A BPMN Extension to Enable the Explicit Modeling of Task Resources

Paolo Bocciarelli, Andrea D'Ambrogio, Andrea Giglio and Emiliano Paglia
Dept. of Enterprise Engineering, University of Rome "TorVergata"
Viale del Politecnico 1, 00133 Rome, Italy
{paolo.bocciarelli,dambro,andrea.giglio,emiliano.paglia@uniroma2.it}

Copyright © held by the authors.

***Abstract*—Business Process Management (BPM) is an holistic approach for describing, analyzing, managing, improving and executing large enterprise business processes, which can be seen as a number of related tasks that have to be executed in order to reach well-defined objectives.**

This paper introduces an approach to the specification and management of the resources that are associated to each BP task and that are required to support the execution of processes and related complex systems. This work provides a notation to define, at design time, the real entities which will perform the activities at different levels of abstraction, also specifying their non-functional properties. This approach makes use of composition patterns in order to model complex resources into tree structures allowing to represent part-whole hierarchies in a flexible and scalable manner. To accomplish these modeling objectives, the Business Process Modeling Notation (BPMN) has been extended. BPMN is the de-facto standard for the high-level description of business processes, but does not support the characterization of the business process or related resources in terms of non-functional properties.

The proposed extension Performability-enabled BPMN (PyBPMN) is based on an approach that exploits principles and standards introduced by the Model Driven Architecture (MDA), thus obtaining considerable advantages in terms of easy customization and improved automation. PyBPMN has been introduced in our previous contributions for performance and reliability analysis, and has been extended in this work for addressing also complex resource modeling and management.

The paper also presents some examples of the proposed extension usage to show how it enables the description of complex resources and their non-functional properties.

***Index Terms*—BPMN, business process management, MDA, resources management, process modeling.**

I. Introduction

Modern society is increasingly dependent on complex collaborations involving multiple and independent systems, ranging from large organizations or bureaucracies to individuals, including software or hardware systems and involving manual or automated tasks eventually executed by heterogeneous resources. Thus, modeling Business Processes (BPs) that enable the analysis of such complex collaborations is a challenging issue for large organizations or governmental institutions, both to improve systems functioning and also to provide a valuable way to better understand the role of individual components of such systems.

To this extent, the overall structure of such complex systems has to be considered, making clear how the role played by resources is crucial. However, although the general meaning of the term *resource* could be similar among different domains [22], as it emerges from many studies in different research areas, obtaining a unified approach for modeling such a pervasive entity is a non-trivial issue. This is due to the fact that the nature of the various researches and the inherent complexity of the different domains often lead to the identification of different characteristics and constraints governing the resources behaviour [22].

As aforementioned, BPs can be very complex as they may involve different systems and may require a large amount of operational resources. Modern BP management approaches strongly recommend the adoption of techniques to concretely support the analysis of BP behaviour (e.g, to verify the conformity of the BP functional or non-functional characteristics to initial requirements, in order to assess whether or not objectives are met and plan appropriate recovery actions when needed). In this context, simulation-based techniques have proven to be effective for analyzing and validating the performance of a BP since the early lifecycle phases, when the BP is commonly specified in Business Process Model & Notation (BPMN) [20].

In previous works we have introduced approaches and tools to improve and to make easier the simulation-based BP performance analysis. In this respect, we have presented: i) *Performability-enabled BPMN* (*PyBPMN*), a BPMN extension that is used to annotate performability-oriented properties (i.e., properties that address both performance and reliability) onto BPMN models [7], ii) *eBPMN*, a Java-based platform to execute BP simulations [9], [10], and iii) a model-driven framework to generate eBPMN code from PyBPMN models by use of automated model transformations [5], [6], [8].

The main objective of this paper is to broaden the scope of the aforementioned contributions by introducing a further BPMN extension which specifically addresses resource management. Such an extension is developed as an enhancement of the PyBPMN metamodel, allowing to deal with the modeling and specification of resources in the generic context of heterogeneous systems and processes.

More specifically, the work we describe in this paper addresses the performance- and reliability-oriented BP characterization by focusing on a more precise understanding and characterization of the resource entities.

Furthermore, the contribution in this paper goes beyond the previous efforts and proposes an holistic approach to resource

specification and management by introducing facilities to enable the dynamic allocation of resources to process tasks (e.g, in case of resource failures and repairs). This dynamic and flexible association of resources (or sets of resources) to BP tasks offers a significant degree of customizability since it allows the use of different policies (e.g, to specify the behaviour of backup resources) and the specification of complex (i.e., composite) systems of resources.

Moreover, the proposed extension is implemented according to a profiling-based mechanism, which allows to obtain significant levels of flexibility and customizability, both in case of BP analysis carried out by use of both analytical and simulation approaches.

Finally, the paper presents some application examples that show how the proposed approach can be effectively used.

The reminder of the paper is structured as follows: Section 2 reviews related work, Section 3 gives an overview of the previous efforts regarding PyBPMN, Section 4 describes the new metaclasses constituting the proposed resource management extension, Section 5 gives an overall description of the improved PyBPMN metamodel, while Section 6 discusses a set of usage examples. Finally, Section 7 gives concluding remarks.

II. Related Work

Multiple aspects of resource management have been investigated from different perspectives and various fields of research. The aim of this section is to provide a concise outline of that work.

In the workflow and process languages research areas various approaches for resource specification are presented, such as [28], [16]. Among the workflow and process languages that deal with resource management, APPL/A [26], Process Weaver [11], BPEL4WS [16], APEL [12], and MVP-L [24] are the most representative.

Despite their significance, the aforementioned languages present a limited resource specification effectiveness. Moreover, they provide restricted support for critical issues as describing resource relationship, resource allocation or request specification. More specifically, BPEL4WS only deals with resources defined as web services, and BPEL4People [14] focuses on human participation in web services [22].

A significant approach is presented in [25], where the authors introduce the idea of *workflow resource patterns*. However, they use a restrictive concept of resource, in which a step can have only a single resource, without the capability of specifying any additional resources. Unlike our approach, they did not address the potential dynamic nature of complex and composite resources.

Furthermore, differently from the aforementioned approaches, our work aims at providing a lightweight extension realized in full compliance with the BPMN metamodel. The PyBPMN extension in fact, being engineered following MDA [17] principles and standards, enables model transformations from annotated BPMN models to executable languages.

In [15], an extension of BPMN to enable the specification of performance properties is presented. In such contribution, the authors explain that BPMN does not allow to specify non-functional requirements and thus they introduce a metamodel for evaluating non-functional aspects of BPs, as well as the related extension.

The aforementioned contributions are limited to the introduction of BPMN extensions, in order to enhance its expressive capabilities. This contribution instead, proposes the adoption of a BPMN extension as part of a model-driven approach that seamlessly enables the use of automated model transformations.

In [29], the authors analyze BPMN and its ability to represent the resources utilization and to support business process simulation. Such contribution states that BPMN needs an extension in order to provide the ability to specify resources and eventually support BP simulation enactment. As a solution to this issue, the authors proposed the use of BPSim [31], which is a standard that provides a framework for structural and capacity analysis of BP models specified by use of BPMN or XPDL (XML Process Definition Language) [30].

In this respect, this paper approach is quite different, since differently from BPSim, the proposed PyBPMN extension has been implemented as a BPMN metamodel extension, in order to be suitable for various operational contexts.

III. Performability-enabled BPMN

In order to provide a clear understanding of the proposed approach, the following subsection motivates and outlines the previous contributions upon which this work places its basis.

More specifically, this section provides a description of the Performability-enabled Business Process Modeling Notation (PyBPMN) metamodel. Such metamodel is an extension of the standard BPMN metamodel [20] and has been designed and implemented exploiting standards, principles and methods suggested by MDA [17], i.e., metamodeling techniques allowing formal abstract definition of models as well as transformations between such models.

To this purpose, MOF [18] constitutes the OMG standard enabling the specification, construction, management of technology neutral metamodels, taking advantage of its abstract language and its framework.

As specified in the MOF abstraction architecture [18], a level M1 model is an instance of a MOF metamodel which is specified at level M2. The latter, is in turn an instance of the comprehensive MOF meta-metamodel at layer 3 which is specified in [18]. Using the XMI specification [19], models defined at level M1 as well as metamodels defined at level M2, can be seamlessly serialized into XML documents and XML schemas, respectively. In this context, the PyBPMN extension process exploits MOF and XMI, implementing a metamodel at M2-level which extends the BPMN metamodel at M2-level and specifying a mapping between such metamodels in terms of their elements.

The model-to-model transformation enables the automatic generation of a PyBPMN model at level M1, from the

BPMN model at level M1. The starting BPMN model is annotated with *TextAnnotation* elements according to a specific syntax. Such annotation allows the automatic generation of the PyBPMN model applying the model-to-model transformation. The aforementioned syntax is described in section VI.

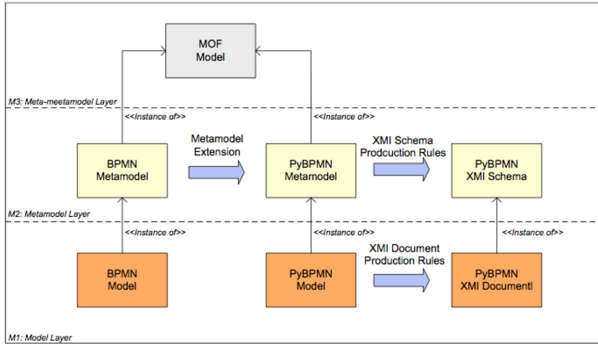

Fig. 1. PyBPMN extension process.

Figure 1 depicts the PyBPMN extension process. PyBPMN (Performability-enabled BPMN) is a lightweight BPMN extension that deals with the specification of performance and reliability properties of business processes. The work for this paper places its basis on previous contributions [7], [9], [10]. With the aim of enriching BPMN models with non-functional properties, the Modeling and Analysis of Real-Time Embedded systems (MARTE) [21] UML profile has been originally taken into account.

Moreover, the method in [1], [3] for MARTE UML profile usage has been analyzed, in order to extract PyBPMN metaclasses and syntax for the annotation of BPMN elements.

As suggested in [1], the concepts of fault, error and failure constitute the basis of the PyBPMN extension. In the following a brief recall of such concepts is given.

In order to be in a correct state, a business process execution has to be fully compliant to its requirements and specifications. Differently, if the business process execution differs from the correct state, it means that a **failure** occurs and consequently the process reaches an incorrect state. Thus, in an incorrect state, the service which is delivered is not the expected one.

A process execution can be seen as a sequence of observable states, i.e., external states [1]. In this scheme, a **failure** occurs when at least an external state deviates from its correct state. Such a deviation is called an **error**. The cause of an error is called a **fault**.

Faults can be of different nature, and can be categorized as follows:

- human-made faults, effects of human action, typically omissions or malicious actions;
- natural faults, not related to human actions, but due to natural phenomena;
- interaction faults, related to different component interfaces misalignment or incompatibilities;
- physical faults, related to hardware failures;
- development faults, typically programming inaccuracies, introduced in the development phase.

A fault is said to be active only if it determines an error. Furthermore, an error can be observed only if it generates a failure. In the case of a business process execution, a resource failure prevents the process to perform its activities.

Given the discussion above, it is worth noting that the concepts of faults, errors and failures as well as the relationship among them, imply the following notions:

- a business process is a set of collaborating elements. An element can involve a single resource or more resources to perform its job;
- an execution platform is defined by a finite set of resources. The execution platform represents the infrastructure that allows the actual business process execution;
- if the execution platform is capable of providing all the resources required by a process element, such element is said to be in a correct state; differently, if at least one of the required resources is not available, the element is said to be in an incorrect state;
- a business process is said to be in a correct state if and only if all its elements are in a correct state.

The revised PyBPMN metamodel deals with four main areas of non-functional properties:

- **workload definition**: responsible for modeling the workload related to the whole business process or to the tasks associated to the process (i.e., the execution of single activities). The key metaclasses for workload specification are the *GaWorkloadEvent* metaclass and the *ArrivalPattern* metaclass;
- **performance properties definition**: responsible for specifying the performance properties associated to both the process and the single task. The most common performance properties are the service demand (service time), the time spent to accomplish the demand (response time), and the throughput. The key metaclasses for specifying performance properties are of type *PaQualification*, such as *PaResponse* and *PaService*;
- **reliability properties definition**: responsible for modeling the reliability related properties of the resources involved in a process or associated with a task. The most common reliability properties are the occurrence rate of the failure, the occurrence distribution of the failure, the mean time to failure (MTTF) and the mean time to repair (MTTR). In order to specify such properties, a metaclass of type *DaQualification* is used, such as *DaFault* and *DaFailure*;
- **resource management**: responsible for specifying the actual resource which is used to execute an activity. Constituting the main contribution of this paper, the purpose of the resource management section of the PyBPMN metamodel is threefold, since it allows to define non functional properties for atomic resources (*PyPerformer*), as long as groups of resources (*PySubsystem*), and also alternative resources (*PyBroker*).

The aforementioned sets of PyBPMN metaclasses are not

exhaustive. In fact, the proposed BPMN extension can be fully automated and the collection of new metaclasses enriching the original BPMN metamodel can be customized in order to address specific needs, with a negligible effort.

As for workload, performance, and reliability perspectives, the PyBPMN metamodel has been already described in our previous contributions. The reader who is interested in a more precise discussion is sent to [7], [8].

From the resource perspective, the next section explains in more detail the PyBPMN metamodel enhancement for resources modeling and characterization, which constitutes the main innovative contribution presented in this paper. For the sake of simplicity, only the metaclasses and attributes related to that perspective are shown. Attributes which have already been described will be omitted.

Next, in Section V, an overall description of the revised PyBPMN metamodel will be given to sum up the different perspectives (i.e., preexisting metaclasses and latest enhancements related to resource modeling).

IV. Resource Characterization

As for the workload, performance and reliability extensions, new metaclasses have been introduced with the aim of specifying non functional properties of process elements. Differently, the resource extension introduces new key notions to the process execution.

More specifically, the standard BPMN resource definition is augmented with the resource definition of PyBPMN.

The standard BPMN resource definition is abstract, because it only defines the resource notion without specifying its details (i.e., the real resource which is executing the activity). Differently, the PyBPMN extension enables to define, at design time, the actual entities which will execute the activities also specifying their non-functional properties.

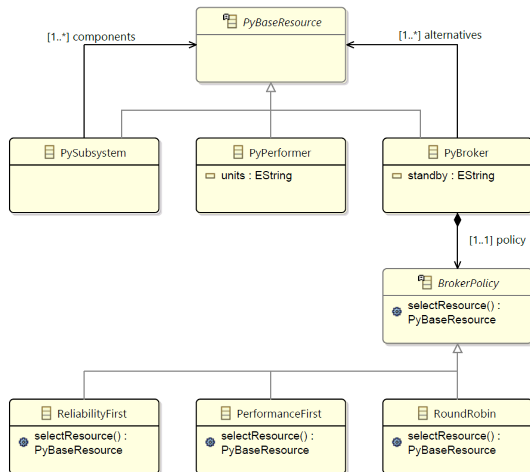

Fig. 2. PyBPMN classes for resources characterization.

The idea is that a PyBPMN resource is capable of modeling a wide range of different entities, such as an employee, an hardware equipment, a functional division of an organization, a web service, an autonomous system, or any other entity which can be modeled as an element able to perform a service request. In this perspective, a PyBPMN resource can be modeled at different levels of abstraction during the design phase. The corresponding level of abstraction depends on both the amount of information available to model the resource and the analysts point of view. In fact, the aim of the proposed extension is to provide a unifying way of modeling resources, in a general context, rather than for specific domains.

The following is a brief description of the metaclasses introduced by PyBPMN to describe the resources used by the process to perform its activities, as depicted in Figure 2:

- *PyBaseResource*: this is an abstract metaclass representing a theoretical resource. It can be specialized as follows:
- *PyPerformer*: this metaclass represents a real work performer that accomplishes the service requests. It has one attribute:
  - *units*: represents the number of concurrent working units which constitute the performer.
- *PyBroker*: this metaclass represents a resource manager which enables the selection of a resource from a set of candidate resources. The chosen resource becomes responsible for the execution of the service request. More specifically, the *PyBroker* acts as dispatcher which forwards the service request to only one resource at a time (i.e., the selected resource). The resource selection is also dynamic, in case of resource failure or resource repair. On the first hand, when the current resource is not available anymore (e.g. due to a failure), and on the other hand, when a more suitable resource is available (i.e., a better resource has been repaired yet). In both cases the selection process is repeated and a different resource can be selected as the currently used resource. The *PyBroker* metaclass has the following attributes:
  - *alternatives*: represents the collection of resources which can be chosen to accomplish the service request;
  - *policy*: represents the strategy used for selecting the current resource. A number of selection strategies can be adopted. Strategies differ as regards the implementation of the selectResource() method. In this respect, the PyBPMN metamodel provides the following concrete metaclasses, which inherit from the abstract metaclass *BrokerPolicy*:
    - *RoundRobin*: in this strategy the next resource is selected with the aim of ensuring a fair use of all the available resources;
    - *PerformanceFirst*: in this strategy the next resource is selected in order to maximize performance;
    - *ReliabilityFirst*: in this strategy the next resource is selected in order to maximize reliability;
  - *standby*: defines the management strategy used for the unselected resource, i.e., the nature of the standby redundancy, as better illustrated later on.

- *PySubsystem*: this metaclass represents a collection of resources in the case they are all required to satisfy the service request. In this case, the *PySubsystem* sequentially dispatch the request to all the resources composing the subsystem. This is a different behaviour compared to the *PyBroker* approach, where the request is sent only to the currently selected resource. In fact, the *PySubsystem* can be understood as a complex resource whose non functional properties are defined in terms of its constituting components. The *PySubsystem* metaclass has the following attribute:
  - *components*: the set of resources that compose the subsystem.

As for the implementation perspective, the PyBPMN resource extension has been modeled conforming to the well-known *Composite* design pattern defined by [13].

The strength of this approach is the capability to compose complex elements into tree structures allowing to represent part-whole hierarchies. In fact, the proposed PyBPMN extension enables the specification of a resource in two different ways. On the first hand, a single and simple definition (*PyPerformer*) can be adopted for the resource, which can be actually used to accomplish a service request. On the other hand, more complex structures (*PyBroker* and *PySubsystem*) can be used, thus reaching a higher level of detail. With the latter approach, it is feasible to specify complex patterns of resource usage:

- *PyBroker* defines a set of resources that are all capable of performing the required activity (i.e., the service request). According to a selection policy the actual resource is chosen, and the remaining resources are kept in one of the standby states, as defined in the system reliability literature [2], [4], [27], [23]:
  - *hot standby*: in this case there is the choice of preference for performance rather than reliability. More specifically, in the hot standby redundancy policy the unselected resources (i.e., redundant or backup units) are all up and running as the selected one. This implies that also the unselected resources are aging during the process execution, even if they are not involved. As a result, the switching mechanism is efficient, but the redundant resources can fail even in reserve state, since they are all ready to go;
  - *cold standby*: in this case, reliability is preferred instead of performance. In more detail, in this policy the redundant resources are not active while the selected resource is in service. This implies that the failure rate in reserve state is assumed to be zero, since the redundant resources do not age while they are not operating. It is clear how in this case the switching mechanism is less efficient than the hot standby, since the newly selected resource needs to be started up. The strength of this approach is that the redundant resources can not fail in reserve state, ensuring an higher degree of reliability;
  - *warm standby*: this policy implements an hybrid mechanism and can thus be seen as a particular case of the hot standby approach. In this case the unselected resources are all powered on but they show a lower failure probability. This is due to the fact that they have no load, since the service requests are forwarded only to the selected resource.

  As aforementioned, the *alternatives* attribute is of type *PyBaseResource*, thus implying that each candidate resource can be a *PyPerformer*, a *PySubsystem* or even another *PyBroker*. Since a *PyBroker* has its own alternative resources, it can manage alternative resources in terms of a hierarchical tree structure. Moreover, a *PyBroker* fails only when all its alternative resources are failed.
- *PySubsystem* provides a way for defining a collection of resources which all together constitute a more complex resource. This approach enables to describe a subsystem, from both the performance and the reliability perspective, specifying such properties for each elementary component constituting it. Moreover, the component attribute is of type *PyBaseResource*, and subsequently each constituting resource can be a *PyPerformer*, a *PyBroker* or even another *PySubsystem* which in turn has its own sub components. As a consequence, similarly to the broker approach, a *PySubsystem* can regulate a hierarchical tree structure of basic components and other complex subsystems. As a difference from the *PyBroker*, it is worth noting how a *PySubsystem* fails when at least one the its components is failed.

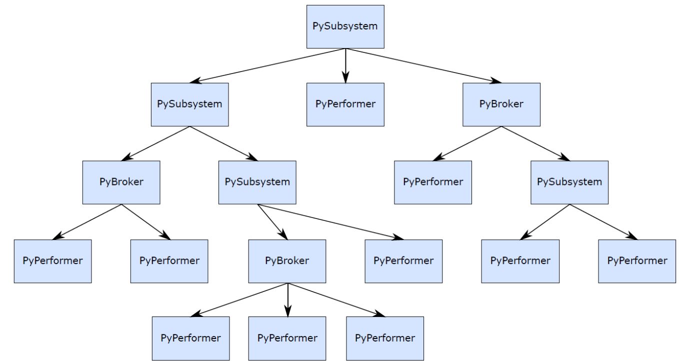

Fig. 3. A very complex resource definition allowed by the composite design pattern.

Taking advantage on the aforementioned composite design pattern, one can define very complex resource structures, as illustrated in Figure 3. One can easily understand that every leaf of the tree structure must be a *PyPerfomer* which actually accomplishes the service request (i.e., executes the activity).

V. Revised PyBPMN metamodel description

In the previous section the fundamental elements and the resource characterization enrichment of the PyBPMN

metamodel have been described. In this section, a description of such revised metamodel is given, focusing on how it can be exploited for the specification of both the non-functional properties of BPMN process elements and the non-functional properties of the resources used by the process to execute its activities. The revised PyBPMN metamodel, as shown in

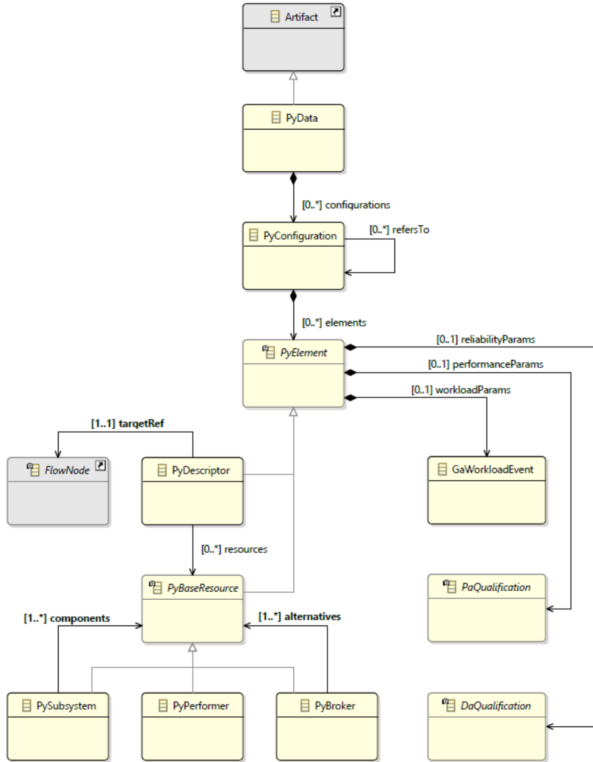

Fig. 4. Overall description of PyBPMN metamodel.

Figure 4, is based on the following fundamental metaclasses:

- *PyData*: this metaclass acts as a top-level container for all elements that can be defined for the process. With the aim of separating the standard elements from the extended elements, this approach enables to arrange PyBPMN extension elements under a single node within the BPMN model. To this purpose, the PyData metaclass derives from the standard Artifacts metaclass, and occurs as an element into the artifacts attribute of the process. Furthermore, in order to augment a model with a complete set of data over the business process, the PyData element enables to gather both the simulation data and the execution data in the same model. This approach enables business analysts to master the business process, evaluating both collected data and business process behavior, and eventually taking appropriate actions in the redesign phase. A single attribute characterizes the PyData root element:
  - *configurations*: a set of PyConfiguration elements. Each of these elements defines a particular configuration of the execution platform;
- *PyConfiguration*: a set of non functional parameters related to a configuration. Such parameters can be originated in different ways: required in the design phase, estimated through simulation or measured during business process execution. In the case of values obtained from design or simulation, each configuration can be assumed as a candidate execution platform for process execution. Differently, in case of values obtained from a real process execution, they are acquired from a process instance. The PyConfiguration metaclass owns the following attributes:
  - *refersTo*: this attribute is responsible for creating relations among different configurations (i.e., to relate the actual measured values to those predicted in the design phase);
  - *elements*: a set of PyConfiguration elements, used to specify the business process non functional parameters;
- *PyElement*: this metaclass is responsible for the workload, performance and reliability characterization of the business process, since it contains the related non-functional properties. *PyElement* is defined as an abstract metaclass, and therefore can be specialized in *PyDescriptor* and *PyBaseResource*, the latter described in section V. The following attributes are owned by the *PyElement* metaclass and therefore by the aforementioned derived metaclasses:
  - *workloadParams*: responsible for specifying the workload characterization exploiting the *GaWorkloadEvent* introduced in [7], [8];
  - *performanceParams*: responsible for specifying the performance characterization as introduced in [8], [7]. It exploits a metaclass derived from the abstract metaclass *PaQualification* (i.e., the *PaService* metaclass or the *PaResponse* metaclass);
  - *reliabilityParams*: responsible for specifying the reliability characterization as introduced in [7], [8]. It exploits a metaclass derived from the abstract metaclass *DaQualification* (i.e., the *DaFailure* metaclass or the *DaFault* metaclass.

With the purpose of providing a better understanding of a *PyElement* data, the PyBPMN metamodel exploits the *source* attribute defined for basic data type in compliance with the UML MARTE profile [21].

As illustrated in [5], the *source* attribute can take different values, each one having a different meaning: *req* (i.e., required), *est* (i.e., estimated), *calc* (i.e., calculated) and *meas* (i.e., measured). A required parameter is utilized to specify a value which has to be satisfied by the system, as in the case of a constraint or a requirement. An estimated parameter is utilized to specify a prediction for a parameter value, as typically happens in the case of the output produced by a simulation. A calculated parameter is utilized to specify a value which is a result of a combination of other parameter values. Finally, a

measured parameter is utilized to specify the value of data obtained during the execution of a business process.

- *PyDescriptor*: this metaclass is responsible for putting in relation non functional properties, which are defined exploiting the aforementioned PyBPMN extension metaclasses, with standard BPMN elements of type *FlowNode*. This metaclass owns the following attributes:
  - *resources*: this attribute represents the collection of resources of the execution platform exploited by the BPMN element;
  - *targetRef*: this attribute represents the unique identifier of the BPMN *FlowNode* element. The referred element will be augmented with resources and non functional properties.

  A *PyDescriptor* and a BPMN element can be associated in two ways:
  - in the first case, the non functional properties for the BPMN element are owned by the related *PyDescriptor* element, which is of kind *PyElement*. In this way the BPMN element can be characterized at a high level of abstraction with workload, performance and reliability properties, leaving the details related to the execution platform in background. This approach could be suitable for a coarse analysis;
  - differently, adopting the second approach, one can explicitly define the execution platform through the resources metaclasses, such as *PyPerformer*, *PyBroker* or *PySubsystem*. After defining the execution platform, the *PyDescriptor* is allowed to realize the association between one or more resources to the BPMN element. In this case several benefits are obtained:
    * since the execution platform is clearly defined, the business process behavior can be more precisely analyzed;
    * since the *PyDescriptor* can reference different resources, a sequential usage of a set of resource can be easily modeled;
    * several *PyDescriptor* elements can refer the same resource. The case in which the same resource can be used by more than one BPMN elements can thus be modeled. In such a scenario, a failure of such resource will have an impact on all the associated BPMN elements.

VI. Extension Usage

This Section briefly outlines the typical usage of PyBPMN and the proposed extension for resource management presented in Section IV. As introduced in Section II, the PyBPMN model can be automatically obtained applying a model-to-model transformation to a standard BPMN model.

While the elements used to define the flow of tasks and activities are natively part of the BPMN meta-model, the non-functional aspects along with resource management and characterization attributes are included as *TextAnnotation* elements compliant with a specific syntax, according to the proposed extension meta-model.

Figure 5 gives a graphic representation of the annotation step, as seen through a BPMN visual editor. A *TextAnnotation*

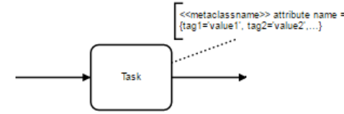

Fig. 5. PyBPMN extension by use of text annotations.

including some PyBPMN specification is called a *PyAnnotation*. Such annotations conform to the following general syntax:

```
<<metaclass name>> attribute name =
    {tag1='value1', tag2='value2', ...}
```

In order to clarify the specification of PyBPMN elements, the rest of this section gives a short application example.

Let us consider a business process for managing the delivery of a commercial proposal, which includes a task (`SendHardCopy` that sends hard copy contracts to the legal office for internal review. In this respect, the task requires a resource, named *PrinterA*, to both scan the contract and send the resulting file. Moreover, for reliability purposes, a backup solution consisting of a scanner (e.g., *ScannerB*) and a file server (*FileServerB*) is adopted as a backup resource for the `SendHardCopy` task. In order to represent such a scenario, several PyBPMN text annotations are required. First of all, the following three resources are defined (for the sake of conciseness, the serviceTime, MTTF and MTTR attributes have been omitted from the specification of resources *ScannerB* and *FileServerB*):

```
<<PyPerformer>> {
    name=PrinterA,
    serviceTime=(value=12, unit=s),
    MTTF=(value=3, unit=year),
    MTTR=(value=6, unit=hour)}
<<PyPerformer>> {
    name=ScannerB, ...}
<<PyPerformer>> {
    name=FileServerB, ...}
```

In addition, the subsystem composed of resources *ScannerB* and *FileServerB* is specified as follows:

```
<<PySubsystem>> {
    name=SubSystem-B,
    components=(ScannerB, FileServerB)}
```

The broker that manages the access to the different resources is then introduced. Let us suppose that the cold standby policy is adopted in order to maximize the overall reliability. Then the broker is specified as follows:

```
<<PyBroker>> {
    name=SendHardCopy,
    standby=Cold,
    alternatives=(PrinterA, SubSystem-B)}
```

Finally, the association between the BPMN `SendHardCopy` task and the broker is carried out through a *PyDescriptor* element, as shown in the following listing:

```
<<PyDescriptor>> {
    name=SendHardCopy_descriptor,
    resources=(SendHardCopy)}
```

VII. Conclusion

This paper has introduced a metamodeling approach to address the problem of resource management in dynamic and complex processes.

More specifically, a BPMN metamodel extension has been presented for managing resources involved in the execution of a BP. Since the concept of resource can assume a wide range of meanings (resources can be humans, software or hardware systems, etc.) the proposed approach aims to provide an efficient support to model complex resources exploiting composition patterns.

The aforementioned approach is domain-independent and enables a flexible and scalable way to model subsystems of resources, possibly in a hierarchical structure.

Moreover, in the proposed extension a BP task can have a set of different resources (each one characterized by its own non-functional parameters) associated to it, thus allowing, during BP execution, the use of different resource selection and allocation policies.

This work is focused on resources characterization and configures itself as further enhancement of PyBPMN, a lightweight BPMN extension to enable the non-functional characterization of BPs. PyBPMN and its resource management extension presented in this paper are part of a model-driven framework that enables simulation-based BP performance and reliability analysis [6], [9], [10].

References

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[2] E. Bauer, X. Zhang, and D.A. Kimber. *Practical System Reliability*. Wiley, 2009.

[3] Simona Bernardi, Jos Merseguer, and Dorina C. Petriu. Adding dependability analysis capabilities to the marte profile. In *Proc. of 11 t h International Conference on Model Driven Engineering Languages and Systems (MoDELS), volume 5301 of Lecture Notes in Computer Sciences*, pages 736–750. Springer, 2008.

[4] A. Birolini. *Reliability Engineering: Theory and Practice*. Engineering online library. Springer Berlin Heidelberg, 2003.

[5] P. Bocciarelli, A. D'Ambrogio, A. Giglio, and E. Paglia. Towards performance-oriented perfective evolution of BPMN models. In *Proceedings of the Symposium on Theory of Modeling and Simulation, part of the SCS SpringSim 2016 conference*, Pasadena, CA, USA, 2016.

[6] P. Bocciarelli, A. D'Ambrogio, A. Giglio, E. Paglia, and D. Gianni. A Transformation Approach to Enact the Design-Time Simulation of BPMN Models. In *2014 IEEE 23rd International WETICE Conference*, pages 199–204. IEEE, jun 2014.

[7] Paolo Bocciarelli and Andrea D'Ambrogio. A BPMN Extension for Modeling Non Functional Properties of Business Processes. In *Proceedings of the Symposium on Theory of Modeling and Simulation*, DEVS-TMS '11, Boston, MA, USA, 2011.

[8] Paolo Bocciarelli, Andrea D'Ambrogio, Andrea Giglio, and Emiliano Paglia. Simulation-based performance and reliability analysis of business processes. In *Proceedings of the Winter Simulation Conference 2014*, volume 2015-Janua, pages 3012–3023. IEEE, dec 2014.

[9] Paolo Bocciarelli, Andrea D'Ambrogio, Andrea Giglio, Emiliano Paglia, and Daniele Gianni. Empowering business process simulation through automated model transformations. In *Simulation Series*, volume 46, pages 278–286. The Society for Modeling and Simulation International, 2014.

[10] Paolo Bocciarelli, Andrea D'Ambrogio, and Emiliano Paglia. A Language for Enabling Model-driven Analysis of Business Processes. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, MODELSWARD '14, Lisbon, Portugal. SciTePress.

[11] M. Le Brasseur and G. Perdreau. Process weaver: from case to workflow applications. In *CSCW (Computer Supported Co-operative Working) and the Software Process (Digest No. 1995/036), IEE Colloquium on*, pages 7/1–7/5, Feb 1995.

[12] S. Dami, J. Estublier, and M. Amiour. Apel: A graphical yet executable formalism forprocess modeling. *Automated Software Engg.*, 5(1):61–96, January 1998.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] Koenig D. Leymann F. Pfau G. Rickayzen A. von Riegen C. Schmidt P. Trickovic I. Kloppmann, M. Ws-bpel extension for people. Technical report, 2005.

[15] Azeem Lodhi, Veit Küppen, and Gunter Saake. An extension of bpmn meta-model for evaluation of business processes. *Sci. J. Riga Tech. Univ.*, 43(1):27–34, January 2011.

[16] OASIS Web Services Business Process Execution Language (WSBPEL), Diane Jordan, and Alexandre Alves. Web Services Business Process Execution Language Version 2 . 0. *Language*, 11(April):1–264, 2007.

[17] OMG. MDA Guide, version 1.0.1., 2003.

[18] OMG. *Meta Object Facility (MOF) Specification, version 2.0*. 2004.

[19] OMG. *XML Metadata Interchange (XMI) Specification, version 2.1.1*. 2007.

[20] OMG. *Business Process Modeling Notation (BPMN), version 2.0, http://www.omg.org/spec/BPMN/2.0/*. 2011.

[21] OMG. *UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems, volume 33*. 2011.

[22] M S Raunak and L J Osterweil. Resource Management for Complex, Dynamic Environments. *IEEE Transactions on Software Engineering*, 39(X):384–402, 2013.

[23] M. Rausand and A. H?yland. *System Reliability Theory: Models, Statistical Methods, and Applications*. Wiley Series in Probability and Statistics - Applied Probability and Statistics Section. Wiley, 2004.

[24] H. Dieter Rombach. Mvp-l: A language for process modeling in-the-large. Technical report, College Park, MD, USA, 1991.

[25] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and David Edmond. Workflow resource patterns: Identification, representation and tool support. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, pages 216–232, Berlin, Heidelberg, 2005. Springer-Verlag.

[26] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Appl/a: A language for software process programming. *ACM Trans. Softw. Eng. Methodol.*, 4(3):221–286, July 1995.

[27] M. Tortorella. *Reliability, Maintainability, and Supportability: Best Practices for Systems Engineers*. Wiley Series in Systems Engineering and Management. Wiley, 2015.

[28] W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: Yet another workflow language. *Inf. Syst.*, 30(4):245–275, June 2005.

[29] Olegas Vasilecas, Evaldas Laureckas, and Audrius Rima. Analysis of using resources in business process modeling and simulation. *Appl. Comput. Syst.*, 16(1):19–25, December 2014.

[30] WfMC. XML Process Definition Language (XPDL. 2012.

[31] WfMC. Business Process Simulation Specification (BPSim). pages 1–35, 2013.