applied sciences

MDPI

Article

Models@Runtime: The Development and Re-Configuration Management of Python Applications Using Formal Methods

Mohammed Mounir Bouhamed [1], Gregorio Díaz [2,*], Allaoua Chaoui [1] and Oussama Kamel [1,3] and Radouane Nouara [1]

[1] MISC Laboratory, Department of Computer Science and Its Applications, University Constantine 2 Abdelhamid Mehri, Constantine 25016, Algeria; mohammed.bouhamed@univ-constantine2.dz (M.M.B.); allaoua.chaoui@univ-constantine2.dz (A.C.); oussama.kamel@univ-constantine2.dz or oussama.kamel@univ-constantine3.dz (O.K.); redouane.nouara@univ-constantine2.dz (R.N.)
[2] Instituto de Investigación en Informática, Universidad de Castilla-La Mancha, 02071 Albacete, Spain
[3] Faculty of Medicine, University Constantine 3 Salah Boubnider, Constantine 25016, Algeria
* Correspondence: gregorio.diaz@uclm.es; Tel.: +34-650-29-95-87

Abstract: Models@runtime (models at runtime) are based on computation reflection. Runtime models can be regarded as a reflexive layer causally connected with the underlying system. Hence, every change in the runtime model involves a change in the reflected system, and vice versa. To the best of our knowledge, there are no runtime models for Python applications. Therefore, we propose a formal approach based on Petri Nets (PNs) to model, develop, and reconfigure Python applications at runtime. This framework is supported by a tool whose architecture consists of two modules connecting both the model and its execution. The proposed framework considers execution exceptions and allows users to monitor Python expressions at runtime. Additionally, the application behavior can be reconfigured by applying Graph Rewriting Rules (GRRs). A case study using Service-Level Agreement (SLA) violations is presented to illustrate our approach.

Keywords: models@runtime; python application; petri nets; formal methods; graph rewriting rules; application re-configuration; application management

Citation: Bouhamed, M.M.; Díaz, G.; Chaoui, A.; Kamel, O.; Nouara, R. Models@Runtime: The Development and Re-Configuration Management of Python Applications Using Formal Methods. *Appl. Sci.* **2021**, *11*, 9743. https://doi.org/10.3390/app11209743

Academic Editor: Alessandro Di Nuovo

Received: 13 September 2021
Accepted: 9 October 2021
Published: 19 October 2021

1. Introduction

This work is motivated by two interrelated necessities of software development: computational reflection and change control. Computational reflection, as described by Maes [1], is "*the activity performed by a computational system when doing computations about its own computation*". In other words, computational reflection is a program's ability to modify itself while running. Furthermore, software development is based on addressing two problems already stated in the 1970s by the first two of Lehman's laws: "*continuous change*" and "*increasing complexity*" [2]. These problems arise from the need for the software to be adapted to the new user requirements, leading to software changes that increase the complexity unless measures are taken. The interrelation between both concepts, computational reflection and change control, can be observed when we analyze the connection between the initial model designs and the software itself. They drift apart after changes are performed in the original code. This situation complicates monitoring and controlling the running instances of the developed software. Little research [3] has focused on bridging this gap by enriching the connection between models and code.

Therefore, the complexity is increased in two different aspects in this context: the change management and how to cope with the already running instances. Change management involves not only updating the software but also analyzing whether the new software version satisfies the new requirements. Another important aspect to be considered is those instances that are being used: How can they be updated to attend the new requirements? Traditional software development offers two different options: either wait until they finish

or kill their execution. The first option is not always available if we consider infinite behaviors, which is omnipresent in the sever–client architecture.

We propose to address these two issues by providing a middleware technology able to orchestrate the execution of a Python program using a Model-Driven Engineering (MDE) approach based on the Petri Net formalism bridging the gap between models and its execution. MDE aims to design complex software systems in terms of models to reduce the complexity of modern systems. Among the advantages of MDE, we can cite the higher productivity, abstraction, etc. In the literature, models are one of the best tools to analyze and verify software. For instance, Bucchiarone et al. [4] presented models@runtime as one of the challenges of MDE. It can analyze software at runtime when they cope with ever-changing software using models@runtime. Guérin et al. [3], as well as other works [5–8], used models@runtime as a technique for monitoring and changing systems. Hence, the main advantage of models@runtime is the use of an abstraction of the running software in the form of a model to enable its reconfiguration by changing its behavior while the software is still running. To achieve this goal, the approach must enable the technology to make reflections in both directions possible. It must also enable technologies to visualize and analyze the software at runtime. In this work, the proposed technology can change the behavior of a running Python program to attend new incoming requirements and to inspect the new execution to analyze whether these new requirements are being satisfied. In addition, the ability to interact with a running instance using its model means that it can be updated at runtime. The model proposed is based on one of the most used formalisms, Petri Net (PN), combined with Graph Rewriting Rules (GRRs), bringing mathematical rigor and reconfiguration capabilities, respectively.

However, this benefit comes at a cost. Developers need to acquire knowledge about the model used and the technology implementing the proposal. Furthermore, the software execution is more complex, because the model orchestrates the execution using the implemented middleware. Therefore, the execution performance will be affected by the time used in the communication process.

The contributions are the following:

- We extend the Petri Net (PN) formalism with Python statements that are executed when transitions are fired.
- We enrich these transitions with guards (Python conditions). Guards are Boolean expressions that evaluate true or false to check which transitions are enabled. The evaluation considers the runtime data of the Python program.
- The behavior of Python applications are modeled using the extended PN.
- We provide a tool to apply the approach with a Client/Server architecture (loosely coupled) to implement the models@runtime technique.
- Developers are able to monitor and evaluate the values of Python expressions at runtime.
- The behavior of applications can be changed by applying Graph Rewriting Rules (GRRs).
- We describe several examples, including a use case, to study the service reconfiguration in the context of Service-Level Agreement (SLA) violations [9].

2. Background

This section outlines the background concepts used in this work. First, the Python language is described, and we then focus on the PN formalism used in this paper to model Python applications. We also present Graph Rewriting, which is used to model the reconfigurations in Python applications. Finally, we explain the necessary concepts related to the models@runtime.

2.1. Python

Python (www.python.org, accessed on 1 August 2021) is a high-level general-purpose language. It is an interpreted programming language. It was first released in 1991 by Guido Van Rossum. It is currently considered one of the most popular programming languages. It is used in many areas, such as big data analysis and server-side programming. Server-side

executions may contain several running instances where the change of the running code may be inevitable due to the change in requirements [10]. Moreover, Python may cause runtime errors, as it is a dynamic typing language where the type of variable is associated at runtime. Therefore, change and monitor programs at runtime can alleviate these problems.

The present work is based on the use of two built-in Python functions, which are *exec* and *eval*. *exec* is a Python function that takes a piece of Python code as an input and executes it. For instance, when executing *exec('print(2 + 3)')*, Python prints *5*. *eval* is a Python function that evaluates a Python expression. For instance, the output of *eval('math.sqrt(25)')* is 5.

We illustrate the use of Python with an example that is part of the PROGRES platform. This software platform is used by the Algerian Ministry of Higher Education to manage student records, e.g., to assess their academic performance every year. In this example, we consider only the annual assessment. The Python application shown in Listing 1 encodes this assessment. First, it reads the averages obtained for both semesters. Students pass if they achieve an average of 50 or above (50 out of 100) in both semesters. This decision is evaluated in a variable called *decision*. The program will print "passed" or "failed" based on the value of this variable. Figure 1 shows the different dialogues obtained by the sequential execution of this example in the case of a student failing.

Listing 1. Grading example in Python.

```
from example import Grade_Frame
from example import Passed_Frame

grade_average_1 = Grade_Frame.enter_grade_average(1)
grade_average_2 = Grade_Frame.enter_grade_average(2)
decision = grade_average_1 >= 50 and grade_average_2 >= 50

if decision:
    Passed_Frame.show_decision(''Passed!'')
else:
    Passed_Frame.show_decision(''Failed!'')
```

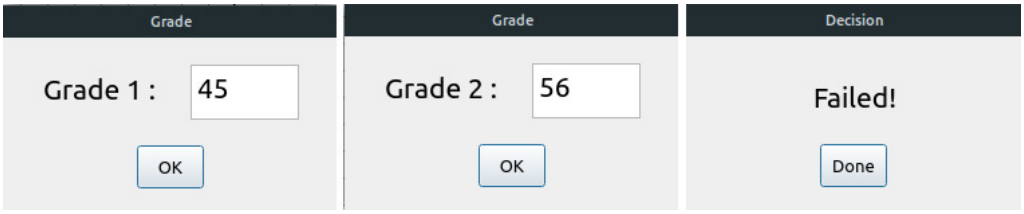

Figure 1. The sequential execution of the grading example.

2.2. Petri Nets (PN)

A PN [11] is a graphical formalism used for modeling the behavior of systems. It is a bipartite graph composed of two types of nodes, which are linked using weighted arcs. Nodes are places and transitions represented as circles and rectangles, respectively. Places can contain a finite number of tokens.

Definition 1. *Petri Net (PN)*
A PN is a tuple $\Gamma = (P, L, F)$*, where*

(i) $P = (P_0, P_1 \cdots P_m)$ *is a finite ordered set of places;*
(ii) $T = (T_0, T_1 \cdots T_n)$ *is a finite set of transitions;*
(iii) $F : (P \times T) \cup (T \times P) \longrightarrow \{0, 1, 2, \cdots\}$ *is a set of directed weighted arcs that represent flow relations.*

A marking m denotes the current state that is represented by tokens in each place of P. For instance, let $P = \{P_0, P_1\}$ be a set of places, and $m = [0, 2]$ denotes that P_0 and P_1 contain 0 and 2 tokens, respectively. In this paper, we represent this marking using $m = \{P_1, P_1\}$.

A transition T_l is enabled if and only if all its input places $P \in (P_{in} \times T_l) \longrightarrow j$ contain a sufficient number of tokens, j. Firing T_l, denoted as $m \xrightarrow{\Gamma, T_l} m'$, consumes and produces new tokens. Firing T_l consumes the corresponding j tokens from each input place and produces new tokens in the out places of T_l, respectively. The number of tokens produced is the weight of the arc $(T_l \times P_{out}) \longrightarrow k$. This firing leads to a new marking m' including the new tokens of every output place $\{P^1_{out}, P^2_{out}, \ldots P^k_{out}\}$.

Example 1. *Figure 2 depicts the initial structure of the grading example. Each arc has a weight equal to one.*

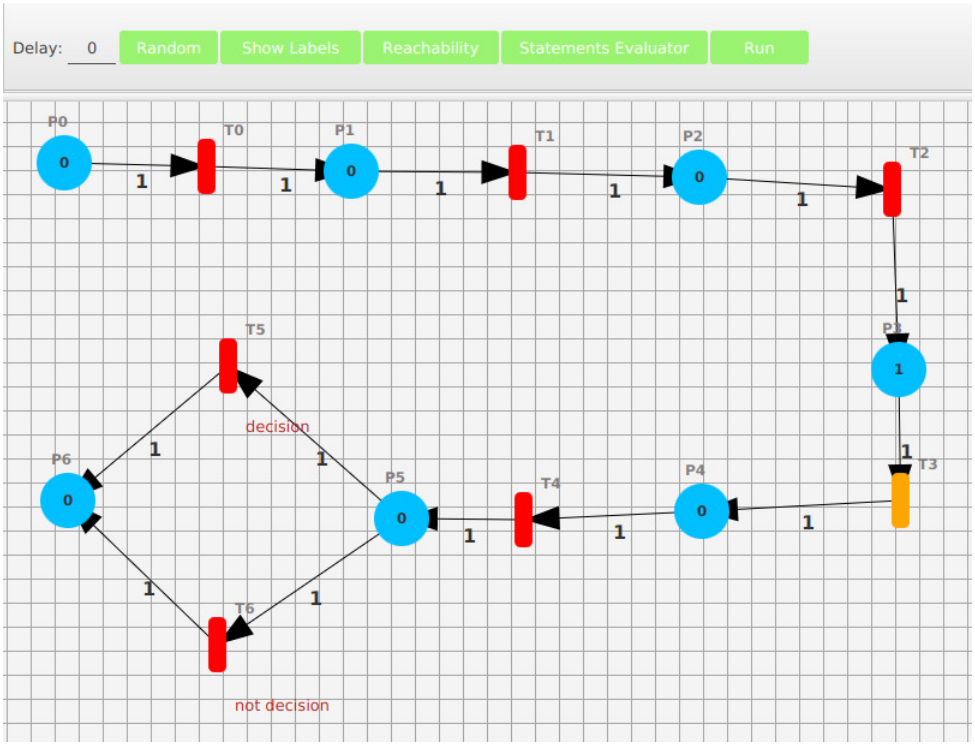

Figure 2. A snapshot of the tool showing a state of the grading example.

2.3. Graph Rewriting

Graph Rewriting [12,13] applies GRRs to rewrite the structure/flow of a graph—in this case, the PN structure.

Definition 2. *A graph G is a pair (V, E), where*

- *V is a set of nodes (i.e., vertices);*
- *E is a set of links (i.e., edges), which connect nodes.*

Definition 3. *A GRR r_i is a pair (LHS_i, RHS_i), where LHS_i and RHS_i are graphs that represent the Left-Hand Side (LHS) and the Right-Hand Side (RHS), respectively. Rewriting a graph G using r_i replaces a sub-graph LHS_i from G with RHS_i. The existence of LHS_i inside G is mandatory to apply r_i.*

Example 2. *The application of the GRRs r_0, depicted in Figure 3, on the PN from Figure 2, produces the PN illustrated in Figure 4.*

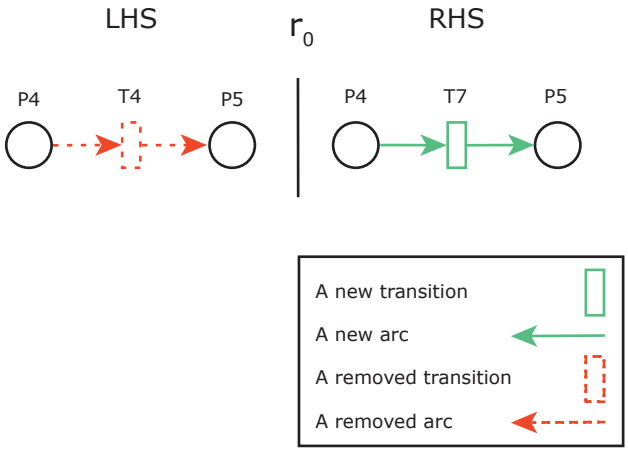

Figure 3. A GRR r_0 to replace transition T_4 with T_7.

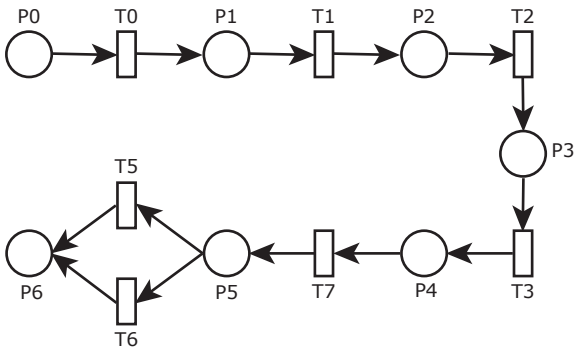

Figure 4. The rewritten structure of the grading example.

2.4. Models@Runtime

MDE constitutes a software abstraction paradigm, in their building, maintenance, and reasoning, to increase productivity. It is an iterative and incremental software development process based on the notion of a model. A model [14] is an abstraction of a system, where models@runtime [15,16] is a solution of a running system. The system and its corresponding model are causally connected, so the model is updated according to the current state of the system, and vice versa. "*A key benefit of using models@runtime is they can provide a richer semantic base for runtime decision-making related to runtime system concerns associated with autonomic and adaptive systems*" [16]. For instance, the PN presented in Figure 2 shows a state of the Python program shown in Listing 1, where the system has successively executed the first three instructions represented by T_0, T_1, and T_2 and is ready to execute the third, T_3. Orange represents an executable transition.

Thus, the adoption of the models@runtime will support the development of Python applications. The general principal of this paradigm is to offer an accurate model reflection of the running system at any moment. The models@runtime offers new techniques to deal with the dynamic adaptation of systems and satisfy the increasing complexity of user requirements. Therefore, in our work, it enables the dynamic re-configuration management of Python applications.

The models@runtime vision consists in the use of models not only at the design time but also during runtime. The underlying systems and their corresponding models evolve together and affect each other during the execution of these systems. The models@runtime paradigm enables the running systems to cope with the dynamic change of environments and satisfy the complex requirements of users.

3. Proposal

This section describes our proposed solution for models@runtime. It presents how to use our approach to develop Python applications and reconfigure their behavior at runtime. In addition, it shows how developers can inspect Python expressions at runtime. First, we present an overview of our proposal, providing detailed information about our framework. Next, we describe the process of developing a new Python application using our framework. Finally, we show this application can be reconfigured.

3.1. Antecedents

In software development, a model is an abstract representation of a software system and its environment. Models are primarily used for *documentation* and *communication* purposes in the software life cycle. Model-Driven Engineering (MDE) increases the importance of the notion of models because they are considered central artifacts in the development process. One of the challenges of the MDE community is to use models, as central artifacts, at runtime, to cope with dynamic aspects of ever-changing software and its environment [4], which inspired the notion of models@runtime.

Research works on models@runtime seek to extend the applicability to the runtime environment of models produced in MDE approaches [16]. Models@runtime cope with the computational reflection bridging the gap between these domains, models, and runtime execution. Models@runtime can be regarded as a reflexive layer causally connected with the underlying system. Hence, every change in the runtime model involves a change in the reflected system, and vice versa [16].

The necessity of monitoring systems through runtime models is presented in many works [3,5–8]. Recent works, such as [6,17,18], address the problem of reconfiguring and changing the behavior of systems at runtime. Although, to date, little research has focused on providing generic tools that are independent of the application domain. To the best of our knowledge, no work has considered providing tools for models@runtime for Python applications. Python developers are obliged to verify the behavior of their applications and change their execution when requirements change.

For instance, regarding the grading example, the method to determine successful students may unexpectedly change. For instance, many universities have changed their assessment methods during the SARS-CoV-2 pandemic (COVID-19). In Algeria, for example, a compensation system is used considering the average of both semesters. Thus, a pass is awarded to students with a total average for both semesters of 50 or above. Developers need to adapt the program to this new requirement. In an ordinary software development process, developers have to stop the execution, change the model, generate the code, and re-start the execution. However, using the models@runtime technique, developers could perform this change without stopping the execution. That is, they would be able to execute models using an execution engine where an engine reflecting the software execution enables the developer to monitor the execution. If the execution engine is able to automatically perform the change in the running instance, developers can monitor the change itself. In addition, a better connection between the design model and the execution can provide certain advantages, such as being able to see the execution state in order to inspect the value of variables before and after the change. Another benefit can be achieved by the adoption of a loosely coupled solution between the model and the execution, which can improve the system flexibility and allow for its deployment in distributed systems.

Table 1 shows some of the previous works addressing the open challenges left by them. We can observe that there is a lack of works addressing the problem when Python programs are considered, and few of them are able to reflect the running system.

Table 1. Open challenges in previous works.

Approach	Issues
Kamel et al. [9]	The proposal analyzes the dynamic behavior of SLA in cloud environments, but the authors do not provide a tool implementing the proposed approach.
PAMELA Guérin et al. [3]	It is only related to Java.
Llorens and Oliver [12]	It does not reflect changes between the model and the system at runtime.
Brand and Giese [5]	The focus is on the system architecture and not on changing the behavior at runtime.
Criado et al. [7]	The objective is to develop a smart architecture at runtime. Therefore, it is not focused on performing changes at runtime.
Tankovic and Grbac [8]	The proposal domain is not general, because it is centered on interpreting information systems.
Valero et al. [19]	The proposal is focused on event systems, and it does not deal with unpredictable requirements.
Leroy et al. [20]	The focus is on runtime monitoring and not on changing the behavior at runtime.
Meghzili et al. [21]	The proposal studies the verification of model transformations and does not deal with unpredictable changes.
Kerkouche et al. [22]	The authors deal with the transformation of UML models, and unpredictable changes are not considered.

3.2. Proposal Overview

The proposed framework comprises the two components depicted in Figure 5: the Model Execution Engine (MEE) and the Python Execution Environment (PEE). The former executes the application model, which consists of a PN and a set of GRRs. The latter executes the instructions (statements) that reflect this model.

The MEE fires an enabled transition whenever the model has enabled transitions. Transitions are square boxes labeled with a name and extended with two entities:

(i) a statement that should be executed by the PEE, and
(ii) a guard that should be satisfied to fire this transition.

The MEE invokes the statement by sending it to the PEE. The latter executes the instruction, then notifies the MEE whether the instruction is correctly executed or not. The MEE reflects, at runtime, the marking, the history of fired transitions, and the states of transitions in the model. The user is able to analyze the reachability of the application structure based on the current marking. The color of each transition changes depending on its state: enabled (green), disabled (red), or running (orange).

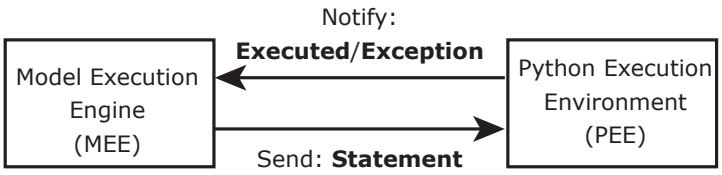

Figure 5. The framework architecture.

The framework supports the creation of new applications and the reconfiguration of running applications. The process of creating a new application, depicted on the left-hand side of Figure 6, starts in the requirement elicitation phase, that is, when the initial requirements are specified. Programmers develop the classes/functions that are invoked by transitions. They model the initial structure as a PN and enrich transitions with the statements and guards to compose the whole Python application. Finally, they execute the application using the MEE. Once the application is running, the monitoring process starts. During this process, developers can inspect Python expressions. In addition, the framework is able to deal with Python exceptions.

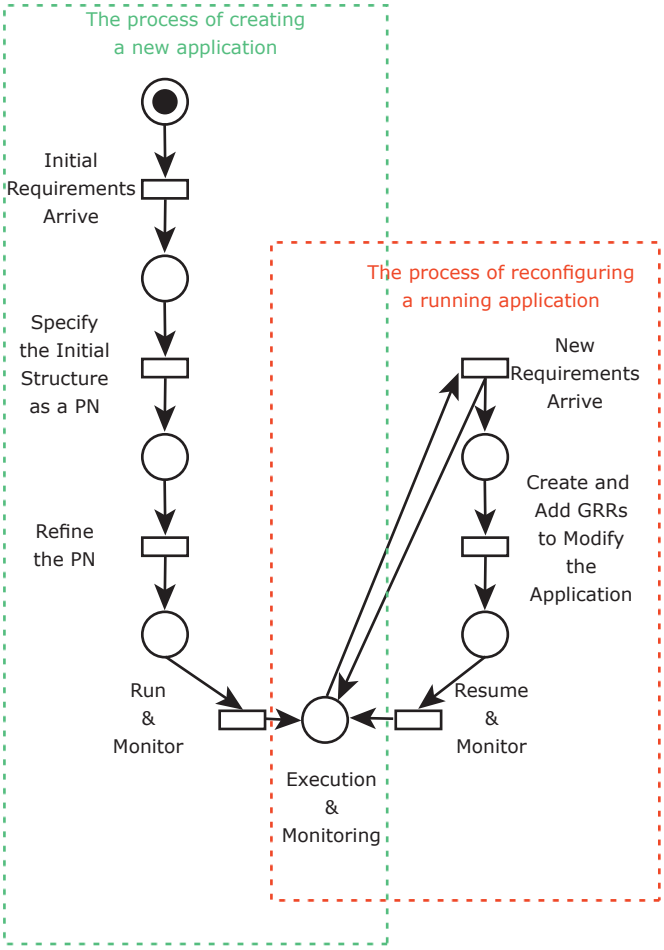

Figure 6. Process of developing and re-configuring an application.

The right-hand side of the figure shows the reconfiguration process, which starts after the execution begins, when a new requirement materializes. Next, the developers add the corresponding GRRs to rewrite the current structure and, therefore, satisfy the new requirement. For instance, if the new requirements include executing new statements, then developers must associate them with new transitions in the RHS and specify in which part of the PN model those statements are injected, that is, create the LHS. This process is similar to the creation of the initial model.

Through this paper, we assume that developers use built-in instructions and functions they define themselves in transitions. In addition, we assume that their execution will terminate correctly or throw exceptions. Invoked functions may contain control structures such as loops and if–else statements. The transitions guards are Python Boolean expressions, where a programmer's functions can also be used.

3.3. Framework Architecture and Behavior

The framework components exchange messages through TCP/IP sockets. The MEE is used to model a Python application, execute the model, and reconfigure the application. The MEE is used to model the structure of the application as a PN. In the next example, we introduce a transition enriched with a statement and a guard.

Example 3. *Transition T_5 from the grading example depicted in Figure 2 is enriched with the statement* `Passed_Frame.show_decision("Passed!")` *and the guard* `decision`*. The statement calls the function* ***show_decision("Passed!")*** *from the Python file* ***Passed_Frame****. The guard ensures that this statement can only be executed when the variable* ***decision*** *is True.*

The MEE sends the transition guards to the PEE to be reevaluated using the Python instruction *eval* any time the marking of the PN changes. The PEE answers the MEE with the set of guards satisfied, and the MEE sends the transition statements to the PEE when a transition is fired, and then awaits to receive the execution outcome.

The MEE supports both automatic and manual firing of transitions. When firing a transition, it follows the protocol depicted in Figure 7. This protocol is modeled as a PN. The MEE sends the corresponding statement to the PEE when firing a transition. It waits for a response that can be either an "EXECUTED" or an "EXCEPTION" message. In the former case, the MEE updates the marking. In the latter case, it restores the marking to the original state and opens a popup dialog showing the exception thrown. The color of transitions can be green, red, or orange to denote the states of enabled, disabled, or running, respectively.

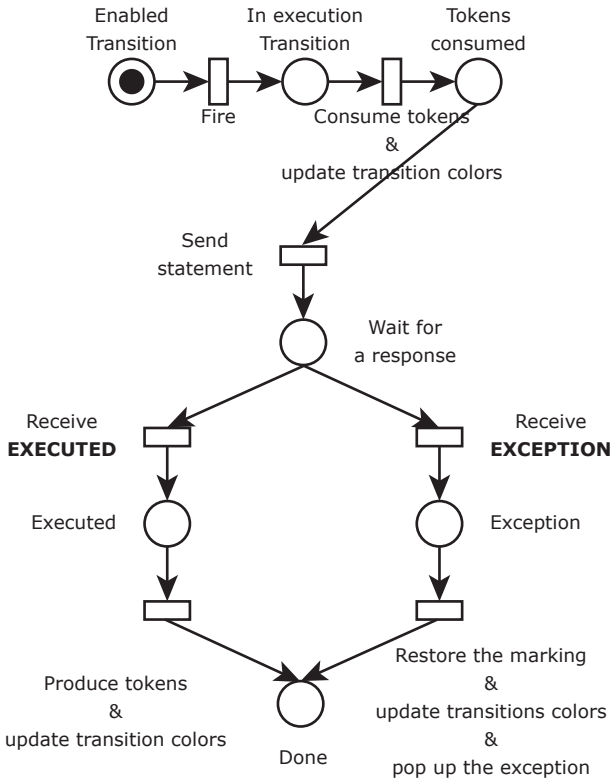

Figure 7. The protocol followed by MEE when firing a transition.

The PEE is a Python program implementing server features. It receives guards and instructions, and returns the outcome of their evaluations and executions, respectively, to

the MEE. For instance, when the PEE receives an instruction, it executes the instruction. If the instruction is executed without an exception, the PEE sends an "EXECUTED" message. Otherwise, it sends an "EXCEPTION" message concatenated with the exception content.

Listing 2 shows the Python function to be executed when receiving an instruction. It uses the *exec* command inside a *try except* block to execute the received instruction.

Listing 2. PEE basic code.

```
def execute(instruction):
    try:
        exec(instruction)
        reply ('' EXECUTED'')
    except Exception as e:
reply ('' EXCEPTION:'' + str(e))
```

In contrast, in the evaluation using the instruction *eval* when an exception is raised, we assume that the evaluation has not succeeded and the guard is not fulfilled. In this case, we consider that the transition is disabled. We follow this strategy because expressions may eventually be evaluated as true, once, for example, the variable used in the expression is defined or used (dynamic typing).

3.4. The Development, Reconfiguring, and Monitoring of Python Applications

We show how a Python application is developed using the proposed framework with the following example based on the grading program.

Example 4. *First, we need to develop the appropriate functions for the grading example. Subsequently, we can enrich the transitions of the PN model in Figure 2 with the statements and guards shown in Table 2.*

Table 2. Statements and guards of the grading example transition.

Transition	Statement	Guard
T_0	from example import Grade_Frame	True
T_1	from example import Passed_Frame	True
T_2	grade_average_1 = Grade_Frame.enter_grade_average(1)	True
T_3	grade_average_2 = Grade_Frame.enter_grade_average(2)	True
T_4	decision = int(grade_average_1) >= 50 and int(grade_average_2) >= 50	True
T_5	Passed_Frame.show_decision("Passed!")	decision
T_6	Passed_Frame.show_decision("Failed!")	not decision

Figure 2 illustrates how the framework enables users to start the execution by using the green button labeled with *Run*. This action enables the monitoring process shown in Figure 2. This figure shows the application state during the execution of transition T_3. At this point, the application awaits the value for the second assessment.

Example 5. *We assume that a new requirement specifies that students pass their exams when they obtain an annual average of 50 or above, otherwise failing. This new requirement is modeled as GRR r_0 from Figure 3. r_0 replaces T_4 with the new transition T_7. T_7 is enriched with the statement and guard from Table 3. The resulting structure after applying r_0 is presented in Figure 4. The sequential execution of the modified application is depicted in Figure 8.*

Table 3. Statements and guards of new transitions of the grading example.

Transition	Statement	Guard
T_7	decision = (grade_average_1 + grade_average_2)/2 >= 50	

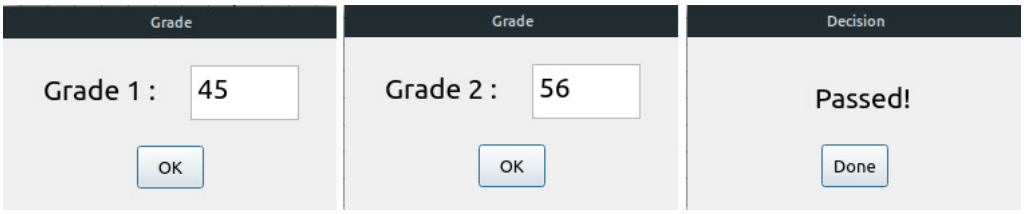

Figure 8. The sequential execution of the grading example after the reconfiguration.

The framework developed enables users to evaluate Python expressions at runtime. It uses the *eval* instruction to evaluate those expressions. These expressions are reevaluated every time the model is updated.

Example 6. *Figure 9 shows the expression evaluator where several expressions are under inspection. For instance, the variable* ***grade_average_1*** *is evaluated to 45.*

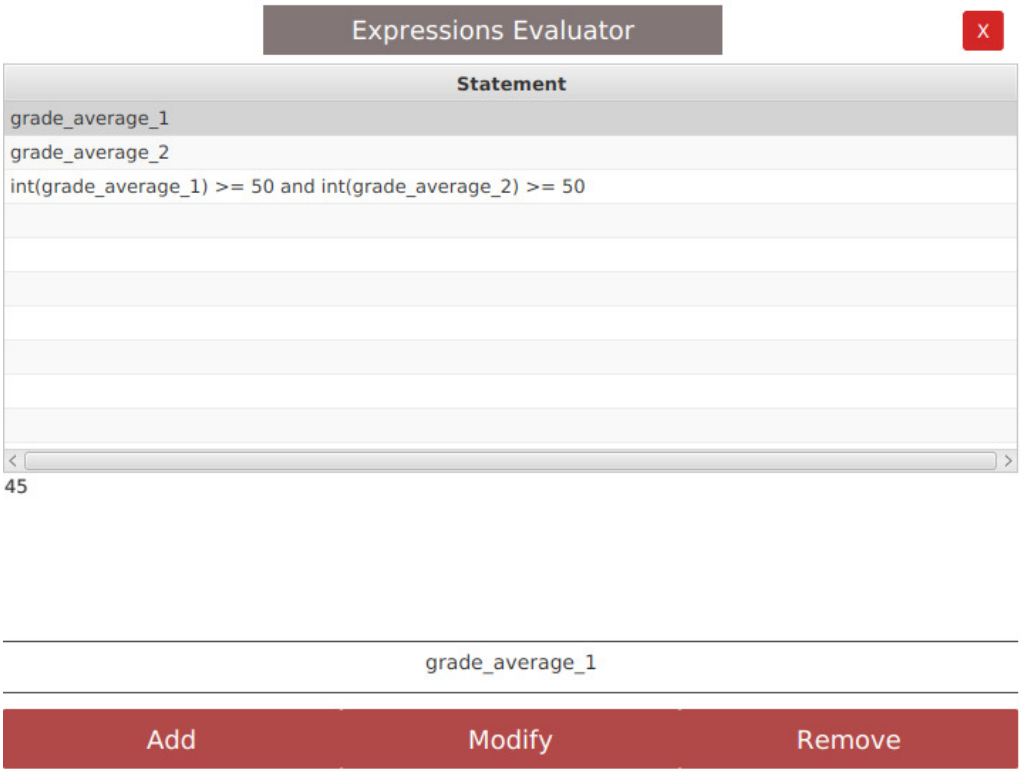

Figure 9. The expression evaluator.

Our framework deals with exceptions that may occur. These exceptions may appear because of typos in statements, or because of a disruption when executing the statements. For instance, Figure 10 shows a dialog that appears when the user enters a String instead of an Integer as a grade.

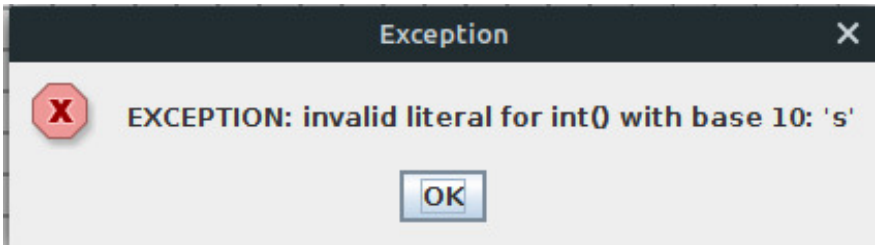

Figure 10. Exception popup.

4. Performance Evaluation

The complexity of the model and the change to be applied determine the performance of our proposal. Therefore, we have designed three sets of experiments to determine how these parameters affect the throughput, where nodes for both the PN model and GRRs are generated automatically. Every PN model includes the same number of places and transitions, and each transition is connected to each place in both directions. Regarding GRRs, the node number in the LHS depends on the experiment objective. In the first set, we analyze how the performance is affected by the model size. The second analyzes the effect of the GRR size, while the third compares both aspects. The three experiments show the performance of our approach studying the consumed time for the graph rewriting process. It includes the time taken to find the LHS in the original model and the time taken to change this structure by the RHS. Other aspects related to the interconnection between the MEE and the PEE are not considered, because the technology used here, TCP/IP sockets, relies on the delays introduced by the underlying infrastructure. The experiments have been performed in a PC with an Intel© Core™ i7-6700HQ CPU at 2.60 GHz with 4 cores, with 16 GB of internal memory, and running a Linux Mint 20 Cinnamon, version 4.6.7 (Linux Kernel 5.4-84-generic).

Figure 11 analyzes the time required to rewrite a model with a GRR consisting of six nodes for both the LHS and the RHS. In the first graph, we have considered an increasing number of nodes in the model from 10 to 800 nodes with intervals of 10 nodes. The LHS structure is always located at the end of the model. Experiments are performed 10 times to alleviate the effect of multitasking, and the figure shows the average rewriting time. Readers can observe a constant increasing in the rewriting time, as the number of nodes is higher. For those systems working in real-time environments, times shorter than 1 s should be considered (see the left hand side graph). Therefore, the recommended model size in this context is smaller to 200 nodes, taking into consideration the hardware configuration used here. In the second graph (right hand side), we have extended the study to find the trend line that shows a polynomial curve with quadratic equation $y = 0.001x^2 - 0.225x + 17.448$ with a significance of $R^2 = 0.981$ (red dotted line). This result confirms the complexity of the algorithm used to implement the rewriting process with order $O(n^2)$. We also show the cubic equation with the best fitting ($R^2 = 0.999$—black dotted line). This value is obtained because the initial values are close to zero.

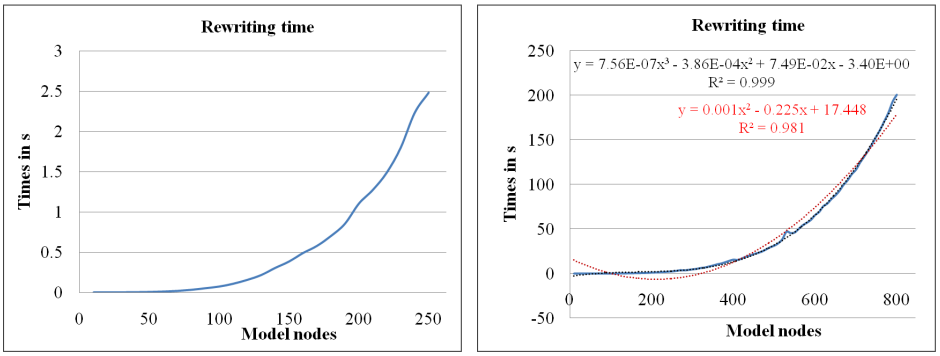

Figure 11. Analysis of the model size.

The second set of experiments analyzes the size of the applied GRR considering different sizes of the LHS structure. The first graph of Figure 12 depicts this scenario where the size of the LHS structure varies from 6 to 80 nodes in intervals of 2 nodes, while the size of the model is always 160. The LHS structure is always located at the end of the model. Each experiment has been performed 10 times, and the figure shows the average value obtained for each case. We can observe behavior similar to that in the previous scenario, and real-time scenarios should consider LHS structures under 48 nodes under this hardware configuration (left-hand side graph). In the second graph (right-hand side), we have extended the study to find the trend line that shows a polynomial curve with quadratic equation $y = 0.0008x^2 - 0.0305x + 0.7331$ with a significance of $R^2 = 0.9855$ (red dotted line). This result confirms again the complexity of the algorithm, order $O(n^2)$, with a similar result for the cubic equation ($R^2 = 0.9988$—black dotted line).

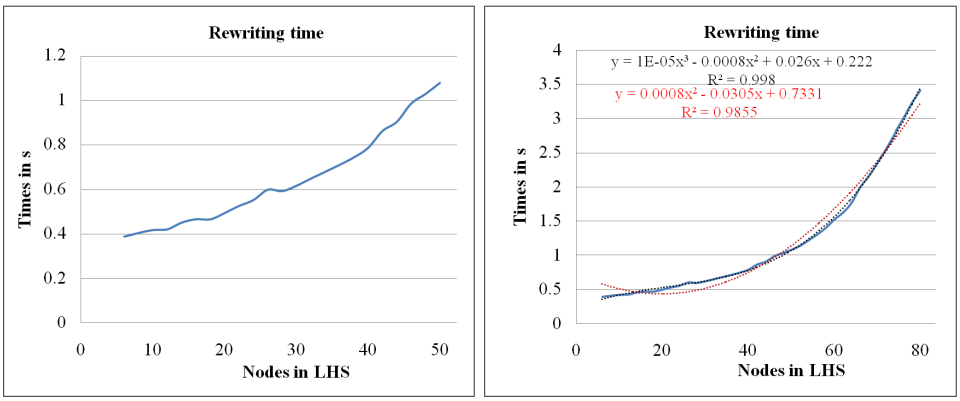

Figure 12. Analysis of the GRR size.

In the last set of experiments, we have studied the correlation between the model size and the GRR size. Figure 13 shows this analysis where the model size increases from 10 to 200 in intervals of 10 nodes. We have considered five sizes of GRR in this study with 10, 20, 30, 40, and 50 nodes for the LHS. We can observe that the rewriting time increases, as the size of the LHS increases. The maximum size of the model is 130 nodes to obtain a real-time response in all cases for this hardware configuration.

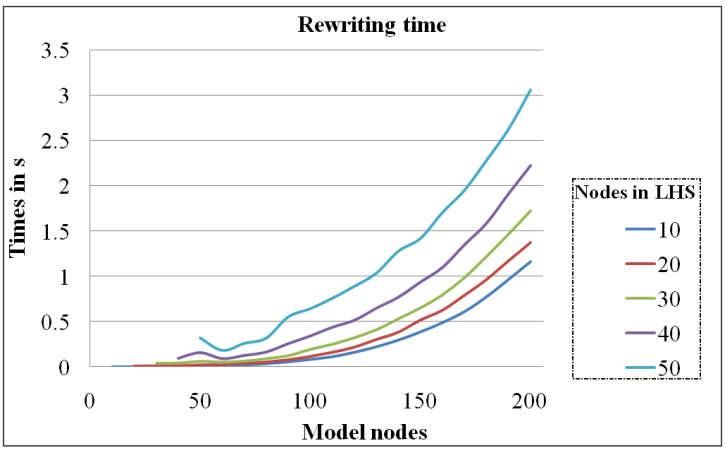

Figure 13. Correlation between the model size and the GRR size.

5. Case Study

To illustrate our approach, we present a case study about monitoring SLA and reducing violations. The relationship between service providers and their consumers is controlled and regulated using SLAs. An SLA includes a set of terms that specify, among others, the expected functional and Quality of Service (QoS) guarantees of the offered service. This contract also specifies penalties that will be applied if its terms are violated.

Let us consider a provider offering a service that allows consumers to compress their files. This service receives a file from a client and schedules its compression. Once the compression is performed, it delivers the compressed file to the client. Regarding the QoS properties considered in this example, we focus only on the response time. As a penalty, consumers will be compensated with a *0.1 Euro* discount if the deliverance is delayed after the agreed deadline.

As illustrated in Figure 14, we model the behavior of the compression service as a PN. First, the application imports the necessary packages by firing the transitions T_0, T_1, T_2, and T_3. The file to be compressed is received when firing T_4, and the receiving time is saved using T_5. Firing T_6 waits for the file turn. Subsequently, the file is compressed using transition T_7. Transition T_8 obtains the size of the received file, which is used to calculate the cost of the compression operation by T_9. Depending on the obtained result, two scenarios may occur. In both scenarios, the time consumed is compared against the pre-agreed time, less than 5. The statements and guards of the transitions are shown in Table 4.

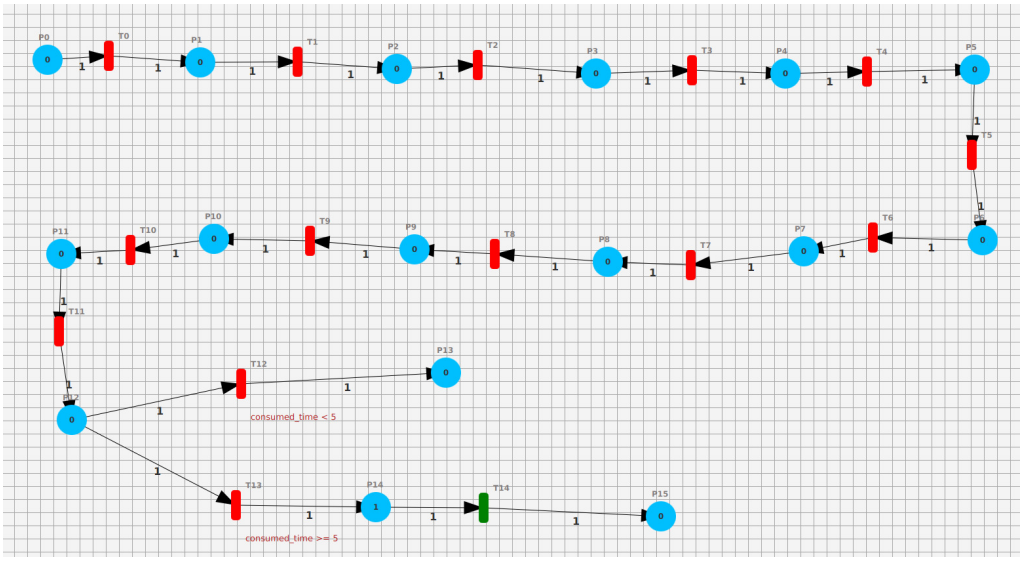

Figure 14. Case study—monitoring of SLA violations.

Table 4. Statements and guards of the case study transition.

Transition	Statement	Guard
T_0	import os	True
T_1	import time	True
T_2	from caseStudy import case_study	True
T_3	from compression import compression_server	True
T_4	input_file = case_study.receive_file()	True
T_5	received_time = time.time()	True
T_6	case_study.wait_turn()	True
T_7	output_file = compression_server.zip_lzma(input_file)	True
T_8	file_size = os.stat(input_file).st_size/(1024 $*$ 1024)	True
T_9	cost = case_study.estimate_cost(file_size)	True
T_{10}	sending_time = time.time()	True
T_{11}	consumed_time = sending_time $-$ received_time	True
T_{12}	case_study.send_file(output_file, cost)	consumed_time $<$ 5
T_{13}	cost = cost $-$ 0.1	consumed_time $>=$ 5
T_{14}	case_study.send_file(output_file, cost)	True

- Respected SLA: In the first scenario, the expected time consumed is met (i.e., response time <5). This scenario is executed by firing transition T12 (see Figure 14). This transition sends the compressed file and the cost of the service to the consumer. This situation can be observed in marking $\{P_{13}\}$.
- Violated SLA: In the second scenario, the deliverance is delayed after the expected deadline (i.e., response time >=5). This scenario is executed by firing transition T_{13} (see Figure 14). This transition applies a penalty, as specified in the SLA, to the provider by subtracting *0.1 Euro* from the cost of the compression operation. The transition T_{14} sends the compressed file and the final cost to the consumer. The situation of the SLA violation can be monitored by observing markings $\{P_{14}\}$ and $\{P_{15}\}$.

Another way to monitor the SLA violations is to inspect runtime data using the expression evaluator. Figure 15 shows the evaluation of the expression `consumed_time >= 5`, which is evaluated as True. This is the deadline for the compression, so the SLA is violated at this point.

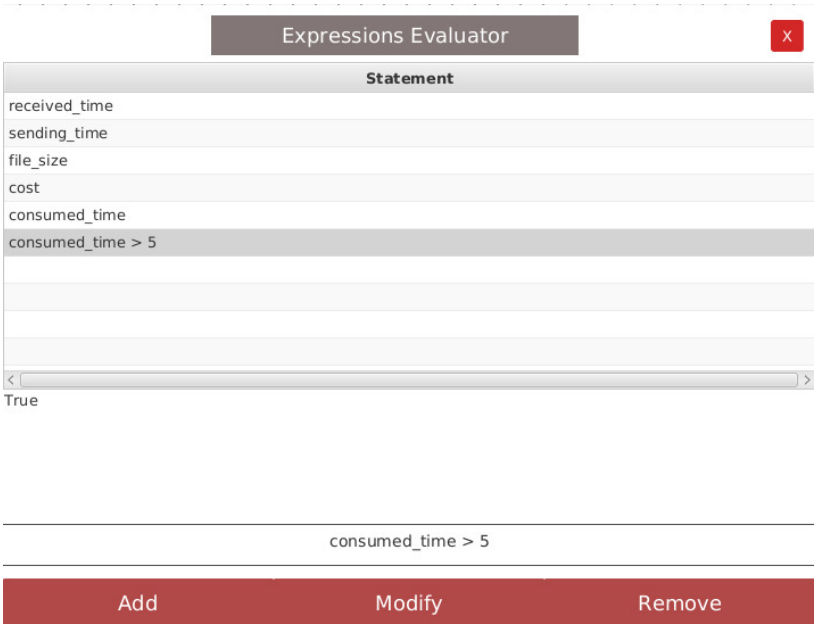

Figure 15. The expression evaluator -case study-.

A new requirement consisting of replacing the compression algorithm with a faster one is considered to reduce the number of SLA violations. Therefore, developers may apply a new GRR to comply with this requirement, as depicted in Figure 16. This GRR replaces transition T_7 with new transition T_{15}. The statement and the guard of new transition T_{15} are shown in Table 5.

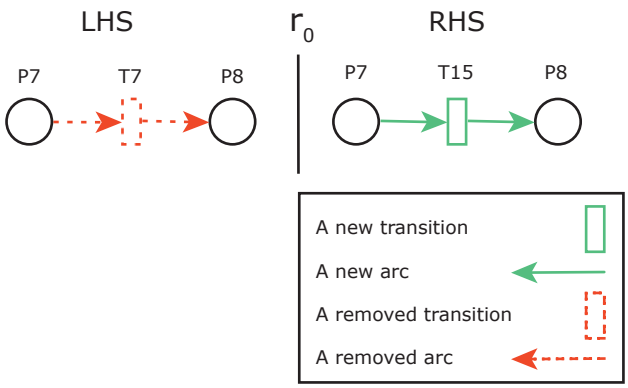

Figure 16. A GRR to replace the compression algorithm.

Table 5. Statements and guards of the new transition of the case study.

Transition	Statement	Guard
T_{15}	output_file = compression_server.zip_deflated(input_file)	True

6. Discussion

The goal of our approach is bidirectional:

(i) to execute a code that conforms to the behavior of its executable model;
(ii) to model at runtime the execution of the application.

Thus, the execution of a Python application and its executable model should be causally connected in both directions, that is, the model (marking, history, structure, etc.) reflects the execution (the behavior of the application) and vice versa.

We achieve the first goal by orchestrating the execution using the MEE. This is similar to what Remote Procedure Calls (RPCs), Java RMI (www.java.com, accessed on 1 August 2021), and BP engines (e.g., Activiti (www.activiti.org, accessed on 1 August 2021)) do. In those paradigms, servers are prepared to execute a pre-fixed set of methods (functions), while the PEE can execute any type of Python instruction using the *exec* command.

We achieve the second goal by reflecting the state of the execution in the PN model using both the MEE and the PEE, which allows the execution to be monitored. Thus, we are able to control the model using real data. The level of abstraction of the model is based on the granularity of the statements invoked. If the developers need to monitor and control more execution details, they use statements that are close to Python built-in/library instructions. Otherwise, they may develop and invoke high level functions.

Other approaches based on code generation techniques have shown how difficult it is to prove the conformity of the executable code to the source model [21]. In contrast, our solution, thanks to its *correct-by-construction* approximation, ensures the execution's conformity with respect to the running model (see Figure 7). That is, the dualism model and application working together allows the application behavior to be reflected.

A running application could be configured using the *meta-programming* paradigm, where the program is treated as data [23]. In fact, our framework can be classified as a *meta-programming* framework. In this work, we use GRRs to rewrite the structure of applications. Therefore, the new structure reflects the new behavior of the application. For instance, the GRR depicted in Figure 16 rewrites the initial structure of the case study (Figure 14). The combination of GRRs with PNs is not new. For instance, Llorens et al. introduced NRS in [12]. In this approach, the GRRs introduced the matching strategy of the LHS, defining a structure without specifying place and transition names. Therefore, several matchings may be found in the same structure. In contrast, we implement a different solution using unique names for places and transition in the definition of GRRs, which results in a single match.

In our approach, there is a clear separation between the design and the program, that is, between the model and the execution. This has certain advantages, since this separation provides an opportunity to substitute one part without affecting the other part. This is a result of the architecture used, i.e., the two modules, the MEE and the PEE, and their interaction via TCP/IP sockets. This architecture may support any type of behavioral model, such as UML behavioral diagrams [22], process algebra [24], BPEL [25], and BPMN [26]. On the other hand, the language used may be any programming language capable of supporting reflection/meta-programming, such as Scala (www.scala-lang.org, accessed on 1 August 2021) or Java (www.java.com, accessed on 1 August 2021). In addition, it is a general-purpose approach that is independent of the application domain. As a proof of concept, the case study presented in Section 5 studies the SLA offered by a service and analyzes its reconfiguration to reduce SLA violations.

In the communication process, we may use any paradigm that guarantees communication among components to connect the two modules, such as RMI, CORBA, publish/-subscribe, and SOA techniques. The aim is to connect the model with its execution using

loosely coupled components. Thus, this solution may be used to model distributed systems at runtime. As a disadvantage, this communication introduces delays, which may be easily avoidable by assuming a monolithic approach calling the exec function directly when a transition is fired.

The literature includes several related works in the area of models@runtime. Table 6 compares our approach with these related works, where several aspects are considered: the modeling technique, the research objective, the application domain, and the runtime technique. Most approaches in this context are related to a specific domain, and few works consider a general programming perspective. To the best of our knowledge, there is no work that considers Python applications. Having tools to model at runtime applications based on their programming language gives the ability to model at runtime application without considering their application domain. In addition, the reconfiguration process is not generally considered in these works. Furthermore, the loosely coupled architecture presented in this work enables its use on distributed environments, which is not a common feature in the related work.

Regarding the work by Guerin et al. [3], they presented a framework called PAMELA to model execution where the model and the code are developed at the same time. The model is a Java program with annotations, and the framework interprets this model at runtime. The framework calls the methods through Java interfaces. This work is focused on the design, validation, and verification phases of the software development life cycle and does not propose a framework focused on the program reconfiguration when changes are applied. The proposed architecture is not loosely coupled. Criado et al. [7] proposed a heuristic solution based on the abstract definition of a software component and a set of available components to generate a configuration of a software architecture in the context of smart architectures for smart cities. They proposed a Domain-Specific Language (DSL) as the modeling language. The main difference from our work is the abstraction level. While we focus on the program behavior, they focus on the architecture. They focus on component base programming rather than on a general programming language. Cedillo et al. [18] described a generic method to monitor the satisfaction of non-functional requirements in cloud environments using models at runtime and SLAs. They proposed a middleware that interacts with services. This middleware retrieves data at runtime and analyzes it to provide a report of unsatisfied non-functional requirements. This work is focused in the context of cloud services where non-functional requirements are monitored at runtime, and the service reconfiguration is not considered. Valero et al. in [19] established the basis to provide a formal semantics for the event processing language (EPL) using an extended version of Colored PNs. They cover a subset of the operators specified in the EPL. This language implements the complex event processing (CEP) paradigm used in the literature to provide complex reasoning about events produced at runtime, providing a set of patterns to be detected in a given event stream. The authors focus in this work on the CEP context, and the runtime reconfiguration is not considered. Tanković et al. [8] proposed a framework to build architectures for distributed information systems in mobile cloud environments. The models are interrupted during execution. The model is represented as a directed graph. A procedural scripting language is used to express complex behavior by end users. The framework allows the modeled systems to be adapted at runtime to accelerate the process of delivering software. The difference with respect to our work is that they focused on mobile environment domains where the information system can be interpreted at runtime. In addition, the reconfiguration of system at runtime is not considered. Búr et al. [27] proposed a distributed model to capture the states of different nodes of cyber-physical systems (CPSs) at runtime. A time triggered protocol is used to update the models. They used a publish-subscribe middleware for communications. The monitoring system captures the critical situations of interest. The focus of this work is not on the system reconfiguration at runtime. Leroy et al. [20] proposed a temporal property language for runtime monitoring of any kind of executable discrete event model supported in a development framework. This work is independent on the modeling language or the programming language. To

deal with reconfiguration in live programming, Rozen et al. [28] used incremental deltas with respect to the original code of texture domain-specific model (DSLs). They applied the deltas on the running programs by migrating them based on their state, using a dynamic patch architecture. The approach does not depend on a specific programming language. It does not focus on monitoring the state of the program while applying changes.

Table 6. Approach comparison.

Approach	Modeling Technique	Research Objective	Application Domain	Runtime Monitoring Technique
Our approach	PN & GRRs	Behavior of Python applications, runtime analysis, Python expressions, continuous change	Independent	Reflection
Guérin et al. [3]	PAMELA meta-model	Runtime analysis of Java applications, continuous change	Independent	Reflection
Criado et al. [7]	DSL	Adapting architectures based on component syntax and semantics	Independent	Heuristics
Cedillo et al. [18]	Meta-model	Monitoring non-functional requirements	Cloud services	Monitoring
Valero et al. [19]	BPCPN Meta-model	Monitoring	Independent	Model transformation
Tankovic and Grbac [8]	AGM Meta-model	Interpreting information systems	Mobile cloud environments	Monitoring
Búr et al. [27]	Meta-model	Monitoring	Cyber-physical systems	Publish-Subscribe
Leroy et al. [20]	Executable DSLs	Monitoring	Independent	Temporal property language
van Rozen and van der Storm [28]	DSLs	Live programming	Independent	Model Comparison
Poggi et al. [29]	Ontologies	Adaptation	Adaptive software	Semantic Web
Chatzikonstantinou and Kontogiannis [30]	Fuzzy goal models	Runtime requirement verification	Systems-of-systems	Reasoning
Heinrich [31]	iObserve mega-model	Architectural runtime applications	Dynamic Cloud	Model transformations

Based on the architectural pattern MAPE-K (Monitor, Analyze, Plan, Execute- Knowledge) [32] and semantic Web technologies, Poggi et al. [29] proposed an approach that enables the management of heterogeneous knowledge and the creation of runtime queryable models. The authors use ontologies, as a semantic technology, for the representation and management of real-world systems and their environment. The proposed approach is evaluated on the system that manages the entire IT infrastructure of the University of Bologna. In this work, the authors do not focus on a general programming language, but on concurrent Java components for the four MAPE-K phases, and the monitoring process is not based on expression inspection, but on the use of OWL ontologies.

Chatzikonstantinou et al. [30] proposed an efficient parallel reasoning framework on fuzzy goal models to assess the compliance of critical requirements at runtime. They take into consideration the application logs as a fuzzified data stream to monitor these situations

in over-medium and large-scale systems of systems. In contrast to our work, the proposed approach is specific to systems-of-systems environments. In addition, this approach relies on a model transformation engine and fuzzy reasoners enabling the evaluation of systems at runtime. Heinrich et al. in [31] proposed the iObserve approach, addressing the adaptation and evolution of applications in cloud environments. The proposed approach adopts the MAPE control loop focusing on the monitoring and analysis phases. In comparison to our work, iObserve addresses a specific type of systems that are based on cloud services focusing only on their architecture. In addition, the authors use model transformation techniques to update the run-time models.

We suggest the following surveys for further details on the state of the art [16,33–35].

7. Conclusions

This work presents an approach to develop, monitor, and reconfigure Python applications at runtime, offering a solution to the challenges addressed by the models@runtime initiative. The main benefit of our proposal is that maintainers and developers are able to make runtime decisions to attend new incoming requirements based on the state of the running system provided by the runtime PN marking and the Python evaluated expressions. In addition, they are able to reconfigure Python applications at runtime by adding GRRs. The proposed approach was implemented as a framework supported by a tool consisting of two components: a Model Execution Engine (MEE) and a Python Execution Engine (PEE). The former component uses a new extension of PNs to model Python applications. Transitions in the extended PN are enriched with Python *statements* and *guards*. Statements are the instructions to be executed when transitions are fired. The guards are Python conditions that must be true to fire these transitions. The evaluation of these guards considers the data of the Python program at runtime. Guards are evaluated by the PEE using the Python built-in instruction *eval*. This extended PN is used to model the behavior of the program and to reflect the program execution. The MEE is used to execute the model. The MEE sends the corresponding statements to the PEE when a transition is fired. It executes them using the Python built-in instruction *exec*. The reconfiguration in the running application is achieved by adding GRRs implementing a new requirement. The GRRs modify the application model, which affects the running application. We adopt the instruction *eval* to allow developers to monitor Python expressions. In the framework proposed, developers can add expressions to inspect them during the execution. There is a clear separation between the model and its execution provided by the client/server architecture, that is, by the two interconnected components, the MEE and the PEE. Another benefit is achieved by the adoption of a loosely coupled solution between the model and the execution, which can improve the system flexibility and allow for its deployment in distributed systems. The proposed approach does not depend on the application domain.

Regarding future works, the separation between the model and its execution in our approach provides an opportunity to model the behavior of Python applications using other formalisms. Therefore, we plan to design a tool to meta-model different formalisms and execute them using the architecture presented in Figure 5. We also plan to adapt our tool to the domain of Internet of Things (IoT) using the Raspberry pi (https://www.raspberrypi.org/, accessed on 1 August 2021) single-board as an IoT device. Currently, we are working on the problem of a Business Process (BP) change [10], where the reconfiguration at runtime still remains a challenge because of the large number of running instances for a given BP. We are building a workflow engine to support both the modeling and the instance migration at runtime, where we plan to apply the techniques developed in this work. Furthermore, the approach can be combined with other proposals to automatically generate the source code. This capability is an advantage of the MDE perspective adopted in this work. For instance, we can use other frameworks using UML models and artifacts to generate the source code and orchestrate its execution using our framework.

Author Contributions: M.M.B.: conceptualization; investigation; methodology; software; writing—original draft. G.D.: conceptualization; funding acquisition; investigation; methodology; software; supervision; writing—review & editing. A.C.: conceptualization; funding acquisition; supervision; writing—review & editing. O.K.: conceptualization; investigation; writing—original draft; writing—review & editing. R.N.: conceptualization; supervision; writing—review & editing. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Spanish Ministry of Science, Innovation and Universities, European Union FEDER funds under Grant RTI2018-093608-B-C32, the JCCM project co-financed with European Union FEDER funds, ref. SBPLY/17/180501/000276, and the UCLM group research grant with reference 2020-GRIN-28708. It was also supported by DGRSDT of the Algerian Ministry of Higher Education and Scientific Research.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Readers may find all the data obtained to reproduce the results obtained in the performance evaluation and the source code in the link: http://doi.org/10.17632/39ddjkyx6d.1, accessed on 14 October 2021.

Conflicts of Interest: The authors declare that there is no conflict of interest.

References

1. Maes, P. Concepts and Experiments in Computational Reflection. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87), Orlando, FL, USA, 4–8 October 1987; Meyrowitz, N.K., Ed.; ACM: New York, NY, USA, 1987; pp. 147–155. [CrossRef]
2. Herraiz, I.; Rodríguez, D.; Robles, G.; González-Barahona, J.M. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Comput. Surv.* **2013**, *46*, 28:1–28:28. [CrossRef]
3. Guérin, S.; Polet, G.; Silva, C.; Champeau, J.; Bach, J.C.; Martínez, S.; Dagnat, F.; Beugnard, A. PAMELA: An annotation-based Java Modeling Framework. *Sci. Comput. Program.* **2021**, *210*, 102668. [CrossRef]
4. Bucchiarone, A.; Cabot, J.; Paige, R.F.; Pierantonio, A. Grand challenges in model-driven engineering: An analysis of the state of the research. *Softw. Syst. Model.* **2020**, *19*, 5–13. [CrossRef]
5. Brand, T.; Giese, H. Towards software architecture runtime models for continuous adaptive monitoring. In *MODELS Workshops*; CEUR: Germany, 2018; pp. 72–77. Available online: http://ceur-ws.org/Vol-2245/mrt_paper_4.pdf (accessed on 1 August 2021).
6. Brand, T.; Giese, H. Modeling Approach and Evaluation Criteria for Adaptable Architectural Runtime Model Instances. In Proceedings of the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, Germany, 15–20 September 2019; Kessentini, M., Yue, T., Pretschner, A., Voss, S., Burgueño, L., Eds.; IEEE Computer Society: Los Alamitos, CA, USA, 2019; pp. 227–232. [CrossRef]
7. Criado, J.; Iribarne, L.; Padilla, N. Heuristics-based mediation for building smart architectures at run-time. *Comput. Stand. Interfaces* **2021**, *75*, 103501. [CrossRef]
8. Tankovic, N.; Grbac, T.G. Run-time interpretation of information system application models in mobile cloud environments. *Comput. Sci. Inf. Syst.* **2020**, *17*, 1–27. [CrossRef]
9. Kamel, O.; Chaoui, A.; Díaz, G.; Gharzouli, M. SLA-Driven modeling and verifying cloud systems: A Bigraphical reactive systems-based approach. *Comput. Stand. Interfaces* **2021**, *74*, 103483. [CrossRef]
10. Song, W.; Jacobsen, H. Static and Dynamic Process Change. *IEEE Trans. Serv. Comput.* **2018**, *11*, 215–231. [CrossRef]
11. Murata, T. Petri nets: Properties, analysis and applications. *Proc. IEEE* **1989**, *77*, 541–580. [CrossRef]
12. Llorens, M.; Oliver, J. Structural and Dynamic Changes in Concurrent Systems: Reconfigurable Petri Nets. *IEEE Trans. Comput.* **2004**, *53*, 1147–1158. [CrossRef]
13. Nouara, R.; Chaoui, A. Checking Behavioural Compatibility in Service Composition with Graph Transformation. In Proceedings of the CS & IT Conference Proceedings, Dubai, United Arab Emirates, 28 January 2017; Volume 7.
14. Schmidt, D.C. Model-driven engineering. *Comput.-IEEE Comput. Soc.* **2006**, *39*, 25. [CrossRef]
15. Blair, G.S.; Bencomo, N.; France, R.B. Models@ run.time. *Computer* **2009**, *42*, 22–27. [CrossRef]
16. Bencomo, N.; Götz, S.; Song, H. Models@run.time: A guided tour of the state of the art and research challenges. *Softw. Syst. Model.* **2019**, *18*, 3049–3082. [CrossRef]
17. Shirazi, S.K.G.; Mohseni, M.; Darvishan, M.; Yousefzadeh, R. RSCM technology for developing runtime-reconfigurable telecommunication applications. *Comput. Stand. Interfaces* **2017**, *51*, 43–55. [CrossRef]
18. Cedillo, P.; Insfrán, E.; Abrahão, S.; Vanderdonckt, J. Empirical Evaluation of a Method for Monitoring Cloud Services Based on Models at Runtime. *IEEE Access* **2021**, *9*, 55898–55919. [CrossRef]

19. Valero, V.V.; Diaz-Descalzo, G.; Boubeta-Puig, J.; Macia, H.; Brazalez-Segovia, E. A Compositional Approach for Complex Event Pattern Modeling and Transformation to Colored Petri Nets with Black Sequencing Transitions. *IEEE Trans. Softw. Eng.* **2021**. [CrossRef]
20. Leroy, D.; Jeanjean, P.; Bousse, E.; Wimmer, M.; Combemale, B. Runtime Monitoring for Executable DSLs. *J. Object Technol.* **2020**, *19*, 1–23. [CrossRef]
21. Meghzili, S.; Chaoui, A.; Strecker, M.; Kerkouche, E. Verification of Model Transformations Using Isabelle/HOL and Scala. *Inf. Syst. Front.* **2019**, *21*, 45–65. [CrossRef]
22. Kerkouche, E.; Chaoui, A.; Bourennane, E.; Labbani, O. On the Use of Graph Transformation in the Modeling and Verification of Dynamic Behavior in UML Models. *J. Softw.* **2010**, *5*, 1279–1291. [CrossRef]
23. Lilis, Y.; Savidis, A. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* **2020**, *52*, 113:1–113:39. [CrossRef]
24. Chama, I.E.; Belala, N.; Saïdouni, D. Formalizing Timed BPEL by D-LOTOS. *Int. J. Embed. Real Time Commun. Syst.* **2014**, *5*, 1–21. [CrossRef]
25. Ings, D.; Clément, L.; König, D.; Mehta, V.; Mueller, R.; Rangaswamy, R.; Rowley, M.; Trickovic, I. WS-BPEL Extension for People (BPEL4People) Specification Version 1.1. 2010. Available online: https://www.bibsonomy.org/bibtex/22c1f4af4f7124255c44899aaaeccb263/porta (accessed on 1 July 2021).
26. OMG. Business Process Model and Notation (BPMN), Version 2.0. 2011. Available online: https://www.bibsonomy.org/bibtex/2d71e28d8b21b73a067683ba4bfe0321a/porta (accessed on 1 July 2021).
27. Búr, M.; Szilágyi, G.; Vörös, A.; Varró, D. Distributed graph queries over models@run.time for runtime monitoring of cyber-physical systems. *Int. J. Softw. Tools Technol. Transf.* **2020**, *22*, 79–102. [CrossRef]
28. Van Rozen, R.; van der Storm, T. Toward live domain-specific languages—From text differencing to adapting models at run time. *Softw. Syst. Model.* **2019**, *18*, 195–212. [CrossRef]
29. Poggi, F.; Rossi, D.; Ciancarini, P. Integrating Semantic Run-Time Models for Adaptive Software Systems. *J. Web Eng.* **2019**, *18*, 1–42. [CrossRef]
30. Chatzikonstantinou, G.; Kontogiannis, K. Efficient parallel reasoning on fuzzy goal models for run time requirements verification. *Softw. Syst. Model.* **2018**, *17*, 1339–1364. [CrossRef]
31. Heinrich, R. Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications. *SIGMETRICS Perform. Eval. Rev.* **2016**, *43*, 13–22. [CrossRef]
32. Kephart, J.O.; Chess, D.M. The vision of autonomic computing. *Computer* **2003**, *36*, 41–50. [CrossRef]
33. Erazo-Garzón, L.; Román, A.; Moyano-Dután, J.; Cedillo, P. Models@runtime and Internet of Things: A Systematic Literature Review. In Proceedings of the 2021 Second International Conference on Information Systems and Software Technologies (ICI2ST), Quito, Ecuador, 23–25 March 2021; pp. 128–134. [CrossRef]
34. Szvetits, M.; Zdun, U. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *Softw. Syst. Model.* **2016**, *15*, 31–69. [CrossRef]
35. Thiry, M.; Schmidt, R.A. Self-adaptive Systems Driven by Runtime Models. In Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering, Pittsburgh, PA, USA, 5–7 July 2017; He, X., Ed.; KSI Research Inc.; Knowledge Systems Institute Graduate School: USA, 2017; pp. 248–253. Available online: http://ksiresearch.org/seke/seke17paper/seke17paper_168.pdf (accessed on 1 August 2021). [CrossRef]