A Specification Language for Performance and Economical Analysis of Short Term Data Intensive Energy Management Services

Alberto Merino, Rafael Tolosana-Calasanz, José Ángel Bañares (✉), and José-Manuel Colom

Dpto. de Informática e Ingeniería de Sistemas. Universidad de Zaragoza, Spain
`albertomerort@gmail.com,{rafaelt,banares,jm}@unizar.es`

Abstract. Requirements of Energy Management Services include short and long term processing of data in a massively interconnected scenario. The complexity and variety of short term applications needs methodologies that allow designers to reason about the models taking into account functional and non-functional requirements. In this paper we present a component based specification language for building trustworthy continuous dataflow applications. Component behaviour is defined by Petri Nets in order to translate to the methodology all the advantages derived from a mathematically based executable model to support analysis, verification, simulation and performance evaluation. The paper illustrates how to model and reason with specifications of advanced data flow abstractions such as smart grids.

1 Introduction

The Smart Power Grid (SG) domain is a prototypical scenario to illustrate the need of different services that will require short and long term processing. The existing paradigm of passive distribution and one-way communications from large suppliers to final consumers will be replaced by an active and responsive system. The development of not only generation but also micro generation at a large scale may pose a big challenge for different actors interacting within a physically constrained network through various ICT systems to efficiently operate the power grid. Actors in SG comprise Bulk Generation, Transmission System Operators, Distribution System Operators, End-users becoming proactive actors, markets, and third service providers to support processes of others actors [5].

From the ICT point of view, data flow applications processing real-time data from thousands of devices make possible new ways to monitor the grid networks and provide new means to accurately predict and effectively respond to events. In the SG scenario events related to different actors occur anywhere in the power generation, distribution and demand chain requiring the joint analysis of heterogeneous massive data sources. It implies different coordinated data flows (forecasting, monitoring, control, etc.) that should be fed with several heterogeneous data sources and that will require the composition of different data flow abstractions to define its functionality.

Recently, several works have presented the benefits of using large data centers with massive computation and storage capacities operated by Cloud providers [4]. Traditional electric utilities and new actors in the SG lack the resources and expertise on all ICT background. Cloud service providers will be in charge on the design, deployment, maintenance and upgrades of the plethora of data intensive services that will make possible the next power system generation. Sharing services executed on scalable platforms that adapt the resources to the evolving demand of the different actors in the SG domain allow electric utilities reduce operational cost. From the cloud service provider point of view, it requires the orchestration of different expertise by means of a common framework that facilitates the integration of data intensive analysis tools, and provides data-intensive engineers the mechanisms to deploy services on cloud infrastructures. This trend may also inspire new services in the SG domain, such as personal services derived from smart meters that analyze user behaviors and optimize the appliance consumption; planning electrical vehicle charges to protect components of the distribution network from being overloaded; or advanced trading services to support more flexible power grid asset buying and selling operations.

The requirements for several Energy Management services that will facilitate effortless energy monitoring and control have been presented in [1]. These requirements of combining long and short-term processing in different scenarios have motivated the Lambda Architecture proposed by Nathan Marz[9]: a generic, scalable and fault-tolerant data processing architecture. In a Lambda architecture data entering the system are dispatched to both the *batch layer* and the *speed layer* for processing. The batch layer manages dataset and pre-computes batch views; and the speed layer compensates high latency of batch layer dealing with recent data only and providing real-time views. The speed layer is fundamental to develop services related to network operation or real state estimation of the grid network. Uncertainties in the real-time views of the system state will still remain, but a certain amount of unpredictability should be assumed, as the expected systems behaviour is intrinsically random, heterogeneous and adaptive. One of the Lambda Architecture strengths is the complexity isolation, meaning that complexity is pushed into the speed layer whose results are only temporary. However, isolating complexity does not means it disappears resulting in a speed layer that is far more complex than the batch layer.

In our previous paper [15], we presented the principles of a methodological approach to specify and analyze real-time data intensive applications executed over a general cloud software architecture. The execution may incur an economical cost, and it can be therefore important to conduct an analysis prior to any execution. Such an analysis can explore how economic cost is interrelated to performance and functionality. The proposed methodology is based on the intensive use of formal executable models used to obtain qualitative information and analysis on performance and economic behaviours under different scenarios. The use of formal executable specifications has a twofold objective: 1) To allow the engineer to conduct and analyse the models in an intensive way before the deployment of the application in order to understand its behavior, and to identify the boundaries of QoS parameters of the different adopted solutions; and 2) Reason on functional and non-functional requirements with functional and performance models, which constitute an essential element of the system knowledge.

In this paper we go on in the development of the methodology presenting a specification language to support the proposed methodology. Then, we show the reasoning capabilities that provides the use of a formal model. The rest of this paper is structured as follows. In Section 2 a brief overview of the data flow language specification is presented. It is illustrated by means of the Matrix-Vector Multiplication problem in streaming fashion, an example of the kind of advanced data flow abstractions required for real-time SG state estimations in the cloud. This problem can be adequately modeled by the wavefront pattern [18]. Section 3 analyses the use of different analysis possibilities of the Petri net (PN) underlying the model, and finally conclusions are given in Section 4.

2 Specification Language for basic and advanced data flow applications

Once delegated the ICT complexity to third cloud service providers, the first step is the definition of a methodology providing the strategy to afford the complexity and defining the framework to support it. Different aspects of information processing information integration, data mining, complex event processing, machine learning and SG state monitoring must be considered and coordinated. Several of the focus areas identified above resemble those seen in eScience and the Big Data.

In [15] we presented the principles of the methodology to cope with the inherent complexity of Continuous Data Flow Applications (CDFAs). The methodology considers all involved elements at different abstraction levels. In this paper, we present the specification language to support the methodology with the following requirements that go beyond pure functionality: (1) The development of CDFAs must be supported by a specification language that provides different views of the constructive elements at different abstraction levels. The language should support complementary capacities for the description of the application or components of the application: behavioral specification of concurrent processes, transformations operated over the data flow, and structural description of components that configure the application. (2) A formal component-based development to build models from existing components and capability to reason about the resulting composition. Reuse of components allows developers to use knowledge of their properties to predict the new system properties [12]. (3) Workflow and data flow patterns have been widely studied [11, 18]. More advanced data flow abstractions such as MapReduce, or Bulk Synchronous parallel (BSP) constitute the essence of parallel programming frameworks such as Hadoop and Hama. However, these frameworks are only instances of many possible parallel execution templates. The reasonable approach is a general specification language that combines eScience workflow and large-grain dataflow abstractions [13].

These principles have guided the design of our component based specification LANGuage of Layers and tIERS (*Langliers*) to support a methodology for building trustworthy CDFAs. That is, the design of layers concerned with the logical division of components and functionality, and the design of tiers concerned with the physical distribution of components. The Hierarchical construction of the specification of CDFAs requires the definition of composition and refinement primitives. The semantics of the component-based language will be defined formally through standard PNs in order to translate to the methodology all the advantages derived from a mathematically based model: Analysis, Verification, Equivalence Relations, etc. Due to space limitations, the paper sketches a CDFA language specification focusing on the structural and behavioural specifications.

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 infrastructures [11]. Our methodological approach to identify the elements to compound our hybrid specification can be summarised with the equation 'Specification of 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.

The constructive elements of a CDFA application are presented in this section, starting from the most basic building blocks and their interpretation as constructive primitives of distributed applications and continuing with their composition by means of simple operators (subsection 2.1). Composition operators provide the way to configure components with complex behaviours . Then, we show how *Langliers* represents the basic components identified in [15], Computational and Data Transmission Processes, to describe a CDFA network as a graph showing various connected processing components that operates over a data stream. This processing network model is an abstraction that describes the **functional** behaviour of a CDFA made up of a number of platform-independent components. The explicit network specification allows developers to visualize the functional model and apply analysis techniques. The last remaining step to specify a complete model is to set the implementation of the functional model with the specification of the resources that will be used to execute the model. This **operational** specification can be used to conduct performance optimizations selecting a good mapping of Computational Processes and Data Transmission Processes to computational and network resources.

2.1 Basic building blocks and composition operators

The precise and formal specification of components requires a description beyond architectural units. It is not enough to specify the interface. Specification of nonfunctional or quality attributes is also required to enable different types of reasoning. In *Langliers* a component is expressed in the form of an *interface* and a *behaviour* description [12]. The **interface** provides a specification for the services the component provides and publishes its input and output points. It gives information at the syntax level that enables *data type checking*, and publishes the *events* that trigger computations and state changes. This way, our data stream model follows a data driven execution model where events lead to computations that may generate events on other components. The ***behaviour*** represents components internal states and state changes. A component behavior description is specified either by one explicit PN or by the PN resulting from the behavior composition of subcomponents.

Once defined the elements of a component description, we present the basic building primitive components to describe a distributed system. The behaviour specification of these building primitives is represented by one of the PNs shown in Figure 1. Transitions and places used for the composition of behaviours are part of the interface declaration and can have respectively associated a set of parameters or data type. Behaviour specification can have inscription expressions labelling arcs in the same way as high-level PNs (HLPNs) [6] (e.g., colored PNs or predicate/transition nets), which provides more compact and manageable descriptions. Internal transitions can also be labelled with actions to represent atomic actions that may be executed after the firing of the transition. It allows conventional programming language methods to describe operations and provides an executable functional specification in a similar way to `Renew`[1], which utilises tuples and Java expressions as the primary inscription language. For the sake of simplicity, in this paper a detailed description of parameters and PN inscriptions are left out of the specification.

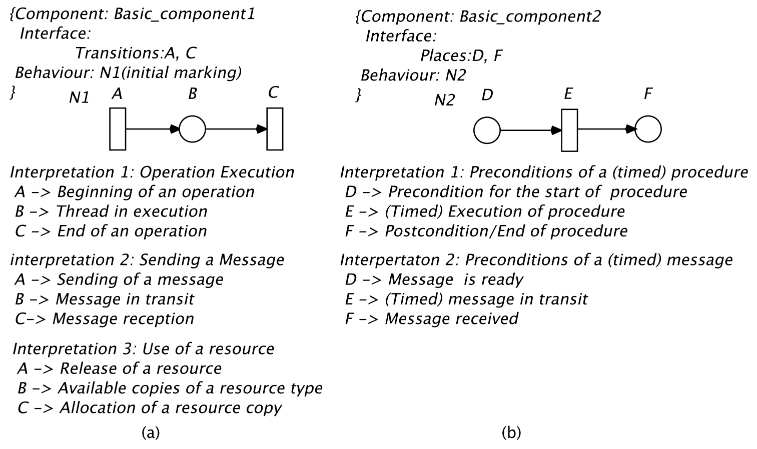

{Component: Basic_component1
Interface:
Transitions:A, C
Behaviour: N1(initial marking)
}
N1
A
B
C
Interpretation 1: Operation Execution
A -> Beginning of an operation
B -> Thread in execution
C -> End of an operation
interpretation 2: Sending a Message
A -> Sending of a message
B -> Message in transit
C-> Message reception
Interpretation 3: Use of a resource
A -> Release of a resource
B -> Available copies of a resource type
C -> Allocation of a resource copy
(a)
{Component: Basic_component2
Interface:
Places:D, F
Behaviour: N2
}
N2
D
E
F
Interpretation 1: Preconditions of a (timed) procedure
D -> Precondition for the start of procedure
E -> (Timed) Execution of procedure
F -> Postcondition/End of procedure
Interpertaton 2: Preconditions of a (timed) message
D -> Message is ready
E -> (Timed) message in transit
F -> Message received
(b)

Fig. 1. Component specification of basic component primitives and interpretations.

We propose two basic building-blocks with different precise interpretations. Figure 1(a) shows the *Basic Component1* that declares as behaviour the Petri net $\mathcal{N}_1$. Interface Transition A requires a component with a Transition (output

[1] `http://www.renew.de`

port) that triggers transition A. Transition A represents the beginning of the execution of an operation in the first interpretation, the beginning of the sending of a message in the second interpretation and the release of a resource in the third interpretation. Interface Transition C requires a component to trigger the execution of an event synchronised with the end of the operation in the first interpretation, the reception of a message in the second interpretation and the allocation of a resource copy in the third interpretation. Place B in Figure 1(a) is an internal state with the interpretations presented. An initial marking can be defined when this component is declared as subcomponent of another component.

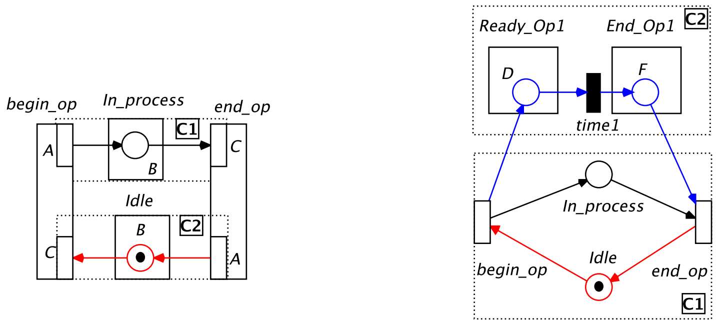
Figure 1(b) shows the *Basic Component2* that declares as behaviour the PN $\mathcal{N}_2$. Interface Place D represents events received and requires a component that sends events to this place, and Interface Place F represents generated events waiting to be consumed requiring a component that consumes these events. Place D represents preconditions to start the execution of an operation in the first interpretation and that a message is ready in the second interpretation. Place F represents the ending of an operation or a received message waiting to be consumed. Transition E represents an internal action, for example a procedure executed in a machine or a network transmission. It can be timed to represent non-instantaneous actions.

The declaration of a "Composite" component is similar to a basic component with an additional declaration of subcomponents, connection of subcomponents and definition of its interface (ports and data-flow) as a mapping of input and output ports to the input and output ports of subcomponents. Three simple composition operators are provided to define Composites. They are based on the fusion of transitions and places in the interface: ***split***, ***fusion*** and ***copy***. The ***split*** operator replaces the argument, place or transition, by a couple of places or transitions, where the first resulting place or transition has the input arcs of the split argument, and the second one the output arcs. The ***fusion*** operator replaces the arguments, places or transitions, by a new place or transition that has the input and output arcs of the arguments. Finally, the ***copy*** operator creates a new place or transition identical to the argument.

The construction of the functional model is based on the identification of the basic components that configures a CDFA. These components 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 computational infrastructures will be added, such as limitations in parallelism, capacity, etc. The behavioural description of CPs and DTPS was presented in [15]. Figure 2 shows the *Langliers* specification of an untimed and timed CPs.

Figure 2.(a). shows a CP defined by *Op_Component*. This basic component is defined by the composition of two instances of *Component_1*, the first one with the first interpretation and the second with the third interpretation semantic.

The second component declaration includes an initial marking of one token in Place B. On the left is represented the composition of PNs describing the sub-components behaviour, and on the right the textual component specification. Composites consist in a declaration of subcomponents, and a declaration of port mappings or links by means of the PN based composition operators. The rename declaration is used to rename places and transitions of subcomponents or the result of composition operators.



```
{Component: Op_Component
 Interface:
   Transitions :begin_op, end_op
 Behaviour:
  Components: C1:Component_1, C2: Component_1 (1)
  Links: begin_op:C1(A)+C2(C), end_op:C1(C)+C2(A)
  Rename:  In Process: C1(B), Idle: C2(B)
}
```

(a)

```
{Component: Timed_Op_Component
 Interface:
  Transitions: begin_op, end_op
 Behaviour:
  Components: C1: Op_Component
                  C2: Component2(time1)
  Links: impl (C1(Operation1), C2(D), C2(F)),
  Rename:  C2(D): Ready_Op1, C2(F): End_Op1,
}
```

(b)

Fig. 2. A Computational Process with Resources. a) A CP composed by 1 state. b) Timed model assigning $time_i$ units of time to the execution of the $operation_i$.
Legend: Textual representation of components on the right follows the frame-based tradition of concepts as attribute-value pairs: `Component`: Name of component; `Interface`: List of Place and Transitions that can be used to compose components; `Behaviour`: Either a reference to a PN $\mathcal{N}$, or a list of `Components`, `Links` that connect components, and `Rename` declarations; `Places` and `Transitions` are referenced by the component name followed by the Place/Transition name placed in parenthesis.

The model presented in Figure 2.(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 labelled with time information representing the duration of the computational operation. In Figure 2.(b), the CP from Figure 2.(a) is refined by assigning *time1* units of time to the execution of the operation 1. Note that the `Link` declaration can be complex enough to require the use of a procedural language for the composition of operators. Two new operators are defined using basic operators. ***Split-Copy*** splits a copy of a place, and ***impl*** operator specifies the way an operation is implemented. The ***impl*** makes a ***split-Copy*** of the first argument and fusions the first returned place with the second argument, and the second place with the third argument. The `Links` declaration in Figure 2.(b) uses the ***impl*** operator to refine *In_process*.

2.2 *Langliers* specification of a Cloud based Operational Model of a Matrix-Vector Multiplication in streaming

To illustrate a more complex specification with *Langliers* we present the specification of an operational model for Matrix-Vector Multiplication in streaming based on the wavefront abstraction [18]. The wavefront abstraction is a paradigmatic regularly structured framework that allow developer to focus on simple sequential programs to create very large parallel programs. Using the regular structure and declarative specification, a wavefront may be materialized in different ways on distributed, multicore, and distributed multicore systems showing different performances. The election of a wavefront pattern illustrates the use of an advanced data flow abstraction that neither is it present as primitive in workflow languages, nor in advanced parallel frameworks. However, this kind of data flow abstraction is essential in SG real time monitoring services. Accurate state estimation has become a key function in supervisory control and planning of electric power grids and is widely expected to play a significant role in the advance of smart grid [8]. Traditional methods compute state estimations by the least squares solution of linear equations. In the more simple case, as soon the measurements are obtained, the estimate is obtained by matrix multiplication. The matrix that converts the measurements to the state estimate is constant as long as the grid network does not change [3].

The functional PN model of the wavefront algorithm for $Z^{(k)} = Y^{(k)} + A \cdot X^{(k)}$, $k = 1, 2, ...$ was presented in [15]. In this section is presented a functional and operational specification in *Langliers*.

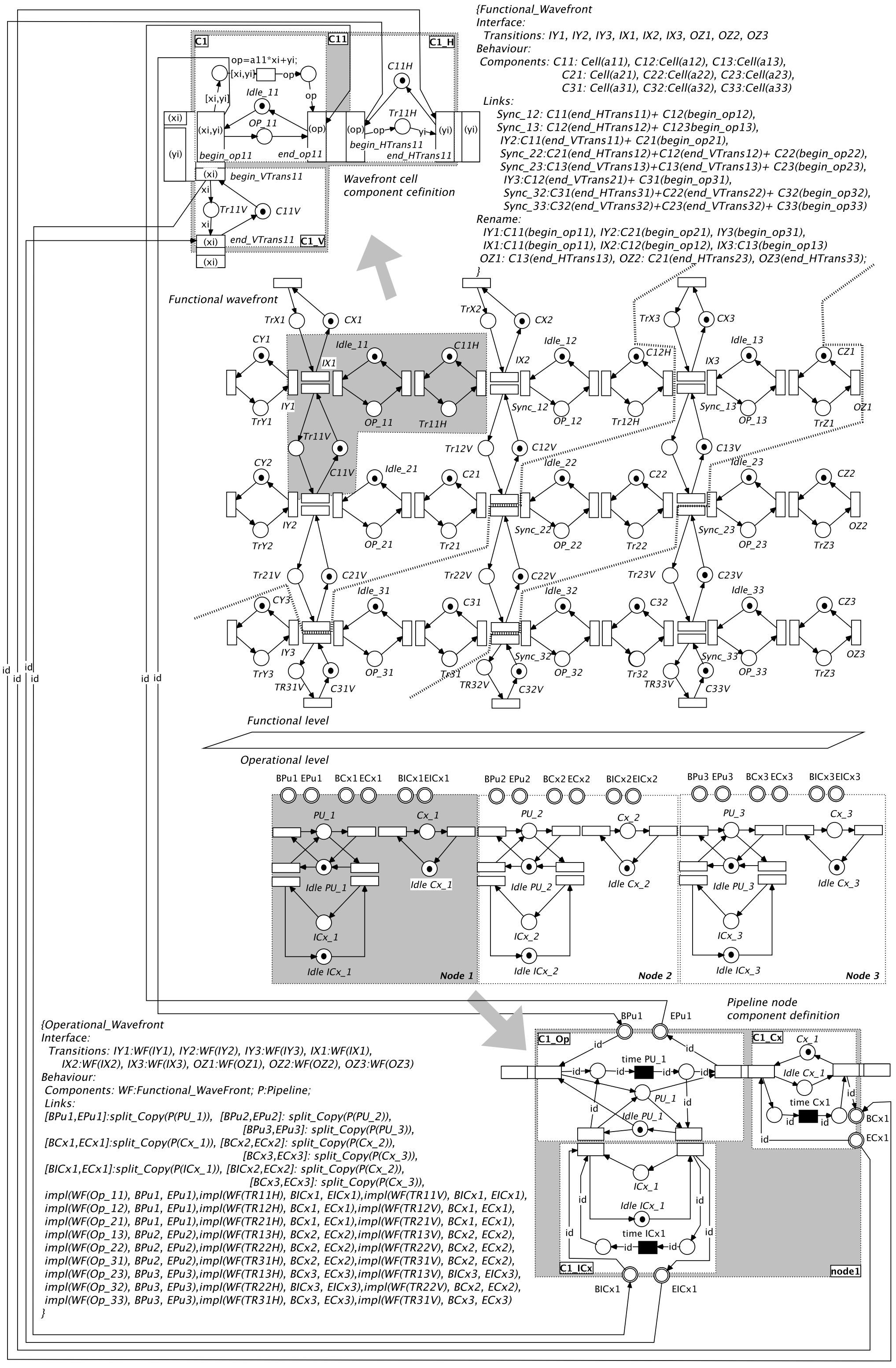
The untimed functional PN specification of the wavefront algorithm is shown in the upper part of the Figure 3. The functional model is constructed in a modular fashion. Each element of the wavefront array is defined as a composite *Cell* component made up of basic subcomponents (see upper Figure 3 *Cell* component specification): (1) A subcomponent to describe the Computational Process carried out in a node of the wavefront array; and (2) two subcomponents to describe Data Transmission Processes of the data streams from *Cell*s at the north/east to *Cell*s in the south/west. Observe that *OP_ij* places are refined adding a sequence place-transition-place in parallel with the place representing the operation that is executed over each data of the data flow. The new added transition is labeled with the procedural action that must be carried by each *Cell* component: $op = aij * xj + yi$. Each *Cell* component is initialised with its *aij* element of matrix A. Arcs and transitions are annotated with variables defining a complete functional specification.

To construct the global functional model, nine instances of the *Cell* component are needed (Figure 3.). Each *Cell_ij* component is connected by a *Sync_ij* transition resulting from the fusion of the transition *begin_opij* with the transition $end\_HTransi(j-1)$ of the east $Cell$ and the the transition $end\_VTransij$ with the $begin\_op(i+1)j$ of the south $Cell$.

Each one of $Sync\_1j$ transition at the first row does not have a $Cell$ component connected at the North, and constitute the IYi interface transitions representing the input stream of the corresponding i-th component of the vector $X^{(k)}$ via the fusion of the transitions $begin\_op\_1j$ with a the $end\_Transmission$ of a DTP component. In the same way, each one of $Sync\_i1$ transition at of the first column does not have a left $Cell$ component, and constitute the IXj interface transitions representing the input stream of the corresponding j-th component of the vector $Y^{(k)}$ via the fusion of the transitions $begin\_op\_i1$ with a the $end\_Transmission$ of a DTP component. Each one of $End\_HTransi3$ transitions in $Cell$ components of the last column is the output stream of the corresponding i-th component of the vector $Z^{(k)}$ and constitute the OZi interface transitions.

Timed Operational specification for the wavefront algorithm. We consider a cloud computing infrastructure where each $Cell$ component can be executed over a different Virtual Machine (VM). We assume a cloud platform (OpenNebula, OpenStack, etc.) provides networking and computing virtualisation. For each token in the Idle places $Idle\_ij$ we assume that there is a VM implementing a computational resource that can accomplish the same functionality with similar performance, and for each token in the $CijH$ and $CijV$ places we assume a transmission over the network assuming than the use of virtualisation technologies does not alter the original incoming data injection rate at each data flow [16]. In this case, the addition of time to the the model of Figure 3 will be done in the way described in the previous section: adding a sequence place-transition-place in parallel with the places representing the activities that consume time by the $Cell$ located at row i-th, column j-th: OP_ij representing the duration of the computation, and in parallel with places $CijH$ and $CijV$ representing the consumption of time in the transmission of a data element in the corresponding DTPs. Following this approach we obtain an operational net model that is isomorphous to the functional model in Figure 3.

An alternative more complex operational model is shown at the bottom part of the Figure 3. It shows a pipeline of three nodes and graphically the $Node$ component specification. Each node component is composed of: (1) A subcomponent to describe the available Computational Resources at each node, and (2) two subcomponents to describe DTPs between computational resources in the same node, and to the computational resources of the next step. Each $Node$ component has a Ci_OP component with a PU_i place representing the VM resources in use, and an $IdlePU\_i$ place representing available VM allocated for this step. Transition labelled with time $timePU\_i$ represents the time for a VM to execute the operation in a wavefront $Cell$. The timed $C1\_Cx$ subcomponent represents the DTPs with the next step in the pipeline or with the component that receives the OZi output of the wavefront; and the timed $C1\_ICx$ subcomponent represents DTPs between VM in the same node.



```
{Functional_Wavefront
Interface:
  Transitions: IY1, IY2, IY3, IX1, IX2, IX3, OZ1, OZ2, OZ3
Behaviour:
 Components: C11: Cell(a11), C12:Cell(a12), C13:Cell(a13),
                  C21: Cell(a21), C22:Cell(a22), C23:Cell(a23),
                  C31: Cell(a31), C32:Cell(a32), C33:Cell(a33)
  Links:
     Sync_12: C11(end_HTrans11)+ C12(begin_op12),
     Sync_13: C12(end_HTrans12)+ C123begin_op13),
      IY2:C11(end_VTrans11)+ C21(begin_op21),
      Sync_22:C21(end_HTrans12)+C12(end_VTrans12)+ C22(begin_op22),
      Sync_23:C13(end_VTrans13)+C13(end_VTrans13)+ C23(begin_op23),
       IY3:C12(end_VTrans21)+ C31(begin_op31),
       Sync_32:C31(end_HTrans31)+C22(end_VTrans22)+ C32(begin_op32),
       Sync_33:C32(end_VTrans32)+C23(end_VTrans32)+ C33(begin_op33)
Rename:
  IY1:C11(begin_op11), IY2:C21(begin_op21), IY3(begin_op31),
  IX1:C11(begin_op11), IX2:C12(begin_op12), IX3:C13(begin_op13)
 OZ1: C13(end_HTrans13), OZ2: C21(end_HTrans23), OZ3(end_HTrans33);
}
```

```
{Operational_Wavefront
Interface:
  Transitions: IY1:WF(IY1), IY2:WF(IY2), IY3:WF(IY3), IX1:WF(IX1),
     IX2:WF(IX2), IX3:WF(IX3), OZ1:WF(OZ1), OZ2:WF(OZ2), OZ3:WF(OZ3)
Behaviour:
 Components: WF:Functional_WaveFront; P:Pipeline;
 Links:
 [BPu1,EPu1]:split_Copy(P(PU_1)),  [BPu2,EPu2]: split_Copy(P(PU_2)),
                                   [BPu3,EPu3]: split_Copy(P(PU_3)),
 [BCx1,ECx1]:split_Copy(P(Cx_1)), [BCx2,ECx2]: split_Copy(P(Cx_2)),
                                   [BCx3,ECx3]: split_Copy(P(Cx_3)),
 [BICx1,ECx1]:split_Copy(P(ICx_1)), [BICx2,ECx2]: split_Copy(P(Cx_2)),
                                    [BCx3,ECx3]: split_Copy(P(Cx_3)),
 impl(WF(Op_11), BPu1, EPu1),impl(WF(TR11H), BICx1, EICx1),impl(WF(TR11V), BICx1, EICx1),
 impl(WF(Op_12), BPu1, EPu1),impl(WF(TR12H), BCx1, ECx1),impl(WF(TR12V), BCx1, ECx1),
 impl(WF(Op_21), BPu1, EPu1),impl(WF(TR21H), BCx1, ECx1),impl(WF(TR21V), BCx1, ECx1),
 impl(WF(Op_13), BPu2, EPu2),impl(WF(TR13H), BCx2, ECx2),impl(WF(TR13V), BCx2, ECx2),
 impl(WF(Op_22), BPu2, EPu2),impl(WF(TR22H), BCx2, ECx2),impl(WF(TR22V), BCx2, ECx2),
 impl(WF(Op_31), BPu2, EPu2),impl(WF(TR31H), BCx2, ECx2),impl(WF(TR31V), BCx2, ECx2),
 impl(WF(Op_23), BPu3, EPu3),impl(WF(TR13H), BCx3, ECx3),impl(WF(TR13V), BICx3, EICx3),
 impl(WF(Op_32), BPu3, EPu3),impl(WF(TR22H), BICx3, EICx3),impl(WF(TR22V), BCx2, ECx2),
 impl(WF(Op_33), BPu3, EPu3),impl(WF(TR31H), BCx3, ECx3),impl(WF(TR31V), BCx3, ECx3)
}
```

Fig. 3. Operational 3 × 3 wavefront array executed over a pipeline of cloud VMs with three steps.

CPs in a left to right diagonal can operate concurrently and propagate in wavefronts. For this reason, a balanced deployment of the wavefront *Cell* components over the pipeline steps is executed in the first node $Cell11$, $Cell12$ and $Cell21$ that corresponds from left to right to the first and second diagonal. *Cell*s in the middle diagonal are executed in the second step, and the rest of *Cell*s are executed in the last node. Observe that the horizontal connection of the $Cell11$ is an internal transmission between VM executing in the same node, while its vertical connection is a transmission with a VM in the second node. In the case of the $Cell22$, all its connections are external transmissions, and the same happens with the *Cell*s in the diagonal deployed in the middle step. In this case, the addition of time to the pipeline model will be done by adding a sequence place-transition-place at each node i in parallel with the places PU_i representing the execution of procedures in VMs and places Cx_i and ICx_i representing respectively external and internal transmissions. The textual component specification of the operational wavefront model is presented at the bottom of Figure 3.

The $Operational\_Wafefront$ component refines the $Functional\_wavefront$ with a *Pipeline* by means of the **split-Copy** operator of timed activities represented by places PU_i, Cx_i and ICx_i, and the **impl** operator applied to map activities in the functional model with this timed activities in the pipeline. All mappings are specified in the *Links* of the $Operational\_Wavefront$ component, and a sample of the result of the mapping is illustrated with the arcs connecting the $Cell11$ with places PU_1, Cx_1 and ICx_1. Observe than arcs are labelled with the *id* label. It represents the identifier of the *Cell* instance to differentiate the beginning and ending of activities of cells deployed in the same node. Finally, observe that in the case that only one VM is used at each node, internal communications will be with the same machine.

3 PN models for performance and economical analysis.

The proposed methodology supported by a PN based component language aims at providing different analysis and prediction techniques that allow developer assessing functional and non-functional properties. *Qualitative analysis*, that is desirable/good properties can be formulated and validated with PN models of complex concurrent and distributed systems. *Qualitative analysis* can be conducted by different techniques: 1) The construction of the state space of the model (Reachability analysis) providing a complete knowledge of all is properties if sate explosion does not hamper the use of this technique; 2) Structural techniques that allow to reason about some properties of the model just from the structure of the net using graph theory, linear algebra, convex geometry, or linear programming. The use of this technique is not constrained by the state explosion, however it has a limited decision power (semidecision algorithms) except for syntactical subclasses of PNs. On the other hand, reduction techniques allow the simplification of the nets preserving some properties. It is intimately bound up with the top-down and bottom-up design supported by the proposed methodology.

Quantitative analysis allow performance-oriented interpretations of the model such as throughput, utilization rates, queue lengths, etc. from which is is possible compute *reward functions*. PNs are executable models, therefore extensive simulations may detect errors, which are rare and elusive, and provide us with some performance and reward functions. However simulation cannot guarantee the absence of errors neither identify under with conditions the simulated performance values will be reproduced. The most common analytical model used for the derivation of exact performance measures are stochastic PNs. 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 assume exponential distributions for transition delays, which may not be acceptable to model CPU performance, memory speed, sequential and random I/O, or network bandwidth.

Let us to explore the analysis possibilities of the presented model in Figure 3 considering the operational net model that is isomorphous to the functional flow model. We add a sequence place-transition-place in parallel with the places representing the activities that consume time by the *Cell* located at row i-th, column j-th: OP_ij representing the duration of the computation, and in parallel with places $CijH$ and $CijV$ representing the consumption of time in the transmission of a data element in the corresponding DTPs.

Qualitative analysis based on Structural Analysis. In [15] we shown how the structural analysis could be used to determine the correction of the obtained design. The functional model obtained was a strongly connected marked graph. From this characteristic we recovered some interesting analysis results that we summarize: (1) The net is live, therefore the modeled solution is deadlock-free. (2) The wavefronts propagate in a orderly manner without colliding one into another validating the functional results. (3) The original solution presented in [15] showed the model cannot fully operate concurrently, and a solution based on the structural analysis was suggested to get all CPs working concurrently. The solution is shown in *Cell* Component of Figure 3 that incorporates two $DTPs$. In this way, we can observe that the 4 transitions $Sync\_ij$, $Sync\_i(j+1)$, $Sync\_(i+1)j$ and $Sync\_(i+1)(j+1)$ are covered by an elementary circuit containing four tokens. All these results can be obtained using the structure net resulting from the component based language. A survey of PN tools can be found in [14].

Quantitative analysis based on Stochastic PNs. A stochastic PN is a Timed PN that adopts a probability density function (pdf) for the specification of random delays. Only the use of 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. From the structural analysis it is 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 annuller of the incidence matrix of the net). Therefore, the relative firing frequency vector is **1**, and we have the same mean cycle time for all transitions.

We translated our *Langliers* resulting operational PN to the GreatSPN2.0.2. The tool generates the reachability state with 1392640 states from which a Markov chain is derived. The steady-state numerical analysis compute performance indices (place markings probability distribution and transitions throughputs). The calculated throughput for all transition is 3,99258 data/sec. The result obtained by the GSPN analysis shows a poor throughput with a performance lost of 60% for each processor. If we repeat the analysis with a wavefront of dimension 2x2 the throughput for all transitions is 4.69182 data/sec with a performance lost of 53% for each processor.

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: (1) The explosion of the computational complexity of the analysis algorithm and (2) Only the use of n exponential pdf for the specification of temporal characteristics makes the analysis mathematically tractable.

In [2] is shown that it is possible to compute, in polynomial time, upper and lower bounds for the performance of timed and stochastic marked graph. This bounds are computed with independence of the probability distribution function of random variables that describe the timing of the system. The lower bound for the mean cycle time is obtained by solving the following linear programming problem:

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

Where Γ^{min}, minimum mean cycle time, Y^T is the left annuller of the incidence matrix of the net (P-semiflow), Pre is the pre incidence matrix(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} finding the maximum value of mean cycle times computed by each circuit. In our model Γ^{min} is 100 ms ($1/\lambda$), i.e., a maximum throughout of 10 data/sec. On the other hand, the upper bound for the mean cycle can be computed by:

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

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. In our sample, the maximum liveness noun is 1 for all transitions. Therefore, Γ^{max} is given by the addition of $theta_j$ corresponding to the longer circuit given $\Gamma^{max} = 404$ ms and a minimum throughput of 2.47 data/sec.

Quantitative analysis based on Simulation. The distribution of response time for a cloud modeled as a queue system when service time is not exponential is complex and rely in some approximation. These approximations are very sensitive to the probability distribution of task service times, and they become increasingly inaccurate when the Coefficient of Variation (CoV) increase towards the value of 1 [17, 7, 10]. The difficulty increases when we try to analyze performance of applications implementing advanced data flow abstraction with a high level of concurrency constraints. In our previous quantitative analysis based on stochastic PNs throughput is near the minimum bound. The election of a exponential distributions for task service times is not adequate and results in poor and not realistic performance results.

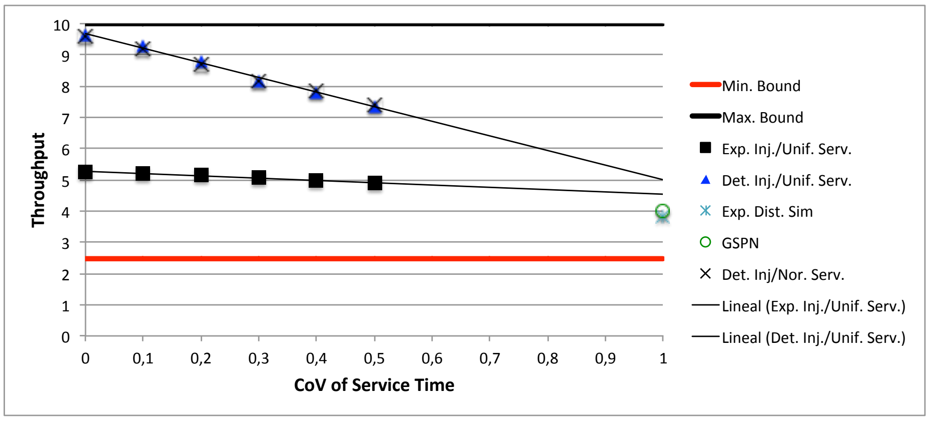


Fig. 4. Simulated throughputs with different Coefficient of variation of the service time.

Once known the throughput bounds, and the fact that CoV can have a high impact on performance, we conducted simulations to evaluate its impact. Figure 4 shows the results of different `Renew` simulations with mean services and injection times of 10 data/sec. The Figure also shows the point *GSPN* that represents the computed performance by the GSPN tool[2], and the *Exp. Dist. Sim.* shows the point with the same scenario obtained by simulation, i.e, assuming both service delivery times and and interarrival times are exponential. 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 [17] between the time of the faster node and the time of the slower node we conduct different simulations in `Renew`. *Exp.Inj./Unif. Serv.* shows the impact of CoV on performances assuming an exponential distribution in injections, and uniform distributions of service delivery times. *Det.Inj./Unif. Serv.* shows the same simulations with a deterministic injection time. Finally, *Det.Inj./Nor. Serv.* shows that simulations with normal distribution of service delivery times provide the same results than uniform distribution. These results shows that some mechanism is required

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

to regulate injections rates because assuming exponential distribution of interarrival times result in performance near 50%. On the other hand with deterministic injection, it is clear the impact of CoV on performance.

In our previous paper [15] we computed the costs for the processing of streams of length $k = n$, assuming the cost of the time unit per CPU, p: $Cost_{functional} = max\{\alpha, \beta, \gamma\} * n * p * 9$ from the functional level analysis. In this paper we have shown this costs will be proportional to the service time Coefficient of Variation CoV of the cloud platform $Cost_{operational} = k * CoV * Cost_{functional}$ assuming a mechanism to regulate the data injection to the wavefront is provided and approximately $2 * Cost_{functional}$ in case this mechanism is not provided. Depending of the CoV value of the platform, the developer can evaluate the costs of alternative operational models such as the pipeline version presented in Figure 3.

4 Conclusions and Future work

The SG scenario is a prototypical scenario for the Big data where a trusthworthy design of real time streamming applications is essential. Although the design of real time streaming applications is recognised as a complex task in the Big Data context [9], neither of proposed development languages and frameworks allow developers a component based reasoning that guides design decisions at all phases of the life cycle, and to address functional and non-functional requirements together with the specification of the execution infrastructure and the involved resources. This work has presented a component PN based specification language that allows developers to specify functional and operational levels at different levels of abstraction and reason with the models. A complex wavefront operational model for the Matrix-Vector Multiplication problem in streaming fashion has been presented and the way the developer can reason with the model has been illustrated. A detailed analysis of advanced data flow abstraction, or a composition of them, allow the developer reason about the functional correctness of the proposed solutions, and guide the simulations to evaluate different proposals to obtain the best performances as the base to analyze alternative solution costs.

Future work will consider the modelling of parallel and architectural patterns, and the deployment of specifications directly on cloud platforms. The characterization of the models and a limited set of building blocks will allow us to know the limits of the formal analyses and develop tools to validate the models.

Acknowledgments This work was supported 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 id TIN2013-40809-R.

References

1. Aman, S., Simmhan, Y., Prasanna, V.: Energy management systems: state of the art and emerging trends. Communications Magazine, IEEE 51(1), 114–119 (2013)

2. Campos, J., Chiola, G., Colom, J.M., Silva, M.: Properties and performance bounds for timed marked graphs. Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on 39(5), 386–401 (1992)
3. Elizondo, D., Gardner, R., Leon, R.: Power and Energy Society General Meeting, 2012 ieee. In: Synchrophasor technology: The boom of investments and information flow from North America to Latin America. pp. 1–6 (July 2012)
4. Fang, X., Misra, S., Xue, G., Yang, D.: Managing smart grid information in the cloud: opportunities, model, and applications. Network, IEEE 26(4), 32–38 (2012)
5. Government, U.: NIST Framework and Roadmap for Smart Grid Interoperability Standards, Release 3.0. General Books (Sep 2014)
6. Jensen, K., Rozenberg, G. (eds.): High-level Petri nets: theory and application. Springer-Verlag, London, UK (1991)
7. Khazaei, H., Misic, J., Misic, V.: Performance analysis of cloud computing centers using m/g/m/m+r queuing systems. Parallel and Distributed Systems, IEEE Transactions on 23(5), 936–943 (2012)
8. Maheshwari, K., Lim, M., Wang, L., Birman, K., van Renesse, R.: Toward a reliable, secure and fault tolerant smart grid state estimation in the cloud. In: IEEE PES Innovative Smart Grid Technologies Conference, ISGT 2013. pp. 1–6 (2013)
9. Marz, N., Warren, J.: Big Data: Principles and best practices of scalable realtime data systems. Manning Publications Co. (2015)
10. O'Loughlin, J., Gillam, L.: 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. LNCS, vol. 8914, pp. 133–145 (2014)
11. Pautasso, C., Alonso, G.: 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. pp. 1–10 (2006)
12. Seceleanu, C.C., Crnkovic, I.: Component models for reasoning. IEEE Computer 46(11), 40–47 (2013)
13. Simmhan, Y., Kumbhare, A.G.: Floe: A continuous dataflow framework for dynamic cloud applications. CoRR abs/1406.5977 (2014)
14. Thong, W., Ameedeen, M.: A survey of Petri Net Tools. In: Sulaiman, H.A., Othman, M.A., Othman, M.F.I., Rahim, Y.A., Pee, N.C. (eds.) Advanced Computer and Communication Engineering Technology, Lecture Notes in Electrical Engineering, vol. 315, pp. 537–551. Springer International Publishing (2015)
15. Tolosana-Calasanz, R., Bañares, J.Á., Colom, J.M.: 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. LNCS, vol. 8914, pp. 189–205 (2014)
16. Tolosana-Calasanz, R., Bañares, J.Á., Rana, O.F., Pham, C., Xydas, E., Marmaras, C.E., Papadopoulos, P., Cipcigan, L.: Enforcing quality of service on opennebula-based shared clouds. In: CCGRID'14 Workshop on Data-intensive Process Management in Large-Scale Sensor Systems, DPMSS'14, Chicago, IL, USA, May 26-29, 2014. pp. 651–659. IEEE (2014)
17. Yeo, S., Lee, H.H.: Using mathematical modeling in provisioning a heterogeneous cloud computing environment. Computer 44(8), 55–62 (2011)
18. Yu, L., Moretti, C., Thrasher, A., Emrich, S.J., Judd, K., Thain, D.: Harnessing parallelism in multicore clusters with the All-Pairs, Wavefront, and Makeflow abstractions. Cluster Computing 13(3), 243–256 (2010)