skeletons to yield more efficient programs.

[13], [1], [4], and [12] assign the responsibility of optimising performances to the framework implementing the parallel design patterns, which however can optimise application performances up to a certain extent. Application developers are in a better position to optimise their code, but too-low level information cannot be conveyed to a developer, and this makes it difficult for her to optimise her code. [2], [7], [19], and [20] follow this motivation, and are suitable for scenarios where execution environment has to be hidden to developers.

[2] permits developers to provide a contract defining the desired QoS of a parallel design patterns-based application. Autonomic controllers continuously monitor an application, and in case of a mismatch between the QoS contract and the actual application performances, a reconfiguration of the application is planned and executed. [7] not only detects performance degradations in parallel design patterns-based applications, but also provide developers with explanations and suggestions on how to address them. Performance degradations and improvement suggestions are obtained by continuously monitoring applications and by using ad-hoc performance metrics. [19] and [20] are other examples of hierarchical autonomic management where autonomic managers cooperate to ensure a certain QoS. A wider discussion about autonomic managers can be found in [17].

Other approaches worth mentioning are [6], [8], [9] and [10]. [6] estimates the throughput of a stream parallel application in a given network configuration, by analysing the corresponding PEPA model [16]. [6] differs from our approach since it only considers throughput, and since it is limited to compositions of *Pipe*s.

[8] takes a composition of *sequence*, *pipe* and *farm*, and generates all alternative compositions having the same denotational semantics. Then, it computes the minimum number of threads needed by each alternative, and it selects the alternative requiring the lowest amount of cores to run. [8] differs from our approach since it does not consider non-determinism, and since it assumes that all activities take the same amount of time to be executed, which can be unrealistic in some scenario.

[9] and [10] instead focus on identifying the most appropriate configuration (in terms of CPU frequency and number of employed cores) to meet some given performance goals. [9] proposes an approach to reconfigure stream parallel patterns-based applications at runtime. Essentially, it monitors system utilisation and dynamically selects a new configuration if such configuration minimises energy consumption. [10] shows that testing all possible configurations to identify that yielding the optimal QoS requires a long time. To address this problem, [10] proposes an approach that can execute and monitor the application on few configurations, and then performs a linear regression on the monitored data to estimate the QoS of all remaining configurations.

Summing up, [1], [4], [12] and [13] focus on optimising the QoS of parallel design patterns-based application at compile time, hiding all QoS information to the developer. [10] permits determining the optimal configuration for a parallel design patterns-based application at deployment time, by requiring to run some application configurations, to monitor their performances, and then to exploit the retrieved information to estimate the performances of other possible configurations. [2], [7], [9], [19], and [20] instead monitor QoS at runtime, and reconfigure the running application if given QoS requirements are violated.

To the best of our knowledge, the only approaches focusing on estimating QoS at design-time are [6], [18], [15], and [8], which however are tightly coupled with a fixed set of patterns and do not deal with non-determinism. Our approach differs from all aforementioned approaches since it tackles the problem from a different perspective. Its novelty indeed resides in reducing whatever composition of parallel design patterns to the composition of two cost compositors (i.e., `Both` and `Delay`), and in exploiting Monte Carlo simulations to deal with the non-determinism introduced by input types distribution and by feedback loops.

Finally, it is worth highlighting the potentials and generality of our approach, which is not limited to parallel design patterns, but can also be adapted to predict QoS of other kinds of composite software applications. For instance, our previous work [5] shows how to employ `Both`, `Delay`, and Monte Carlo simulations to predict the QoS (e.g., response time, availability, or cost) of web service orchestrations.

7 Conclusions

When implementing a parallel design patterns-based application, QoS is one of the main discriminants to decide which of the possible pattern compositions to employ for implementing such application.

In this paper we have proposed a probabilistic technique that permits predicting the QoS of given compositions of parallel design patterns, and which relies on two basic ideas. First, in order to deal with arbitrarily complex compositions of parallel design patterns, we reduce each composition of patterns to a combination of two cost operators, which permit analysing the cost of independently executing two activities (`Both`) and of delaying one activity after the other (`Delay`). Second, to deal with the non-deterministic nature of the input stream and of `Feedback` loops, we employ Monte Carlo simulations [11].

Our approach is thought to be easily extensible. On the one hand, including additional QoS attributes in the analysis requires only to specify how the values of such properties have to be combined when two activity run independently or one after the other (i.e., by specifying how `Both` and `Delay` compose their values). On the other hand, to include other parallel design patterns in the analysis, it is enough to extend the abstract syntax for parallel design patterns-based applications (Sect. 3.1), and to extend the definition of the function `exec` to coherently simulate the added patterns. The extension of the set of the analysable QoS attributes, and of the set of considered patterns is in the scope of our immediate future work.

We have also illustrated PASA (Probabilistic Analyser of Skeleton-based Applications), a proof-of-concept implementation of our approach developed in F# [22]. Our first experimental results demonstrate that PASA provides highly reliable predictions for completion time (with an average relative error of 1.816%), while the prediction of energy consumption is currently less accurate. This is mainly due to the quite simplistic model we employed for estimating energy consumption. We are currently analysing more precise ways to combine the energy consumption of separate activities to improve the estimation of energy consumption.

Our experiments with PASA also prove that our probabilistic approach can be fruitfully exploited to qualitatively compare different compositions of parallel design patterns. Hence, PASA can be considered a promising technique for supporting parallel application developers at design time (e.g., for selecting whichever of the suitable combinations of patterns to employ to implement a parallel application). The extension of PASA prototype to support new patterns and frameworks (also easing the way of providing the necessary input) is in the scope of our future work.

PASA is not only useful for helping parallel application developers, but it can also be of great value for improving the compile time optimisation frameworks and the monitoring and reconfigurations solutions that we discussed in Sect. 6. In particular, we plan to develop autonomic managers that rely on the multi-attribute QoS analysis performed by PASA to determine which of the suitable configurations to employ to respond to a QoS or SLA violation and reconfigure an application.

Acknowledgements The research leading to these results has been partly supported by the project *Through the Fog* (PRA_2016_64) funded by the University of Pisa, and by the project *REPARA* (ICT-609666) funded by the European Union within the FP7 program.

References

1. Aldinucci, M., Danelutto, M.: Stream parallel skeleton optimization. In: Proceedings of the 12th International Conference on Parallel and Distributed Computing Systems (PDCS 1999), pp. 955–962. International Society for Computers & Their Applications (1999)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P.: Autonomic management of non-functional concerns in distributed & parallel application programming. In: Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS 2009), pp. 1–12. IEEE (2009)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: S. Pllana, F. Xhafa (eds.) Programming Multi-core and Many-core Computing Systems, Parallel and Distributed Computing, chap. 13. Wiley (2014)
4. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: The FAN skeleton framework. Parallel Algorithms And Applications **16**(2), 87–121 (2001)
5. Bartoloni, L., Brogi, A., Ibrahim, A.: Probabilistic Prediction of the QoS of Service Orchestrations: A Truly Compositional Approach. In: X. Franch, A.K. Ghose, G.A. Lewis, S. Bhiri (eds.) Proceedings of the 12th International Conference on Service-Oriented Computing (ICSOC 2014), *Lecture Notes in Computer Science*, vol. 8831, pp. 378–385. Springer (2014)
6. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Evaluating the performance of skeleton-based high level parallel programs. In: Computational Science-ICCS 2004, pp. 289–296. Springer (2004)
7. Caromel, D., Leyton, M.: Fine Tuning Algorithmic Skeletons. In: Proceedings of the 13th International Euro-Par Conference on Parallel Processing (Euro-Par 2007), pp. 72–81. Springer (2007)
8. Castro, D., Hammond, K., Brady, E., Sarkar, S.: Structure, Semantics and Speedup: Reasoning about Structured Parallel Programs using Dependent Types. Under consideration for publication in J. Functional Programming (2015)
9. Danelutto, M., De Sensi, D., Torquati, M.: Energy driven adaptivity in stream parallel computations. In: Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2015), pp. 103–110. IEEE (2015)
10. De Sensi, D.: Predicting Performance and Power Consumption of Parallel Applications. In: Proceedings of the 24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2016), pp. 200–207. IEEE Computer Society (2016)
11. Dunn, W.L., Shultis, J.K.: Exploring Monte Carlo Methods. Elsevier (2011)
12. Emoto, K., Matsuzaki, K., Hu, Z., Takeichi, M.: Domain-specific optimization strategy for skeleton programs. In: Euro-Par 2007 Parallel Processing, pp. 705–714. Springer (2007)

13. Gorlatch, S., Lengauer, C.: (De)Composition Rules for Parallel Scan and Reduction. In: Proceedings of the 3rd working conference on Massively Parallel Programming Models (MPPM 1997), pp. 23–32. IEEE (1997)
14. Hähnel, M., Döbel, B., Völp, M., Härtig, H.: Measuring energy consumption for short code paths using rapl. SIGMETRICS Perform. Eval. Rev. **40**(3), 13–17 (2012)
15. Hayashi, Y., Cole, M.: Static performance prediction of skeletal parallel programs. PARALLEL ALGORITHMS AND APPLICATION **17**(1), 59–84 (2002)
16. Hillston, J.: A compositional approach to performance modelling, vol. 12. Cambridge University Press (2005)
17. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing — degrees, models, and applications. ACM Comput. Surv. **40**(3), 7:1–7:28 (2008)
18. Jay, C.B.: Costing parallel programs as a function of shapes. Science of Computer Programming **37**(1), 207–224 (2000)
19. Kandasamy, N., Abdelwahed, S., Khandekar, M.: A hierarchical optimization framework for autonomic performance management of distributed computing systems. In: Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS 2006), pp. 9–9. IEEE (2006)
20. Khargharia, B., Hariri, S., Yousif, M.S.: Autonomic power and performance management for computing systems. In: Proceedings of the 3rd International Conference on Autonomic Computing, pp. 145–154. IEEE (2006)
21. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer-Verlag London (2003)
22. Syme, D., Granicz, A., Cisternino, A.: Expert F# 4.0, fourth edn. Apress (2015)