Model-Driven Development of Data Intensive Applications[☆] over Cloud Resources

Rafael Tolosana-Calasanz[a,*], José Ángel Bañares[a], José-Manuel Colom[a]

[a] *Computer Science and Systems Engineering Department*
Aragón Institute of Engineering Research (I3A)
University of Zaragoza, Spain

Abstract

The proliferation of sensors over the last years has generated large amounts of raw data, forming data streams that need to be processed. In many cases, cloud resources are used for such processing, exploiting their flexibility, but these sensor streaming applications often need to support operational and control actions that have real-time and low-latency requirements that go beyond the cost effective and flexible solutions supported by existing cloud frameworks, such as Apache Kafka, Apache Spark Streaming, or Map-Reduce Streams. In this paper, we describe a model-driven and stepwise refinement methodological approach for streaming applications executed over clouds. The central role is assigned to a set of Petri Net models for specifying functional and non-functional requirements. They support model reuse, and a way to combine formal analysis, simulation, and approximate computation of minimal and maximal boundaries of non-functional requirements when the problem is either mathematically or computationally intractable. We show how our proposal can assist developers in their design and implementation decisions from a performance perspective. Our methodology allows to conduct performance analysis: The methodology is intended for all the engineering process stages, and we can (i) analyse how it can be mapped onto cloud resources, and (ii) obtain key performance indicators, including throughput or economic cost, so that developers are assisted in their development tasks and in their decision taking. In order to illustrate our approach, we make use of the pipelined wavefront array.

Keywords: Cloud computing, model-driven development in cloud computing, Petri net performance modelling, big data application development.

1. Introduction

The confluence of cloud technologies, Internet of Things (IoT) sensors, and big data analytics has been recognised as the key enabling combination of technologies for innovation in a broad range of sectors, including military applications, the emerging smart grids, smart buildings, applications for e-health, or natural disaster prevention.

A common characteristic of all these applications is that they are data intensive, with data being generated continuously and coming from heterogeneous sources such as sensors or scientific devices. Furthermore, data generation rates can vary significantly, and the applications may often need to process data in a timely manner enabling systems to take corrective / strategic operational actions, or react at situations [1] urgently. For this reason, such applications make often use computational resources. We will refer to this kind of applications as continuous, data flow applications (CDFA), involving a wide range of applications, such as scientific workflows, pipelines, streaming applications, or any other data intensive application where data dependencies and concurrency play an important aspect.

It is difficult to have one solution valid for all these applications in any circumstance. An approach for their conception can make use of any of the existing, enabling framework. Some representative examples can be found

[☆] This work was co-financed by the Industry and Innovation department of the Aragonese Government and European Social Funds (COSMOS research group, ref. T93); and by the Spanish Ministry of Economy under the program "Programa de I+D+i Estatal de Investigación, Desarrollo e innovación Orientada a los Retos de la Sociedad", project identifier TIN2013-40809-R

[*] Corresponding author

in commercial clouds (e.g. Amazon, Google, or Microsoft), or in any of the more than 40 projects of the Apache Big Data Stack [2], which include the pioneer MapReduce computation framework, or others such as Flume, Spark, Storm, or Flink.

Such solutions often provide high-level operators that hide the inherent complexity of a distributed resources: Programmers can make use of the operators, but they typically do not contemplate the computational resource usage in their functional implementation (see the Spark implementation in Section 2).The availability of this rich enabling ecosystem can make the design process to abandon too soon the analysis to produce an implementation, avoiding a deep investigation of properties, or a wider exploration of alternative architectural designs and task mapping solutions onto distributed resources. This lack of analysis may result in some areas of the design space being insufficiently explored. In other words, if the analysis is done without an enabling technology in mind, a variety of alternative implementations might emerge [3].

Indeed, lifting the level of abstraction is the most effective way to manage problem complexity. In order to illustrate this statement, it is easy to see that programmers do not need to worry about writing parallel code [4, 5, 6], or about the cluster of machines where the application will be finally executed, i.e. these aspects can be abstracted away from the high-level design process of the application. However, since there is a strong relationship between the application logic and the execution environment that can affect its functionality and disturb the non-functional properties to be fulfilled, it is advisable to contemplate the execution environment in the abstraction of the system.

This paper is about how to manage the complexity of developing the logic of data intensive applications; initially, by taking into account the functional and non-functional requirements, to gradually incorporate the restrictions imposed by the implementation in later phases –a typical non-functional requirement can be scalability. An application can scale if it can exploit concurrency and data dependencies, which require a careful analysis of the application (e.g. via model analysis) to determine the existing opportunities to execute multiple activities at the same time, considering eventually features such as the variability of data and the executing environment. Hence, our main contributions are linked with the shortcomings, gaps and difficulties found in the development of cost-effective and performance sensitive CDFAs, managing the complexity of developing these applications from three different fronts [7]: (i) Lifting the level of abstraction for applications based on cloud frameworks, (ii) Providing a collection of patterns for parallel applications and cloud platforms that describe high-quality solutions to recurring problems, and (iii) Developing ad-hoc performance models to forecast the behaviour of particular patterns on specific platforms.

L. Lamport emphasizes two key ideas related to specifications in [8]: (i) It is better to handle the complexity of applications by abstractions, instead of hiding it; and (ii) specifications should not be written in program code. For an engineer designing a CDFA, these ideas can be translated to consider resource aware functional specifications instead of pure functional specifications; and avoid functional specifications that are written in the abstractions provided by a concrete implementation framework. The incorporation of resources *early* at the functional specification will allow developers to perform an analysis from the beginning that will lead to a wider exploration of the design space. This analysis must help find errors, and to think about concurrency and data dependencies as the main features on which the scalability, performance or economical cost requirements rest. The level of abstraction that requires the incorporation of resources to the functional level is not the same as the detail that might require the implementation on a *particular infrastructure*. It is easier to find and fix errors in higher-level abstractions than in the code of the implementation. It implies that a hierarchical model is required that can be refined for different execution platforms. Finally, the need for reasoning about performance and finding design errors lead to start writing formal specifications. The complexity that arises from the combination of distributed functionality, performance, and variability of the cloud environment involves the use of predictive models with different accuracy degrees [9, 10]. Summarizing, our research objective is a step-by-step methodology to guide developers for developing specification that incorporates the abstractions required by a CDFA, and help them to support the analysis and design with the use of formal models.

The general contribution in this paper is a model-driven and stepwise refinement methodological approach for data intensive applications executed in distributed systems.The central role in this methodology is assigned to a set of Petri Net models describing the functionality required and its behaviour from a non-functional perspective. Such Petri Net models allow a developer to analyse the behaviour of the system prior to and during its implementation and deployment. In particular, the analysis enables the exploration of minimal and maximal boundaries of the economic cost of the execution of the application, in relationship with its performance and its workload. Hence, in many cases, as an outcome of the analysis of the system, changes may be recommended or induced into the system with the purpose of modifying parts of the design and enforcing the agreed specifications. Such changes can happen at different

levels, i.e. at task specification level or at resource management level. Therefore, as a byproduct of the approach, the proposed methodology can improve the design and implementation's decisions taken during the development process of a streaming application.

On the other hand, cloud infrastructures are particularly well-suited for the computations of such applications due to their properties and they have typically represented the technology of choice for the aforementioned current framework projects. In particular, our approach considers the following properties of the cloud: (i) *on-demand resource provisioning*, (ii) the *dynamism of the cloud* execution environment caused by performance variation of machines, services competing for shared resources, and changing user quality of services requirements [11, 12]. The on-demand resource provisioning characteristic provides with a great versatility to our methodology. As a result of our analysis, as already stated, we can explore the alternative mapping solutions for the application tasks and their performance. Then, we can choose the most suitable one and exploit the cloud in order to configure and provision the computational resources at runtime. Furthermore, in case of performance degradation – it is known that cloud resources are subject to resource contention, leading to performance interference [13, 14] over time, and potentially impacting performance metrics, making the execution time to vary up to an order of magnitude – our approach enables the application to exploit the versatility and elasticity of cloud infrastructures to migrate into an alternative mapping solution. In this paper, however, our focus is on how to obtain the models and how to perform the subsequent analysis from them so that they can assist developers across the CDFA lifecycle. We are not focused on how to exploit a cloud infrastructure, but we will provide a discussion about it.

In a detailed way, the contributions of this paper can be summarised as follows: (i) A methodology that guides the design of CDFAs at all phases of the development life cycle. The methodology addresses functional and non-functional requirements together with the specification of the execution infrastructure and the involved resources. (ii) A formal component-based development to build models from existing components and capability to reason about the resulting composition. (iii) A guide of the possibilities of model reasoning for efficient and reliable design and / or optimisation, combining simulation, and approximate analysis. (iv) An integrated view of functional, performance and economic aspects of data intensive applications.

In [15], we illustrated the model construction of a data intensive application for an economic analysis. In [16], a preliminary specification language to support the methodology described here was provided. The language intends to simplify the definition of Petri net models, by providing high-level constructions on top of the formalism. In this paper, we extend the work in [15] by providing more details on both the methodological refinement process and on the subsequent analysis capabilities. The remaining part of this paper is structured as follows. The requirements of data intensive applications in general and the advantages of cloud technologies are discussed in Section 2, along with an example from the smart building scenario and implemented in the Spark framework. In Section 3, our methodology is presented in the context of streaming applications, we will refer to this kind of applications as Continuous Data Flow Applications (CDFA). In Section 4, the wavefront use case [17] is presented as a use case to validate the methodology. In Section 5, we describe how to build our Functional and Operational models. In Section 6, we illustrate how functional and operational models can be constructed for the wavefront algorithm. Then, Section 7 studies the use of different analysis possibilities of the Petri net (PN) underlying the model, and applies them to the wavefront model. In Section 8, the related work is briefly discussed. Finally, the conclusions are given in Section 9. Supplementary material can be found in the appendix, which presents the specification language to support the methodology, and illustrates the modelling of the wavefront use case.

2. Data-Intensive Applications and Current Technological Practice

Over the last years, the rapid development of science and engineering is generating a wide variety and large amount of datasets. Such data is growing larger and its location, availability and other properties are often dynamic, that is, dependent on time [18]. From a computational perspective, the amount of computational resources required for processing it is typically significant, such datasets often need to be processed within a time threshold; and, as a result of the output generated, automated control actions can be triggered. In this section, we will discuss a number of such applications and their requirements. In particular, we will describe the smart building scenario and we will show an implementation of it in the *Spark framework*. Spark is currently one of the popular frameworks in use, consequently, we believe that its exemplification can be considered as a good representative of current technological practice for this kind of applications.

2.1. Motivating Case Studies

An example of a data-intensive application can be found in the biological sciences. The study of marine ecosystems requires constant monitoring and analysis of undersea life and for such a purpose undersea video data is available. In many cases, human labor was used for undersea video analysis, but this is a tedious task. The EU-funded Fish4Knowledge project [19] developed algorithms and a distributed infrastructure in order to support automated video analysis. The idea is that videos are recorded and transmitted continuously for processing. Nevertheless, the processing of each video is computationally intensive and also needs to be conducted in a timely manner. On the other hand, in the automotive industry, automobile safety is becoming popular in order to reduce the consequences of traffic collisions. There can be complex scenarios within city boundaries that may require immediate reaction: A vehicle at a relatively low speed of 60 km/h can cover more than 3 meters in 200ms. Vehicles will incorporate sensors collecting measurements periodically, and the processing needs to respond fast, acceptable delays for collision avoidance systems should be below 10ms [20].

Besides, the transformation of power networks into smart grids requires a controlled charging of batteries for electric vehicles [21]. From a computational perspective, such processing requires periodic computations of charging schedules which considers electricity price, electrical constraints, and user preferences. The computations need to be done within a time threshold, where a breadth-first search algorithm is executed for each geographic area to prioritize on which vehicle should be selected for charging, given that demand exceeds supply).

Hence, we can conclude that there are a number of applications arising with the proliferation of distributed sensors. Many of them share common characteristics: They are typically computationally intensive, some require immediate response (like in the automobile safety scenario), while some others do not, as data elements arriving into the system need to be processed within a time threshold (deadline). Such a deadline is in the order of seconds, minutes, hours, or even days, rather than in the order of milliseconds, and this deadline is one of the key metrics in the Service Level Agreement (SLA). Moreover, the processing may involve the execution of complex simulations or control algorithms [21].

2.1.1. The Smart Building Scenario

Another similar data-intensive application is the smart building scenario. An advanced intelligent building management aims at reducing operational costs, while increasing the energy saving in large buildings via automated management actions. Therefore, on one hand, some physical variables are measured periodically by sensors deployed across a building infrastructure; such monitored physical variables often include temperature or humidity; and also people density across the building premises. On the other hand, a number of factors of the building are also considered, such as the construction materials, the structure of the building, or the building heat and mass balance. With all such monitored values and characteristics a number of computations can be accomplished, so that an automatic management action is subsequently taken, i.e. for each room, increasing / decreasing its temperature, customising its lighting, etc. In particular, the computations are often based on the EnergyPlus model [22], which is a simulation framework. Often the output of such simulations needs to be obtained before a given deadline, in time for taking any required control actions.

2.1.2. Spark and its Programming Model

Spark [23] is a popular cluster computing framework that arose to overcome some limitations of the MapReduce paradigm, since there are some applications whose control / data flow cannot be expressed efficiently, in MapReduce, as acyclic flows: (i) iterative jobs, when a function needs to be applied repeatedly over the same dataset, and (ii) interactive analytics, when exploratory queries need to be applied on large datasets. Spark offers programmers an interface that builds in implicit data parallelism. The execution is specially designed to be fault tolerant. Such interface consists of parallel operations such as map, filter, reduce, or foreach that are accomplised by passing closures (functions) to Spark. All these operations are based on the *resilient distributed dataset* (RDD) abstraction, which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.

In order to illustrate the usage of Spark, we make use of the aforementioned Smart Building scenario and we code the EnergyPlus based simulation scenario based on the computations specified at [22]. The following code fragment in Spark & Java for the Smart Building scenario can be seen in Listing 1. First, a one-second batch Spark context is created, thereby all the requests arriving during the previous second will be processed. Each request has

all the required data for a building simulation, namely the monitored data and the building characteristics. Then, after an input stream is created, the operations over the datasets are specified: (i) for each request (task), a number of simulations are derived; this is achieved by means of the parallel operator *flatmap*, which applies the lambda function generateJobs to each request (task); (ii) finally, each simulation is computed in parallel by means of the *foreachRDD* method. It should be noted that in Spark, developers specify first the data flow, but the actual computations do not start until the method *start* is invoked (line 16 of Listing 1).

Listing 1: Smart Building Scenario code in Java / Spark

```
1  public static void main(String[] args) throws Exception {
2    // 1 Create the context with a 1 second batch size
3    SparkConf sparkConf =
4        new SparkConf().setAppName("SmartBuildingEnergyPlus");
5    JavaStreamingContext ssc =
6        new JavaStreamingContext(sparkConf, Durations.seconds(1));
7    // 2 Create an input stream on target ip:port
8    JavaReceiverInputDStream<String> task =
9        ssc.socketTextStream(args[0], Integer.parseInt(args[1]),
10           StorageLevels.MEMORY_AND_DISK_SER);
11   // 3 For each task, create the EnergyPlus jobs
12   JavaDStream<Job> jobs = task.flatMap(x ->  EnergyPlus.generateJobs(x));
13   // 4 Execute each job: Perform the EnergyPlus simulation
14   jobs.foreachRDD();
15   // Start the application
16   ssc.start();
17 }
```

This example illustrates how the programmer specifies data flow operations in Spark, regardless of the computational resources required. The advantage of the approach is that the inherent complexity of the distributed and parallel systems is hidden from developers. Once the engine starts the execution of parallel operators on the receiving datasets, the computational resources required will be provisioned. Spark can be configured in a number of different ways, for instance, with the standalone Spark resource provisioning component or with Mesos [24]. Nevertheless, the disadvantage of the approach is that it can result in an inefficient resource provisioning, which is the essence of this work. Indeed, Spark resource provisioning components cannot operate with the information regarding the characteristics of the data flows (e.g. the workload prediction characteristics, such as arrival rate, predicted execution time, etc.) and consequently any elasticity operation at runtime can be challenging.

For each building, the Spark scheduler needs to execute the simulations (line 14). This step can be computational intensive and there is a trade-off between having the output in time (i.e. measured in terms of a significant percentage of simulations finished within a given threshold [22]) and the number of computational resources involved. The amount of computational resources is a key aspect, since too few resources may lead to an SLA violation, but a too high number of resources may incur in an increase in economic cost. However, even in this simple scenario, where jobs (simulations) have no dependencies among them, performance uncertainty may arise, leading to SLA serious violations. Indeed, in line 14 of Listing 1, the scheduler makes a decision on how many computational resource to allocate for the workload, but such decision can seriously affect the future workload to come, as it is completely unknown by the system the amount of computational resources required for the immediate future, and the choice on the number of used resources need to be done immediately. The solution adopted in [22] consists in killing running jobs (simulations) when more resources are needed and there was no free resource left, thus reducing the EnergyPlus simulation accuracy.

With our approach, by taking into account performance models that also consider computational resources, we can better estimate the number of resources required at any step and move from a purely reactive provisioning approach (that can potentially kill jobs) into a predictive one (that can potentially make a more efficient and intelligent use of resources).

2.2. Computational Capacity Requirements

Given all the previous characteristics of the data intensive applications, and also based on related previous work [18, 25], we have identified the following key computational resource requirements:

- Support for data-intensive workloads: The type of processing required is computationally intensive and it may involve a great number of computational resources. The computations may require complex distributed and parallel simulations, optimisation and control algorithms, or the computation of predictive models.
- Quality of Service (QoS) enforcement: Typically, each data element may need to be processed within a time threshold. Sometimes, such threshold is similar to the processing and transmission times, therefore no delay is allowed (like in the vehicle safety applications). In other cases, the processing time (S) is less than the overall due time for the control period (deadline), allowing data elements to be buffered prior to their processing. Moreover, in some scenarios, exceeding the overall amount of time for performing the computation may be allowed by the SLA of the application. The resource management policy should provide mechanisms for enforcing QoS.
- Elastic/On-demand Provisioning: The computational capacity must be adjusted to the overall requests. Therefore, a resource management strategy needs to allocate the appropriated number of computational resources to process unpredictable and variable workloads while satisfying QoS constraints, as described above. The underlying system must be able to support admission control and enable a variable processing rate per stream. The adopted mechanisms should be based on autonomic principles, so that they are resilient and self-adaptive to variations in the historical traces without requiring human intervention.

The cloud computing paradigm and related technologies are appropriate for these computational capacity requirements. Computational resources (i.e. CPU, network and storage) can be customised for the specific needs of the data intensive workload. Then, vertical / horizontal (de-)provisioning of resources can be exploited in order to accommodate the computational power to the workload demand, in such a way that QoS is enforced and the economic cost of the resource usage is minimised. In the following sections, we describe our methodology for building Petri-net based performance models that can be subsequently exploited to choose the appropriate target cloud paradigm. Our model supports two types of analysis, namely, performance and economic based.

3. Methodology for the Functional, Performance and Economic Analysis of CDFAs on the Cloud

In this section, we outline the principles for the modular construction of specifications that will guide the analysis, design and execution of CDFAs on the cloud. Application development typically starts with the requirements capture phase of functional, performance, and economic aspects of the problem domain. Any software development methodology must be founded on scientific-based, predictable, and rigorous models to be considered an engineering method. Formal models support the verification of design, coding and testing phases for finding a mismatch between a system's requirements and its actual implementation [26, 27, 8]. Unfortunately, most computer scientists are either not familiar with or reluctant to use formal methods [8]. Moreover, formal-model based analysis tools are only useful under certain assumptions (e.g. imposing some boundaries). Hence, it has so far been reasonably effective to tune an optimisation. On the other hand, simulation may be useful to discover some (un)desirable behaviours, but in general it does not allow to prove the (in)existence of some properties. Therefore, the combination of simulation and formal models for functional, performance, and economic analysis is necessary for efficient and reliable design and/or optimisation.

3.1. Requirements for our Methodological Approach

In order to support the methodology, we have identified the following modelling requirements that go beyond pure functionality:

- The methodology should define a **development process**, which is guided by the identified abstraction levels, and should provide a number of modelling artefacts, analytical methods, and guidelines to support it. The guidelines should include a *cyclical process* including analysis, design, and testing. It is not necessary to define

the whole set of requirements from early stages. The process can be based on a top-down approach, starting with functional requirements, and following a refinement process with non-functional requirements. Alternatively, we can start modelling the resource management of a concrete framework to accomplish some non-functional requirements, and following a bottom-up approach specify the functional specification on top. A mixed *bottom-up* and *top-down* design is required depending on the level being modelled, which enables the generation of reconfigurable and scalable specifications.

- The development of CDFAs must be supported by a **specification language** to describe a them as a collection of platform-independent building blocks. There are several streaming processing engines and frameworks based on particular parallel patterns to build and deploy tasks as distributed applications for commodity clusters and clouds. The language should support complementary capacities for the description of the application: Behavioural specification of concurrent processes, transformations operated over the data flow, and structural description of components that constitute the application. The reasonable approach is a general specification language that combines a *high-level language, describing the concurrency pattern or abstraction* to create very large programs. With such a general specification language, a user can plug in simple sequential programs expressed in whatever language to create very large parallel programs [28, 29, 17, 30, 31]. The specification should also support a formal *component-based development* to build models from existing modules and a capability to reason about the resulting composition. Hence, the specification must be executable to support both *analytical analysis* and *simulation*.

3.2. Synoptical View of the Methodology

There are many ways in which a system can be built to provide the same functionality with different concurrent behaviours and different deployments over distributed computing resources. Starting with the modelling artefacts, our methodological approach can be summarised with the equation '*Specification of Continuous Data Flow Applications (CDFA)* = Functional Entities + Communication / Synchronisation mechanisms + Data Dependencies + Resources'. The identification and characterisation of each building block of the proposed equation defines the basic specification elements. Figure 1 depicts a synoptical view of our methodology. In contrast to the Spark model of computation discussed in Section 2, it identifies three levels that are interleaved, namely functional, operational, and implementation. The overall life-cycle is finalized with implementation and monitoring phases:

- **Functional Level** The process starts by identifying the functional requirements of the problem domain and the outcome of this step is a functional model.
- **Qualitative Analysis** A functional model analysis can identify problems and help guide the redesign of the functional model aiming to achieve the maximum level of concurrency.
- **Operational Level** The operational level takes into account the execution platforms, and it explores the design space to select the design pattern that most effectively defines how to map processes to resources. The outcome is an operational model.
- **Quantitative Analysis** The integration of the functional and the operational model allows the designer to evaluate performance and reward functions. The analysis can help guide the redesign of the functional and operational models to meet non-functional requirements.
- **Implementation Level** This phase transforms the model into a flat model of processes that are deployed in a topology of cloud resources.
- **Monitoring**. The last step collects monitoring data from all used resources and applications. Collected data and developed models can help identify performance anomalies, and provide support to the autonomic principles of a Platform as a Service. The primary aim is to reduce human intervention, cost, and the perceived complexity by enabling the autonomic platform to self-manage applications [32].

The smart building scenario discussed at Section 3 was developed by means of the Spark model of computation, thereby the problem is solved in functional terms exclusively. In our approach, the functional level also contains

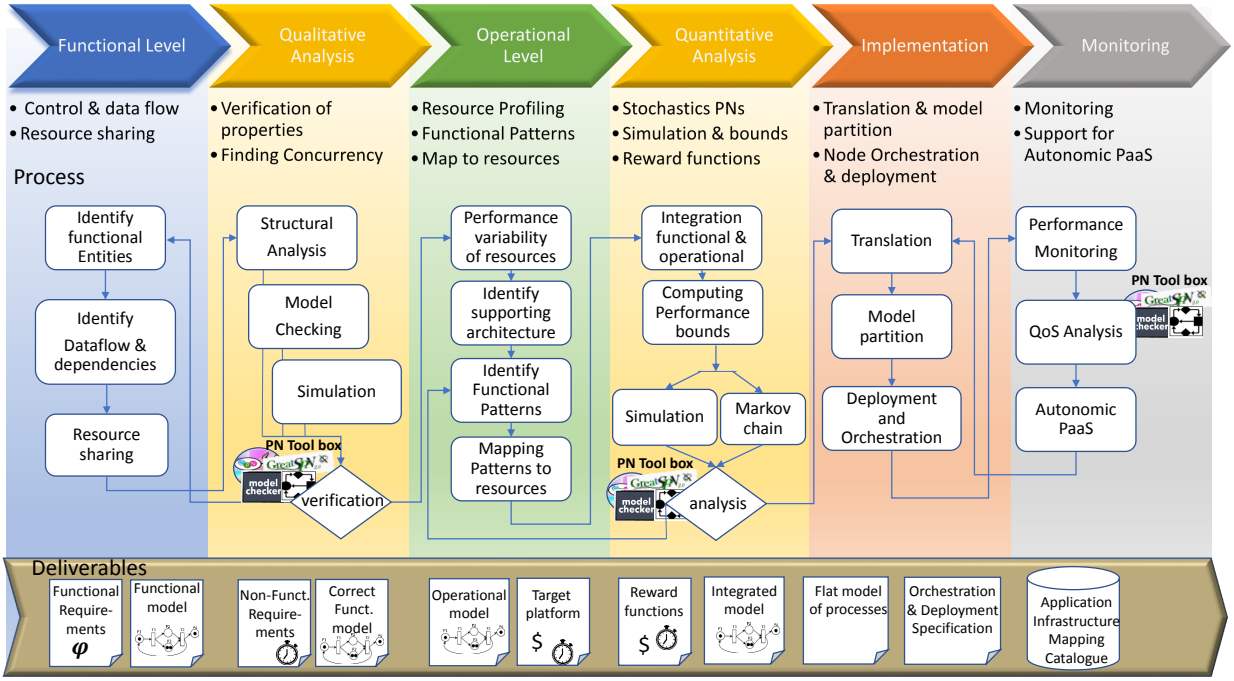

Figure 1: Life cycle and management methodology for CDFAs over cloud platforms.

the required resources but in an abstract way. For instance, each EnergyPlus job can be modeled with an associated resource, indicating the maximum degree of parallelism. Then, in the operational level, the actual resources from a resource pool could be incorporated. If this number is less than the number of jobs, then the maximum degree of parallelism could be accomplished. The advantage of doing so is that we can now simulate what-if scenarios (e.g. by varying workload prediction characteristics, such as arrival rate, predicted execution time, etc.), and study how the execution of EnergyPlus jobs arriving at time t with a number of resources can affect the performance of jobs arriving at time $t + 1$. As a result the analysis, the implementation could follow a policy thereby resources are used in a more efficient way in comparison to Spark's approach.

On the other hand, the semantics of the component-based language will be defined formally in terms of ordinary PNs [33] in order to translate all the advantages derived from a mathematically based model to the methodology –e.g. Analysis, Verification, or Equivalence Relations. The consideration of PNs is based on the natural descriptive power of concurrency, but also on the availability of analytic tools coming from the domain of Mathematical Programming and Graph Theory. Moreover, taking into account that PNs are executable specifications, PN models can also be simulated. In summary, PNs are a formalism that combine simulation and analysis techniques. They can be exploited in order to improve the understanding of the system, and to obtain performance or economic boundaries. A survey of PN tools can be found in [34].

The final Petri Net models can be used as a universal specification across multiple platforms and languages. Figure 1 shows how the PN tool box can be exploited at all the methodological steps. The translation of the models specified by a Petri net to an actual system with the same behavior, or the development of efficient interpreters that directly execute the models have received considerable attention from researchers over the years [35, 36]. Regarding the last step, we also described our autonomic-based architectural approach in [32].

3.3. Methodological Phases Addressed in this Paper

As we have just seen, our methodology consists of a sequence of phases. This paper focuses on the first two ones, where the functional and operational levels are developed, but we also highlight how the Cloud infrastructures can benefit from our approach. Our models for CDFAs can be exploited to understand the most appropriate approach for

solving the problem: Strategy(ies) for decomposing the problem into processes, communication needs and resources required for satisfying functional and non-functional requirements.

The proposed approach decomposes the construction of the model in phases:

- *Functional level*. Parallel programs solve big problems by simultaneously executing different parts of the problems on different processing resources, which is possible if the problem contains exploitable concurrency, and the first design phase is *Finding concurrency* [3]. This level can be itself divided at least into two different levels:
  - *Control level*. At this phase, the designer perceives a CDFA as a set of computational threads, following predefined computational processes. Such computational threads can request, in a competitive way, different quantities of a finite number of shared conservative resources, and they can interact throughout the production/consume of dataflows. The attention is focussed on the study of the problems arising when the shared resources must be granted to a set of concurrent and communicating processes.
  - *Data-flow level*. The data-flow level may be the result of a refinement of the control level incorporating data and functions to the model. The result of a data-flow can be part of the control flow represented by conditions or guards, and the control level triggers the execution of functions and data transfers.
- *Operational level*. The model is transformed to take into account the execution environment, adapting the functionality to the set of available resources and physical channels used to routing dataflows. At this phase, the actual characteristics of the resources used arise, such as resource capacity, economic cost, or performance. The aim of the operational level is to refine the functional design and move it closer to the involved resources. This level may be again divided at different levels, the most noteworthy of which are as follows:
  - *Enrichment of the model with target platform features*. At this phase, the model is enriched with specific information of the target platform such as execution time, cost or performance variability. We can analyse at this phase how expansive (cost/time) is to compute a function or to share information. We can specify a number of what-if scenarios such as if there is hardware support for a shared memory, or if nodes are connected by slow or fast connections.
  - *Mapping to the resources*. The functional level specifies the concurrency in terms of computational processes, data transfers and involved resources. The next step in this phase is to specify how this concurrency can be mapped onto the resources. Mattson et al. [3] call this phase the *algorithm structure design space*, and identify three major principles: Organisation by tasks, data decomposition, and flow of data. For example, a matrix multiplication can exploit the data structure by decomposing it in chunks that can be operated concurrently, but if the matrix multiplication is done in the context of a problem with data injected in a stream fashion, a pipeline pattern can be more adequate.
  - *Modelling of Supporting Structures*. We can immerse in the modelling of details of the supporting specific target frameworks and programming mechanism. For example, the framework can support master/worker for dynamically balance the work among resources and queues for supporting a large number consumers accessing it.

4. Case Study: the Wavefront Algorithm

In order to illustrate our methodology, we are making use of the Matrix-Vector Multiplication problem in streaming fashion, in particular, the Wavefront Algorithm, which represents a simple solution for large arrays. The wavefront is a paradigmatically, regularly structured abstraction that allows developers to focus on simple sequential programs to create very large parallel programs. In [37], authors present abstractions as *All-Pairs*, *Wavefront* and *Makeflow* highlighting the idea that due to their regular structure, they are more tractable to provide robust and predictable implementation of workloads. In this sense, abstractions are similar to *systolic arrays* and *wavefront arrays* studied by Kung and Leiserson, sets of regularly interconnected simple processor operating respectively in a synchronous or asynchronous way [38]. By using the regular structure and declarative specification, a wavefront may be materialised in different ways on distributed, multicore, and distributed multicore systems showing different performances.

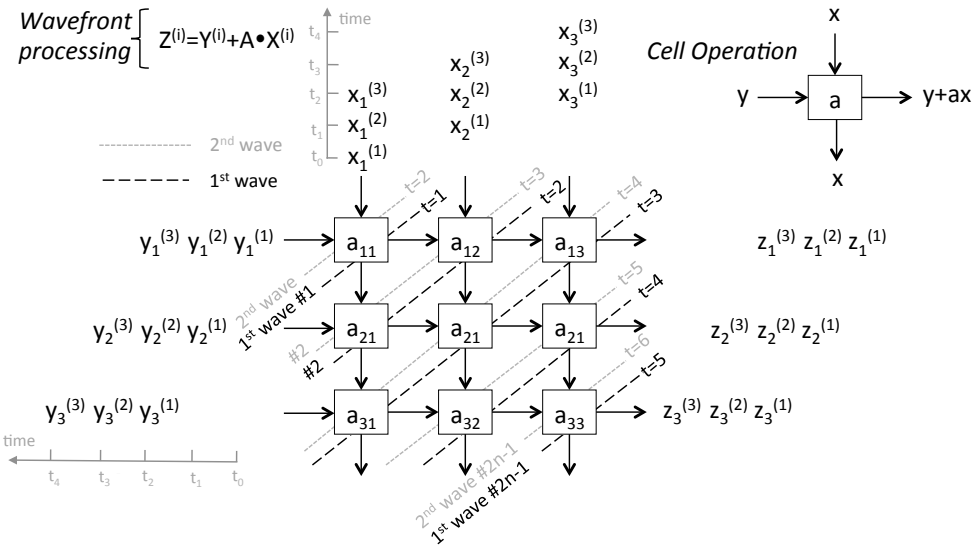

Figure 2: Wavefront processing of $Z^{(k)} = Y^{(k)} + A \cdot X^{(k)}$, $k = 1, 2, ...$

The parallel pipelined wavefront algorithm was originally described by Lamport in his classic paper [39] to execute loops on a multiprocessor computer. Since then, researchers have proposed performance models and solutions for specific applications and platforms: Algorithms for determining the optimum wavefront partition into sections assigned to individual processors [40], models for performance analysis of wavefronts implemented on message passing environments in large-scale distributed systems [41], large scaled shared memory multiprocessors [42], Commodity Processor Cluster Systems [43], or distributed clusters of distributed of CPUs and GPUs [44]. However, previous analytic performance models can not be generalised or reused in unexplored new generations of high performance computing system such as the cloud [45, 43]. The interest in models for performance analysis of wavefronts is due to the broad spectrum of applications. It can be found in simulation problems in economics and game theory [37], problems of sequence alignment via dynamic programming in genomics [46, 47, 48], real-time video coding [49], stereo vision and obstacle avoidance [50], or optimal motion planning of multiple robots [51].

Beyond its applicability, the election of a wavefront presents interesting implementation and performance modelling challenges on distributed memory machines because they exhibit a subtle balance between processor utilisation and communication cost [52, 53, 37]. Due to its regular structure, the wavefront array provides a concise specification, but it also imposes some message passing constraints that can limit the available parallelism in the algorithm, and it is not easy to reason about its behaviour or performance in an intuitive way.

Let us examine how an algorithm from Linear Algebra, a Matrix-Vector Multiplication problem in streaming fashion, can be executed on a square, orthogonal 3×3 wavefront array (Figure 2).

Let $A = (a_{ij})$ be a 3×3 matrix, and let $X^{(k)} = (x_i^{(k)})$, $Y^{(k)} = (y_i^{(k)})$ and $Z^{(k)} = Y^{(k)} + A \cdot X^{(k)} = (z_i^{(k)})$ be 3×1 matrices for $k = 1, 2, 3$. Initially, the elements of A are stored in the array of processors (a_{ij} in processor P_{ij}). The elements $x_i^{(k)}$, for $k = 1, 2, 3$ are stored from data streams on the top of the i-th column of processors. The elements $y_i^{(k)}$, for $k = 1, 2, 3$ are stored from data streams to the left of the i-th row of processors. The computational process starts with processor P_{11}, where $y_1^{(k)} + a_{11}x_1^{(k)}$ is computed. The appropriate data is then propagated to the neighbour processors P_{12} (the result of P_{11}) and P_{21} (the input data on the top of P_{11}, $x_1^{(k)}$), which execute their respective (similar) operations. The next front of activity will be at processors P_{31}, P_{22} and P_{13}. At the end of this step, P_{13} outputs $z_1^{(1)}$. A computation wavefront that travels down the processor array appears. Once the wavefront sweeps through all the cells, the first computation for $k = 1$ is finished. Similar computations can be executed concurrently with the first one by pipelining more wavefronts in succession immediately after the first one. The wavefronts of two successive computations never intersect, since once a processor performs its share of operations for a given computation, the next set of data that it will receive can only be from the next computation.

5. Modular specification of a CDFA as a Native Cloud application

In this section, we discuss about the characteristics of Native Cloud Applications (CNA) and we describe how to build our Petri-net-based functional and operation models from our methodology.

5.1. Cloud Native Applications

Native cloud applications (CNA) are designed to take full advantage from the cloud computing characteristics. Fehling et al. identified five essential architectural properties to make up a CNA [54], which sets out the steps and patterns that need to be taken into account. The first property is the **decomposition** of the functionality in chunks of distributed functionality. A CNA is made up of components that can be scaled out independently. Decomposition can be based on layers, processes, or data flows. Wavefront and dataflow computations are characterized by a data dependent flow of computation [45]. When the data flow is regular and static, a pipeline pattern is the best option to represent the function decomposition, and alternatively an event processing network (EPN) is the the more adequate pattern to represent a data flow with irregular interactions and unpredictable intervals [3, 28]. Independently of regular or irregular features, data flows are made up of two type of constructive elements that can be found in popular stream processing engines designed for the cloud such as Flume, Storm, Spark, or Flink: Computational processes that provide a certain function that is performed on input data and produces output data, and data transmission processes that support communication among computational processes.

The next two properties correspond to the design space or operation model that comprises the analysis of the application **workloads** and the analysis of how the application handles **state** and how expensive it is to share information. The analysis of application workloads involves the way a cloud deployment model can provide a pool of resources that can be automatically managed to provide elasticity to react to varying workloads, and mechanisms to enable a pay-per-use billing. *Application profiling* is strongly associated with the workload analysis. Profiling must collect a large amount of data generated by the cloud resources and forecasting models are fed with these data to analyse resource contention and service degradation. A survey on forecasting and profiling models for cloud applications can be found in [55]. An important aspect related to the workload is to consider the dynamic nature of the cloud, which can be caused by performance variation of machine instances offering the same capability, and by services that are deployed, updated and destroyed all the time giving rise to a dynamic competition for shared resources [11, 12]. CDFAs on the cloud are complex systems where customers and resources have not identical characteristics , and exponential distribution does not adequately model observed inter-arrival and service times. Therefore, traditional queuing systems are not feasible as forecasting models to obtain accurate performance evaluations. In these cases, we can use approximate methods to compute performance bounds, and complemented with simulation [9, 56, 10]. In addition to the mean service time and mean inter-arrival time, the coefficient of variation of resources and inter-arrival time has been proposed to introduce the dynamic nature of cloud applications and streaming applications on the cloud [11, 57, 10, 31]. In our case, profiling data is essential to feed our models with time distribution annotations to estimate bounds. Besides, the explicit modeling of resources at different levels of abstractions supports the subsequent analysis of resource contention by simulation. As regard **state analysis**, in streaming applications, pipeline tasks or stages act as data transformation activities and can store or not the state between different executions. With elasticity in mind, it is important to specify which components of the application are stateful and which are stateless. In our wavefront case study, all components are stateless.

Finally, the two last properties are **Component refinement** and **Management components** to support *elasticity* and *resilience*. Solutions for the latter are presented by patterns such as load balancers, or elastic queues. Component refinement models can be based on middleware such as message-oriented middleware to support loose coupling, schedulers that run tasks on opportunistic resources or prioritise QoS enforcements, or multi-tenancy solutions to share resources and components. A catalog of patterns for implementation mechanisms can be found in [3, 54]. Additionally, elasticity and resilience are also intrinsically linked with resource management, scheduling and autonomic principles with direct effect on performance and cost [58]. Although this paper focuses on the first four properties, the utility of formal models for the modelling of components supporting the two last properties is shown in previous works: In [59, 60, 32, 61, 62] are presented specifications of strategies on cloud for resource management following autonomic principles at the application level for streaming and scientific workflows. It is also important to highlight the work of Brogi et al. [63] that propose to extend TOSCA with the use of Petri nets for modelling management operations of complex applications over heterogeneous clouds.

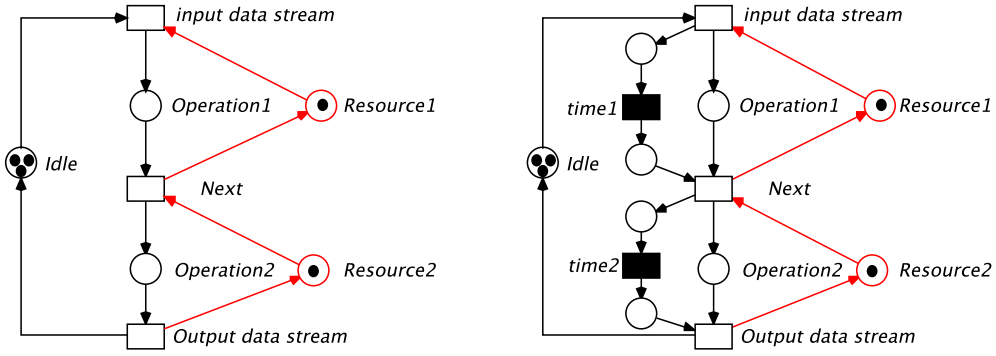

Figure 3: A Computational Process with Resources composed by 2 sequential states, each one requiring a different type of resource. a) Untimed model; b) Timed model assigning $time_i$ units of time to the execution of the $operation_i$.

5.2. Modular construction of the functional Petri Net model of a CDFA

The construction of the functional model is based on the identification of the basic modules that compose an application of this class. These modules are the Computational Processes (CPs) and the Data Transmission Processes (DTPs). CPs accomplish functional operations and transformations on data, and DTPs allow data dependencies to be conducted among CPs. Both CPs and DTPs need resources to accomplish the corresponding operation, and these resources also appear in the model, but at a conceptual, and generic way. Later, in subsequent model refinements, specific resource constraints of different characteristics be added, such as limitations in parallelism, capacity, or economic cost.

- ***Characterisation of a Computational Process***. A CP can be viewed as a type of elementary computational task of an application to be applied to a set of data elements coming from different input data streams. We assume that a CP can consist of multiple instances, called Computational Threads (CT), that are executed concurrently. By analogy with a programming language, we could see a CP as a concurrent program which consists of multiple processes (threads).

  A CP is modelled as a Petri Net, $\mathcal{N}$, and a CT as a token that moves through $\mathcal{N}$. The places (partial states) of $\mathcal{N}$ are related to the different operations (either transformations, handling or assembly/disassembly operations) to be carried out by the thread (See Figure3). There is a special place named *Idle* representing the inactive state of the threads and its initial marking is the maximum number of supported threads executing simultaneously this CP. Each transition of $\mathcal{N}$ represents a state (e.g. task) of a process. Therefore, when the flow of the process makes progress through the transitions, the final state can be reached –representing the end of the computation for the input data elements, the production of the output data elements and the restarting of the thread for the processing of the next data elements on the stream. A CP has distinguished input points (output transitions of the *Idle* place) of data elements from the input streams and output points (input transitions of the *Idle* place) of data elements of the output streams. The execution of a CP is achieved by the execution of a computation path, and several of them can exist in the same CP. A computation path is a sequence of transitions fireable in $\mathcal{N}$, whose occurrence represents the processing steps for a computed data record.

  In Figure 3.a, a CP with two sequential states is represented. A token in the place *Operation1* or *Operation2* represents a thread executing the code corresponding to the operation 1 or 2, respectively, required by the computational task modelled by means of this CP. A thread executes these operations sequentially following the firing sequence: (1) *Input Data Stream* representing the acquisition of the data records from the input stream to realise the computational task; (2) *Next* representing the end of the operation 1 and the beginning of the operation 2; (3) *Output Data Stream* representing the delivery of the data records obtained after the computation to the output stream. The model presented in Figure 3.a is untimed. The addition of timed information to a CP is introduced by the addition of a sequence place-transition-place in parallel with a process place representing an operation of the computational task that consumes time. The transition added is labeled with time information representing the duration of the computational operation. In Figure 3.b, the CP from Figure 3.a is represented

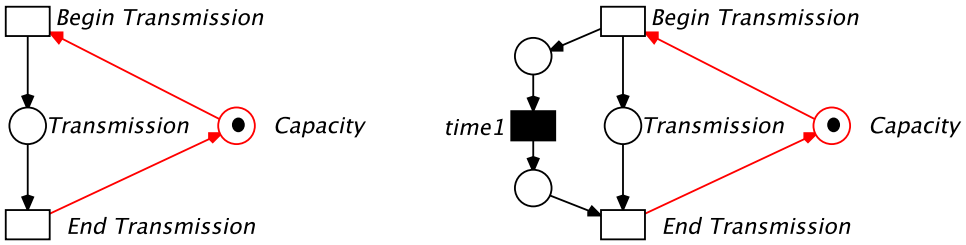

Figure 4: A Data Transmission Process with capacity for a single data record. a) Untimed model; b) Timed model

by assigning *time1* and *time2* units of time to the execution of the operations 1 and 2, respectively, according to the previously announced construction for the introduction of timing in the model. Observe that all transitions of Figure 3.a are immediate that is, do not consume time.

- ***Characterisation of a Data Transmission Process***. A CP can transmit data elements to other CP in the form of a stream sent by means of a physical/virtual device such as a FIFO queue implemented in memory or a communication channel in a communication network. That transmission behaviour is captured by a *Data Transmission Process* DTP. A data element to be transmitted is modelled as a token that moves through a Petri Net, $\mathcal{N}$, representing an elementary DTP with capacity for a single data record (see Figure 4). The places of $\mathcal{N}$ are related to the states in which a data element can be in the transmission device. The transitions of $\mathcal{N}$ allow a data record to progress from the source to the destination. The construction of a model that represents the transmission of k data elements (sequentially ordered) requires the concatenation of k of these elementary DTPs. The model in Figure 4.a is untimed and the firing sequence of transitions *Begin Transmission* and *End Transmission* represents the movement of a single data element of a stream from the source (the final transition of any kind of Process) to a destination (the initial transition of any kind of Process). In Figure 4.b, the DTP of Figure 4.a is represented by assigning $time_1$ units of time to the transmission of a data element (according to the previously announced construction for the introduction of timing in the model).

- ***Incorporation of Resources to each Computational and Data Transmission Process***. We consider any hardware/software element part of the execution environment (e.g. a processor, a buffer, a server, or a communication channel) as a resource with a given capacity. In the case of a buffer, its capacity can be the number of positions to allocate elements. Similarly, a processor may have a number of cores that can be considered as its capacity. Moreover, in the execution environment, there are several resource types and for each of them, a number of identical instances can be available, representing either the number of available copies of the resource to be used (or its capacity). In all cases, the considered resources are conservative, i.e. there is no resource leakage. On the other hand, each state of a CP, for its corresponding processing step, requires a (multi-)set of resources (including the buffering capacity to hold the thread itself). In our model, a resource type is represented by means of a place whose initial marking represents either the number of available copies of the resource or its capacity. A resource place has input (output) arcs to (from) those transitions of a CP that moves a Thread to (from) a state that required (used) a number of copies of this resource type. In the case of DTPs, resource places represent the capacity of the storage device for transmission. The CP of Figure 3.a requires two different types of resources that are modelled by means of places *Resource1* and *Resource2*. It should be observed that the CP requires a copy of *Resource1* to realise the operation 1 and a copy of *Resource2* to realise the operation 2. In the DTP of Figure 4.a, the resource place is the place named *Capacity*, which represents the size of the storage in the transmission device measured in terms of the number of data records. In the figure, it is equal to one (the initial marking of the place *Capacity*).

- ***Construction of the global model by composition of the Modules with resources***. In order to obtain the global model of the Streaming application, a number of CPs with their corresponding DTPs (accomplishing the data dependencies among them) must be composed. Besides, the resources needed must also be considered at this step. The composition is based on the fusion of the resource places representing the same resource type in the

different Modules. The initial marking of the resources, after the fusion, is often computed as the maximum of the initial markings of the instances that have been merged. The other composition operation is the fusion of a transition representing the production of data records of an output stream in a module with the transition representing the consumption of data records of an input stream belonging to a different module. Observe that it is possible to connect directly two CP without intermediate DTPs, one of the processes acts as producer of data records and the other as consumer of data records but without any intermediate buffer. Furthermore, it is also possible to connect directly two DTPs, this represents the construction of a DTP for a Stream of data elements with a capacity of storage equal to two. We can see examples of these fusions in the wavefront example through the rest of the paper, for instance in Figure 5.

Last but no least, the construction of the model will be done under the principle of economy of details. That is, only those aspects related with the structure of processes of the application and the use of shared resources by the processes will be explicitly represented in the model. These elementary patterns presented here (i.e. computing and transmission) can be combined to form any data stream pattern, ranging from simple sequential pipelines, to Directed Acyclic Graphs (DAGs), or the complex Wavefront Array.

5.3. Modular construction of the operational Petri Net model of a streaming application

The functional Petri Net model is derived from a specific algorithm that actually processes a number of given data streams. As already seen, it consists of a composition of computational tasks and the data dependencies among them. In consequence, a *minimal* number of constraints coming from the final execution environment can be taken into account and, in many cases, the functional model is constructed under a number of hypothesis that may not hold when targeting a specific infrastructure – i.e. the resources required in order to reach the maximum degree of parallelism inherent in the model will not be available, or in case there are resources available, but the economic cost of its usage exceeds the budget. Therefore, refining the functional model with the operational submodel aims at introducing specific resource constraints that may alter either economic cost, performance or even functionality. The alteration of the expected and observed behaviour at the functional model may even induce changes into the functional model in order to better target a particular execution environment. In other words, the reason for the operational submodel is to consider explicitly those actual characteristics of a final execution environment, or to compare the response of the application under different deployment scenarios. In this section, we refine the Functional Petri Net model according to the characteristics of a given execution environment. Nevertheless, there is a huge variety of different characteristics arising from different execution environments, leading to different requirements and constraints. As a result, there can be many different ways of refining the functional level. Hence, the following is an illustration about how the operational Petri Net model can be constructed from the Functional Petri Net model in two different situations.

The first one corresponds to the case in which the functional Petri Net model has several DTPs that were initially independent, but finally in the operational model they have to be merged together within the same low-level DTP. The actual refinement procedure is depicted in Figure 5. There, three independent DTPs, P_1, P_2 and P_3, are displayed that were already present in the Functional Petri Net Model. Nevertheless, the design decision to be taken is that the three Processes must share the same Low-Level Data Transmission Process of capacity 2. The refinement of the model requires the splitting of each place s_i of a DTP P_i in 2 places: (1) s_{i1} represents the request of transmission to the low level; (2) s_{i2} represents the end of the transmission. These two places are connected with a low-level DTP of capacity 2, as depicted in the figure. Observe that in order to recognise the process requesting the transmission, in the low-level DTP a Polling Algorithm to serve the requests has been implemented that is equal to the Polling Algorithm to send the acknowledgements to High-Level DTPs. The other aspect to take into account in the refinement activity is that in case of having timing information for the processes P_1, P_2 or P_3, this information must be removed before the actual refinement; since, after the refinement of the original information, it has no significance. The reason for this is that in the refined model the consumption of time is in the Low-Level DTP.

The second case arises if several CPs of the Functional Petri Net Model, which use resource types in isolation, must share the resources between all CPs. This provokes the rise of competitive relationships. A typical scenario for this transformation appears when the number of CPs is higher than the number of processors and the actual parallelism is limited.

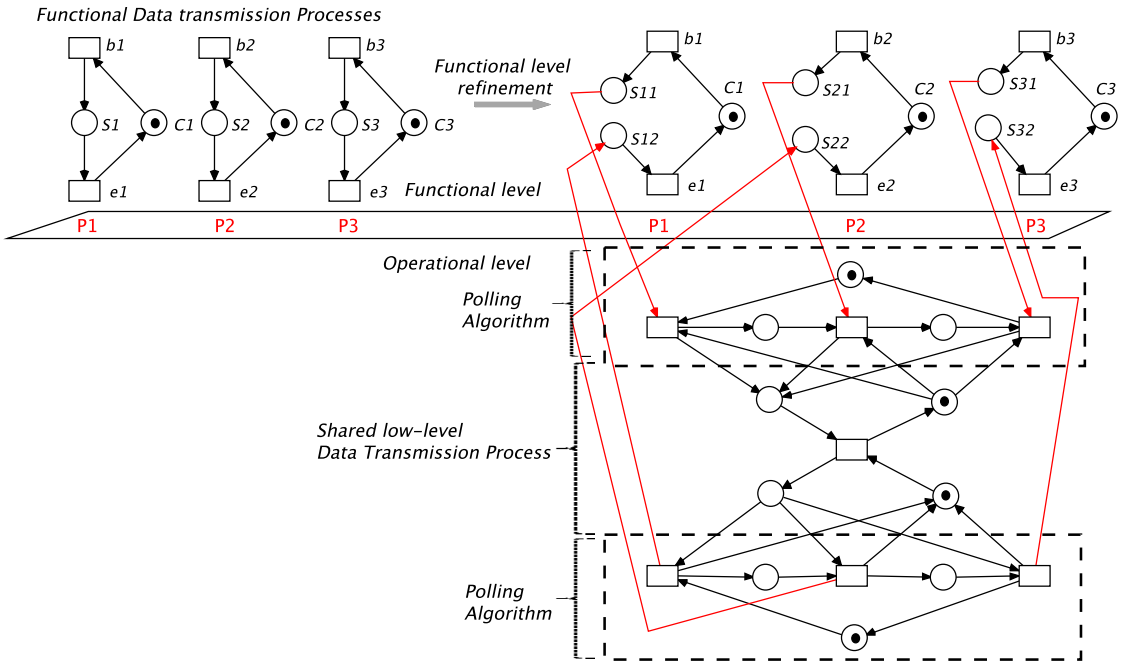

Figure 5: Refinement of the Functional Petri Net model to take into account the operational data transmission process that must be shared for three functional data transmission processes

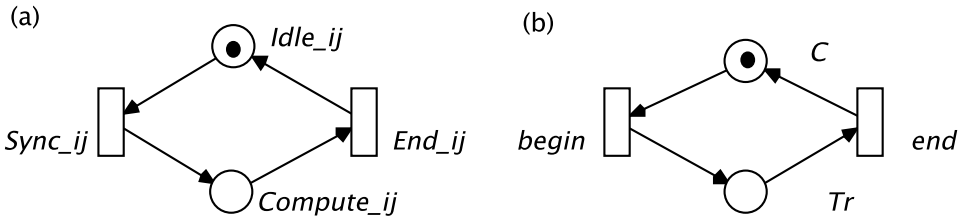

Figure 6: Basic modules for the construction of the functional Petri Net model of the wavefront algorithm: (a) Computational Process associated to a node; (b) Data Transmission Process for the external/internal data streams.

6. Functional and Operational models of the pipelined Wavefront

6.1. A functional Petri Net model for the Wavefront Algorithm

The functional Petri Net model of the wavefront algorithm sketched in Figure 2 is constructed in a modular fashion. The basic models we need in this case are: (1) A module to describe the Computational Process carried out in a node of the wavefront array; (2) A module to describe the Data Transmission Processes of the input and output data streams to/from the wavefront array. These modules are depicted in Figure 6. It should be noticed that they share the same structure with the patterns provided in Figures 3 a) and 4 a). In order to build the global model, 9 instances of the CP of Figure 6.a are needed. The modules of this type belonging to the same row are composed via the fusion of the transition End_i1 with the transition $Sync\_i2$; and the transition End_i2 with the transition $Sync\_i3$. These fusions of transitions represent the transmission of the result elaborated by the column 1 or 2, as input to the columns 2 or 3, respectively, without any intermediate buffering. Each one of the CPs of the first column is composed by a DTP representing the input stream of the corresponding i-th component of the vector $Y^{(k)}$ via the fusion of the transitions $Sync\_i1$ and *end*. Each one of the CPs of the last column is composed by a DTP representing the output stream of the corresponding i-th component of the vector $Z^{(k)}$ via the fusion of the transitions End_i3 and *begin*. Each one of

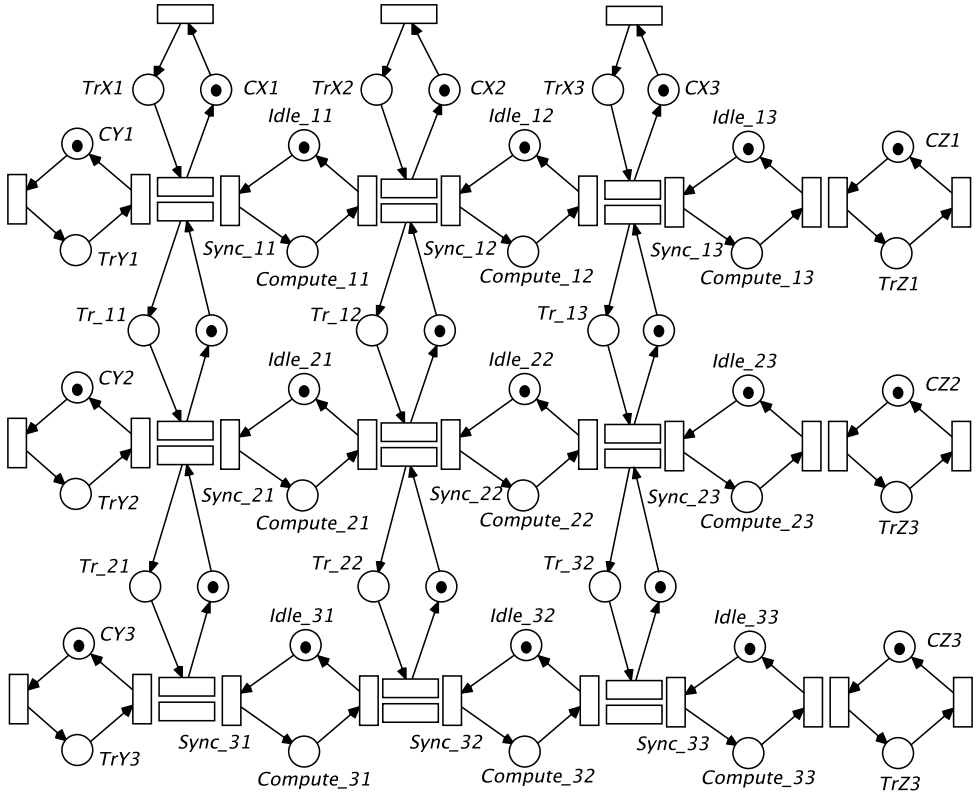

Figure 7: Untimed functional Petri Net model of the wavefront algorithm

the CPs of the first row is composed by a DTP representing the input stream of the corresponding i-th component of the vector $X^{(k)}$ via the fusion of the transitions $Sync\_1i$ and *end*. Finally, we connect two CPs belonging to the same column but located in rows 1 and 2, or in rows 2 and 3, by means of a Internal DTP describing the flow of the corresponding component of the vector $X^{(k)}$ through the rows of the array. This connection is done by the fusion of the transitions $Sync\_1j$ and one transition *begin* and the corresponding transition *end* with the transition $Sync\_2j$ (similarly for the case of rows 2 and 3).

Figure 7 depicts an untimed functional Petri Net model of the wavefront algorithm for for $Z^{(k)} = Y^{(k)} + A \cdot X^{(k)}$, $k = 1, 2, ...$ This net model is isomorphous to the flow model in Figure 2.

The addition of time to the model of Figure 7 will be done in the way described in the previous section: adding a sequence place-transition-place in parallel with the place representing the activity that consumes time. The new added transition will be labeled with the time information. In the example, a timed sequence will be added in parallel with each place $Compute\_ij$ representing the duration of the computation accomplished by the CP located at row i-th, column j-th. Moreover, a timed sequence will be added in parallel to each place Tr representing the consumption of time in the transmission of a data element in the corresponding DTP.

6.2. Operational Petri net Models for the Wavefront Algorithm

The previous functional level can be refined with a number of different operational models with distinct layouts and resources and, hence, achieving different degrees of parallelism. In other words, a functional model can also be connected to an operational model in a number of ways. We propose here the strategy depicted in Figure 8, which extends the basic modules for the construction of the wavefront algorithm from Figure 6. It can be seen how both modules (CP and DTP) can be connected to their corresponding computational resource / transmission process in the operational model, by adding places and arcs as specified in the figure. In particular, in Figure 8 a), if the

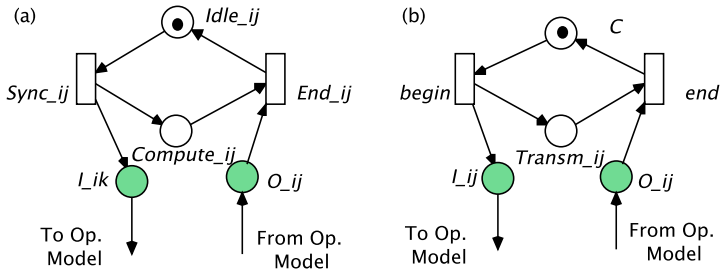

Figure 8: A Strategy for Connecting a Functional Model with an Operational Model

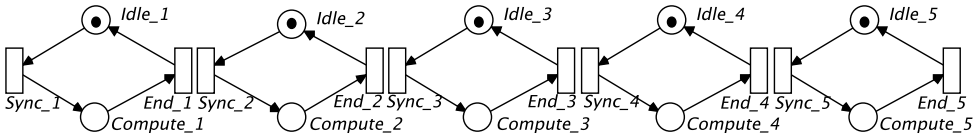

Figure 9: A Pipeline Operational Model

execution starts, Transition Sync_ij is fired and the flow then goes to the operational model. After the operational model conducts the execution, the flow comes back from the operational model to the functional model. The same mechanism is designed for the DTP module in Figure 8 b).

6.2.1. A Grid Operational Model

Perhaps, one of the operational models that can be proposed in the first place is the grid operational model –based on the computational processes that appear in the functional model. Therefore, one would need a computational resource for each CP. Therefore, the operational model needs to be the same in essence as the functional model from Figure 7: a 3x3 matrix of CPs connected with the right and bottom neighbour with a DTP.

The wavefront functional model can be connected to the grid operational model by adding to the functional model the places, transitions and arcs as in Figure 8 and then, in particular, by following this specific mapping: For each CP and DTP in the functional model, each Place I_ij will be connected to the underlying transition Sync_ij of the operational model, whereas the underlying transition End_ij of the underlying model will be connected to Place O_ij. We do not provide a graphical representation model of both the functional and operational models linked together because we understand it will be difficult to display. For that reason, we developed an abstraction and specification language [16] that can be found in the Appendix.

6.2.2. A Pipeline Operational Model

An alternative operational model could be derived from the fact that the wavefront array works forming sequences (*waves*) of operations. This characteristic was represented in Figure 2, thereby the first diagonal (left to right direction) starts an operation which is subsequently continued by the rest of diagonals. In this case, the functional model can be mapped into a sequence (pipeline) of resources, as depicted in Figure 9. Therefore, each diagonal from the functional model is mapped onto a computational resource in the operational model. As there are 5 diagonals in the functional model matrix, a 5 sequential pipeline of computational resources was arranged. It should be noted that each diagonal consists of a cluster of CPs –with a variable number of CPs depending on the diagonal. We do not provide a graphical representation model of both the functional and operational models linked together because we understand it will be difficult to display. For that reason, we developed an abstraction and specification language [16] that can be found in the Appendix.

Therefore, the connection between the functional and the operational models can be done analogously:

- Each Place Compute_ij of the functional model will be split in two places. One of them will be connected to the underlying transition Sync_k, with $k = i + j - 1$, of the operational model, whereas the underlying transition

End_k of the underlying model will be connected to the remaining place of the functional model. It should be noted that the nodes belonging to the same diagonal will be mapped to the same computational resource.

- Each Place I_ij of the functional model will be connected to the underlying transition Sync_k, with $k = i + j$, of the operational model, whereas the underlying transition End_k of the underlying model will be connected to Place O_ij.

6.3. Linking the Models to Real Cloud Resources

The enormous popularity of the cloud has its origin in the combination of both utility and service computing paradigms. As a result, a great number of technologies have been developed in the last years, leading to an overwhelming variety of options of choice, including resources, framework systems, and packaging and tools. In this subsection, we will briefly highlight the type of cloud of resources available and their particular characteristics that can make any performance prediction challenging.

6.3.1. Cloud Resource Types

In general terms, developers can choose among the following cloud resource types: (i) The conventional physical machines (also known as *bare metal*), (ii) virtual machines that imitate physical hardware (it contains the operating system and the applications), (iii) containers either comprising the same layers (operating system and application) or only the application, (iv) application packages running on a middleware system (e.g. a web server or application server), and more recently (v) lambda functions.

Virtual machines (VMs) were one of the first type of cloud resources. A VM can be seen as a piece of software that emulates hardware, and typically multiple VMs are executed over the same physical host machine, sharing the same hardware. A key component for them is the hypervisor, responsible for conducting the emulation, and they also divide the hardware across the physical machines. One of the most important benefits of using VMs is the full isolation they achieve. Nevertheless, VM utilization can sometimes be difficult to achieve, as oftentimes applications to be run do not consume all the resources of a VM. Developers could alternatively try to map multiple applications onto the same VM, but, in such a case, applications would not be isolated.

Containers, on the other hand, represent a way to solve that isolation problem for improving utilisation. A container can be seen as a set of processes where an application is executed in isolation. Multiple containers typically coexist on the same host machine, and each container in it uses the resources that the application on it consumes. Nevertheless, the degree of isolation achieved by VMs is still higher than the one achieved by containers, but containers have much less overhead. The reason for it is that all containers deployed in the same host (physical or virtual) machine share the same OS kernel, and therefore virtualisation is not required. Furthermore, while a VM needs to boot before any application can be executed on it, a container is a group of processes whose execution can be initiated almost immediately, i.e. the overhead is much lower. For this reason, containers are rapidly replacing Virtual Machines (VMs) as the compute instance of choice in cloud-based deployments. One of the reasons is the significantly lower overhead of deploying and terminating containers in comparison to VMs. Understanding performance associated with deploying, terminating and maintaining a container is therefore significant.

More recently, *lambda functions* or *serverless-infrastructures* are recently emerging. The main idea behind a function is that they allow developers to execute their code without the need for both provisioning and managing servers. Therefore, developers only load their code into the function infrastructure that is subsequently wrapped as a service, and there is no need to provision for the machines (e.g. switch on a number of VMs or containers). Instead, when triggered by an event, the code is executed and the resource management is completely automated. As being invoked as a service, functions are closely related to the service oriented paradigm, and to micro-services. Furthermore, functions are often implemented by container technologies.

6.3.2. Cloud Resource Sharing

As discussed previously, virtualisation technologies and containers, on one hand, allow applications to request the required cloud computational power on demand, as needed, and potentially paying for the use done. On the other hand, from a provider's perspective, they also aim at maximising resource utilisation, and this is achieved by executing multiple tasks simultaneously over the same computational resource. This is appropriate as tasks are not making use of the required resources at a maximum level over time, but in practice the usage of a resource can vary. Nevertheless,

this sharing of resources in the cloud is subject to resource contention. As a result, performance interference have been reported, for instance involving virtual machines [12, 13], and also involving container technologies [14]. Such effects can have a great impact in the execution times of tasks, in some cases, making the execution time to vary an order of magnitude. Therefore, the methodology discussed in this paper can be of great interest, as it can analyse the convenience of a deployment configuration depending on the execution circumstances. On the other hand, communication networks can also be virtualised, facilitating the centralisation of control, therefore reducing economic cost and complexity of operating and maintenance. In the following subsection, we explain how our methodology could be used to integrate real cloud resources.

6.3.3. Integrating Cloud Resources in our Methodology

Any of the cloud types described above can be integrated into our model. We propose two ways of capturing the actual cloud resources used and integrate them into our proposed methodology:

- *Real-time monitoring* of performance execution time and *feeding* of our *operational* models with it. By monitoring these cloud resources (both computational and network resources), real-time performance data can be obtained, and subsequently the operational models can be fed. The addition of time to the obtained model will be done in the way described in the previous sections: Adding a sequence place-transition-place in parallel with the place representing the activity that consumes time, which will happen in the operational model (rather than in the functional one). Such a new added transition can be labeled with the time information: A timed sequence will be added in parallel with each place $Compute\_ij$ representing the duration of the computation realised by a CP. Moreover, a timed sequence will be added in parallel to each place Tr representing the consumption of time in the transmission of a data element in the corresponding DTP. As modelling performance interference can be very complex in practice, this option could capture performance unexpected variation in real-time and the models can provide insight on the better alternatives for the mapping of tasks.

- *Modelling of the actual infrastructure and* integrate it into the models, by refining the operational models. By means of this option, a model of the cloud infrastructure is developed which serves as a refinement of the operational model. For instance, in [14], the authors provide a Petri-net based performance model of Kubernetes [1]. Kubernetes is an open source platform that abstracts and automates the deployment of applications across a number of distributed, computational resources. It makes use of container technologies in order to manage and provide computational resources. The model described in [14] can be annotated and configured with deterministic time, probability distributions, or functions obtained from monitoring data acquired from a Kubernetes deployment. It can be used by an application developer / designer: (i) to evaluate how pods and containers could impact their application performance; (ii) to support capacity planning for application scale-up / scale-down. Such models of the infrastructure can be integrated into the functional and operational models analogously by following the same approach in which the operational model was integrated to the functional model: It was described in Section 5. This approach can capture the whole behaviour of the computational resources, and therefore it can obtain more opportunities for performance analysis. However, it can require a substantial effort, due to the complexity of the cloud distributed infrastructures.

6.4. Exploiting our Approach in a Cloud Infrastructure

When using a cloud infrastructure, resources are pooled together in order to process the requests of multiple clients simultaneously. What cloud computing brings as a novelty from traditional data centers is the flexibility in choosing hardware and operating system on demand, as well as a rapid elasticity. At least in theory, cloud resources promise a rapid automated provisioning to quickly scale out and in. The client, on the other hand, if using a public cloud just pays for the consumption, as a utility. Nevertheless, as already discussed in above in the paper, the sharing and the virtualization technologies are not yet capable to isolate resources one another and performance interference can arise. This poses some challenges in terms of meeting SLAs, as the amount of the computational resources that can help meet the SLAs is dependent on the performance interference phenomenon. We believe that our technology can benefit form

[1] http://kubernetes.io/

the cloud elasticity and our obtained models can be exploited in order to mitigate the cloud performance interference problems. In this section, we are providing a way to exploit our methodology based on autonomic principles.

Automated resource provisioning in the cloud often makes use of autonomic principles and exploits the elasticity of the cloud, thereby computational power can be increased / decreased on demand, offering cost-effective solutions. Some of these approaches [25] are often based on the MAPE-K (Monitoring, Analysis, Planning and Execution, and Knowledge) loop. These autonomic systems typically have an objective, related to enforcing QoS as specified in their SLA and which is used as the basis for triggering actions. The loop goes through four phases continuously: (i) During the *monitoring* phase, (near) real-time data is gathered from computational resources of the distributed infrastructure – this would correspond to our monitoring phase in our methodology, also depicted at Figure 1; (ii) such monitored data is used during this autonomic *analysis* phase in order to determine whether current computational resources are enforcing SLA; in case the objective is not met, (iii) then the planning phase is executed, thereby a (sequence) of action(s) can be triggered, such as adding or reducing the computational power (e.g. this can be accomplished by vertical or horizontal elasticity); finally, (iv) the actions are executed to move current state of the system to a new state where the objective is met.

The essence for accomplishing this MAPE-K effectively lies in the knowledge (K). Indeed, during the planning phase, the system needs performance models in order to decide which resource mapping option and which resource configuration is more convenient. To this regard, our methodology aims at building performance models and to exploit them with different performance analysis techniques. As we have seen, we can obtain performance minimal / maximal boundaries, and then during the simulation (see Section 7.5), we can improve the prediction and narrow the gap distance between them, by exploring the combination of parameters, e.g. data income rate and processing throughput. Then, the monitored data, e.g. the actual data income rate and processing throughput, can be used to feed our models and estimate what the CDFA performance will be. The autonomic controller thereafter could decide which is the most convenient mapping solution among the existing ones, and trigger an action that may involve a dynamic change of the current mapping solution. Thus, by means of this mechanism, we can find a relationship between our methodology, the elasticity of the cloud infrastructures, and the phenomenon of performance interference. In our previous work in [32], we explored how to exploit these formal-based performance models for the enactment of scientific workflow systems.

7. Performance and Economic Analysis

In the following subsections, we explore the possibilities for both qualitative and quantitative analysis of the functional net model presented in Figure 7. Then, we perform an analysis with an enriched operational model (i.e. by adding time inscriptions) that is isomorphous (grid operational model) to the functional model.

7.1. Background: Types of Analysis

The proposed PN model-driven methodology aims at providing different analysis and prediction techniques that allow developers to assess functional and non-functional properties by means of Qualitative and Quantitative analysis. **Qualitative analysis** aims to detect qualitative properties of concurrent and distributed systems, that is, to decide whether the model is correct and meets the given qualitative functional properties (e.g deadlock freedom). Qualitative PN analysis can be conducted by means of different techniques: (i) The construction of the state space of the model (reachability analysis) providing a complete knowledge of all its properties – in case state explosion does not hamper the use of this technique; (ii) Structural techniques in order to reason about some properties of the model, from the structure of the net.

Among the techniques based on the construction of the state space, standard *Model Checking* techniques can be used to explore the state space, which corresponds with the complete set of reachable makings of the PN by the occurrence of transitions. Then, any property to be verified can be expressed in logic terms. If the property is satisfied, then the answer of the Model Checker is just a confirmation of this, but if the property is false, then the model checker gives counterexamples that prove that the property does not hold. The main advantage is that usual properties like deadlock-freeness, home space, maximal sets of concurrently fireable transitions, or mutual exclusions can be decided. In practice, the applicability of the approach is limited to *bounded* systems with a finite state space and with a moderate size. Conclusions hold *only for the initial marking* being considered.

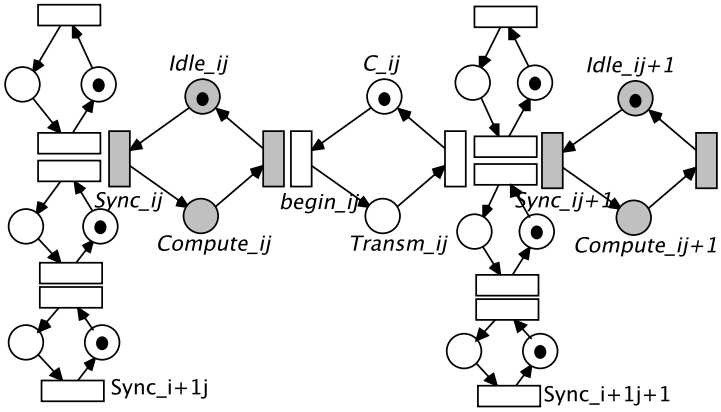

Figure 10: Functional Model of Two adjacent Computational Cells (in dark colour) of the Wavefront Algorithm without Concurrency Limitations

The second group exploits the models through the extraction of *structural information* from the net. From the structural information, some properties can be obtained, i.e. the places, the transitions and the token flow relation represented by means of the arcs. They can make use of graph theory, linear algebra, convex geometry, or linear programming. Although the use of this technique is not restricted by the state explosion effect, it has a limited decision power (semi-decision algorithms) except for syntactical subclasses of PNs.

From the above comments, we can say that a first phase for exploiting the models is to determine *the correctness of the adopted solution*, verifying all the good properties expected are satisfied. As a second step, if a property is not satisfied, the model can be used for *detecting the causes of the problems* or anomalies in it that prevent it from behaving as expected. Then, the understanding of the causes can lead to a *modification of the model* to correct bad behaviours and to maximise concurrency.

Quantitative analysis seeks performance-oriented interpretations of the model such as throughput, utilisation rates, or queue lengths, from which it is possible compute *reward functions*. PNs are executable models, therefore extensive simulations can detect errors, which are rare and elusive, and provide us with some performance and reward functions. However, simulations, as an analytical model, cannot guarantee the absence of errors neither identify under which conditions the simulated performance values will be reproduced. The most common analytical model used for the derivation of exact performance measures are stochastic PNs (SPNs)[64]. The techniques consist on the derivation of performance measures from the reachability graph of the model from which a Markov Chain is obtained, under certain assumptions on the stochastic specification. Once again, state explosion can hamper the use of the technique. Additionally, the model assumes exponential distributions for transition delays, which may not be acceptable to model CPU performance, memory speed, sequential and random I/O, or network bandwidth.

7.2. Qualitative analysis based on Structural Analysis.

First, we determine the correctness of the obtained design at Figure 7. This can be realised by exploiting the structural properties of the net to conclude about behavioural properties. It is a *strongly connected marked graph* (a subclass of Petri nets in which each place has only one input and one output transition, being strongly connected in the sense of graph theory). Moreover, all its circuits contain at least one token. From these structural properties and conditions fulfilled by the initial marking, we obtain the following functional/behavioural properties of the model:

- Any transition of the net is fireable from any reachable state of the net (the net is *live*, thus deadlock-free). Moreover, the minimal repetitive sequence of transition firings contains all transitions of the net exactly once (it is guaranteed in live marked graphs from the existence of only one fireable T-invariant equal to vector **1**: right anuller of the net incidence matrix). The liveness property is a necessary property of the model to guarantee the execution of a well-behaved wavefront in the array.

- The wavefronts propagate in an orderly manner, that is, a data element d_i being processed cannot be overtaken by the following data element in the stream d_{i+1}. Besides, data element d_i, in turn, cannot overtake the processing

data element d_{i-1} that started its execution in the previous time slot. This can be concluded after the computation of the maximal difference between firings (in any firing sequence) of a transition $Sync\_ij$ with respect to its:

- right neighbor transition $Sync\_i(j+1)$, that is equal to 1. To see this, observe that both transitions are covered by a circuit containing only one token. This circuit enforces a strict alternation in the firing of both transitions starting with the firing of $Sync\_ij$;
- left neighbor transition $Sync\_i(j-1)$, that is equal to 0. The reason is the same as in the previous case: the existence of a circuit with a token that enforces the alternation of both transitions starting with the transition $Sync\_i(j-1)$;
- bottom neighbor transition $Sync\_(i+1)j$ is equal to 1. Once again, this can be proven by means of the vertical circuit covering both transitions and containing only one token; and
- top neighbor transition $Sync\_(i-1)j$ is equal to 0. All these values can be obtained from the so called marking invariants of the net (that in the case of marked graphs are the elementary circuits of the net) and can be computed in a structural way.

This computation can be realised in a Structural Analysis by determining the structural bounds of Synchronic Lead and Synchronic Distance properties between the referred transitions, by means of the corresponding Linear Programming Problems appearing in [65].

- In Figure 7, CPs in a *top-right to bottom-left* diagonal can operate concurrently, but CPs belonging to two *top right to bottom left* and *adjacent* diagonals cannot fully operate concurrently. There is a structural limitation in the model, which can be easily proven. Indeed the four transitions $Sync\_ij$, $Sync\_i(j+1)$, $Sync\_(i+1)j$ and $Sync\_(i+1)(j+1)$ are covered by an elementary circuit containing only two tokens. This means that only two transitions out of the four transitions can be concurrently fired. But taking into account the firing relations enumerated in the previous point, only two scenarios are possible: (i) concurrent firing of the transitions $Sync\_ij$ and $Sync\_(i+1)(j+1)$; (ii) concurrent firing of the transitions $Sync\_i(j+1)$ and $Sync\_(i+1)j$. This points out the initial statement about the mutual exclusion in the execution of *top-right to bottom-left* neighbour diagonals.
- From the previous property, if we extend it to the full array, then we can conclude that we cannot have the nine CPs running concurrently. In contrast, due to the structural limitation discussed before, the maximal concurrency that can be achieved for the model from Figure 7 is as follows: When the CP associated to Place $Compute_{31}$ (diagonal 1) is in execution, then CPs associated to Places $Compute_{11}$, $Compute_{22}$, and $Compute_{33}$ (diagonal 3) and the CP associated to Place $Compute_{13}$ (diagonal 5) can be in execution concurrently, and the remaining CPs from diagonals 2 and 4 cannot be in execution. On the other hand, when the CPs associated to Places $Compute_{21}$ and $Compute_{32}$ (diagonal 2) are in execution, then the CPs associated to $Compute_{12}$ and $Compute_{23}$ (diagonal 4) can be in execution concurrently, but not the rest of CPs.

The previous analysis, without the need for an exhaustive simulation or construction of the state space, points out that it is not possible to have the nine CPs working concurrently. As we discussed, this anomaly or bottleneck limiting concurrency is due to a structural limitation: The existence of the circuits covering four transitions, but containing only two tokens. In order to enable a fully concurrent operation of all the nine CPs, we can decouple any two consecutive CPs in a row by adding a DTP between them. Such a structural modification can be seen in Figure 10, it shows two adjacent computational cells of row i of a Wavefront array: $Cell_{ij}$ and $Cell_{ij+1}$. Both cells are highlighted in dark colour and a DTP module between them. The achieved effect is a decoupling between both cells, so that $Cell_{ij}$ can now start its execution independently from $Cell_{ij+1}$ (Transition $Sync_{ij}$ can now be fired without requiring a direct synchronisation with Transition $Sync_{ij+1}$). From a structural analysis point of view, we modified the initial design where we had circuits containing only two tokens, to obtain a structure where we enforce circuits with four tokens. In other words, the four transitions $Sync\_ij$, $Sync\_i(j+1)$, $Sync\_(i+1)j$ and $Sync\_(i+1)(j+1)$ from Figure 10 are now covered by an elementary circuit containing four tokens.

7.3. Quantitative analysis based on Stochastic PNs

For the quantitative analysis, we refined the wavefront functional model with the grid-based operational model. In order to perform the analysis, we enriched the model with time. As discussed above, this can be done by adding

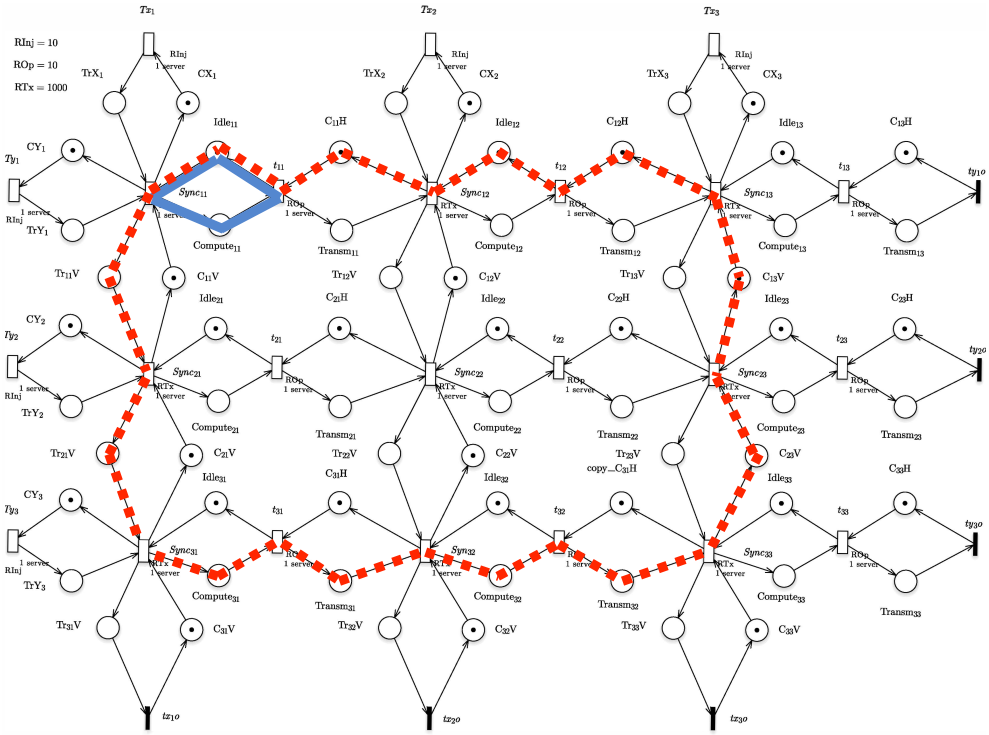

Figure 11: Timed operational GSPN model of the wavefront without concurrency limitations. Blue straight line shows a P-semiflow that makes maximum the summation of all delays involved in the circuit divided by the tokens in the circuit. Red broken line shows one of the longest circuits. `RInj=10`, `ROp=10` and RTx=1000 represents respectively injection, operation and transmission rates.

a sequence place-transition-place in parallel with the places representing the activities that consume time. Such places appear at the operational level, which is built upon the modules from Figure 8: The duration of computations (Transitions $Sync\_ij$ from Figure 8a), and the consumption of time in the transmission of a data element in the corresponding DTPs (Transitions *begin* from Figure 8b). When the time inscriptions added are a probability density function (PDF), our model is by definition a stochastic PN. Only the use of a negative exponential PDF for the specification of temporal characteristics makes the analysis mathematically tractable. Let us assume that processors service delivery time ($1/\lambda$) and injection timed transitions ($1/\gamma$) follow an exponentially distributed random amount of time, with average 100ms (rate =10 data/sec); and a transmission time 100 times faster with average ($1/\beta$=1ms) also following an exponential distribution.

Then, we can introduce this resulting operational PN model directly to GreatSPN2.0.2 (See Figure 11)[2] [66]. GreatSPN2.0 (GSPN) is a software package for the modelling, validation, and performance evaluation of distributed systems using Generalised Stochastic Petri Nets and their coloured extension. Recall that, from the previous structural analysis, it can be derived that all transitions will have the same throughput: The minimal repetitive sequence of transition firings contains all transitions of the net exactly once (it is guaranteed from the existence of only one T-invariant: right annuler of the incidence matrix of the net). Therefore, the relative firing frequency vector is $\mathbf{1}$, and we have the same mean cycle time for all transitions. The relative firing frequency vector is a vector that for each component, it contains the mean interfiring time of each transition t_i, i.e. the inverse of its throughput. The tool generates the reachability graph with 1,392,640 states from which a Markov chain is derived. Performance indices like place markings, probability distribution and transition throughputs can be computed. The calculated throughput for all transitions is 3.99 data/sec. The result obtained by the GSPN analysis shows a poor throughput with a performance loss of 60% for each processor, in comparison with the model that achieves maximum degree of concurrency and all its CPs are executed concurrently.

Furthermore, if we repeat the analysis with a wavefront of dimension 2x2, the throughput for all transitions is 4.69 data/sec with a performance loss of 53% for each processor in comparison with the model that achieves maximum degree of concurrency and all its CPs are executed concurrently. This is due to the concurrency limitations found in the structural analysis. All transitions will have the same throughput, and as a result throughput is determined by the slower transition. Intuitively, the use of a negative exponential PDF, with a high coefficient of variation, makes more likely a slower transition than the expected throughput with a larger number of processors.

7.4. Quantitative analysis based on Structural Analysis: Computing performance bounds

The use of stochastic PNs for the derivation of exact performance measures and rewards functions is hampered by two factors: (i) The explosion of the computational complexity of the analysis algorithm and (ii) only the use of an exponential probability distribution function for the specification of temporal characteristics makes the analysis mathematically tractable.

In [56], authors present upper and lower bounds on the steady-state performance of marked graphs that can be computed efficiently. To do that, they derive bounds for the throughput of transitions, defined as the average number of firings per time unit (or its inverse, called the *mean cycle time of transitions*, Γ). From this value, applying Little's Law, it is possible to obtain other average performance estimates of the model. Under these restrictions they showed results that can be computed in polynomial time on the size of the net model, and that depend only on the mean values and not on the higher moments of the probability distribution functions of the random variables describing the timing of the system. Finally, they found that both upper and lower bounds, computed by means of Linear Programming Problems, are tight, in the sense that, for any marked graph, it is possible to define families of stochastic timings such that the steady-state performances of the timed Petri net models are arbitrarily close to either bound.

For the case of throughput upper bounds for strongly connected marked graphs, the computation method is obtained by applying Little's Law to each place of the net. Besides, special marking invariants, derived from the P-semiflows, of the net are used. A P-semiflow is a special sequence of firing of transitions of a Petri net, so that when such firing sequence occurs in it, its marking (i.e. the token distribution) is invariant. Further description about Petri nets and these topics can be found in [33]. Furthermore, the dynamic behaviour of Petri nets can be expressed in terms of matrix equations and the following Linear Programming Problem can be derived. Its optimal solution (which

[2] `http://www.di.unito.it/~greatspn`

can be computed in polynomial time) is a *lower bound* for the *mean cycle time of transitions* (inverse of the average throughput) that is denoted as Γ^{min}.

$$\begin{array}{ll} \Gamma^{min} = \text{maximum} & Y^T * Pre * \theta \\ \text{subject to} & Y^T * C = 0, Y^T * M_0 = 1, \\ & Y \geq 0 \end{array}$$

where Γ^{min} is the minimum mean cycle time, Y^T is the left annuler of the incidence matrix of the net (P-semiflow), *Pre* is the pre incidence matrix (i.e. denoting tokens removed by transition firing), and M_0 is the initial marking. This means that the mean cycle times can be computed by the summation of all time delays involved in a circuit (P-semiflow) divided by the tokens in the circuit. And we obtain the Γ^{min} by finding the maximum value of mean cycle times computed by each circuit. In our model, Γ^{min} is 101 ms $(1/\lambda)$, i.e., a maximum throughout of 9,9 data/sec. Figure 11 shows one of the possible circuits highlighted by a blue straight line. It returns the maximum value of mean cycle time computed by the summation of all time delays. `RInj=10 data/sec`, `ROp=10 data/sec` and RTx=1000 data/sec represents respectively injection, operation and transmission rates. Therefore, time delays in the blue circuit is the addition of operation in `t11` is 1000/10=100 ms, and transmission 1000/1000 = 1 ms giving a total of 101 ms divided by one token in this circuit.

A tight *upper bound for the mean cycle time* (its inverse is a lower bound for the steady-state throughput) is obtained in polynomial time, from the knowledge of the given average service times and the liveness bounds of transitions, which are computed by solving proper linear programming problems. This bound cannot be improved unless more information from the service times of transitions than their mean values is used.

$$\Gamma^{max} = \sum_{j=1}^{m} \frac{\theta_j}{LB(t_j)}$$

where θ_j denotes the average service time of transition t_i, and $LB(t_j)$ the liveness bound of t_j, i.e. its maximum degree of concurrency (maximum number of concurrent firings of a transition). In our example, the maximum liveness bound is 1 for all transitions. Therefore, Γ^{max} is given by the addition of $theta_j$ corresponding to the longer circuit. The dotted red line in Figure 11 shows one of the possible longer circuits that defines the lower throughput bound. This circuit comprises four operations (100 ms) and eight transmissions (1ms), and the the maximum degree of concurrency of each transition is one because there is only one token in previous places. More specifically, in our model: Γ^{max} = 408 ms and a minimum throughput of 2.45 data/sec.

Under the light of this analysis, we can obtain best- and wort-case scenarios in terms of throughput, which is inline with the results obtained from previous analysis (i.e. the GSPN analysis also reveals that the 2x2 wavefront has better throughput for each output than a 3x3 wavefront). But the most important result of this analysis is that we can estimate upper and lower bounds that specify the space of possible performance solutions (see bounds in Figure 12).

7.5. Quantitative analysis based on Simulation

If the obtained stochastic PN model is too large, the generation of all of its states to obtain the underlying Markov chain or even solving the previous optimization problem may not be feasible. In consequence, an alternative is to make an analysis based on simulations. Simulations can also be used to explore different what-if scenarios between the obtained performance bounds. One of its main drawbacks is that it only reports on simulated situations, since it is not an exhaustive enumeration technique. Besides, it can also be a time-consuming approach, as each of the different settings need to be simulated and it can be difficult to determine whether the system reaches stability.

Furthermore, one of the challenges for conducting simulations on the model is to choose the appropriate probability distribution functions of response time of processes (i.e. computations and transmissions in our case), in order to provide with a realistic exploration of all the variability. In our previous quantitative analysis based on stochastic PNs, the system throughput is near the minimum bound. Then, the selection of an exponential distribution for task service times is not adequate and results in poor and not realistic performance results. However, if the service times are not exponential, it can be complex and it relies on a set of approximations. Such approximations are sensitive to the probability distribution of service times and they even become increasingly inaccurate when the Coefficient of Variation (CoV) increases towards the value of 1 [11, 10, 12]. The CoV is a measure of dispersion of a probability distribution. It can be used to show the extent of variability in relation to the mean of the population. Therefore,

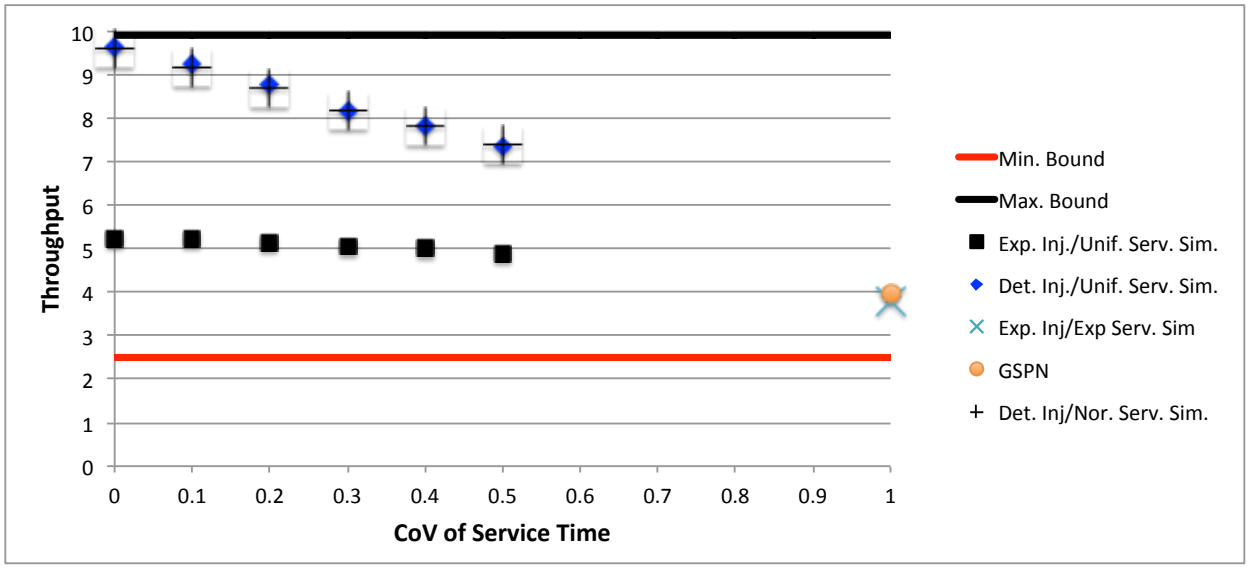

Figure 12: Simulated throughputs with different Coefficient of variation of the service time.

CoV is formally defined as $CoV = \frac{\sqrt{Var[X]}}{E[x]}$. The higher the values of CoV, the higher the dispersion in service times. Furthermore, the difficulty increases when we try to analyse performance of applications implementing advanced data flow abstractions with a high level of concurrency constraints.

Once the throughput bounds for our model are known from the previous analysis, and given the fact that the CoV on service time can have a high impact on performance, we conducted simulations to evaluate its impact. Such a characteristic can be of great interest, for instance, when computational resources are subject to unexpected performance variations (e.g. in some clouds). In order to conduct the simulations, we made use of the Renew tool. Renew [3] [67] is a Java-based Reference net interpreter and a Reference net graphical modelling tool.

Figure 12 shows the results of different simulations with mean services and injection times of 10 data/sec. In order to perform the simulations, we introduced our operational PN model into the `Renew`[4] PN interpreter, In the example, we considered the transmission time to be negligible compared to the average service time. The figure shows the point *GSPN* that represents the computed performance by the GSPN tool obtaining the Markov chain, and the point *Exp. Inj./Exp. Serv. Simulation* shows the same scenario obtained by simulation in Renew, i.e, assuming both service delivery times and inter-arrival times are exponential. The CoV of an exponential distribution is 1. The proximity of these points shows the accuracy of simulations. Assuming the computing nodes in the cloud are heterogenous and that the performance capabilities of these computing nodes are uniformly distributed [11] between the time of the faster node and the time of the slower node, we conducted different simulations in `Renew`. Service time CoV ranges in simulations from 0 to 0,5 with uniform and normal distributions. It is not possible to obtain higher values of CoV with these distributions. *Exp.Inj./Unif. Serv. Simulation* shows the impact of CoV on performance assuming an exponential distribution in injections, and uniform distributions of service delivery times. *Det.Inj./Unif. Serv. Simulation* shows the same simulations with a deterministic injection time. Finally, *Det.Inj./Nor. Serv. Simulation* shows that simulations with normal distribution of service delivery times provide the same results as a uniform distribution (cross and triangles overlap in Figure 12). These results show that the throughput depends on the CoV, but it is independent on the probability density function.

These results also show the relevance of a mechanism to regulate injections rates and avoid bursty behaviours. As it can be seen in Figure 12, assuming exponential distribution of inter-arrival times results into performance near 50%. Without an injection rate regulation of a bursty flow, the service time CoV is less important. Once the incidence of the

[3] `http://www.renew.de`
[4] `http://www.renew.de`

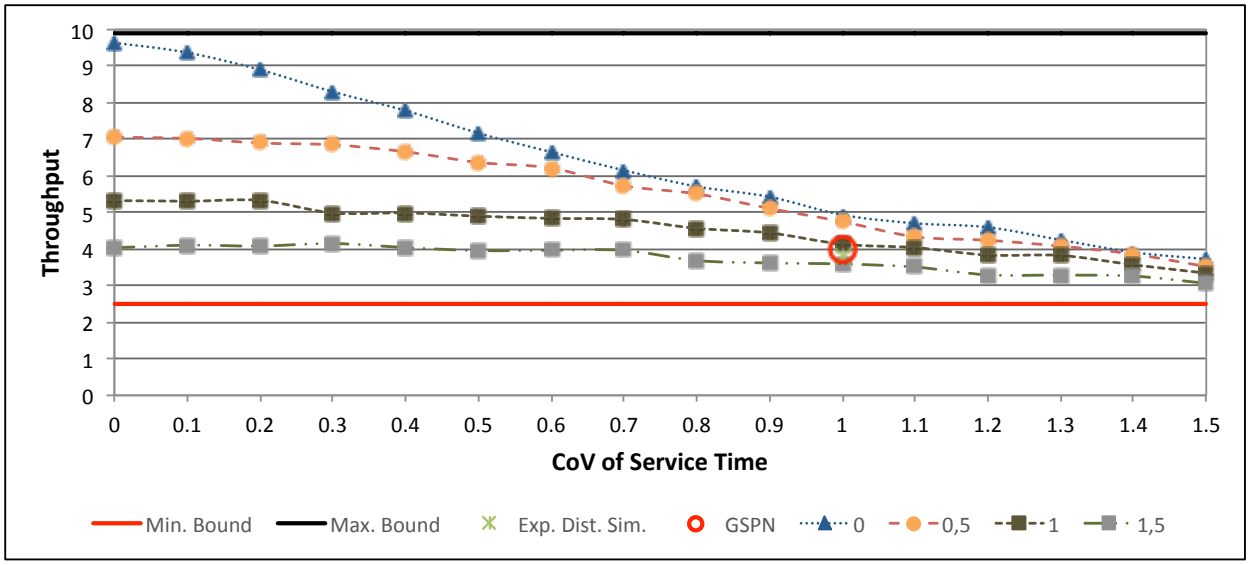

Figure 13: Simulated throughputs with different CoV of service time and injection rates following a Gamma distribution.

inter-arrival time CoV is reduced by any admission control mechanism such as a leaky or token bucket, performance only depends on the Service time CoV improving performance. In [61], a mechanism based on token bucket to regulate bursty streams on multi-tenant cloud environments is proposed.

In Figure 12, we make use of normal, uniform and exponential probability distributions for inter-arrival rates and processing times, whereas in Figure 13, we make use of a gamma distribution. Figure 13 aims to highlight the influence of CoV in the distribution of inter-arrival times and service time using a gamma distribution. Each point shows the mean throughput for four simulations using a gamma distribution with CoV of service time ranging from 0 to 1.5. The figure shows the results for deterministic injection (CoV=0) and CoV=0,5, CoV=1 and CoV=1.5 of injection rate. In this way, we show how simulations are conducted to cover the space of solutions between the minimum and maximum estimated bounds. The purpose is to show that performance does not depend on the probability distribution, but on the Coefficient of Variation (CoV). Besides, with normal and uniform distributions, values of CoV higher than 0.5 cannot be generated. This is why we also used a gamma distribution, which allows us to generate CoV values higher than 0.5.

7.6. Economic Analysis

The knowledge obtained from all the previous analysis techniques can be subsequently used to study the economic cost of executing an application. In this subsection, we also make use of our recurrent example throughout the paper: The wavefront algorithm. We are taking into account the pricing models in [68], which include the pricing cost associated to different data transfers, CPU time or storage usage. For the sake of simplicity, we are just considering an economic cost per CPU usage through time, information that can be added into the model in the following way:

- Let us assume, under a deterministic timing interpretation, a time α for the timed transitions added to represent the duration of the input of data elements from the streams corresponding to the vectors $Y^{(k)}$ and $X^{(k)}$ (timed transitions in parallel with the places $TrYi$ and $TrXi$, a time β for transitions corresponding to the execution of the code of the CPs (timed transitions added in parallel with the places $Comp\_ij$), and a time γ for transitions corresponding to the internal transmissions in the wavefront array. A reachable (exact) bound of the throughput of the system can be computed through presented structural techniques [56], obtaining a value equal to the inverse of $max\{\alpha,\beta,\gamma\}$. This is the first relevant aspect to be taken into account from the economical point of view from our model: The cost derived from the CPU time consumed depends on the slower resource (CPU). Additionaly, under a stochastic timing interpretation, the computed bounds following [56] allow us to derive

economical cost bounds. These bounds can be eventually improved (in [69] a search for embedded queueing networks was considered).

- We can do the following economic analysis on the functional Petri Net model. A bound of the mean cycle time for this net (the elapsed time between two consecutive firings of a transition) is the inverse of the throughput, from this value, we can compute the economic cost for the processing of streams of length $k = n$, assuming the cost of the time unit per CPU, p: $Cost_{functional} = max\{\alpha, \beta, \gamma\} * n * p * 9$, where α, β, γ, n represent the average service time, transmission time, injection rate, inter-arrival time, and the wavefront matrix dimension, respectively. It allows the designer to have an accurate estimation of the cost taking into account the pricing applied to CPU time consumed. Note that this analysis corresponds to the functional level and no information on the operational level has been considered. In this case, the model is designed with nine CPs having each nine computational resources in isolation.

- In case we wanted to introduce a particular operational level, we would have to proceed with the refinement of the previous functional Petri Net model and with a similar analysis to the previous one. Besides, in such a refined model, the relationship between economic cost and performance can be studied: i.e. how performance varies when the economic cost is reduced. Taking into account the quantitative analysis and the simulations, we can also observe the impact that service time and injection rates variance have in performance. The CoV of service time has been pointed as an additional difficulty in estimating the number of resources for optimising cost from the client's point of view, or energy and cost from the cloud provider point of view [11]. The operational cost becomes a function of the service time and injection rate variances: $Cost_{operational} = f(CoV_{ServiceTime}, CoV_{InjectionRate}) * Cost_{functional}$. Function f denotes that the cost depends on the CoV coefficients for injection and processing rates, whereas $Cost_{functional}$ is described in the previous point. Both f and the functional cost are application dependent and can be obtained by conducting the analysis performed in this section.

- The results of previous analysis can be used to reduce the cost in different ways. By knowing the impact of service time and injection rate variances, the performance engineer can integrate and tune a data admission and control mechanism in order to control the performance (as accomplished in [61]). Clients can specify in fixed SLAs the variance of provisioned services and pay for them according to this specification, or at least a way to run meaningful performance experiments to know it [57].

8. Related Work

There is several related work related with the modelling and performance evaluation of parallel programs. A thorough revision of all involved aspects is out of the scope of this paper. As a brief summary, from the modelling point of view, the approaches of I.T. Foster [70] and T.G. Mattson et al. [3] are the main guidelines in conjunction with a set of parallel patterns [29, 17]. More specific to the cloud, Fehling et al. present in [54] the essential patterns for building cloud applications. Besides, there are also works proposing cloud business models, providing reference models for value chains in the Cloud [71]. In this sense, our proposal is compatible with previous methodologies, providing them with a component-based approach and capabilities to reason with the models.

On the other hand, performance models of computer and communications systems have been studied from many years, basically using queueing networks [9]. However, the introduction of synchronisation constraints naturally turned the focus to Petri nets [72]. Performance models of cloud computing have attracted considerable research attention, but likely due to the complexity involved, rigorous analytical approaches have only been adopted by works that focus at the server farm level [10]. Other approaches for performance analysis try to derive Petri net performance models from UML diagrams with standardised annotations in the "UML Profile for Modelling and Analysis of Real-Time Embedded systems" (MARTE) [73]. Unlike these approaches, our emphasis in here is to exploit the inherent nature of PNs for modelling concurrency, and make use of it for modelling the specific aspects of Streaming applications, which involves concurrent computations and data transmissions, and third-party resources.

From the application point of view, for example, we can find some work focusing on performance of the wavefront pattern [74, 45, 75, 17, 76]. However, these works are not based on formal models, and the same happens with other patterns [77, 17]. We can also find some work dealing with scientific workflows and data flows modeled as Directed

Acyclic Graphs (DAG) on grid, cluster and cloud [78]. In general terms in pipelined workflows, performance is typically measured in terms of throughput, and therefore the throughput is conditioned by the slowest task. For such a reason, it is important that all the tasks execute in the same time, which is challenging due to the variability and heterogeneity of computational resources as well as the programs that execute the tasks. Thus, task merging and workflow transformation is essential prior to mapping the tasks onto distributed resources in order to achieve the minimal variation in execution time of tasks. We believe that our proposal in this paper can help analyze the influence of the different design decisions in the task-mapping process on workflow throughput and other properties such as economic cost. Other proposals like [79] utilised Petri Nets for predicting the execution time (makespan) of Taverna workflows at a functional level. It is worth noting the work for bottleneck detection and performance analysis for data intensive applications, related to *Nephele*, the stream engine of *Flink* [80, 6, 31] . However, as these authors point out, the extensive amount of related literature focus only on different abstraction levels.

Finally, framework systems based on specific patterns, or stream processing engines such as *Flink*, *Storm*, or *Spark*, Yahoo's S4 [81], or IBM InfoSphere Streams [82] provide streaming programming abstractions to build and deploy tasks as distributed applications at scale for commodity clusters and clouds.

9. Conclusion and Future Work

In this paper, we described a Petri net-based, model-driven and stepwise refinement methodological approach for CDFA executed over cloud resources. Current available technological solutions (such as Apache Big Data Stack or the solutions by most vendors) provide high-level abstractions that cover different functional and non-functional aspects, but such approaches require from engineers to elaborate every detail of their implementations. Our methodology follows an iterative and cyclic approach, starting from the specification of functional algorithms (specified in the functional models). Then, it continues with the specification of the computational resources available (in the operational models), providing complementary views: Control flow, data-flow, and resources. Such separation of views also enables functional and operational model reuse, and the exploration of what-if scenarios that enable the analysis of the application. Such analysis provides a quantitative and qualitative evaluation of models based on different PN techniques, and the simulation of the executable specification, which are used all together in a synergic manner.

We have illustrated how the separation between the graph-based structure of the model and dynamic properties such as the marking, enables the use of many structure-based analysis techniques. In order to verify some quantitative properties, namely throughput and economic cost, we have added a timing interpretation and have assumed an associated CPU pay-per-use cost. We have estimated performance and cost based on stochastic Petri nets, and conducted simulations under the light of our previous analysis covering the solutions space between the identified bounds. Besides, we made use of the wavefront algorithm, a simple Matrix-Vector multiplication in streaming fashion, as a recurrent example throughout the paper and in order to evaluate our proposal.

To sum up, this paper covers the gap between the capture of functional and non-functional requirements and the design specification given to developers using a specific platform or an architectural solution. However, our approach also comprises some limitations. We believe that it is difficult to change current practice and its inertia, as our approach involves the incorporation of resources early at the functional specification. Additionally, software engineers may not be familiar with formal methods. For these reasons, in order to foster its usage, we also provide a preliminary specification language to support our methodology, which includes high-level abstractions, and which due to space limitations can be found in the appendix.

As future work, we plan to exploit our approach in a real cloud infrastructure in the terms exposed in Section 6.4. Our models can be used in two different ways. On one hand, we can exploit the on-demand provisioning of cloud infrastructures in order to configure and manage the computational cloud resources so that one of the task to resource mapping solutions studied can be arranged. On the other hand, the models can be the basis of an autonomic resource management approach, thereby the elasticity actions on the infrastructure are driven by the models obtained by the methodology. Besides, we also plan to study a formal definition of the Petri net subclass supporting the constructive process of the presented methodology, and we also need to design tools for assisting the realisation of our methodology.

References

[1] D. Mazmanov, C. Curescu, H. Olsson, A. Ton, J. Kempf, Handling performance sensitive native cloud applications with distributed cloud computing and sla management, in: 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing, 2013, pp. 470–475. doi:10.1109/UCC.2013.92.

[2] G. C. Fox, J. Qiu, S. Kamburugamuve, S. Jha, A. Luckow, Hpc-abds high performance computing enhanced apache big data stack, in: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015, pp. 1057–1066.

[3] T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, Addison-Wesley Professional, 2004.

[4] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: A new model and architecture for data stream management, The VLDB Journal 12 (2) (2003) 120–139. doi:10.1007/s00778-003-0095-z.
URL `http://dx.doi.org/10.1007/s00778-003-0095-z`

[5] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, P. Valduriez, Streamcloud: A large scale data streaming system, in: IEEE 30th International Conference onDistributed Computing Systems, ICDCS 2010, 2010, pp. 126 –137. doi:10.1109/ICDCS.2010.72.

[6] B. Lohrmann, D. Warneke, O. Kao, Nephele streaming: stream processing under QoS constraints at scale, Cluster Computing 17 (1) (2014) 61–78.

[7] V. Andrikopoulos, T. Binz, F. Leymann, S. Strauch, How to adapt applications for the cloud environment - challenges and solutions in migrating applications to the cloud, Computing 95 (6) (2013) 493–535.

[8] L. Lamport, Who builds a house without drawing blueprints?, Commun. ACM 58 (4) (2015) 38–41. doi:10.1145/2736348.
URL `http://doi.acm.org/10.1145/2736348`

[9] P. G. Harrison, N. M. Patel, Performance Modelling of Communication Networks and Computer Architectures, Addison-Wesley Longman Publishing Co., Inc., 1992.

[10] H. Khazaei, J. Misic, V. Misic, Performance analysis of cloud computing centers using m/g/m/m+r queuing systems, IEEE Transactions on Parallel and Distributed Systems 23 (5) (2012) 936–943.

[11] S. Yeo, H.-H. Lee, Using mathematical modeling in provisioning a heterogeneous cloud computing environment, Computer 44 (8) (2011) 55–62.

[12] J. O'Loughlin, L. Gillam, Performance evaluation for cost-efficient public infrastructure cloud use, in: Economics of Grids, Clouds, Systems, and Services - 11th International Conference, GECON'14 , Cardiff, UK, September 16-18, 2014., Vol. 8914 of LNCS, 2014, pp. 133–145.

[13] R. Nathuji, A. Kansal, A. Ghaffarkhah, Q-clouds: managing performance interference effects for qos-aware clouds, in: Proceedings of the 5th European conference on Computer systems, ACM, 2010, pp. 237–250.

[14] V. Medel, O. Rana, U. Arronategui, et al., Modelling performance & resource management in kubernetes, in: Proceedings of the 9th International Conference on Utility and Cloud Computing, ACM, 2016, pp. 257–262.

[15] R. Tolosana-Calasanz, J. Á. Bañares, J. M. Colom, Towards Petri net-based economical analysis for streaming applications executed over cloud infrastructures, in: Economics of Grids, Clouds, Systems, and Services - 11th International Conference, GECON'14, Cardiff, UK, September 16-18, 2014., Vol. 8914 of LNCS, 2014, pp. 189–205.

[16] A. Merino, R. Tolosana-Calasanz, J. Á. Bañares, J. M. Colom, A specification language for performance and economical analysis of short term data intensive energy management services, in: Economics of Grids, Clouds, Systems, and Services - 12th International Conference, GECON'15, Cluj-Napoca, Romania, September 15-17, 2015., Vol. 9512 of LNCS, 2015, pp. 1–17.

[17] L. Yu, C. Moretti, A. Thrasher, S. J. Emrich, K. Judd, D. Thain, Harnessing parallelism in multicore clusters with the All-Pairs, Wavefront, and Makeflow abstractions., Cluster Computing 13 (3) (2010) 243–256.

[18] S. Jha, D. S. Katz, A. Luckow, N. C. Hong, O. F. Rana, Y. Simmhan, Introducing distributed dynamic data-intensive (D3) science: Understanding applications and infrastructure, Concurrency and Computation: Practice and Experience 29 (8). doi:10.1002/cpe.4032.
URL `https://doi.org/10.1002/cpe.4032`

[19] Fish4Knowledge Project. http://fish4knowledge.eu/.

[20] M. Simsek, A. Aijaz, M. Dohler, J. Sachs, G. Fettweis, 5g-enabled tactile internet, IEEE Journal on Selected Areas in Communications 34 (3) (2016) 460–473. doi:10.1109/JSAC.2016.2525398.

[21] R. Tolosana-Calasanz, J. D. Montes, O. F. Rana, M. Parashar, E. Xydas, C. E. Marmaras, P. Papadopoulos, L. Cipcigan, Computational resource management for data-driven applications with deadline constraints, Concurrency and Computation: Practice and Experience 29 (8).

[22] I. Petri, O. F. Rana, Y. Rezgui, H. Li, T. Beach, M. Zou, J. D. Montes, M. Parashar, Cloud supported building data analytics, in: 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014, 2014, pp. 641–650.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets., HotCloud 10 (10-10) (2010) 95.

[24] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, I. Stoica, Mesos: A platform for fine-grained resource sharing in the data center., in: NSDI, Vol. 11, 2011, pp. 22–22.

[25] R. Tolosana-Calasanz, J. D. Montes, O. F. Rana, M. Parashar, Feedback-control & queueing theory-based resource management for streaming applications, IEEE Trans. Parallel Distrib. Syst. 28 (4) (2017) 1061–1075. doi:10.1109/TPDS.2016.2603510.
URL `https://doi.org/10.1109/TPDS.2016.2603510`

[26] G. J. Holzmann, Conquering complexity, Computer 40 (12) (2007) 111–113.

[27] C. C. Seceleanu, I. Crnkovic, Component models for reasoning, IEEE Computer 46 (11) (2013) 40–47. doi:10.1109/MC.2013.335.

[28] O. Etzion, P. Niblett, Event Processing in Action, 1st Edition, Manning Publications Co., Greenwich, CT, USA, 2010.

[29] C. Pautasso, G. Alonso, Parallel computing patterns for Grid workflows, in: Proceedings of the HPDC 2006 Workshop on Workflows in Support of Large-Scale Science, WORKS 2006, June 19-23, Paris, France, 2006, pp. 1–10.

[30] Y. Simmhan, A. G. Kumbhare, Floe: A continuous dataflow framework for dynamic cloud applications, CoRR abs/1406.5977.

[31] B. Lohrmann, P. Janacik, O. Kao, Elastic stream processing with latency guarantees, in: Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on, 2015, pp. 399–410.

[32] R. Tolosana-Calasanz, J. Á. Bañares, J. M. Colom, On autonomic platform-as-a-service: Characterisation and conceptual model, in: G. Jezic, R. J. Howlett, L. C. Jain (Eds.), Agent and Multi-Agent Systems: Technologies and Applications, Vol. 38 of Smart Innovation, Systems and Technologies, Springer International Publishing, 2015, pp. 217–226.
[33] T. Murata, Petri nets: Properties, analysis and applications, in: Proceedings of IEEE, Vol. 77, 1989, pp. 541–580.
[34] W. Thong, M. Ameedeen, A survey of Petri Net Tools, in: H. A. Sulaiman, M. A. Othman, M. F. I. Othman, Y. A. Rahim, N. C. Pee (Eds.), Advanced Computer and Communication Engineering Technology, Vol. 315 of Lecture Notes in Electrical Engineering, Springer International Publishing, 2015, pp. 537–551.
[35] F. Basile, P. Chiacchio, On the implementation of supervised control of discrete event systems, IEEE Transactions on Control Systems Technology 15 (4) (2007) 725–739. doi:10.1109/TCST.2006.890281.
[36] R. P. Moreno, D. Tardioli, J. L. V. Salcedo, Distributed implementation of discrete event control systems based on petri nets, in: 2008 IEEE International Symposium on Industrial Electronics, 2008, pp. 1738–1745. doi:10.1109/ISIE.2008.4676963.
[37] L. Yu, C. Moretti, S. Emrich, K. Judd, D. Thain, Harnessing parallelism in multicore clusters with the All-pairs and Wavefront abstractions, in: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, HPDC 2009, ACM, New York, NY, USA, 2009, pp. 1–10.
[38] S.-Y. Kung, K. S. Arun, R. J. Gal-Ezer, D. V. B. Rao, Wavefront array processor: Language, architecture, and applications., IEEE Trans. Computers 31 (11) (1982) 1054–1066.
[39] L. Lamport, The parallel execution of do loops, Commun. ACM 17 (2) (1974) 83–93. doi:10.1145/360827.360844.
URL http://doi.acm.org/10.1145/360827.360844
[40] B. Sinharoy, B. Szymanski, Finding optimum wavefront of parallel computation, in: [1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences, Vol. ii, 1993, pp. 225–234 vol.2.
[41] A. Hoisie, O. M. Lubeck, H. J. Wasserman, Performance analysis of wavefront algorithms on very-large scale distributed systems, in: Workshop on Wide Area Networks and High Performance Computing, Springer-Verlag, 1999, pp. 171–187.
[42] N. Manjikian, T. S. Abdelrahman, Exploiting wavefront parallelism on large-scale shared-memory multiprocessors, IEEE Trans. Parallel Distrib. Syst. 12 (3) (2001) 259–271. doi:10.1109/71.914756.
URL http://dx.doi.org/10.1109/71.914756
[43] G. R. Mudalige, S. A. Jarvis, D. P. Spooner, G. R. Nudd, Predictive performance analysis of a parallel pipelined synchronous wavefront application for commodity processor cluster systems, in: 2006 IEEE International Conference on Cluster Computing, 2006, pp. 1–12. doi:10.1109/CLUSTR.2006.311888.
[44] S. Pennycook, S. Hammond, G. Mudalige, S. Wright, S. Jarvis, On the acceleration of wavefront applications using distributed many-core architectures, The Computer Journal 55 (2) (2012) 138.
[45] E. C. Lewis, L. Snyder, Pipelining Wavefront Computations: Experiences and Performance, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 261–268.
[46] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, K. Tan, Generating parallel programs from the wavefront design pattern, in: Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM, 2002, pp. 8 pp–.
[47] C. E. R. Alves, E. N. Cáceres, F. Dehne, S. W. Song, Computational Science and Its Applications — ICCSA 2003: International Conference Montreal, Canada, May 18–21, 2003 Proceedings, Part II, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, Ch. A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison, pp. 249–258.
[48] D. Serfass, D. Serfass, P. Tang, parallel_dp: the parallel dynamic programming design pattern as an intel® threading building blocks algorithm template, in: ACM Southeast Regional Conference, ACM, 2013, pp. 13:1–13:6.
[49] Y. H. Tan, W. S. Lee, J. Y. Tham, S. Rahardja, Complexity-rate-distortion optimization for real-time h.264/avc encoding, in: Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on, 2009, pp. 1–6.
[50] F. Diotalevi, A. Fijany, G. Sandini, Current Advancements in Stereo Vision, Vol. 2012, Ch. Wavefront/Systolic Algorithms for Implementation of Stereo Vision and Obstacle Avoidance Computations on a Very Low Power MIMD Many-Core Parallel Architecture: Applications for Mobile Systems and Wearable Visual Guidance.
[51] S. M. LaValle, S. A. Hutchinson, Optimal motion planning for multiple robots having independent goals, IEEE Transactions on Robotics and Automation 14 (6) (1998) 912–925.
[52] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, H. Alme, A general predictive performance model for wavefront algorithms on clusters of smps, in: Parallel Processing, 2000. Proceedings. 2000 International Conference on, 2000, pp. 219–228.
[53] P. N. Clauss, J. Gustedt, F. Suter, Out-of-core wavefront computations with reduced synchronization, in: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), 2008, pp. 293–300.
[54] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter, Cloud Computing Patterns - Fundamentals to Design, Build, and Manage Cloud Applications, Springer, 2014. doi:10.1007/978-3-7091-1568-8.
[55] R. Weingärtner, G. B. Bräscher, C. B. Westphall, Cloud resource management: A survey on forecasting and profiling models., J. Network and Computer Applications 47 (2015) 99–106.
[56] J. Campos, G. Chiola, J. M. Colom, M. Silva, Properties and performance bounds for timed marked graphs, Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on 39 (5) (1992) 386–401.
[57] J. Schad, J. Dittrich, J.-A. Quiané-Ruiz, Runtime measurements in the cloud: Observing, analyzing, and reducing variance, Proc. VLDB Endow. 3 (1-2) (2010) 460–471.
[58] D. C. Marinescu, Cloud Computing: Theory and Practice, Morgan Kaufmann, 2013.
[59] R. Tolosana-Calasanz, J. A. Bañares, C. Pham, O. F. Rana, Enforcing QoS in Scientific Workflow Systems Enacted Over Cloud Infrastructures, Journal of Computer and System Sciences To appear (2012) 1–20.
[60] R. Tolosana-Calasanz, J. A. Bañares, O. F. Rana, Autonomic streaming pipeline for scientific workflows, Concurr. Comput. : Pract. Exper. To Appear (2011) 1–31.
[61] R. Tolosana-Calasanz, J. Á. Bañares, C. Pham, O. F. Rana, Resource management for bursty streams on multi-tenancy cloud environments, Future Generation Computer Systems 55 (2016) 444–459.

[62] V. Medel, O. Rana, J. a. Bañares, U. Arronategui, Modelling performance and resource management in kubernetes, in: Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC '16, ACM, New York, NY, USA, 2016, pp. 257–262.
[63] A. Brogi, A. Canciani, J. Soldani, P. Wang, Modelling the behaviour of management operations in cloud-based applications, in: Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'15) satellite event of 36th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2015, Brussels, Belgium, June 22-23, 2015., 2015, pp. 191–205.
URL http://ceur-ws.org/Vol-1372/paper11.pdf
[64] F. Bause, P. S. Kritzinger, Stochastic Petri nets - an introduction to the theory (2. ed.), Vieweg, 2002.
[65] M. Silva, J. M. Colom, On the computation of structural synchronic invariants in P/T nets, in: Advances in Petri Nets 1988, Vol. 340 of Lecture Notes in Computer Science, Springer, 1988, pp. 386–417.
[66] S. Baarir, M. Beccuti, D. Cerotti, M. De Pierro, S. Donatelli, G. Franceschinis, The GreatSPN Tool: Recent Enhancements, SIGMETRICS Perform. Eval. Rev. 36 (4) (2009) 4–9.
[67] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, R. Valk, An extensible editor and simulation engine for Petri nets: Renew, in: International Conference on Application and Theory of Petri Nets, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 484–493.
[68] A. Gohad, N. C. Narendra, P. Ramachandran, Cloud pricing models: A survey and position paper., in: Cloud Computing in Emerging Markets (CCEM), 2013 IEEE International Conference on, IEEE, 2013, pp. 1–8.
[69] J. Campos, M. Silva, Embedded product-form queueing networks and the improvement of performance bounds for Petri net systems, Performance Evaluation 18 (1) (1993) 3–19.
[70] I. T. Foster, Designing and building parallel programs - concepts and tools for parallel software engineering, Addison-Wesley, 1995.
[71] A. B. Mohammed, J. Altmann, J. Hwang, Cloud computing value chains: Understanding businesses and value creation in the cloud, in: Economic models and algorithms for distributed systems, Springer, 2009, pp. 187–208.
[72] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, P. Computing, J. Wiley, V. Almeida, J. Almeida, C. M. P. Analysis, Modelling with generalized Stochastic Petri nets, in: Series in Parallel Computing, John Wiley & Sons, 1995.
[73] M. Woodside, D. C. Petriu, J. Merseguer, D. B. Petriu, M. Alhaj, Transformation challenges: from software models to performance models, Software & Systems Modeling 13 (4) (2014) 1529–1552.
[74] A. Hoisie, O. Lubeck, H. Wasserman, Workshop on wide area networks and high performance computing, Springer London, London, 1999, Ch. Performance analysis of wavefront algorithms on very-large scale distributed systems, pp. 171–187.
[75] P. N. Clauss, J. Gustedt, F. Suter, Out-of-core wavefront computations with reduced synchronization, in: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), 2008, pp. 293–300.
[76] A. J. Dios, R. Asenjo, A. Navarro, F. Corbera, E. L. Zapata, Evaluation of the task programming model in the parallelization of wavefront problems, in: High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on, 2010, pp. 257–264.
[77] D. Jiang, B. C. Ooi, L. Shi, S. Wu, The performance of mapreduce: An in-depth study, Proc. VLDB Endow. 3 (1-2) (2010) 472–483.
[78] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, J. Good, On the use of Cloud computing for scientific workflows, in: 3rd International Workshop on Scientific Workflows and Business Workflow Standards in e-Science (SWBES), in conjunction with Fourth IEEE International Conference on e-Science (e-Science 2008), 2008, pp. 640–645.
[79] R. Tolosana-Calasanz, O. F. Rana, J. A. Bañares, Automating performance analysis from Taverna workflows, in: Component-Based Software Engineering: 11th International Symposium, CBSE 2008, Karlsruhe, Germany, October 14-17, 2008. Proceedings, 2008, pp. 1–15.
[80] D. Battré, M. Hovestadt, B. Lohrmann, A. Stanik, D. Warneke, Detecting bottlenecks in parallel dag-based data flow programs, in: 2010 3rd Workshop on Many-Task Computing on Grids and Supercomputers, 2010, pp. 1–10.
[81] L. Neumeyer, B. Robbins, A. Nair, A. Kesari, S4: Distributed stream computing platform, in: Proceedings of the 2010 IEEE International Conference on Data Mining Workshops, ICDMW '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 170–177.
[82] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, C. Moran, Ibm infosphere streams for scalable, real-time, intelligent transportation services, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 1093–1104.