STRATFram: A framework for describing and evaluating elasticity strategies for service-based business processes in the cloud

Aïcha Ben Jrad, Sami Bhiri, Samir Tata

Version of Record: https://www.sciencedirect.com/science/article/pii/S0167739X18306125
Manuscript_eaaac543c383f2e93117793bd2472901

STRATFram: A framework for describing and evaluating Elasticity Strategies for Service-based business processes in the Cloud

Aicha Ben Jrad[a,b,*], Sami Bhiri[a], Samir Tata[c]

[a] *OASIS, National Engineering School of Tunis, University Tunis El Manar, 2092, Tunisia*
[b] *SAMOVAR, Telecom SudParis, CNRS, University of Paris-Saclay, 9 rue Charles Fourier 91011 EVRY, France*
[c] *Almaden Research Center, IBM Research, San Jose, CA, USA*

Abstract

In the recent years, growing attention has been paid to the concept of Cloud Computing as a new computing paradigm for executing and handling operations/processes in an efficient and cost-effective way. Cloud Computing's elasticity and its flexibility in service delivery have been the most important features behind this attention which encourage companies to migrate their operation/processes to the cloud to ensure the required QoS while using resources and reduce their expenses. Elasticity management has been considered as a pivotal issue among IT community that works on finding the right tradeoffs between QoS levels and operational costs by developing novel methods and mechanisms. However, controlling process elasticity and defining non-trivial elasticity strategies are challenging issues. Also, despite the growing attention paid to the cloud and its elasticity property in particular, there is still a lack of solutions that support the evaluation of elasticity strategies used to ensure the elasticity of processes at service-level. In this paper, we present a framework for describing and evaluating elasticity strategies for Service-based Business Processes (SBP), called StratFram. It is composed of a set of domain-specific languages designed to generalize the use of the framework and to facilitate the description of evaluation elements that are needed to evaluate elasticity strategies before using them in real cloud environment. Using StratFram, SBP holders are allowed to define: (i) an elasticity model with specific elasticity capabilities on which they want to define and evaluate their elasticity strategies, (ii) a SBP model for which the elasticity strategies will be defined and evaluated, (iii) a set of elasticity strategies based on the elasticity capabilities of the defined elasticity model and for the provided SBP model, and (iv) a simulation configuration which identifies simulation properties/elements. The evaluation of elasticity strategies consists in providing a set of plots that allows the analysis and the comparison of strategies. Our contributions and developments provide Cloud tenants with facilities to choose elasticity strategies that fit to their business processes and usage behaviors.

Keywords: Service-based business process, Elasticity Strategies, Evaluation, Elasticity Model, Cloud Computing

1. Introduction

Cloud Computing is gaining more and more importance in the Information Technologies (IT) scope as an emerging computing paradigm for managing and delivering services over the internet. One of the major assets of this paradigm is its economic model based on pay-as-you-go model. It allows the delivering of computing applications as a service rather than a product by enabling ubiquitous, convenient, and on-demand network access to large pools of computing resources (*e.g.*, storage, computing, network, applications and services) that can be dynamically provisioned by increasing and decreasing services capacity to match workloads demands and usage optimization [1].

In today's information society, more and more companies are adopting the Cloud-based technologies in their day-to-day activities to handle customer service in an efficient and cost-effective way. A survey conducted by the IDG Enterprise Cloud Computing across more than 1,600 IT and security decision-makers at a variety of industries in 2014 has shown that 69% of the companies have at least one application or a portion of computing infrastructure in the Cloud [2]. The movement to the Cloud Computing as an IT infrastructure enables companies to operate more efficiently on the continuous incremental change in business operations. Cloud Computing's elasticity and its flexibility in service delivery are the most important features behind this movement which encourage companies and allow the delivery of services with the required quality of service (QoS) to costumers while reducing costs.

Elasticity is defined as the ability of a system to be adjustable to the workload change by provisioning as many resources as needed in autonomic manner in order to meet the QoS requirements [3]. Provisioning of resources can be made using either vertical elasticity, horizontal elasticity, or hybrid elasticity (combination of horizontal and verti-

*Corresponding author
Email addresses: aicha.ben_jrad@telecom-sudparis.eu (Aicha Ben Jrad), sami.bhiri@gmail.com (Sami Bhiri), stata@us.ibm.com (Samir Tata)

cal elasticity). Vertical elasticity, also known as resizing of resources, consists in changing the characteristics/properties (*e.g.*, memory, CPU cores) of the used instances in the system by increasing or decreasing them. Horizontal elasticity, also known as replication of resources, consists in adding/removing instances of system elements to balance the current workload. As a combination of horizontal and vertical elasticity, hybrid elasticity allows to add/remove instances with different characteristics. These elasticity capabilities are the main construction of an elasticity model which defines the ground terms and functionalities that describe the elasticity of the managed system such as the elasticity actions to be undertaken, metrics to monitor to trigger the elasticity actions and properties to access and reconfigure.

Elastic systems are usually managed by elasticity controllers implementing specific elasticity models . The main function of an elasticity controller is to automate the provisioning of resources by controlling the elasticity decisions according to an elasticity strategy that is used to manage elasticity by deciding when, where and how to use elasticity capabilities/actions (*e.g.*, adding or removing resources) defined in the elasticity model of the managed system. Many strategies can be defined to ensure systems elasticity. The abundance of possible strategies requires their evaluation and validation in order to guarantee their effectiveness before using them in real Cloud environments. Few works have targeted the evaluation of elasticity strategies [4, 5]. However, most of them do not allow formal evaluation of elasticity strategies. The formal evaluation of elasticity strategies is an important step to be performed before using the strategies which allows to detect any suspicious behavior that may cost extra-utilization of resources.

In this paper, we present our StratFram framework, for describing and evaluating elasticity strategies for service-based business processes (SBPs). The StratFram framework enables the evaluation, through simulation, different elasticity strategies based on different elasticity models. It is designed based on a set of domain-specific languages (DSLs) for describing different simulation elements from the elasticity model to the simulation configuration in order to conceal the used formal methods/systems and the implementation complexity from SBP holders. Using StratFram, SBP holders are allowed to define (i) an elasticity model with specific elasticity capabilities on which they want to define and evaluate their elasticity strategies, (ii) a SBP model for which the elasticity strategies will be defined and evaluated, (iii) a set of elasticity strategies based on the elasticity capabilities of the defined elasticity model and the provided SBP model, and (iv) a simulation configuration which identifies the elements of the evaluation. As results for an evaluation, StratFram provides a set of plots that allows the analysis and the comparison of strategies.

This paper is organized as follows. Section 2 presents a review of related works for managing and evaluating elasticity in the cloud. In Section 3, we introduce our framework architecture and give an overview of its languages and their relationships. Then, we describe each of its components, namely, SBP language in Section 4, StratModel language in Section 5, Strat language in Section 6, and StratSim language in Section 7. We chain up with the evaluation of our framework in Section 8, which contains the implementation details and some experiment results. The final section presents the conclusions and proposes some lines of future work.

2. State of the art

Since the advent of Cloud Computing, more and more companies are moving their applications to the Cloud. They are now required to manage their cloud services at any time to preserve their QoS. The quality and reliability of the cloud services become an important aspect, as customers have no direct influence on services. QoS has been a critical issue in several customer-centric disciplines such as manufacturing ([6, 7]), healthcare ([8, 9, 10, 11]) and information management ([12, 13, 14, 15]). It denotes the levels of performance, reliability, and availability of a service/process offered by the platform or the infrastructure that hosts it. The expectation of cloud users from providers to deliver the required level of service quality and the neediness of cloud providers to find a good compromise between QoS levels and operational costs, are what make QoS a fundamental issue for both parties.

Elasticity played an important role in many research works that propose methods and mechanisms to harness the ability of services/processes running in the cloud to be elastic regarding the change in workload to ensure that the customer gets the desired level of QoS while avoiding over-provisioning and under-provisioning of resources. In the following, we cite some previous works related to the work presented in this paper.

2.1. Approaches for elasticity strategies evaluation

In [16], the authors proposed an analytical model, using queuing theory, to evaluate the impact of elasticity strategies on the performance of three-tier applications deployed on Cloud infrastructures. They have simulated the logic of scale-out and scale-in actions based on CPU utilization.

In [17], a formal model has been proposed for quantitative analysis of horizontal elasticity at the infrastructure scope using Markov Decision Processes (MDPs). The model has been defined to formally control at runtime the adding and removing of VMs to/from the managed system. However, these works have been proposed to use strategies that are limited to infrastructure metrics (such as CPU utilization) to base their decision and do not consider metrics related to deployed processes. They also focus on evaluating strategies providing horizontal elasticity decisions (*i.e.*, scale-in and scale-out).

In another work, Copil *et al.* [4] have proposed a framework, named ADVISE, for the evaluation of Cloud service/application elasticity behavior based on a learning process and a clustering-based evaluation process that determines at runtime the expected elasticity behavior of Cloud service. It is proposed in order to evaluate different elasticity control processes with different elasticity capabilities exposed by cloud provided and cloud services and determine the most appropriate one regarding the considered Cloud service and a particular situation.

Nevertheless, the existing works do not allow formal evaluation of elasticity strategies before investing in using them in a real Cloud environment and they do not consider the evaluation of SBPs elasticity.

2.2. Languages for describing elasticity strategies

In [18], a Simple Yet-Beautiful Language (SYBL) has been proposed for specifying elasticity requirements for Cloud applications. An elasticity strategy is expressed by SYBL as logical combination of constraints on metric values obtained from one of three main layers that the strategy is associated to: (1) application, (2) component, and (3) programming (*e.g.*, level of infrastructure including rules on CPU usage). The violation or fulfillment of those constraints can lead to triggering particular elasticity actions.

Another DSL language, named Scalability Rule Language (SRL), has been proposed in [19] for specifying event patterns of multi-Cloud application as well as scaling actions. SRL has been inspired from OWL-Q language by adopting some of its terminologies as well as the metric description. It allows specifying elasticity rules as well as metrics and actions as models. Though, its modeling aspect makes it effortful to use and express complex rules. Contrary to our work, an elasticity rule in SYBL and SRL is associated to one specific component identified by its name. SBP holder cannot attach the same rule to different tasks/services. Moreover, these languages don't use symbolic constants and embed rather constant values directly in rule specifications what makes rule definitions and maintenance difficult.

Another work has been presented in [20] where the authors propose a Domain-specific language called SPEEDL that simplifies the specification of elastic strategies of IaaS services. SPEEDL has been proposed to facilitate the creation of event-driven policies for resource management by leasing and releasing VMs. Our interest is to provide a language that triggers the elasticity of SBPs and their services.

Apart from language related aspects; these works don't take into consideration fundamental characteristics of SBPs, which make them unsuitable for defining elasticity strategies for SBPs. Indeed the executions of process instances are scattered over a set of services related to each other according to the process control flow. First these services may have different resource requirements and have thereafter different elastic behavior. Second due to task/service dependencies prescribed by the control flow an elasticity strategy of a given service may need to refer to other related services' states. It is not clear how current approaches can expand their local analysis of the monitored information to have a more global view. Moreover, all the above proposals assume that QoS related requirements (*e.g*, the defined thresholds for QoS metrics) are the same for all requests. However, enactment requests of a SBP are different and require therefore different amount of resources. For example, some process (or service) requests can be more data-intense than others, which could lead to different QoS if we handle them in the same manner.

2.3. Approaches for constructing elasticity controllers

The provisioning of resources is usually conducted automatically by an elasticity controller which has been the focus of many research works. In [21], two adaptive horizontal elasticity controllers has been proposed to control the adding and removing of VMs to prevent QoS violations. In [22], a self-trained elasticity controller has been proposed for managing cloud-based storage services elasticity. It is defined to automatically train itself while serving workload in order to update its control model which is used to make elasticity decisions for adding/removing servers to/from the underlying storage system. In [23], the authors proposed an elasticity approach that uses control theory to synthesize a controller for vertical memory elasticity of cloud applications. Another work tackled the problem of memory elasticity of virtual machines (VMs) has been proposed in [24]. The paper introduced a framework, called CloudVAMP, for monitoring VMs and adjusting their allocated memory to adapt the current memory requirements of their running applications using a cloud vertical elasticity controller/manager. While most of these approaches are recently proposed, they focused on a specific elasticity model for constructing their elasticity controllers, which only tackle either horizontal or vertical elasticity at infrastructure scope.

3. STRATFram: Elasticity Strategies Evaluation Framework for SBPs

Elasticity strategies govern the provisioning of necessary (to respect the agreed QoS) and sufficient (to handle the amount of requests) resources despite variations in enactment requests load. Many strategies can be defined to steer processes elasticity. The abundance of possible strategies requires their evaluation in order to guarantee their effectiveness before using them in real Cloud environments. In this section, we present our elasticity strategies evaluation framework for SBPs, named StratFram. We first present StratFram overview, followed by the relationships between StratFram languages.

3.1. Strat*Fram overview*

Fig. 1 shows an overview of StratFram framework for evaluating elasticity strategies. The framework allows SBP

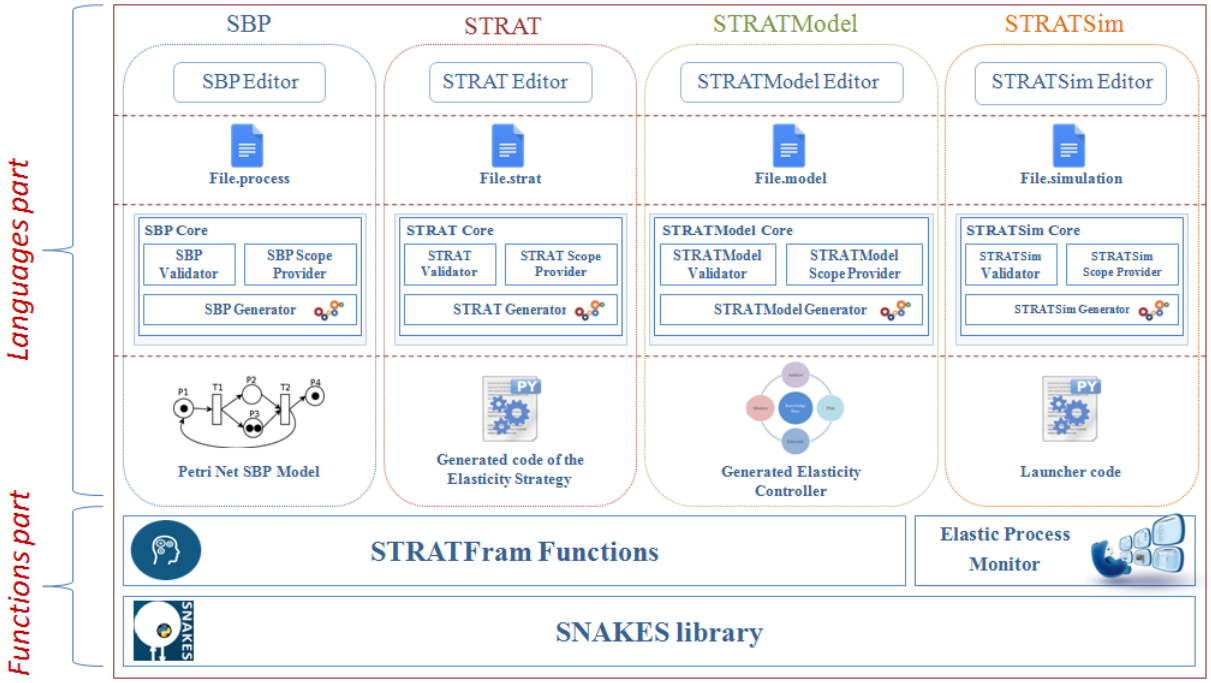


Figure 1: StratFram architecture overview

holders to evaluate, through simulation, elasticity strategies for a given SBP model and under a given usage behavior based on a given elasticity model. As illustrated in Fig. 1, StratFram is composed of two main parts: (1) StratFram languages, and (2) StratFram functions. The first part is composed of a set of languages designed to generalize the use of the framework and to facilitate the description of evaluation elements, *i.e.*, elasticity strategies that will be evaluated, SBP model for which the elasticity strategies will be defined, elasticity model on which the elasticity strategies will be based, and simulation configuration, while concealing the implementation complexity and the used formal method from the users. Each language is provided with its dedicated editor from which the SBP holders define their evaluation elements and perform their evaluation. The StratFram languages part is composed of the following languages:

1. ***SBP language***: It is a domain-specific language for defining a SBP model that represents the elastic execution environment of a SBP in the cloud. It allows SBP holders to specify in declarative way the characteristics of components composing of the elastic execution environment of a SBP and their connections. The use of SBP language releases SBP holders from dealing with the specification of the underline formal method used for the evaluation;

2. ***STRATModel language*** [25]: It is a domain-specific language for describing elasticity models for SBPs which allows to define different elasticity models. It permits business process holders to define different elasticity models, with different elasticity capabilities/mechanisms and customized monitoring metrics, and to generate their associated elasticity controllers in order to use them to evaluate elasticity strategies on a given SBP model.

3. ***STRAT language*** [26]: It is a rule-based domain-specific language for describing elasticity strategies for SBPs deployed in Cloud environments. It was initially proposed based on a specific elasticity model. Thereafter, we generalized the use of Strat to be able to adapt to any elasticity model provided by SBP holders.

4. **STRAT*Sim language***: It is designed as domain-specific language for specifying simulation properties/elements in a declarative manner and providing SBP holders with a simulation launcher for their evaluation scenario.

Each one of the StratFram languages is provided with a code generator that is responsible for generation a concrete implementation of the described element based on the formal method/system used in the second part of StratFram. The latter represents the StratFram engine which provides common classes and functions used and triggered by the generated items either for processing or profiling the SBP model. The provided classes and functions are implemented based on the underlying used api for a specific formal method/system, *e.g.*, SNAKES API for defining and executing petri-nets.

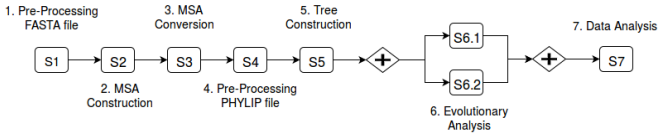


Figure 2: BPMN model of Molecular Evolution Reconstruction Process

Currently, StratFram evaluation results can be displayed as plots showing the behavior of resource allocations for executing each elastic service in the defined SBP model as well as the one of the overall process according to some defined indicators for a specific elasticity strategy.

Example 1. *Let's consider an elastic system composed of (i) a SBP for molecular evolution reconstruction (MER) based on an input protein sequences of genomes, (ii) an elasticity controller implementing a specific elasticity model and (iii) an elasticity strategy that have to be evaluated before using it in a real cloud environment.*

The MER process is a data-centric process. As illustrated in Fig. 2, it is composed of 8 services. Some services (e.g., S2) are more complex than others which make them expensive in terms of time and resource consumption. Their time and resource consumption are strongly related to the size of the input file. Some input files will contain small sets of protein sequences of genomes that are expected to be processed in short time and to use a little amount of computational resources, while others will contain large sets that are expected to take much more time to be processed and its processing occupies a large amount of computational resources. So, at some point of time, the process will get invoked many times with different input files which lead some of the composed services to overcome their capacity which in turn leads to loss of QoS. In such case, the elasticity controller will perform the required elasticity action on the bottleneck services according to the given elasticity strategy which defines when, where and how to use elasticity actions.

The elasticity controller implements an elasticity model for hybrid scaling that performs two main elasticity actions namely 'Duplicate' and 'Consolidate' and defined to use reactive elasticity strategies. The duplication action allows adding a new service copy with a different configuration. Two properties are re-configurable by the action namely the capacity and the group property. The consolidation action allows releasing service copies whenever needed.

The elasticity strategy that will be used in our elastic system and have to be evaluated by our StratFram defines rules for the actions defined in the elasticity model based on waiting time of requests. So, the duplication action creates a new copy of a service for requests under a specific group if at least one of the waiting requests of that group exceeds the maximum waiting time threshold and the same applied for all its copies. Otherwise, if there is no longer waiting requests and the consumed capacity of a service copy is equal to 0 and the response time of the service is below its minimum threshold, a consolidate action is triggered by releasing the service copy.

We show in the following sections how each component of our elastic system can be described using our framework/languages in order to evaluate how the chosen strategy will behave in the system.

3.2. Relationships between StratFram languages

The design of StratFram was based on the idea of providing a language for describing elasticity strategies that is general enough to describe strategies for different elasticity models and allows the adding of new possible actions. The first solution was to provide Strat language grammar with a constant set of actions used in the commercial cloud-solutions and the research papers. Such solution makes the SBP holders constrained to a set of pre-defined actions and parameters and does not enable the adaptation to new elasticity capabilities that can be provided by the community.

The second solution that we adopted in designing our framework is to provide a set of languages that allow users to separately define an elasticity model on which Strat language will be based. So, the latter will provide SBP holders with only the elasticity capabilities defined in the elasticity model by the elasticity manager. This solution is based on the notion of *"cross-reference"* to create links between languages. The elasticity capabilities are defined using StratModel as objects and then their references are used in Strat language as part of its grammar. Along the elasticity capabilities, StratModel allows users to define a set of metrics that can be used as QoS metrics or functions called inside Strat script using their references, and to specify the properties that the users can reconfigure when applying an action according to a Strat strategy. The links created between the two languages, *i.e.*, StratModel and Strat, only constraints the users with the order of defining their system elements. So, they have to first define an elasticity model using StratModel on which Strat will be based and then they can define their elasticity strategies.

Other links has been made between StratModel and SBP language, and StratSim and all the other languages. Using StratModel, a user can (optionally) define his elasticity model for a specific SBP by indicating the reference to an already defined SBP model, using SBP language, as the managed component of the elasticity model. Since it is designed to indicate the elements of an evaluation scenario, StratSim provides users with references to the already defined elements that can be used in an evaluation.

In the following sections, we describe the detail of each language in our framework with its use in defining an element of the elastic system given in Example 1.

4. SBP Language

SBP language is proposed to facilitate the description of the elastic execution environment of a SBP by using

elements composing an elastic execution environment in the cloud. The execution environment of a given elastic SBP can be seen as a network, of service engines (refer to containers such as micro-container [27], Docker [28]) isomorphic to the SBP model. The execution of a SBP instance (request) is routed over several services (execution) engines that match each of the SBP component services via routers which are resources that provide message format transformations and routing. The structure of execution environment of a SBP evolves over time according to requests load by adding/removing service engine copies in order to meet its QoS requirements (*e.g.*, maximum response time). The service engine copies are related to a load balancer service to balance the incoming load between them. In our previous work [29], we argued that it is beneficial to adopt formal models to describe elastic execution environment of SBPs which provides rigorous description and allows formal evaluation and verification of SBP elasticity. We proposed to use petri nets to formally describe the elastic execution environments of SBPs. However, allowing SBP holder to express their processes in terms of petri nets needs knowledge on petri net concepts and how to use them for describing elastic execution environment of SBP. To facilitate the modeling of SBPs elasticity and the extensibility of our framework, we propose a domain-specific language, named SBP, for describing an elastic execution environment for a SBP and generating its corresponding formal model according to the provided SBP generator. So, SBP language can be seen as an interface that provides an abstract representation of SBP model entities. By providing different SBP generators for different formal modeling techniques or systems, we allow STRATFram to be extensible and used across different systems. In the following, we present our formal model of SBP elasticity that is used and generated by SBP language.

4.1. Formal Modelling of SBP elasticity

We proposed, in [29], a formal model for describing elastic execution environments for SBPs. The model is defined based on High-Level Petri Nets [30] to describe the characteristics of service engines, hosted services in a SBP, and their requests in order to allow defining more sophisticated elasticity strategies using different elasticity indicators. The High-Level Petri Net model has been proposed to extend classical petri nets by introducing higher-level concepts, such as representing tokens by complex data structure, and using expressions to annotate net elements (*i.e.*, Places/Transitions/Arcs) which makes it possible to describe the elements of our execution environment of SBPs. In our model, a place denotes either a service engine or a load balancer, a transition denotes a router and a token denotes a service request/instance which may carry a data value referenced by a token color.

A service engine place is characterized by a function representing the time complexity of the hosted service. This complexity function enables estimating the processing time needed for handling each request. Also, it has

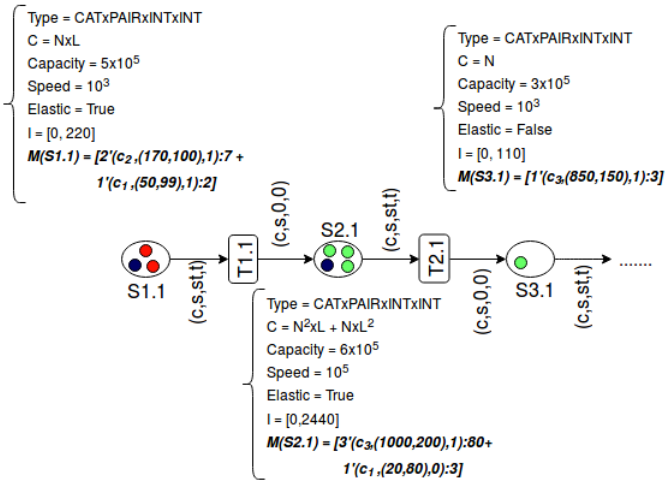


Figure 3: TC-SBP petri net system of MER process

a processing speed and a capacity indicating the quantity of data that can be processed simultaneously by the service engine. It can be either an elastic service engine or not. Moreover, if it is an elastic one, it might have many copies hosting the same service related by an equivalence relation and connected to a load balancer. A load balancer/buffer place is characterised by a capacity indicating its queue length. Both types of places (*i.e.*, service engine place and buffer place) have temporal information enabling to indicate when a request becomes outdated. A service request/token is characterized in our model by its data size, its belonging category (according to the data size or representing a tenant) and a state defining its progress (waiting/under-processing/finished) representing its data value along with the processing/waiting time representing its age. A data value with an age attached to the request/-token named timed token color. Each place in our model can hold many requests/tokens that represent its marking. To connect places in our model, we use a transition as a router to transfer requests between places (service engines/load balancers) according to the behavioral specification of the SBP. Finally, arcs connecting places and transitions in our model allow to modify the characteristics of requests when they are "transferred" from one place to another using expressions on the arcs.

Example 2. *Let's take the SBP presented in Example 1 (cf., Fig. 2). Fig. 3 represents the corresponding petri-net model of sub-SBP for MER. Each place in the petri net model is annotated with its type, its time complexity function C, its capacity, its processing speed, its time interval I, the marking M for no empty places and whether the place is elastic or not. The accepted token format for the places from $S1.1$ to $S3.1$ is of type $CAT \times PAIR \times INT \times INT$ where CAT is a set of groups which contains three groups c_1, c_2 and c_3 respectively for the small, medium and large sequence files (request), PAIR represents a couple of integer values (N, L) for the number of sequences in the file*

and the length of sequences, INT×INT represents the state and the age of the token. For example, the service place $S2.1$ corresponding to the service $S2$ in Fig. 2, is of time complexity $O(N^2 \times L + N \times L^2)$. We assume that its corresponding service engine has a capacity $6{\times}10^5$ (quantity of data that the service engine can process) and can process 10^5 instructions per time unit and it considers any request that has been in it for more than 2440 time units as an outdated request. We assume that at time t_1 the service engine associated to the place $S2.1$ holds four requests: one waiting request (c_1, (20,80), 0) of group c_1 with data size (20,80) and an age equal to 3, three under-processing requests with data value (c_2, (1000,200), 1) and an age 20.

4.2. SBP Grammar

Based on the composition of an elastic execution environment, we define the grammar of SBP language that allows to specify SBP models using elastic execution environment components rather than using formal notation and terminologies of the used formal method. The top-level of SBP specification grammar is given in Grammar 1 using the Backus Normal Form (BNF).

⟨*SBPModel*⟩ ::= '`Process`' ⟨*id*⟩ '{' ⟨*ProcessDescription*⟩ '}' ⟨*Requests*⟩

⟨*ProcessDescription*⟩ ::= ⟨*Reference*⟩ ⟨*Description*⟩ ⟨*Groups*⟩ ⟨*Nodes*⟩ ⟨*Routers*⟩ ⟨*Links*⟩

⟨*Groups*⟩ ::= ⟨*Group*⟩ ⟨*Groups*⟩ | ⟨*empty*⟩

⟨*Nodes*⟩ ::= ⟨*Node*⟩ ⟨*Nodes*⟩ | ⟨*Node*⟩

⟨*Node*⟩ ::= ⟨*ServiceEngine*⟩ | ⟨*LoadBalancer*⟩

⟨*Routers*⟩ ::= ⟨*Router*⟩ ⟨*Routers*⟩ | ⟨*empty*⟩

⟨*Links*⟩ ::= ⟨*Link*⟩ ⟨*Links*⟩ | ⟨*empty*⟩

⟨*Requests*⟩ ::= ⟨*Request*⟩ ⟨*Requests*⟩ | ⟨*empty*⟩

Grammar 1: General SBP Grammar

A SBP model is composed of a structure and a marking that corresponds to a set of requests in the process. The structure description part is given in a block defined by the keyword `Process` (*i.e.*, indicates the beginning of the SBP model) and identified by a name. As described in the formal model, a SBP structure can be composed of four sets: (*i*) a set of groups, (*ii*) a set of nodes where a node can be either a service engine or a load balancer, (*iii*) a set of routers, and (*iv*) a set of links connecting nodes and router to each other. To describe the SBP model, the SBP holder can provide a process reference to indicate its name outside the framework and a descriptive text.

A group represents one set of requests with specific characteristics/requirements. As shown in Grammar 2, it is identified by a name which refers to a tenant, a group of users (such as premium user, normal user, and guest user), or a group of requests according to the indicated type. For the latter, the group is defined depending on the size of requests data. So, the user can indicate the range of data size for request to be belonging to the group. This categorization of requests allows SBP holders to specify different QoS requirements for different groups and assign groups to specific service engine copies.

⟨*Group*⟩ ::= '`group`' '`:`' ⟨*GroupDescription*⟩ '`;`'

⟨*GroupDescription*⟩ ::= ⟨*Name*⟩ ⟨*Type*⟩ ⟨*Range*⟩

⟨*Name*⟩ ::= '`name`' '`:`' ⟨*id*⟩

⟨*Type*⟩ ::= '`type` '`:`' ⟨*GroupType*⟩ | ⟨*empty*⟩

⟨*GroupType*⟩ ::= '`REQUEST_DATA`' | '`REQUEST_USER`' | '`TENANT`'

Grammar 2: Grammar for describing process groups in SBP

A service engine represents the container on which a service is hosted. A container can be either a VM or a container like Docker [28] or micro-container [27]. The service engine, as shown in Grammar 3, is described by a set of attributes specifying its characteristics. A service engine's name is used to identify the elements inside the framework while its reference indicates its actual name or location.

A service engine can be included in the SBP for a specific set of groups. So, SBP holders can associate groups to the service engine by referring to their names as given in the groups section. For the simulation purpose, a complexity function can be given to estimate the processing time needed for handling each request. The capacity and processing speed attributes can be provided to indicate the quantity of data that can be processed simultaneously by the service engine. The latter are given with a cost which may restrict the use of resources. The timeout attribute can be used to indicate when a request becomes outdated inside the service engine. Additionally, a service engine can be specified as an elastic service engine or not. In the first case, it might have many copies hosting the same service connected to a specific load balancer. So, '*initial*', '*copies*' and '*lb*' attributes represent respectively the original service engine copy from which the current one has been duplicated, the list of service engine copies which are given by referring to their names, and a reference to the load balancer node that is associated to the current service. Moreover, the set of belonging requests can be given by referring to their identifiers.

After introducing the service engines composing a SBP, load balancers can be primary described by a name, a reference and an associated service engine (*cf.* Grammar 4). A load balancer represents a service that is responsible for

⟨*ServiceEngine*⟩ ::= '`ServiceEngine`' ':' ⟨*SEDesc*⟩ ';'

⟨*SEDesc*⟩ ::= ⟨*Name*⟩ ⟨*Reference*⟩ ⟨*OriginalCopy*⟩ ⟨*Container*⟩ ⟨*Complexity*⟩ ⟨*NodeGroups*⟩ ⟨*Capacity*⟩ ⟨*Speed*⟩ ⟨*Elastic*⟩ ⟨*Cost*⟩ ⟨*TimeOut*⟩ ⟨*ServiceLoadBalancer*⟩ ⟨*ServiceCopies*⟩ ⟨*NodeRequests*⟩

⟨*OriginalCopy*⟩ ::= '`initial`' ':' [ServiceEngine] | ⟨*empty*⟩

⟨*ServiceLoadBalancer*⟩ ::= '`lb`' ':' [LoadBalancer] | ⟨*empty*⟩

⟨*ServiceCopies*⟩ ::= '`copies`' ':' ⟨*SetCopies*⟩ | ⟨*empty*⟩

⟨*SetCopies*⟩ ::= [ServiceEngine] ',' ⟨*SetCopies*⟩ | [ServiceEngine]

⟨*NodeRequests*⟩ ::= '`requests`' ':' ⟨*SetRequests*⟩ | ⟨*empty*⟩

⟨*SetRequests*⟩ ::= [Request] ',' ⟨*SetRequest*⟩ | [Request]

Grammar 3: Grammar for describing ServiceEngine in SBP

receiving requests and forward them to a specific service engine copy in order to balancer the load between copies of the same service according to a load balancing algorithm such as Round Robin. Moreover, a load balancer has a capacity (or a queue) representing the maximum number of requests that can keep in its queue before forward them to their target service engine copy. Similar to service engines, a load balancer might have a temporal information (*i.e.*, timeout) indicating when a request in the queue will be considered outdated, a set of allowed groups, and a set of requests.

⟨*LoadBalancer*⟩ ::= '`LoadBalancer`' ':' ⟨*LBDesc*⟩ ';'

⟨*LBDesc*⟩ ::= ⟨*Name*⟩ ⟨*Reference*⟩ ⟨*Service*⟩ ⟨*NodeGroups*⟩ ⟨*Queue*⟩ ⟨*TimeOut*⟩ ⟨*Algorithm*⟩ ⟨*NodeRequests*⟩

⟨*Service*⟩ ::= '`service`' ':' [ServiceEngine]

Grammar 4: Grammar for describing LoadBalancers in SBP

In order to transfer requests between services (either service engines or load balancers), a router is used to take requests from services as output and transfer them the next services as input. For simulation purpose, a router can be given with an expression representing a condition on requests to be transferred by the router. As we might have more than one copy of a service engine, we might have also a set of routers related by equivalent relation connected to the copies. So, in the router description, a SBP holder can identify the initial router element, from which the other copies are copied from, by referring to the name of the router. Moreover, the set of router copies can be introduced by referring to their name as a list. the formal grammar for defining routers is given in the Grammar 5.

⟨*Router*⟩ ::= '`router`' ':' ⟨*RouterDesc*⟩ ';'

⟨*RouterDesc*⟩ ::= ⟨*Name*⟩ ⟨*OriginalRouter*⟩ ⟨*Expression*⟩ ⟨*RouterCopies*⟩

⟨*OriginalRouter*⟩ ::= '`initial`' ':' [Router] | ⟨*empty*⟩

⟨*RouterCopies*⟩ ::= '`copies`' ':' ⟨*SetRouters*⟩ | ⟨*empty*⟩

⟨*SetRouters*⟩ ::= [Router] ',' ⟨*SetRouter*⟩ | [Router]

Grammar 5: Grammar for describing process routers in SBP

Therefore, for routers to be able to do their function, they should be connected to services via input and output links (*cf.* Grammar 6). An input link connects a router to a service (either a service engine or a load balancer) which gives to the latter requests as input. In the other hand, an output link connects a service to a router which takes requests from the former as output. In order to describe the change that is made by services on input requests, the SBP holder can provide links with an expression which allows to modify the characteristics of requests when they are transferred from node to node.

⟨*Link*⟩ ::= ⟨*InLink*⟩ | ⟨*OutLink*⟩

⟨*InLink*⟩ ::= '`in`' ':' [Router] '`to`' [Node] ⟨*LinkExpression*⟩

⟨*OutLink*⟩ ::= '`out`' ':' [Node] '`to`' [Router] ⟨*LinkExpression*⟩

Grammar 6: Grammar for specifying links between nodes and routers in SBP

A request can be characterized by a set of attributes defining its static and dynamic aspects. The static characteristics of a request are an identifier and a reference. The same request sent by a user traverses a set of services to attend the final result. So, some request's characteristics might change from service to service, such as its data size and its belonging group (if the type of the group is data based) which can take new values according to the output data of one service and the input data of the next service, and some others might change inside the service such as the request's age which represents the processing/waiting time and the request's status which defines the request's progress (waiting/running/finished/dead).

⟨*Request*⟩ ::= 'Request' ⟨*id*⟩ '{' ⟨*RequestDesc*⟩ '}'

⟨*RequestDesc*⟩ ::= ⟨*Reference*⟩ ⟨*RequestNode*⟩
⟨*RequestGroup*⟩ ⟨*RequestStatus*⟩
⟨*RequestAge*⟩ ⟨*RequestExecutionTime*⟩
⟨*RequestTotalTime*⟩ ⟨*RequestData*⟩

⟨*RequestNode*⟩ ::= 'node' ':' [Node]

⟨*RequestGroup*⟩ ::= 'group' ':' [Group] | ⟨*empty*⟩

⟨*RequestStatus*⟩ ::= 'status' ':' ⟨*Status*⟩ | ⟨*empty*⟩

⟨*Status*⟩ ::= 'WAITING' | 'RUNNING' | 'FINISHED' | 'DEAD'

Grammar 7: Grammar for defining Requests in SBP

As indicated in Grammar 7, more than one attribute can be given to a request to indicate time information. 'age' attribute can be given to indicate the total waiting and processing time of the request since it has been send to the service. 'executionTime' attribute represents the processing time of a request and from which the waiting time can be concluded while 'totalTime' attribute represents the time that has been spend by the request since the beginning of the process.

Example 3. *Listing 1 presents the description of our SBP model presented in Example 1 using SBP language. It presents the description of the group 'c1' for small requests. It is defined depending on the size of requests data. In our example the size of requests data is the size of a two-dimension matrix, cf. NxM where N is the number of lines and M is the number of column (i.e., the length of a sequence). The group 'c1' is one of the groups that are associated to the service engine 'S2'. The service 'S2' is of time complexity $O(N^2 \times L + N \times L^2)$. As we have previously indicated, its corresponding service engine has a capacity 6×10^5 (quantity of data that the service engine can process) and can process 10^5 instructions per time unit and it considers any request that has been in it for more than 2440 time units as an outdated request. In its initial state, the service 'S2' does not need a load balancer to balance the incoming load. With the creation of new service copies to serve the increasing incoming load, a load balancer will be used. So, a load balancer named 'B2' is specified for the service engine 'S2'. It is characterized with a queue indicating how many requests it can keep waiting before forwarding them to a service copy. The requests in a load balance can became outdated after waiting 150 time units. A request is defined to be timed and to have data value as a tuple (x, y).*

```
Process MERProcess{
  referenceID : 'MolecularEvolutionReconstruction'
  group :
        name : c1
        type : REQUEST_DATA
        range : 10000
  ;
    ...
  serviceEngine :
        name : 'S2'
        complexity : 'N*N*L+N*L*L'
        groups : c1, c2, c3
        capacity : 600000
        speed : 200000
        elastic : true
        timeout : 2440
  ;
  loadBalancer :
        name : 'B2'
        service : S2
        groups : c1, c2, c3
        queue: 10
        timeout : 150
  ;
    ...
  router : name : 'T1'
        ...
  out : S1 to R1
  in : R1 to S2
        ...
}
Request Req1 {
        referenceID : 'req1'
        node : S2
        group : c1
        age : 0
        executionTime : 0
        totalTime : 0
        data : 100.0,100.0
}
```

Listing 1: Example of describing the SBP Model of our MER process using SBP language

4.2.1. SBP Core

After describing a SBP model using SBP grammar, the script should be processed to generate its corresponding formal model according to the chosen technique. In order to do so, some basic functionalities are provided to the language composing its core. This part is composed of the three main components that can be adapted to the chosen formal technique by providing a new implementation: (1) a validator that is responsible for checking the model structure coherence and the provided expressions correctness which might be given to routers and links, (2) a scope provider which aids users to edit their script by providing a set of expected elements, and (3) a generator that translates a SBP script into a Petri net model according to the described model.

5. STRATModel Language

In order to allow SBP holders to customize the evaluation framework to their needs, STRATModel language has been designed to allow describing elasticity models and generating their associated elasticity controllers using a pre-defined elasticity controller template. An elasticity

model defines the ground terms and functionalities that describe the elasticity of SBPs such as the elasticity actions to be undertaken, metrics to monitor to trigger the elasticity actions and properties to access and reconfigure. Hence, it is the basis for specifying elasticity strategies and constructing an elasticity controller that manages and evaluates SBPs elasticity. An elasticity controller is used to monitor a SBP execution and analyze its performance by inspecting an elasticity strategy in order to apply some actions whenever needed which may reconfigure some properties of the managed component [29]. So, the monitoring metrics, actions, and properties are the composition of an elasticity model in StratModel.

A StratModel metric can be defined as a basic metric, *i.e.*, obtained directly from the monitored component property, or a composed metric by introducing the composition expression of basic metrics. A StratModel action is defined to make changes on a SBP model and it targets a specific type of component, *e.g.*, Service. StratModel is designed relying on the specification of SBP model.

Describing elasticity model in StratModel depends on what characteristics SBP holders want to provide to their SBP models such as temporal information, how they want to manage them by the generated elasticity controller, and which type of elasticity strategies they want to specify. Such information is required to know whether to include or exclude some functionalities to/from the generated elasticity controller.

5.1. STRATModel Grammar

The top-level of StratModel specification grammar is given in Grammar 8 using the Backus Normal Form (BNF). StratModel documents are composed of two parts: (i) the elasticity model description part which defines the essential elements to describe an elasticity model and (ii) SBP transformation states which define the transformations occurred on the SBP model when applying some defined elasticity actions. In the following, we will discuss each of the composed parts in more details.

⟨*ElasticityModel*⟩ ::= ⟨*ModelDescription*⟩ ⟨*ProcessStates*⟩

⟨*ProcessStates*⟩ ::= ⟨*ProcessState*⟩ ⟨*ProcessStates*⟩ | ⟨*empty*⟩

Grammar 8: General StratModel Grammar

5.1.1. Elasticity Model Description

An elasticity model in StratModel as given in Grammar 9 is composed of two sets of statements, descriptive and functional, encapsulated in a block defined by `ElasticityModel` and identified by a name.

The user is allowed to firstly describe the general aspect of elasticity model such as the managed component, the use of knowledge base and the frequency of monitoring, *etc.* Thereafter, the functional statements can be provided to specify the essential elements for describing the functionality of the elasticity controller implementing the elasticity model. They are split into actions to be undertaken, metrics to monitor to fire the elasticity actions and properties to access and to reconfigure.

⟨*ModelDescription*⟩ ::= '**`ElasticityModel`**' ⟨*id*⟩ '{' ⟨*ModelStatements*⟩ '}'

⟨*ModelStatements*⟩ ::= ⟨*GeneralDescription*⟩ ⟨*ItemsDefinitions*⟩

⟨*ItemsDefinitions*⟩ ::= ⟨*ItemDefinition*⟩ ⟨*ItemsDefinitions*⟩ | ⟨*empty*⟩

⟨*ItemDefintion*⟩ ::= ⟨*Action*⟩ | ⟨*Metric*⟩ | ⟨*Property*⟩

Grammar 9: Grammar for describing elasticity model in StratModel

Example 4. *Let's consider our elasticity model for hybrid scaling that performs two main elasticity actions namely 'Duplication' and 'Consolidation as described in Example 1. We specify our SBP model written in SBP language as the managed component. The elasticity strategies that will be used on the latter are reactive and they do not need a knowledge base for elasticity decisions. Listing 2 presents a general description of the elasticity model named 'ElasticityModel1'.*

```
ElasticityModel ElasticityModel1 {
  referenceID : 'ElasticityModel1'
  managedComponent : MERProcess
  routing : DEFAULT
  timer : true
  knowledgebase : false
  frequency : 3
  ...
}
```

Listing 2: Example of describing an Elasticity Model with StratModel

- ***Action***: An `action`, as shown in Grammar 10, is defined by a set of statements for describing its functionality details and for generating its implementation mechanism. It is defined by a name, a reference ID and a component. The latter is used to specify on which type of elements the action can be applied, *i.e.*, `Process`, `Service`, `Router` or `Request`. For example, a Routing action can be defined for `Router` to allow to control the execution flow of requests between SBP's services. The keywords `delay` and `multiple` are used to define respectively the time delay of applying the action and whether its multiple application is allowed.

The execution of the defined action changes the structure of the managed SBP model. It transforms the SBP

⟨*Action*⟩ ::= '**action**' ':' ⟨*ActionStatements*⟩ ';'

⟨*ActionStatements*⟩ ::= ⟨*Name*⟩ ⟨*Reference*⟩ ⟨*Component*⟩ ⟨*Delay*⟩ ⟨*Multiple*⟩ ⟨*Transformation*⟩

⟨*Delay*⟩ ::= '**delay**' ':' ⟨*int*⟩ | ⟨*empty*⟩

⟨*Multiple*⟩ ::= '**multiple**' ':' ⟨*Boolean*⟩ | ⟨*empty*⟩

⟨*Transformation*⟩ ::= '**cases**' ':' ⟨*Examples*⟩

⟨*Examples*⟩ ::= ⟨*Example*⟩ ⟨*Examples*⟩ | ⟨*empty*⟩

⟨*Example*⟩ ::= '**apply**' '**on**' ⟨*Elements*⟩ '**transform**' [ProcessState] '**to**' [ProcessState]

Grammar 10: Grammar for describing elasticity actions in StratModel

model from one state to another. This transformation can be specified in StratModel as transformation cases. A transformation case is defined by giving an example of an initial state of the SBP model and the resulting state after applying the action by referring to the provided process states (*cf.* Grammar 13). The idea of using examples to specify the transformation on SBP model follows the *by-example paradigm* [31] which allows the software to drive information from a set of examples specifying how things are done or what the user expects. The most prominent approaches for *by-example paradigm* are *Query by-example* [32] which has been developed for querying database systems by allowing users to give examples of query results and *Programming by-example* [33] that permits to create a program from user's actions which are recorded as replayable macros. These approaches allow the use of examples in some way to overcome the complexity of selected problems in the field of computer science. In this work, we argue that providing transformation examples is more friendly for SBP holders than defining complex formal transformations instructions. So, in StratModel, the user is allowed to give a set of examples to describe different cases of applying the action on specific elements.

Example 5. *In the following, we describe the duplication action of our running example. It targets service engines components. Its execution will be delayed by 4 time units. There are two transformation cases. The first case is when the duplication action will be applied for the first time on a service engine to create a new copy and a service engine load balancer. The second case is when there already exists more than one service copy sharing a load balancer in the process model. The description of the action is given in Listing 3.*

```
ElasticityModel ElasticityModel1 {
  ...
  action :
    name : 'Duplicate'
        referenceID : 'D'
        component : Service
        delay : 4
        cases :
          apply on 'S2' transform state1 to state2
          apply on 'S2' transform state3 to state4
  ;
  ...
}
```

Listing 3: Example of describing an action with StratModel

- ***Metric***: A *metric* in StratModel is identified by a name and has a low-level reference. It is related to an entity that can be a process, a service, a load balancer or a requests. It also can be obtained for a specific group by allowing grouping. A metric can be either a basic metric which obtained from a low-level property or a composite metric that is denoted by the values of other base or composite metrics. For example, the metric *executionTime* is a basic metric that refers to the age of a service request. When defining composite metrics, *expression* is used to specify how the value is computed. Also, a metric can be obtained for a specific group of requests by indicating *group* as true. The specification of metrics in StratModel is given by Grammar 11 using also the Backus Normal Form (BNF).

⟨*Metric*⟩ ::= 'metric' ':' ⟨*MetricStatements*⟩ ';'

⟨*MetricStatements*⟩ ::= ⟨*Name*⟩ ⟨*Reference*⟩ ⟨*Entity*⟩ ⟨*OnGroups*⟩ ⟨*Unit*⟩ ⟨*MetricExpression*⟩

⟨*Entity*⟩ ::= 'level' ':' ⟨*MetricLevel*⟩

⟨*MetricLevel*⟩ ::= 'Service' | 'LoadBalancer' | 'Process' | 'Request'

⟨*OnGroups*⟩ ::= 'group' ':' ⟨*Boolean*⟩ | ⟨*empty*⟩

⟨*Unit*⟩ ::= 'unit' ':' ⟨*string*⟩ | ⟨*empty*⟩

⟨*MetricExpression*⟩ ::= 'expression' ':' ⟨*Expression*⟩

Grammar 11: Grammar for describing metrics in StratModel

Example 6. *Listing 4 describes a metric named 'waitingTime' which captures the waiting time of a request in a service. It is the subtraction of the value of two metrics: 'processingTime' and 'executionTime' which refers to the 'age' attribute of request.*

```
ElasticityModel ElasticityModel1 {
  ...
  metric :
    name : 'WaitingTime'
```

```
      level : Request
      expression : executionTime - processingTime
  ;
  metric :
      name : 'executionTime'
      referenceID : 'age'
      level: Request
;
  ...
}
```

Listing 4: Example of describing metrics with StratModel

- ***Property***: In elasticity model, some actions may requires to access or modify some low-level properties/attributes of the managed SBP and its services. So, the user is allowed to define those properties and whether they are configurable or not. A ***property*** is primary defined in StratModel by a name and a reference.

⟨*Property*⟩ ::= 'property' ':' ⟨*PropertyStatements*⟩ ';'

⟨*PropertyStatements*⟩ ::= ⟨*Name*⟩ ⟨*Reference*⟩ ⟨*Config*⟩

⟨*Config*⟩ ::= 'configurable' ':' ⟨*Boolean*⟩

Grammar 12: Grammar for describing properties in StratModel

Example 7. *As we previously indicated, there is two properties that are accessible and reconfigurable by the duplication action. Listing 5 presents how they are specified using* StratModel.

```
ElasticityModel ElasticityModel1 {
  ...
  property :
        name : 'cap'
        referenceID : 'capacity'
        config : true
  ;
  property :
        name : 'cat'
        referenceID : 'groups'
        config : true
  ;
  ...
}
```

Listing 5: Example of describing properties with StratModel

5.1.2. SBP transformation state definition

As previously indicated, the execution of an action transforms the SBP model from one state to another. A state represents the managed SBP model at timestamp t. It is specified in a block defined by ***ProcessState*** and identified by a name which is used to refer to the state in the action description section (*cf.* Grammar 13).

Two ways are allowed to define the SBP model at timestamp t. The first is by describing the process and its components using SBP language notations. So, as described in section 4.2, the item SBPModel is composed of Process block describing the structural representation of the SBP model at timestamp t followed by the description of requests contained in the SBP's services. The block encapsulates the process general description, the groups of requests allowed in the process, its services that are split into service engines and load balancer, the routers that connect its services, and the links between services and routers. The second way for providing the SBP model description at timestamp t, is by indicating the URL of the SBP petri net model encoded in the Petri Net Markup Language (PNML).

⟨*ProcessState*⟩ ::= '***ProcessState***' ⟨*id*⟩ '{' ⟨*ProcessDefinition*⟩ '}'

⟨*ProcessDefinition*⟩ ::= ⟨*SBPModel*⟩
| '***url***' ':' ⟨*string*⟩

Grammar 13: Grammar for defining a SBP state in StratModel

Example 8. *Listing 6 describes two states of the transformation states used to describe the mechanism of the duplication action. We focus on the service 'S2' to show how the states can be described in* StratModel. *The process used to describe the states is our SBP model.*

```
ProcessState state1 {
  Process Process1 {
    ...
        serviceEngine :
          name : 'S2'
          complexity : 'N*N*L + N*L*L'
          groups : c1, c2, c3
          capacity : 600000
          ...
          requests : req1, req2, req3
        ;
        ...
  }
  ...
}
ProcessState state2 {
  Process Process1 {
    ...
        serviceEngine :
          name : 'S2'
          complexity : 'N*N*L + N*L*L'
          groups : c1, c2, c3
          capacity : 600000
          ...
          lb : LB_S2
          copies : S21
          requests : req1, req2, req3
        ;
        serviceEngine :
          name : 'S21'
          initial : S2
          complexity : 'N*N*L + N*L*L'
          groups : c1
          capacity : 50000
          ...
          lb : LB_S2
```

```
        ;
        loadBalancer :
          name : 'LB_S2'
          service : S2
        ;
        ...
    }
    ...
}
...
```

Listing 6: Example of describing transformation states with StratModel

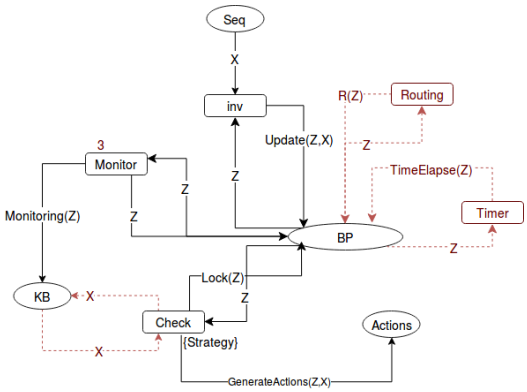


Figure 4: Elasticity Controller Petri Net Model Template

5.2. StratModel core

After describing an elasticity model using StratModel syntax, the StratModel document should be processed to generate its corresponding elasticity controller based on a pre-defined template that groups the common functionalities of a controller. In order to achieve this, StratModel language is provided with a set of functionalities constructing its core which includes: (1) a validator to check the coherence of the provided elements, (2) a scope provider, and (3) a generator that uses a pre-defined controller template grouping the common functionalities of a controller.

Usually, a controller is represented by a control loop to provide autonomic management which gives the system the ability to manage its resources automatically and dynamically whenever needed. This loop consists in (*i*) harvesting monitoring data, (*ii*) analyzing them using (optionally) a knowledge base and (*iii*) generating reconfiguration actions to correct violation (self-healing and self-protecting) or to target a new state of the system (self-configuring and self-optimizing) [34].

The pre-defined template as shown in Fig. 4 is modeled using high-level petri nets [30] to allow the formal evaluation of elasticity strategies on a SBP model [29]. It represents the basic construction of elasticity controllers on which the generation is based. It contains a central place *BP* of type net system that represents the managed component and surrounded with a set of transitions representing the actions to be performed on a SBP model (token in the place BP). The *Monitor* transition is used to trigger the monitoring of the SBP. It is guarded with a delay representing the frequency of monitoring. For example, if the *Monitor* transition is guarded with value 3, it means that there is three cycles between two successive monitoring actions. The firing of the transition adds new metrics values to the place *KB* representing the used knowledge base in which the history of monitoring metrics are stored. The *Check* transition is used to inspect an elasticity strategy and to check for QoS violations. It can optionally use information from the knowledge base component (*i.e.*, place *KB*) according to the defined elasticity model. The firing of the *Check* transition generates a set of actions to be applied according the used strategy and locks the entities on which the actions will be performed. The generated actions are stored in the place *Actions*. The *Inv* transition is used to introduce new requests to the SBP model from the place *Seq* which stores the sequence of requests arrival. According to the description of the elasticity model, two other transitions are optionally used from the template. The first one is the *Routing* transition which is responsible for routing requests between services in the SBP model. It can be omitted from the template in order to allow the user to use its customized routing action. The second one is the *Timer* transition which can be used and included in the template if the SBP model includes temporal information. It is used to increment the clocks in the SBP model.

Given an elasticity model, the elasticity controller petri net model is generated by enriching the pre-defined template with new transitions for the defined actions. Each action is translated to a transition where the name of the transition is the name of the action. It can be associated with a time delay of applying the action. The firing of the transition executes the action on the SBP model if the place named '*Actions*' contains some actions (tokens) that can be consumed by the transition.

Example 9. *Let's take the elasticity system described in Example 1. The elasticity model performs two main actions namely Duplicate and Consolidate. So, the default routing mechanisms is used by the generated controller. The elasticity strategies that will be used on the latter are reactive and they do not need a knowledge base for elasticity decisions. Fig. 5 illustrates the generated elasticity controller petri net model. We added two transitions to the template named Duplicate and Consolidate corresponding respectively to the action 'Duplicate' and the action 'Consolidate' in the elasticity model. Since the used SBP model contains temporal information, the Timer transition is allowed in the final model. The elasticity model also specifies that the default routing mechanism will be used and there is no need for a knowledge base in the analyzing step (i.e., Check transition).*

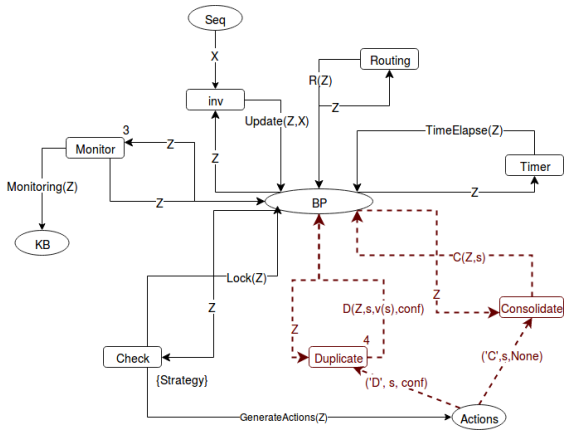


Figure 5: Example of a generated elasticity controller Petri Net Model

6. STRAT Language

Elasticity Strategy Description Language (STRAT for short) [26] has been proposed as a rule-based Domain Specific Language for specifying strategies governing SBP elasticity. It allows to specify QoS requirements of a SBP at different granularity levels (*i.e.*, process, service, and instance level) with taking into consideration the fundamental characteristics of SBP. STRAT is proposed based on a specific elasticity model that defines the actions that users can define to manage their process, the metrics that users can call in action's rules and the properties that users can access and reconfigure. In order to adapt the language to different elasticity models, we link STRAT to STRATModel that gives the description of the elasticity model that the STRAT language will be based on. So, a STRATModel script is required from the SBP holders before defining elasticity strategies to adapt STRAT to the described elasticity model.

6.1. STRAT *Grammar*

The top-level of STRAT specification grammar is given in Grammar 14 using the Backus Normal Form (BNF). A strategy STRAT is composed of two sections encapsulated in a block defined by `Strategy` (*i.e.*, indicates the beginning of the strategy) and identified by a name. The SBP holder is allowed to separate the rules section identified by `Actions` from the definition of constants sets used by the rules like thresholds sets and time constrains. This separation facilitates the adjustment and the maintenance of strategy's code. The latter is an optional section and is identified by `Sets`.

The `Sets` section as given in Grammar 15 could either be empty or consist of several constants sets. A set can be defined as a bound of a quality of service metric by indicating the represented metric, which refers to a metric defined in the provided elasticity model, and whether it is an upper bound or a lower bound. It allows the specification of the invariant requirements and characteristics of a SBP

⟨*ScalingPolicy*⟩ ::= '`Strategy`' ⟨*name*⟩ '{' ⟨*Statements*⟩ '}'

⟨*Statements*⟩ ::= ⟨*Initialization*⟩ ⟨*ActionsBlock*⟩
| ⟨*ActionsBlock*⟩

⟨*Initialization*⟩ ::= '`Sets`' ':' ⟨*Sets*⟩

⟨*ActionsBlock*⟩ ::= '`Actions`' ':' ⟨*Actions*⟩

Grammar 14: General STRAT Grammar

and its services such as the maximum and the minimum capacity of services in a SBP, time constraint, budget for deployment a service, *etc.* These invariant can be specified hierarchically from the top-level component, *e.g.*, a process, to its fine-granular level, *e.g.*, a sub-group of service's requests. This is done by using sets of sets providing the hierarchical construction. So, an item in a set is associated to either a value or another set specializing the item. The term `Default` is used in the latter case to provide the requirement (or characteristic) of the item in its top-level. Listing 7 presents an example of specifying the maximum execution time for two elastic services $s2$ and $s4$ according to their request groups $c1$ and $c2$. This specification allows the SBP holder to define elasticity action rules that incorporate the characteristic of the processing requests into the decision making. We are currently working on providing a formal model of SBP elasticity that allows the distinguishing of services requests by their characteristics.

⟨*Sets*⟩ ::= ⟨*Set*⟩ ';' ⟨*Sets*⟩ | ⟨*empty*⟩

⟨*Set*⟩ ::= ⟨*id*⟩ '=' '{' ⟨*Items*⟩ '}' ⟨*QoS*⟩

⟨*QoS*⟩ ::= '`as`' ⟨*Bound*⟩ [STRATModel::Metric] | ⟨*empty*⟩

⟨*Bound*⟩ ::= '`upper_bound_qos`' | '`lower_bound_qos`'

⟨*Items*⟩ ::= ⟨*Item*⟩ | ⟨*Item*⟩ ',' ⟨*Items*⟩

⟨*Item*⟩ ::= ⟨*id*⟩ '=' ⟨*value*⟩
| ⟨*id*⟩ ':' '{' ⟨*Items*⟩ '}' '`Default`' ⟨*value*⟩

Grammar 15: Grammar of defining Sets in STRAT

```
Strategy ...{
Sets:
  max_ex = {s2:{c1=65, c2=120} Default 70,
            s4:{c1=25, c2=75} Default 30}
  ...
Actions:
  ...
}
```

Listing 7: Example of defined maximum execution time Set with STRAT

⟨*Actions*⟩ ::= ⟨*Action*⟩ ':' ⟨*Rules*⟩ '.'
| ⟨*Action*⟩ ':' ⟨*Rules*⟩ '.' ⟨*Actions*⟩

⟨*Rules*⟩ ::= ⟨*Rule*⟩ | ⟨*Rule*⟩ ';' ⟨*Rules*⟩

⟨*Rule*⟩ ::= ⟨*Condition*⟩ ⟨*Operator*⟩ ⟨*Condition*⟩
| ⟨*Condition*⟩

⟨*Action*⟩ ::= [StratModel::Action] '(' ⟨*id*⟩ ⟨*Copies*⟩ ⟨*Configuration*⟩ ')' ⟨*Multiple*⟩

⟨*Configuration*⟩ ::= ',' '[' ⟨*ConfigItem*⟩ ⟨*ConfigItems*⟩ ']'
| ⟨*empty*⟩

⟨*ConfigItems*⟩ ::= ',' ⟨*ConfigItem*⟩ ⟨*ConfigItems*⟩ | ⟨*empty*⟩

⟨*ConfigItem*⟩ ::= [StratModel::Property] ⟨*ConfigOps*⟩ ⟨*ItemValue*⟩

⟨*Multiple*⟩ ::= 'by' ⟨*int*⟩ | ⟨*empty*⟩

⟨*Copies*⟩ ::= ',' ⟨*id*⟩ ⟨*Copies*⟩
| ⟨*empty*⟩

Grammar 16: Grammar of specifying Actions and their Rules in Strat

Under the `Actions` section, elasticity mechanisms could be provided by setting a set of rules grouped by their intended elasticity action as shown in Grammar 16. The actions allowed in Strat are the ones defined in the given StratModel script. So, by changing the provided script, the actions allowed in Strat change as well. Moreover, we provide the grammar with a syntactic validator and scope provider modules to adapt actions parameters according to their definitions in the used elasticity model by dynamically activate and deactivate parts of syntactic definition of `Action`. The parameters of an action are: (1) the elements on which the action will be applied, (2) the resulting configuration performed by the action, and (3) the multiplicity of applying the action. The first two parameters are recognized from the defined cases of the action while the multiplicity of the action is allowed if the action is defined as multiple. The configurable properties used in the second parameter are referred to the properties defined in StratModel as configurable.

Each provided action might have one or more rules ordered according to their priority. This means that for a specific action the first provided rule is the most priority one from all the action's rules, thereafter the next one is the second most priority and so on. A rule in Strat is composed of a set of conditions connected by logical operators (*i.e.*, `and`/`or`).

Conditions are the reflection of system's state at a certain point of time. They are split into boolean (*i.e.*, `true`/`false`), boolean function, iteration, comparison, negation, time-based condition, or another rule. The time-based condition allows the execution of an action after some period of time or after the persistence of system's state for some period using `for` to specify that period, *e.g.*, the rule "`Duplicate`(s): true `for` 60 min" schedules a duplication of each elastic service s in every "60 min". Unlike existing languages in the literature for controlling elasticity, we incorporate the concept of iteration into the definition of Strat in order to get a global view of the system's state. For example, to be able to determine the state of a service at a given time we have to check the state of all its copies so we get a global perspective instead of a local one (*i.e.*, for only one service's copy). To do so, `foreach` and `exists` operators are used to express respectively '∀' and '∃' symbols of the first-order logic. The `Sequence` element of an iteration represents either a function returning a list or a constant list. The specification of conditions in Strat is given by Grammar 17 using also the Backus Normal Form (BNF).

⟨*Condition*⟩ ::= ⟨*Boolean*⟩
| ⟨*Function*⟩
| ⟨*Iteration*⟩
| ⟨*Comparison*⟩
| ⟨*Condition*⟩ 'for' ⟨*value*⟩
| 'not' ⟨*Condition*⟩
| '(' ⟨*Rule*⟩ ')'

⟨*Comparison*⟩ ::= ⟨*Operand*⟩ ⟨*Ops*⟩ ⟨*Operand*⟩

⟨*Iteration*⟩ ::= 'foreach' ⟨*name*⟩ 'in' ⟨*Sequence*⟩ ':' ⟨*Condition*⟩
| 'exists' ⟨*name*⟩ 'in' ⟨*Sequence*⟩ ':' ⟨*Condition*⟩

⟨*Sequence*⟩ ::= ⟨*Function*⟩ | ⟨*List*⟩

Grammar 17: Grammar for Conditions in Strat

Example 10. *In Listing 8, we describe how the strategy used in our running example (cf., Example 1) can be defined using our* Strat *language. We indicate that the set 'max_t' contains the QoS thresholds for each service and group which is associated to the metric 'executionTime' defined in the elasticity model 'ElasticityModel1'. The Duplicate action is provided to allow to create a new copy for a specific group with a different capacity if the requests of that group with waiting status has been waiting for a certain amount of time. The waiting time thresholds are given in a set named 'max_w'.*

```
Strategy Strategy1{
  Sets :
    max_t = {S1:{ c1=12, ...} Default 195, ...}
            as upper_bound_qos executionTime
    min_t = {S1 = 1, ...}
          ...
  Actions :
```

```
    Duplicate(s,[cat='c3',cap+=600000]) :
      STRAT.has_group(s, 'c3') and
      (exists req in STRAT.waitingRequests(s,'c3')
        waitingTime(req) >= max_w[s]['c3']) and
      foreach ss in STRAT.copies(s) :
        (exists req2 in STRAT.waitingRequests(ss,'c3')
          waitingTime(req2) >= max_w[s]['c3']) .
                    ...
}
```

Listing 8: Describing an elasticity strategy using STRAT

6.2. STRAT Core

It groups the basic functionalities of STRAT which include: (1) a validator to check the coherence of the given rules, (2) a scope provider, and (3) a generator that processes the STRAT scripts and generate their corresponding output.

7. StratSim language

STRATSim is a simulation domain-specific language for specifying simulation properties/elements in a declarative manner and generating a simulator launcher. STRATSim document is composed of a set of (property, value) pairs that represents the parameters that customize the simulator to perform the wanted simulations.

⟨*Simulation*⟩ ::= ⟨*ElasticityModel*⟩ ⟨*Items*⟩ ⟨*Invocation*⟩ ⟨*OutputPath*⟩

⟨*ElasticityModel*⟩ ::= '`elasticityModel`' ':' [STRATModel::ElasticityModel]

⟨*Items*⟩ ::= ⟨*Item*⟩ ⟨*Items*⟩ | ⟨*empty*⟩

⟨*Item*⟩ ::= ⟨*Process*⟩ | ⟨*Strategy*⟩ | ⟨*UsageBehavior*⟩

⟨*Process*⟩ ::= '`process`' ':' [SBP::SBPModel]

⟨*Strategy*⟩ ::= '`strategy`' ':' [STRAT::ScalingPolicy]

⟨*UsageBehavior*⟩ ::= '`workload`' ':' ⟨*string*⟩ ⟨*Values*⟩

⟨*Invocation*⟩ ::= '`frequency`' ':' ⟨*int*⟩ | ⟨*empty*⟩

⟨*OutputPath*⟩ ::= '`output`' ':' ⟨*string*⟩

Grammar 18: General STRATSim Grammar

As described in the Grammar 18, SBP holders have to provide the elasticity model on which the simulations will be based on, by referring to the existing elasticity model. Along this the elasticity model, they can specify a set of items composed of processes defined using SBP language, STRAT strategies and workloads that represent different configuration of specifying the arrival law of process enactment requests. These items indicate to the simulator the series of simulations that have to be performed. Additionally, SBP holders can optionally indicate the frequency of process invocations which represents the distance between two process invocations. Finally, the 'output' property is used to specify the path in which the results will be saved.

Example 11. *Listing 9 presents an example of simulation script writen in* STRAT*Sim for evaluating the elasticity strategy 'Strategy*1*' on the SBP model 'MERProcess' using the elasticity model 'ElasticityModel*1*'.*

```
elasticityModel : ElasticityModel1
process : MERProcess
strategy : Strategy1
workload : '...generators.hpoisson' 4 100
frequency : 12
output : 'path'
```

Listing 9: Example of simulation scenario with STRATSim

8. Implementation and evaluation

We present in this section an overview of the implementation of STRATFram framework. We also provide insights on its use through two evaluation scenarios.

8.1. Implementation

The STRATFram framework[1] is an eclipse-based framework. It is designed and implemented on two parts: (1) the STRATFram languages part and (2) the STRATFram functions part.

The first part is developed using Xtext[2] an eclipse-based development framework for creating DSLs. Each designed language in STRATFram has its editor which provides user with code completion, syntax highlighting, automated parsing and quick fixes functionalities that facilities the edition of scripts. In addition to these functionalities, we provided each language with a scope provider that is customized according to the semantics of the language, along with a validator that is implemented to perform additional constraint checks.

In the current implementation of STRATFram, the second part of the framework is composed of a set of functions and classes designed and implemented based on SNAKES toolkit [35] a Python library that provides all the necessary to define and execute many sorts of Petri nets, in particular algebras of Petri nets. The functions and classes implemented in this part provide mainly the common functionalities of STRATFram that manipulate SBP models and are triggered by the generated elasticity controller. A set of functions are implemented as pre-defined functions for STRAT language that can be used in a strategy.

[1] STRATFram framework and the evaluation projects used in this paper can be found in the following link: `https://github.com/AichaBenJrad/STRATFram`

[2] `https://eclipse.org/Xtext/`

SBP	$\|Services\|$	$\|AND\|$	$\|XOR\|$
1	3	0	0
2	3	0	1
3	4	1	0
4	5	1	3
5	8	1	1

Table 1: Evaluation SBP models

8.2. Evaluation

In order to validate our framework, we present in this section two evaluation scenarios: the first one focuses on simulating the use of a strategy on publicly available SBPs using the same elasticity model, while the second one consists in two use cases for two different elasticity models using a SBP model for molecular evolution construction in which different elasticity strategies are evaluated and compared. Thereafter, we describe the preliminary results.

8.2.1. 1^{st} Evaluation Scenario

In this evaluation scenario, we choose to evaluate the elasticity of a set of publicly available process models selected from the SAP reference model [36], using an elasticity model described according to the elasticity controller used in [37]. The SAP reference model has been widely used in many research papers [38]. It contains 205 business process models. A node in a process model can be a function/service, an event or a gateway. In order to use the available process models in SAP reference model, we transformed the models to SBP models by eliminating the events and their connections and focusing on services/-functions. From the transformed models, we selected a set of 5 exemplary SBP models which feature different complexity degrees in terms of business process patterns. Table 1 shows the characteristics of each selected SBP models in terms of number of services in the SBP, number of AND-blocks and number of XOR-blocks. Fig. 6 illustrates two SBP models from the selected set of SBPs. The first model, named *SBP No. 3*, contains 4 services and one AND-block. The second model, named *SBP No. 5* contains 8 services and one XOR-block in which one branch leads to an AND-block.

For each service in the SBP models, we associate a capacity value randomly generated that indicates the maximum number of requests that can be handled by the service. So, in order to evaluate the elasticity of the SBP models, we choose to describe an elasticity model for the elasticity controller proposed in [37] which has been proposed for stateless SBP models and performs the following elasticity actions: (1) a *Routing* action which controls the way a load of a service is routed over the set of its copies, (2) a *Duplicate* action which creates a new exact copy of an overloaded service in order to meet its workload increase and (3) a *Consolidate* action which releases an unnecessary copy of a service in order to meet its

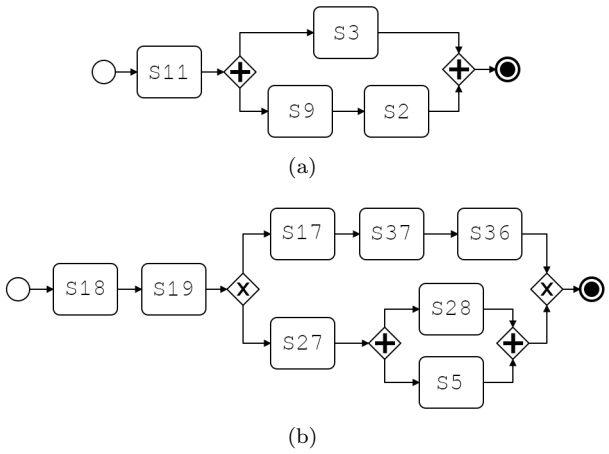


Figure 6: BPMN models of (a) the SBP No. 3 (top) and (b) the SBP No. 5

workload decrease. Along with these actions, we specify a '*capacityUtilization*' metric that provides the number of requests in a given service. In this evaluation scenario, we choose to define elasticity strategies for our SBP models that share the same set of rules while changing the defined threshold sets which have been specified according to the generated capacity values.

The strategy is defined to scale up or down the number of copies according to the threshold of each service. So, the duplication action for a specific service s is triggered when its workload (determined by the '*capacityUtilization*' metric) as well as the workload of all its copies reached its QoS threshold in the set max represented by the metric '*capacityUtilization*'. Otherwise, in case a service copy cp doesn't contain any request and the workload of the service s is below its minimum threshold defined in the set min, a consolidation action is triggered by releasing the service copy cp from the set of copies of s. The Routing action is defined to route a request if the router t does not exceed the capacity of its post-services. It is triggered when neither of the previous actions are allowed. Fig. 7 shows the described elasticity model and the used elasticity strategy. We use in this experiment the Poisson distribution for dynamically generating the sequence of requests arrival. We set the mean of Poisson distribution to 2;

By this scenario, we show how the same elasticity model and strategy can be reused for different SBP models without making any changes as long as the logic and the requirements are the same. The only change made in the strategy is the values in the defined sets which depend on the services in the managed process.

8.2.2. 2^{nd} Evaluation Scenario

Using StratFram interface, we create two projects where in each one we create an elasticity model using StratModel editor, a SBP model using SBP editor, three elasticity strategies using Strat editor, and a simulator script using StratSim editor. In both projects, we pro-

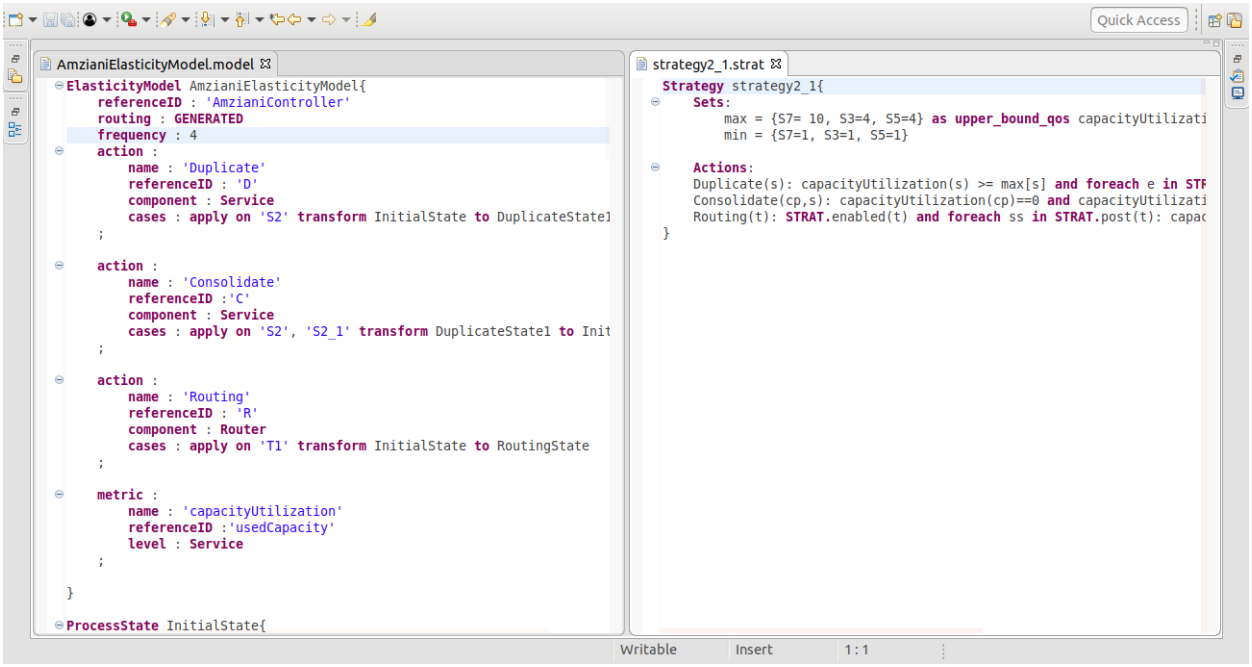


Figure 7: Elasticity model and strategy of the 1^{st} Evaluation Scenario

vide the SBP model for molecular evolution reconstruction (MER) described in Fig. 2 which is composed of eight services and six service out of eight are defined as elastic. We assume that each elastic service engine is provided with a maximum response time thresholds (execution time) depending on request groups as their required QoS given with the elasticity strategies. Above the maximum threshold, the QoS would no longer be guaranteed. We used the following QoS requirement in each elasticity strategy defined in the projects:

- Max_t = {S1:{c1=12,c2=102,c3=202},
  S2:{c1=10,c2=500,c3=1995},
  S5:{c1=4,c2=255,c3=1005},
  S6.1:{c1=4,c2=130,c3=1005},
  S6.2:{c1=4,c2=130,c3=1005},
  S7:{c1=100,c2=1005,c3=2005}}

1. ***Evaluation project 1:*** Along with the SBP model, this project is composed of the following elements :

   (a) The elasticity model on which SPEEDL language [20] is based is composed of four main actions: (1) Request scheduling that represents the mapping of a request to the next service engine in the process, (2) Request migration which re-maps an already existing request to another service engine copy, (3) scale-up that adds a new service engine copy, and (4) scale-down that remove unused service engine copy. With the set of actions, we define in this elasticity model '*executionTime*' metric that provides the age (execution time) of a given request. Fig. 8 illustrates the elasticity model script and its generated controller and actions implementation.

   (b) A strategy named `StrategyResponseTime` that defines rules for scaling-up, scaling-down, and scheduling of request. In this strategy, the conditions of elasticity actions do not consider the distinguishing between services requests and their different QoS requirements. So, we define a fixed maximum response time threshold for each elastic service engine regardless of request groups as an elasticity indicator. The scale-up action is triggered when the response time of at least one service request in each service engine copy has reached the maximum response time threshold. Otherwise, if the consumed capacity of a service engine copy is equal to 0 and the response time of the service is below its minimum threshold, a scale-down action is triggered by releasing the service engine copy.

   (c) A strategy named `StrategyWithMigration` that defines rules for scaling-up, scaling-down, scheduling of requests, and migrating of requests. In this strategy as well, the conditions of elasticity actions do not consider the distinguishing between services requests and their different QoS requirements. So, we use the same defined fixed maximum response time threshold. This strategies provides a rule for migration action in addition to the rules defined in the strategy `StrategyResponseTime`.

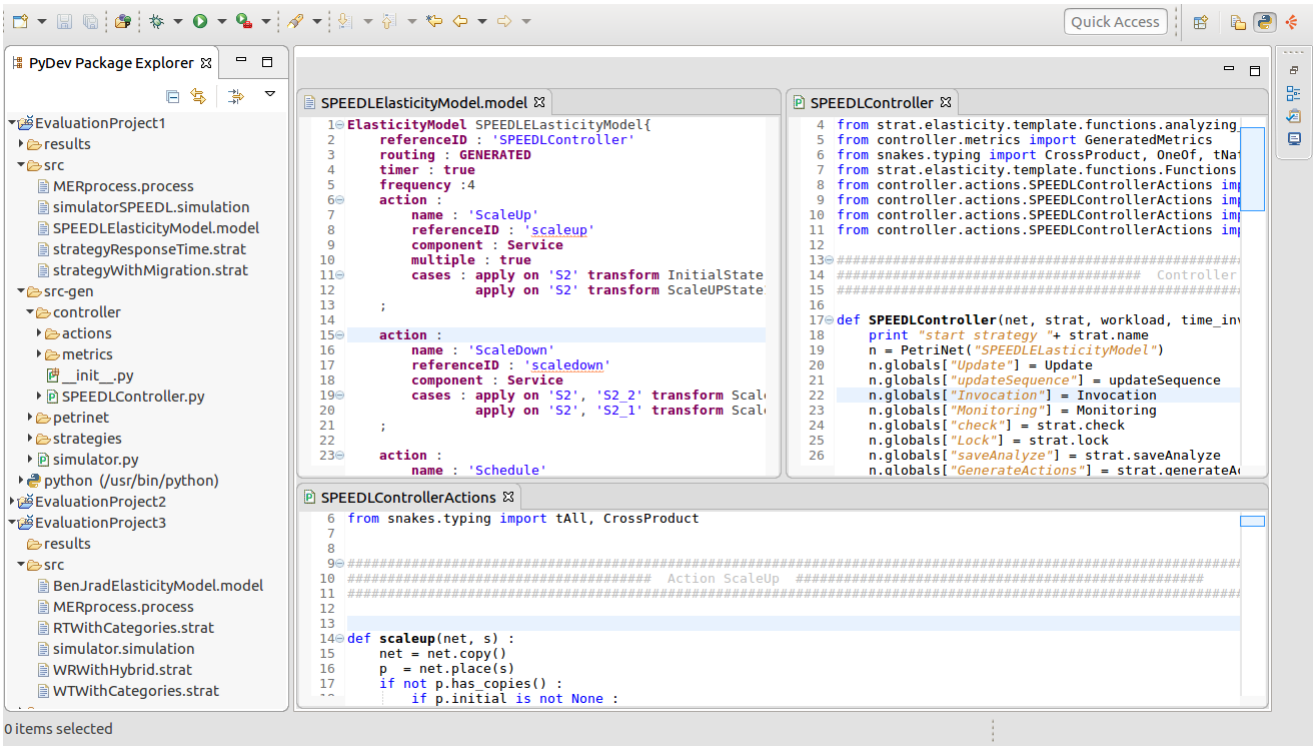


Figure 8: SPEEDL elasticity model and its generated controller and actions

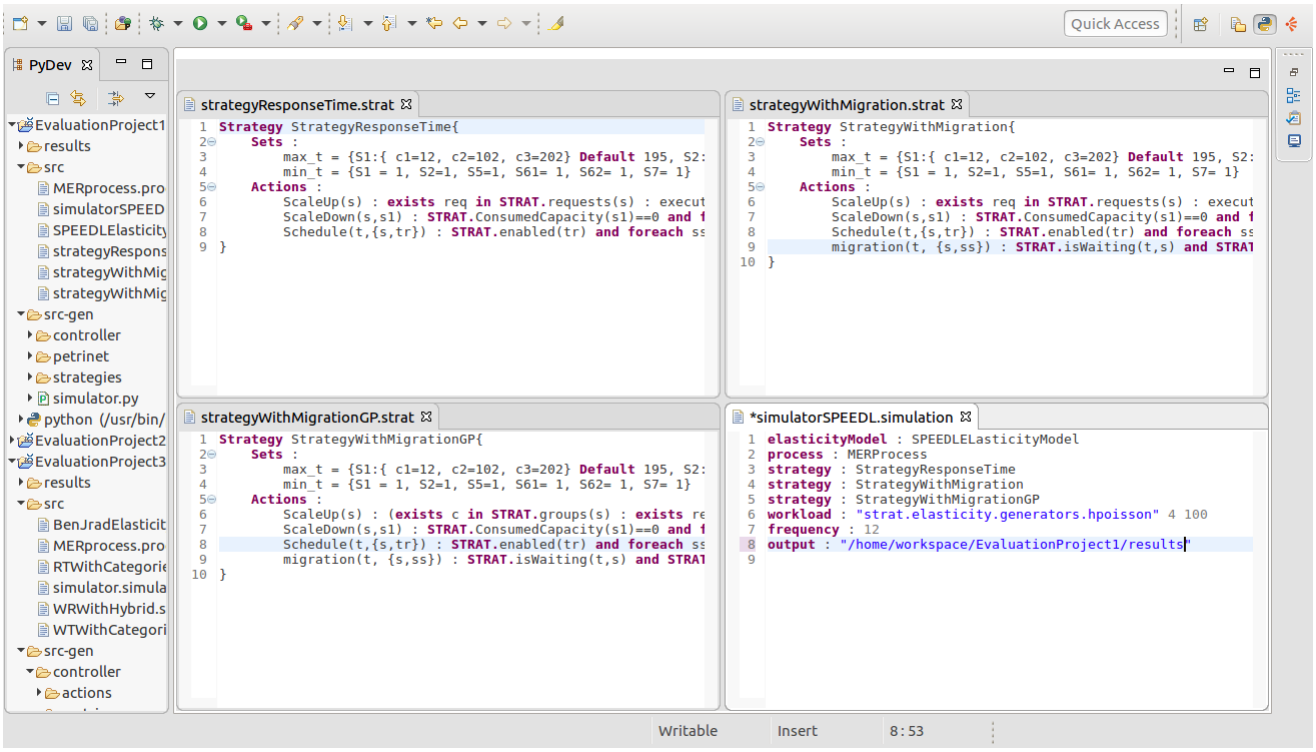


Figure 9: Elasticity strategies and simulation script for Evaluating project 1

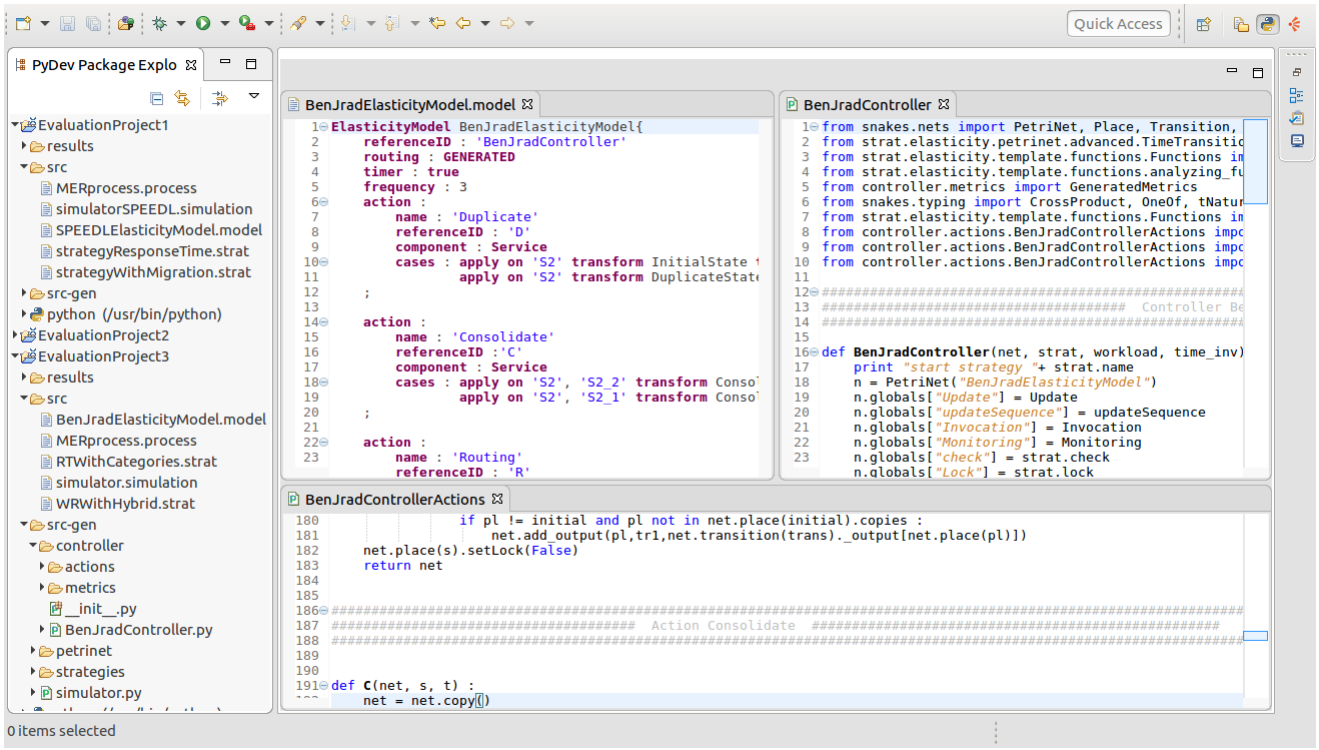


Figure 10: Jrad *et al.* elasticity model and its generated controller and actions

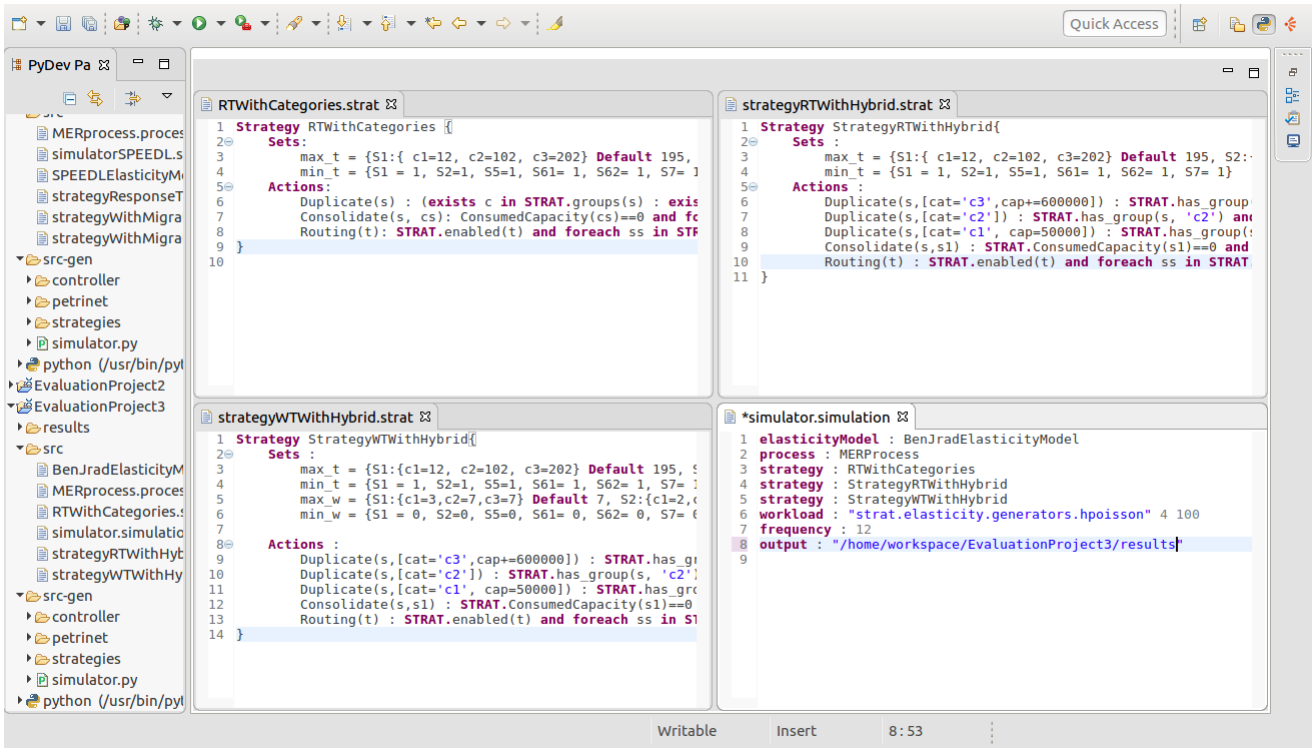


Figure 11: Elasticity strategies and simulation script for Evaluating project 2

(d) A strategy named `StrategyWithMigrationGP` that defines rules for scaling-up, scaling-down, scheduling of requests, and migrating of requests. Contrary to the previous strategies, this strategy uses the provided QoS requirement as an elasticity indicator to scale-up the service engine while uses the same rules as in the strategy `StrategyWithMigration` for the other actions.

(e) A simulation script that specifies that the three provided strategies should be evaluated and compared on the SBP model using the elasticity controller generated from the SPEEDL elasticity model and the usage behavior. Fig. 9 illustrates the defined strategies and the simulation script. We use the Poisson distribution for dynamically generating the sequence of requests arrival. We set the mean of Poisson distribution to 4;

2. ***Evaluation project 2:*** Along with the SBP model, this project is composed of the following elements:

(a) The elasticity model used in our previous work [29] that performs hybrid scaling. It is composed of three main actions: (1) duplication action that adds a new copy of service engine with different configuration (2) consolidation action that removes unused service engine copy, (3) routing action which transfers request from a service engine to the next service engine in the process. With the set of actions, we define in this elasticity model '$executionTime$' metric and '$waitingTime$' metric that provide respectively the age (execution time) and the waiting time of a given request. Also, we specify two properties as re-configurable namely the property 'groups' and the property 'capacity'. Fig. 10 illustrates the elasticity model script and its generated controller and actions implementation.

(b) A strategy named `RTWithCategories` that defines rules for duplicate action, consolidate action, routing action. It uses the provided QoS requirement as an elasticity indicator to duplicate the service engine and the minimum response time thresholds an indicator to consolidate a service engine copy. So, the duplication action for a specific service engine s is triggered when the maximum response time threshold has been reached for at least one of request groups and the same applied for all its copies. Otherwise, if the consumed capacity of a service engine copy is equal to 0 and the response time of the service is below its minimum threshold, a consolidate action is triggered by releasing the service engine copy.

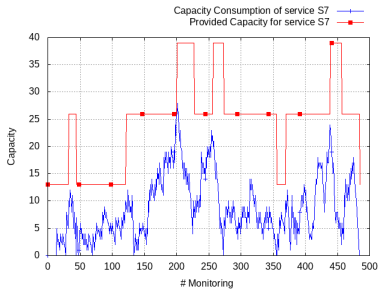
(c) A strategy named `StrategyRTWithHybrid` that defines rules for hybrid scaling that allows to add a new service engine copy with specific configuration whenever needed. So, the duplication action for a specific service engine is triggered depending on the request groups. It creates a new copy of a service engine for requests under a specific group if at least one of requests of that group exceeds the maximum response time threshold and the same applied to all its copies. The consolidation and routing rules are the same for all the strategies defined in this project.

(d) A strategy named `StrategyWTWithHybrid` that defines rules for hybrid scaling that allows to add a new service engine copy with specific configuration whenever needed based on the waiting time of requests. So, the duplication action creates a new copy of a service engine for requests under a specific group if at least one of waiting requests of that group exceeds the maximum waiting time threshold and the same applied to all its copies.

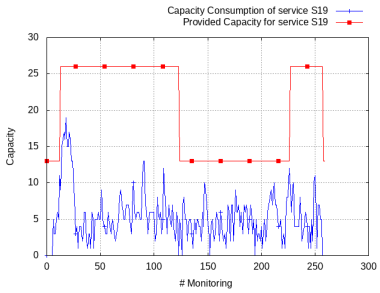
(e) A simulation script that specifies that the three provided strategies should be evaluated and compared on the SBP model using the elasticity controller generated from our elasticity model. Fig. 11 illustrates the defined strategies and the simulation script. We use the Poisson distribution for dynamically generating the sequence of requests arrival. We set the mean of Poisson distribution to 4;

8.2.3. Evaluation results

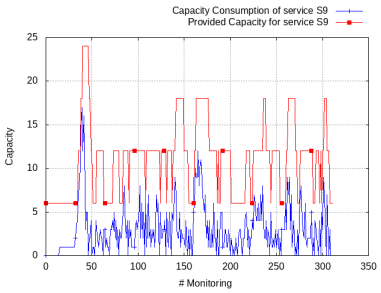
In order to evaluate elasticity strategies, we have defined some evaluation indicators that can be obtained from monitoring a given SBP model. In Fig. 12 and Fig. 13, we compare the amount of used capacity to the total provided capacity over time for a given service engine. The used capacity is computed by summing up the consumed capacity in each copy of the service engine. The provided capacity is computed by summing up the provided capacity for each service engine copy per strategy. The histogram in Fig. 14 illustrates the rate of QoS violation for each strategy in the 2^{nd} evaluation scenario. We compute at each monitoring cycle the number of requests violated the QoS requirement. Then, we compute the violation rate by dividing the number of violations by the total number of process requests. The resulted plots for the first evaluation scenario show how the same strategy logic can behave differently for different process models even when it is used for the same workload. We can see the perfect adjustment of capacity for services in the SBP No. 3 and the SBP No. 4 while for other processes the plots shows more unused capacity which may indicate the necessity to adapt the strategy logic to the specificity of the process by, for example, changing thresholds or adding some conditions. On the other hand, the resulted plots for the second evaluation scenario show how each strategy is performed for



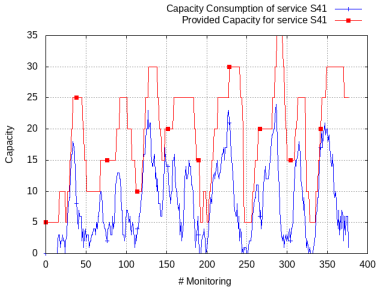
(a) SBP No. 1



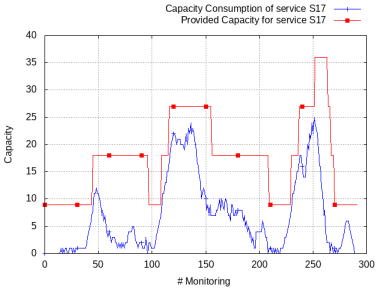
(b) SBP No. 2



(c) SBP No. 3

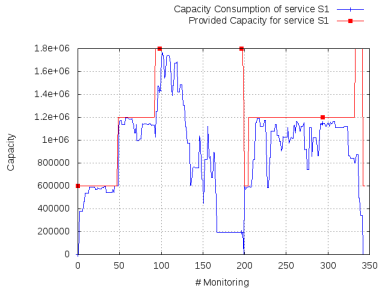


(d) SBP No.4

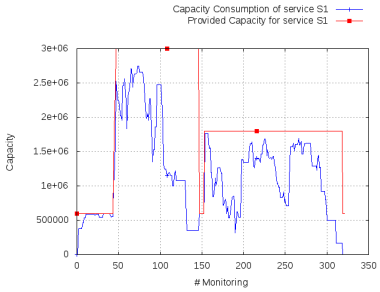


(e) SBP No. 5

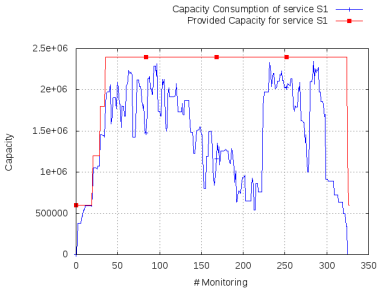
Figure 12: The evolution of capacity of a service in each SBP Models in the 1^{st} Evaluation Scenario



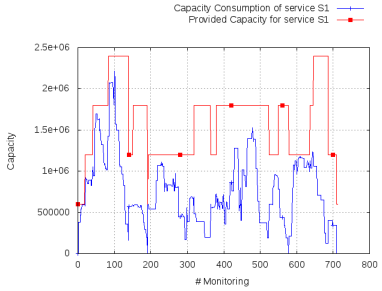
(a) Service S1 using StrategyResponseTime with SPEEDL elasticity model



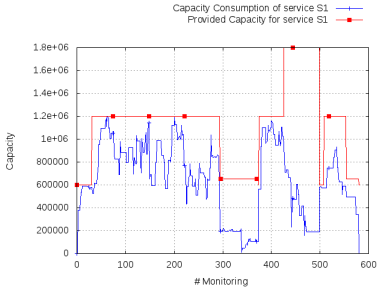
(b) Service S1 using StrategyWithMigration with SPEEDL elasticity model



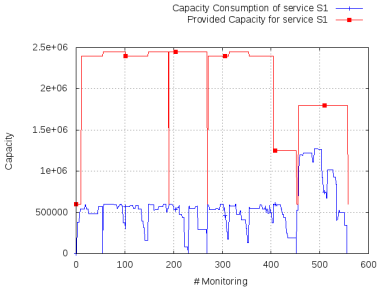
(c) Service S1 using StrategyWithMigrationGP with SPEEDL elasticity model



(d) Service S1 using RTWithCategories with Jrad *et al.* elasticity model

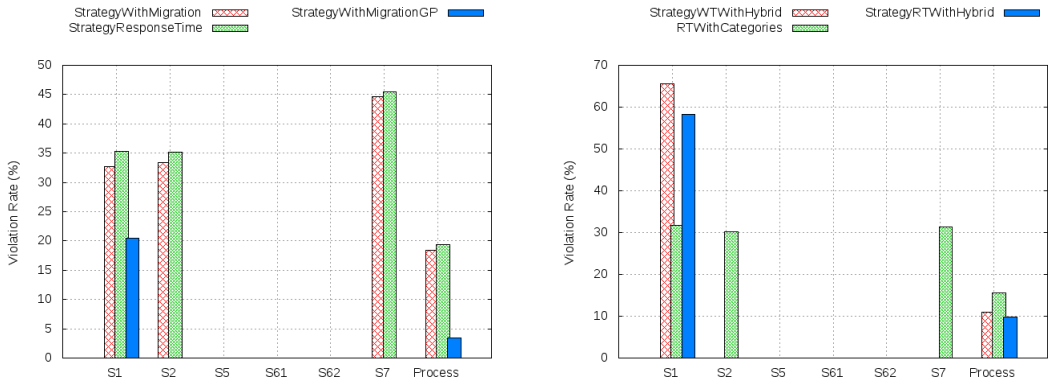


(e) Service S1 using StrategyRTWithHybrid with Jrad *et al.* elasticity model



(f) Service S1 using StrategyWTWithHybrid with Jrad *et al.* elasticity model

Figure 13: The evolution of capacity of SBP Model in the 2^{nd} Evaluation Scenario



(a) (b)

Figure 14: (a) Violation rate (%) in each elastic service engine and process using different strategies of evaluation project 1 (b) Violation rate (%) in each elastic service engine and process using different strategies of evaluation project 2

both elasticity models which allows SBP holders to not only compare strategies but also elasticity models providing them with insight on which one of elasticity models (or actions) is more suitable for their process and their QoS requirements. The SBP holders can observe the elasticity behavior of their processes by analyzing the difference between the allocated and the consumed capacity and the violation rate of each strategy for different elasticity model. This analyses allows to make decision on adjusting the defined thresholds or changing some conditions in the elasticity strategy.

9. Conclusion

In this paper, we presented a framework for describing and evaluating elasticity strategies for service-based business processes, called StratFram. StratFram enables SBP holders through a set of editors for different DSLs to define their proper elastic systems by providing their proper elasticity model along with their SBP models and strategies. It provides SBP holders with a set of languages that facilitate the description and the evaluation of elasticity strategies that can be based on different elasticity models. It has been designed in a way to separate the description part from the functional part of the framework. The description part which represented by StratFram languages is designed to conceal the complexity and the type of underlying techniques and systems used for the evaluation which makes the framework extensible to other techniques and systems without affecting the users interaction with the framework and the previously defined scripts.

As future, we aim to extend StratFram by providing other implementations for the functional part in order to SBP holders to evaluation their strategies not only using formal method such as petri nets but to simulate and analyze their performance using a business process engine such as Activiti[3] or evaluating them in real cloud environments as well using Open Cloud Computing Interface (OCCI) [39]. Moreover, we aim to provide our framework with a domain-specific model checker that considers the specificity of elastic SBPs.

[3] https://www.activiti.org/

References

[1] P. M. Mell, T. Grance, The nist definition of cloud computing, Tech. rep., National Institute of Standards & Technology, Gaithersburg, MD, United States (2011).

[2] IDG, Idg enterprise cloud computing study 2014, http://www.idgenterprise.com/report/idg-enterprise-cloud-computing-study-2014 (2014).

[3] N. R. Herbst, S. Kounev, R. Reussner, Elasticity in cloud computing: What it is, and what it is not, in: ICAC, 2013, pp. 23–27.

[4] G. Copil, D. Trihinas, H. L. Truong, D. Moldovan, G. Pallis, S. Dustdar, M. D. Dikaiakos, ADVISE - A framework for evaluating cloud service elasticity behavior, in: ICSOC, 2014, pp. 275–290.

[5] S. Islam, K. Lee, A. Fekete, A. Liu, How a consumer can measure elasticity for cloud platforms, in: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, ICPE'2012, Boston, Massachusetts, USA, 2012, pp. 85–96.

[6] Z. Zhou, W. Xu, D. Pham, C. Ji, Qos modeling and analysis for manufacturing networks: A service framework, in: Proceedings of the 7th IEEE International Conference on Industrial Informatics, INDIN'2009, Cardiff, Wales, UK, 2009, pp. 825–830.

[7] W. Xu, Z. Zhou, D. Pham, Q. Liu, C. Ji, W. Meng, Quality of service in manufacturing networks: a service framework and its implementation, The International Journal of Advanced Manufacturing Technology 63 (9-12) (2012) 1227–1237.

[8] J. W. Kenagy, D. M. Berwick, M. F. Shore, Service quality in health care, The Journal of the American Medical Association (JAMA) 281 (7) (1999) 661–665.

[9] P. Ray, G. Weerakkody, Quality of service management in health care organizations: a case study, in: Proceedings of the 12th IEEE Symposium on Computer-Based Medical Systems, CBMS'99, Stamford, CT, USA, 1999, pp. 80–85.

[10] A. Aktas, S. Cebi, I. Temiz, A new evaluation model for service quality of health care systems based on ahp and information axiom, Journal of Intelligent and Fuzzy Systems 28 (3) (2015) 1009–1021.

[11] K., Q. Wang, J. Hur, K.-J. Park, L. Sha, Medical-grade quality of service for real-time mobile healthcare, Computer 48 (2) (2015) 41–49.

[12] L. Liu, C. PU, K. Schwan, J. Walpole, Infofilter: Supporting quality of service for fresh information delivery, New Generation Computing.
[13] T. Smith, Quality of service requirements for system wide information management (swim), in: Proceedings of the 24th Digital Avionics Systems Conference, Vol. 1 of DASC'2005, Hyatt Regency Crystal City, Washington, D.C., 2005, pp. 1–8.
[14] A. Garcia-Recuero, S. Esteves, L. Veiga, Towards quality-of-service driven consistency for big data management, International Journal of Big Data Intelligence 1 (1-2) (2014) 74–88.
[15] R. Sandhu, S. K. Sood, Scheduling of big data applications on distributed cloud based on qos parameters, Cluster Computing 18 (2) (2015) 817–828.
[16] B. Suleiman, S. Venugopal, Modeling performance of elasticity rules for cloud-based applications, in: EDOC, 2013, pp. 201–206.
[17] A. Naskos, E. Stachtiari, P. Katsaros, A. Gounaris, Probabilistic model checking at runtime for the provisioning of cloud resources, in: RV, 2015, pp. 275–280.
[18] G. Copil, D. Moldovan, T. Hong-Linh, S. Dustdar, Sybl: An extensible language for controlling elasticity in cloud applications, in: CCGRID, 2013, pp. 112–119.
[19] K. Kritikos, J. Domaschka, A. Rossini, SRL: A scalability rule language for multi-cloud environments, in: Proceedings of the IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom'2014, Singapore, 2014, pp. 1–9.
[20] R. Zabolotnyi, P. Leitner, S. Schulte, S. Dustdar, Speedl - a declarative event-based language for cloud scaling definition, in: IEEEServices, 2015.
[21] A. Ali-Eldin, J. Tordsson, E. Elmroth, An adaptive hybrid elasticity controller for cloud infrastructures, in: NOMS, 2012, pp. 204–21.
[22] Y. Liu, D. Gureya, A. Al-Shishtawy, V. Vlassov, Onlineelastman: Self-trained proactive elasticity manager for cloud-based storage services, in: ICCAC, 2016.
[23] S. Farokhi, P. Jamshidi, E. B. Lakew, I. Brandic, E. Elmroth, A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach, Future Generation Computer Systems 65 (2016) 57–72.
[24] G. Molt, M. Caballer, C. de Alfonso, Automatic memory-based vertical elasticity and oversubscription on cloud platforms, Future Generation Computer Systems 56 (2016) 1–10.
[25] A. B. Jrad, S. Bhiri, S. Tata, Stratmodel: Elasticity model description language for evaluating elasticity strategies for business processes, in: On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE, Rhodes, Greece, 2017, pp. 448–466.
[26] A. B. Jrad, S. Bhiri, S. Tata, Description and evaluation of elasticity strategies for business processes in the cloud, in: SCC, 2016, pp. 203–210.
[27] S. Yangui, M. Mohamed, S. Tata, S. Moalla, Scalable service containers, in: CloudCom, 2011, pp. 348–356.
[28] D. Merkel, Docker: Lightweight linux containers for consistent development and deployment, Linux Journal 2014 (239).
[29] A. B. Jrad, S. Bhiri, S. Tata, Data-aware modeling of elastic processes for elasticity strategies evaluation, in: CLOUD, 2017.
[30] K. Jensen, G. Rozenberg, High-level Petri Nets: Theory and Application, Springer-Verlag, 1991.
[31] A. Cypher (Ed.), Watch What I Do – Programming by Demonstration, MIT Press, Cambridge, MA, USA, 1993.
[32] M. M. Zloof, Query by example, in: Proceedings of National Compute Conference, AFIPS Press, 1975, pp. 431–438.
[33] H. Lieberman (Ed.), Your Wish is My Command: Programming by Example, organ Kaufmann Publishers Inc., 2001.
[34] B. Jacob, R. Lanyon-Hogg, D. K. Nadgir, A. F. Yassin, A Practical Guide to the IBM Autonomic Computing Toolkit, IBM redbooks, IBM Corporation, International Technical Support Organization, 2004.
[35] F. Pommereau, SNAKES: A flexible high-level petri nets library (tool paper), in: Proceedings of the 36th International Conference on Application and Theory of Petri Nets and Concurrency, Brussels, Belgium, 2015, pp. 254–265.
[36] SAP R/3 Business Blueprint: Understanding the Business Process Reference Model, Inc. Prentice-Hall, 1998.
[37] M. Amziani, T. Melliti, S. Tata, Formal modeling and evaluation of stateful service-based business process elasticity in the cloud, in: CoopIS, 2013, pp. 21–38.
[38] J. Mendling, H. Verbeek, B. van Dongen, W. van der Aalst, G. Neumann, Detection and prediction of errors in epcs of the sap reference model, Data & Knowledge Engineering 64 (1) (2008) 312–329.
[39] R. Nyren, A. Edmonds, A. Papaspyrou, , T. Metsch, Open cloud computing interface - core, Technical report, Open Grid Forum (OGF) (2011).