Using the ORIS Tool and the SIRIO Library for Model-Driven Engineering of Quantitative Analytics

Laura Carnevali[1], Marco Paolieri[2], Riccardo Reali[1](✉), Leonardo Scommegna[1], Federico Tammaro[1], and Enrico Vicario[1]

[1] Department of Information Engineering, University of Florence, Florence, Italy
{laura.carnevali,riccardo.reali,leonardo.scommegna,federico.tammaro, enrico.vicario}@unifi.it

[2] Department of Computer Science, University of Southern California, Los Angeles, USA
paolieri@usc.edu

Abstract. We present a Model-Driven Engineering (MDE) approach to quantitative evaluation of stochastic models through the ORIS tool and the SIRIO library. As an example, the approach is applied to the case of a tramway line with reduced number of passengers to contain the spread of infection during a pandemic. Specifically, we provide a meta-model for this scenario, where, at each stop, only a certain number of people can ride the tram depending on the current tram capacity, the length of the queue of people waiting at the stop, and the number of passengers on the tram. Then, the ORIS tool and the SIRIO library are used as a software platform to derive a Stochastic Time Petri Net (STPN) representation for each tramway stop and to perform its regenerative transient analysis to obtain quantitative measures of interest, such as the expected number of people waiting at each stop and the expected number of tram passengers over time. Experimental results show that the approach facilitates exploration of the space of design choices, providing insight about the effects of parameter changes on quantitative measures of interest and allowing balanced queue sizes at different stops.

Keywords: Quantitative evaluation · Model Driven Engineering (MDE) · Software tools and libraries · Intelligent transportation systems

1 Introduction

Models are generally used to replace the system under study with a domain-focused although simplified view [5]. By shifting the attention primarily to models, the concepts expressed are less bound to a precise technology or framework, and closer to the problem domain [13], allowing better insight into the issues of interest: this approach eases system understanding by domain experts, improves the expressivity of the system description, and facilitates the maintainability of the adopted solution. At the same time, models also support early evaluation of the impact of design choices on the final behavior before data become available, enabling fast exploration of the space of possible solutions.

Model-Driven Engineering (MDE) is an approach that considers models as primary artifacts during all software development phases [5] and connects them with the practice of software engineering, making them living components of the system rather than using them for documentation and study purposes only. MDE usually consists in the use of Domain-Specific Languages (DSLs) and Domain-Specific Modeling Languages (DSMLs), specialized in formalizing the structure and behavior of applications, and described using meta-models to map relationships, semantics, and constraints between concepts expressed in a domain [12]. This practice is also common in industrial contexts, where small DSLs are developed for narrow and well-understood domains [16]. Another important aspect that led to the widespread use of MDE and Model-Driven Design (MDD) are automated transformations: Model-to-Text (M2T) transformations are more commonly used to transform a particular model instance into text-based file formats or software artifacts available as source code (code generation), while Model-to-Model (M2M) transformations are used to translate a model into another model. Both techniques are referred to as "correct-by-construction" [12] given that they do not require any subsequent modification and they avoid manual, and thus error-prone, changes to the considered artifacts.

In a broader perspective, models can be generated at system runtime with an inverse approach, i.e., by connecting models to operational data, possibly persisted in a data layer, and by enabling the management of high volumes of concrete running instances derived from a meta-model, with variability determined by runtime changes (e.g., time-dependent parameters). When combined with the use of formal semantics and solution techniques to compute results of a service request, this two-way approach supports dynamic state monitoring and system control during execution as well as understanding of runtime behavior, including the identification of behavioral phenomena [1]. In so doing, the approach allows agile validation of choices in the design phase.

In this paper, we outline an MDE approach which demonstrates how to leverage the ORIS suite [11,14] to develop practical applications of stochastic modeling and analysis. In particular, we focus on how to manage crowding of a tramway line to contain the spread of a pandemic, which comprises a problem of resource assignment: at each stop, a transit pass can be granted only to a certain number of people, depending on the current capacity of the tram, the length of the queue of people waiting at the stop, and the number of passengers on the tram. To this end, we perform a context analysis to derive a meta-model of the considered scenario. Then, we exploit MDE practices and M2M transformations to derive a Stochastic Time Petri Net (STPN) representation for each tramway stop, and then we perform regenerative transient analysis [8] of each tramway stop model. Specifically, the analysis of each stop provides the expected number of queued people over time and the expected number of tram passengers over time, showing that the distribution of the number of queued people and the distribution of the number of tram passengers at the beginning of each tram period reach a steady state within a limited number of tram periods. Then, the steady state distribution of the number of tram passengers at the beginning of tram periods is used in the analysis of the subsequent tramway stop as the initial distribution of the number of tram passengers. In the theoretical perspective, the tramway stop model fits in the class of polling systems [9], where the periodic arrival pattern of trams is impacted by a stochastic delay (jitter). Experimental results shows that the approach facilitates exploration of the design space, providing insight about the effects of parameter changes on performance measures of interest, such as the expected number of queued people at each stop and the expected number of passengers on the tram.

The rest of this paper is organized as follows. In Sect. 2, we describe the application domain and we provide a meta-model to define context-specific models. In Sect. 3, we illustrate how the ORIS tool supports MDE of quantitative analytics, transforming a tramway line model into an STPN and performing its evaluation through regenerative transient analysis. In Sect. 4, we explain how to support modeling and evaluation steps through the SIRIO library. In Sect. 5, we illustrate the experimental results obtained analyzing the models of a tramway line. Finally, conclusions are drawn in Sect. 6.

2 A Tramway Line Meta-model

Figure 1 shows a meta-model of the tramway scenario, designed to express a wide range of models at different levels of granularity and derive (through M2M transformations) analytic representations to get insight into the scenario. Specifically, the meta-model represents a network of tramway lines (represented by the `Network` and `Route` classes) each characterized by a `TimeBand` identifying the time period of tramway service and the frequency of tram departures. An itinerary is an ordered sequence of `StopPoint` instances, each characterized by a `StopPointCapacity` which can be affected by different *restriction policies*, e.g., due to maintenance works or social distancing. Each `Tram` has a maximum *capacity* in terms of number of passengers (possibly subject to restriction policies), runs on a specific route, and stops at each stop point of the route itinerary.

The `Stop` class identifies the event that a tram of a certain route arrives and stops at a specific stop point. It is described by: an `ArrivalTimeTable` instance, characterized by a *jitter* delay with respect to the nominal arrival time; a maximum number of passengers that can board the tram (with the possibility to define an ad-hoc restriction policy); a flow of *passengers* of different *types*. The latter association enables the representation of a stop point visited by a heterogeneous group of passengers with a different arrival rate, outgoing rate (i.e., the tendency to abandon the stop point) and disembark rate. The stop event is characterised by parameters that may vary with the time of day (e.g., a tram stop near a school is characterized by different flows of passengers immediately after school hours and during a night time slot), and thus it is also characterized by a *time chunk*.

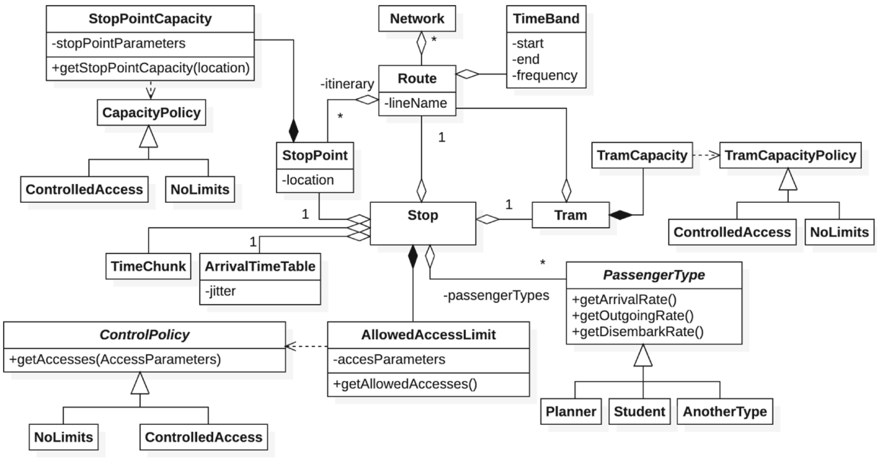


Fig. 1. Meta-model of a network of tramway lines.

Note that the class diagram representing the meta-model could be exploited to define the domain model of a software architecture. In this context, the entire framework could be transposed into a *Software as a Service* (SaaS) system of a cloud infrastructure. On the one hand, the class diagram is designed to enable an easy mapping of object instances to the database through standard Object-Relational Mapping (ORM) technologies (e.g., JPA for Java EE). On the other hand, the M2M transformation procedure, as well as the quantitative analysis through the SIRIO library, represent business capabilities that could be conveniently encapsulated into a set of microservices, generating a cloud architecture able to fulfill various use cases such as those illustrated in Fig. 2. In particular, to avoid crowding on the tram and at tramway stops, the mentioned M2M transformations (see Sects. 3 and 4) enable the evaluation of the expected value of queued people at a stop and of people traveling on a tram. Such information could be exploited by a tramway schedule planner to modulate the frequency of trams during a certain time band, to change the maximum capacity of a tram, or, more generally, to adapt system parameters to guarantee some safety conditions.

3 A Tramway Stop Model

We define a model of a tram stop (Sect. 3.1), and we use the ORIS tool to derive its STPN representation (Sect. 3.2) and to compute rewards of interest through regenerative transient analysis of the STPN model (Sect. 3.3).

3.1 Model Description

We consider a model in the class of polling systems [1], i.e., systems where queue processes are served by a component, which does not operate continuously, but

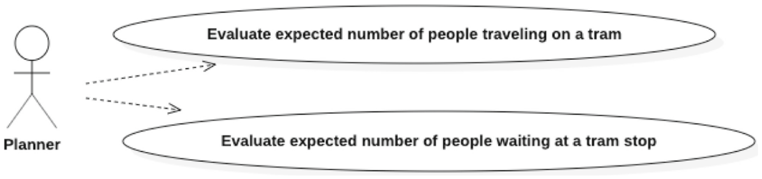

Evaluate expected number of people traveling on a tram
Planner
Evaluate expected number of people waiting at a tram stop

Fig. 2. Use case diagram showing relevant use cases for tramway line planning.

intermittently and recurrently. In the tramway scenario, trams arrive at different stops according to a probability distribution that combines the deterministic interarrival time of trams with a stochastic jitter. At each stop, the tram serves people waiting in a queue that is populated according to an arrival process.

Figure 3 shows the object model that illustrates the considered polling system, in conformity with the meta-model described in Sect. 2. The system targets a single *stop point* (`SP-A`) of a tramway line (`R1`). To avoid crowding, a stop point is associated with a *capacity* (`CPT-A`) characterized by a rejection policy, which disables queue access to people arriving after the saturation of maximum queue capacity, here equal to 10. A route is characterized by a *frequency*, which is the inverse of the deterministic time between the arrival of two consecutive trams at the stop point. Each individual *stop* event (`SA1`) is also characterized by a stochastic delay defined by a *jitter*. In our model, we assume an interarrival time equal to 220s seconds and a jitter distributed as an expolynomial function $f(x) = 0.075468 \exp(-x/10) + 0.002516\, x \exp(-x/10)$ with support $[0, 60]$ s. Different classes of people (`STD-P`) may arrive at the stop point with a specific probability distribution; for simplicity, we only consider *standard passengers* arriving at the queue according to a Poisson process with rate 0.08 passengers/second. Finally, we consider trams with maximum capacity (`CPT-T`) equal to 4; additional people waiting to board the tram are rejected (`CA-S` control policy).

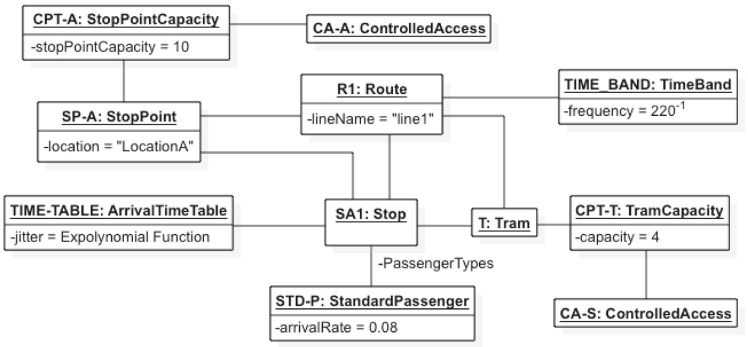


Fig. 3. Object model of a tram stop.

3.2 STPN Representation

Figure 4 shows the STPN model of a single tram stop. In an STPN, time processes are modeled with transitions, represented as bars with different colors depending on the probability density function (PDF) of their associated timers: transitions with exponential PDFs (EXP) are represented as white thick bars (e.g., transition `passengerArrival`); transitions with deterministic times (DET) are represented as gray thick bars (e.g., transition `serviceArrivalNominal`); transitions with non-exponential general distributions (GEN) are represented as black thick bars (e.g., transition `serviceArrival`). A deterministic transition with time 0 is called *immediate* (IMM) and it is represented as a black thin bar (e.g., transitions `leaving` and `boarding`). Discrete logical states of the system are modeled as tokens within places, which are represented as dots or numbers within circles (e.g., places `TramCapacity` and `QueueCapacity`, respectively). Places and transitions are connected with directed arcs, modeling precedence relations among processes. Directed arcs from input places to transitions define preconditions, while arcs from transitions to output places defines postconditions (e.g., the tuple (`WaitJitterDelay`, `serviceArrival`) is a precondition).

A transition becomes *enabled* by a marking (i.e., an assignment of tokens to one or more places) when each of the input places contains at least one token (e.g., if a token is added to place `WaitJitterDelay`, transition `serviceArrival` is enabled), and when an *enabling function* (if any) is satisfied for the transition (e.g., transition `passengerArrival` is enabled when the tokens in place `Queue` are less than or equal to those in place `QueueCapacity`). For each enabled transition t, a time-to-fire is sampled between its earliest firing time (EFT) and latest firing time (LFT), according to the PDF $f_t(x)$ of the transition. When its time-to-fire elapses, a transition becomes *firable*; when it fires, one token is removed from each of its input places and one token is added to each of its output places (e.g., if transition `passengerArrival` fires, then a token is moved from place `WaitJitterDelay` to place `TrainArrived`), or the marking is modified according to an *update function* (e.g., the firing of transition `boarding` removes all tokens from place `Queue`, when the queue is less than the tram capacity, or subtracts from place `Queue` a number of tokens equal to the remaining capacity of the tram, when the queue is larger). In case of multiple firable transitions, transitions with lower *priority number* have precedence; among transitions with the same priority, one is selected with a random switch where probabilities are

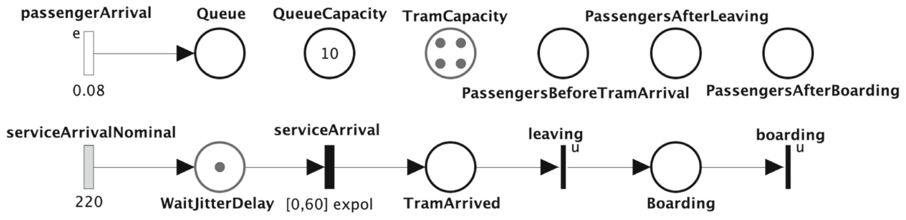


Fig. 4. The STPN of a tram stop.

proportional to *transitions weights.* After a firing, the times-to-fire of transitions enabled before and after the token moves are reduced by the time-to-fire of the fired transition (e.g., if `passengerArrival` fires before `serviceArrival`, the time-to-fire of the latter transition is decreased by the time elapsed since the previous firing).

The STPN model of Fig. 4 can be evaluated to get insight about the model described in Fig. 3. Transitions `serviceArrivalNominal` and `serviceArrival` model the nominal time and the stochastic jitter employed by the tram to periodically reach the considered stop. Transition `passengerArrivial` models the arrival process of people at the tram stop, which is a Poisson process, as described in Sect. 3.1. Finally, IMM transitions `leaving` and `boarding` are introduced to represent the processes of exiting and entering the tram, which are assumed to happen in zero time. Note that, in the model of Fig. 4, places are used not only to represent logical states of a specific process, but also to represent model parameters (e.g., `QueueCapacity` and `TramCapacity`) or variable counters (e.g., `Queue`, `PassengersBeforeTramArrival`, `PassengersAfterLeaving` and `PassengersAfterBoarding`).

3.3 Model Evaluation and Rewards of Interest

STPN models enable the application of analysis techniques that can produce insight, analytics or predictions on such models. The ORIS tool provides a GUI to draw system models and to apply such analysis methods. Evaluation of a model involves identifying certain quantities of interest, which can be formulated as *rewards.* In ORIS, a reward is an expression including constants and token counts, which defines a real-valued function over markings of the STPN. A reward is used to specify quantitative measures of interest to be evaluated by the analysis engines of ORIS; for example, to evaluate the evolution over time of the number of queued people in the model of Fig. 4, it is sufficient to evaluate the reward "`Queue`".

The ORIS tool provides different analysis engines [11] enabling rewards evaluation through different techniques. The *nondeterministic* engine is designed to enable nondeterministic analysis of the state space of an STPN; the *Markovian* engine implements methods to evaluate models with underlying Continuous Time Markov Chains (CTMC); the *enabling restriction* engine is tailored for the evaluation of Markov Regenerative Processes (MRPs) under the enabling restriction [7]; the *forward* and the *regenerative* engines implement different techniques for the evaluation of MRPs without restrictions on the number of regenerative states. The *regenerative* engine enables the evaluation of the transient behavior of an STPN by numerical integration of the Generalized Markov Renewal equations $P_{ij}(t) = L_{ij}(t) + \sum_{k\in\mathcal{R}} \int_0^t dG_{ik}(u)\, P_{kj}(t-u)$ for all i in the set of reachable regenerations $\mathcal{R}$ and for all j in the set of markings $\mathcal{M}$, where the *global kernel* $G_{ik}(t) := P\{X_1 = k, T_1 \leq t \mid X_0 = i\}$ characterizes the next regeneration point $T_1 \geq 0$ and regeneration $X_1 \in \mathcal{R}$, while the *local kernel* $L_{ij}(t) := P\{M(t) = j, T_1 > t \mid X_0 = i\}$ defines transient probabilities of the process until the next regeneration point. Kernels can be evaluated using the method of *stochastic state classes* [8]; this method encodes the marking, the support of the times-to-fire of enabled transitions after each sequence of firings between any two regeneration points, and the continuous joint PDF with a piecewise representation over Difference Bounds Matrix (DBM) zones [3], that can be efficiently evaluated in closed-form by the ORIS tool when each transition has expolynomial PDF [15].

As discussed in Sect. 2, we are interested in two use cases: predicting the expected number of people waiting at a stop and predicting the expected number of people traveling on a tram. According to the semantics of rewards in ORIS, these metrics can be evaluated for the model of Fig. 4 by using the rewards "`Queue`" and "`PassengersAfterBoarding`", respectively. The evaluation can be carried out with the GUI of ORIS, or through its analysis library SIRIO, as presented in Sect. 4.

4 Combining SIRIO with MDE Practices

SIRIO is a Java library from the ORIS tool suite which collects modeling, analysis and simulation features for STPNs. In this section, we briefly illustrate how to use the SIRIO library to model and analyze the polling system presented in Sect. 3 (Sect. 4.1) and we describe how MDE practices can transform models from the specification of the meta-model provided in Sect. 2 to the SIRIO API (Sect. 4.2) to evaluate quantitative measures of interest.

4.1 SIRIO Modeling and Evaluation

Figure 5 shows the UML class diagram of the SIRIO STPN modeling component. An STPN can be modeled by instancing a `PetriNet` object as an aggregation of instances of `Place` and `Transition`. Precedence constraints between places and transitions can be obtained by adding instances of `Precondition` and `Postcondition` to the `PetriNet` object. `Precondition` and `Postcondition` classes model precedence constraints between places and transitions, and transitions and places, respectively. `TransitionFeature` instances can be added to a transition: a transition can have an `EnablingFunction`, a `PostUpdater`, and a `StochasticTransitionFunction`, which represent the condition to enable a transition, the policy to update the marking after the transition fires, and the PDF of the time-to-fire of the transition, respectively.

In Listing 1, we provide the SIRIO implementation of the model of the polling system, described in Sect. 3. Initially, a `PetriNet` and a `Marking` are created (lines 1 and 2). Then, places are added to the `PetriNet` through the method `addPlace()` (lines 3 to 11). The method returns a `Place` object, which is used to reference the newly created place to add preconditions and postconditions. Similarly, transitions are created through the method `addTransition()` (lines 12 to 16), which is called on the `PetriNet` object and returns a `Transition` object. After creating places and transitions, it is possible to add `Precondition` and `Postcondition` instances through the methods `addPrecondition()` and

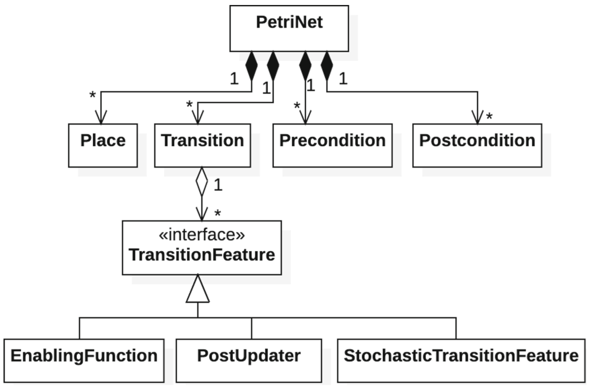


Fig. 5. Class diagram of the SIRIO STPN modeling component.

`addPostcondition()`, respectively (lines 17 to 23). To set the initial marking of the STPN, the method `setTokens()` can be called for each place (lines 24 to 32). Finally, additional features can be added to transitions through the method `addFeature()` (lines 33 to 50). Objects of different types can be passed to this method to add different features: `EnablingFunction` objects (line 42) to add enabling functions, `PostUpdater` objects (lines 33–35 and 38–39) to add update functions, and `StochasticTransitionFeature` objects to add stochastic behavior. Note that the `StochasticTransitionFeature` class implements static constructors to facilitate the creation of the different stochastic transition types presented in Sect. 3.2: `newDeterministicInstance()` for DET transitions (lines 48–49) and IMM transitions (lines 36 and 40); `newExponentialInstance()` (lines 43–44) for EXP transitions; and `newExpolynomialInstance()` (lines 45–47) for expolynomial transitions.

After the creation of the model in the SIRIO specification, regenerative transient analysis can be performed. In Listing 2, analysis time limit, time tick and error threshold are set (lines 1 to 3); these parameters are used to create a builder of a `RegTransient` object (lines 6 to 10), which is then instantiated and used to perform the analysis (lines 12 to 14); finally, the reward `Queue` (or `PassengersAfterBoarding`) is evaluated on the solution (lines 17 and 18).

4.2 Mapping Meta-models to SIRIO Petri Nets

Model-to-model (M2M) transformations are used to move from one domain to another one much closer to the solution domain [5]. In our case study, the transformation between the proposed meta-model and the Petri net classes used by SIRIO is automated by following a set of mapping rules for the entire process. While there is loss of information in the transition from the source model to the target model, STPNs are used only for analysis purposes and all the required information is present in the solution domain. Following the classification in [4], this M2M process can be seen as a unidirectional transformation that computes a target model from a source model; it uses a relational approach with a set of mapping rules to link source and target element types.

```java
PetriNet net = new PetriNet();
Marking marking = new Marking();
Place Boarding = net.addPlace("Boarding");
Place WaitJitterDelay = net.addPlace("WaitJitterDelay");
Place TramArrived = net.addPlace("TramArrived");
Place TramCapacity = net.addPlace("TramCapacity");
Place Queue = net.addPlace("Queue");
Place QueueCapacity = net.addPlace("QueueCapacity");
Place PassengersAfterBoarding = net.addPlace("PassengersAfterBoarding");
Place PassengersAfterLeaving = net.addPlace("PassengersAfterLeaving");
Place TramPassengersBeforeTramArrival = net.addPlace("TramPassengersBeforeTramArrival");
Transition serviceArrivalNominal = net.addTransition("serviceArrivalNominal");
Transition serviceArrival = net.addTransition("serviceArrival");
Transition leaving = net.addTransition("leaving");
Transition boarding = net.addTransition("boarding");
Transition passengerArrival = net.addTransition("passengerArrival");
net.addPostcondition(serviceArrivalNominal, WaitJitterDelay);
net.addPrecondition(WaitJitterDelay, serviceArrival);
net.addPostcondition(serviceArrival, TramArrived);
net.addPrecondition(TramArrived, leaving);
net.addPostcondition(leaving, Boarding);
net.addPrecondition(Boarding, boarding);
net.addPostcondition(passengerArrival, Queue);
marking.setTokens(WaitJitterDelay, 1);
marking.setTokens(TramArrived, 0);
marking.setTokens(Boarding, 0);
marking.setTokens(Queue, 0);
marking.setTokens(QueueCapacity, 10);
marking.setTokens(TramCapacity, 4);
marking.setTokens(TramPassengersBeforeTramArrival, 0);
marking.setTokens(PassengersAfterLeaving, 0);
marking.setTokens(PassengersAfterBoarding, 0);
boarding.addFeature(new PostUpdater(String.join(
        "PassengersAfterBoarding=min(PassengersAfterLeaving+Queue,TramCapacity);",
        "Queue=max(0, Queue-TramCapacity+PassengersAfterLeaving);"), net));
boarding.addFeature(StochasticTransitionFeature.newDeterministicInstance("0"));
boarding.addFeature(new Priority(0));
leaving.addFeature(new PostUpdater(
        "PassengersAfterLeaving=max(0,PassengersBeforeTramArrival-1);", net));
leaving.addFeature(StochasticTransitionFeature.newDeterministicInstance("0"));
leaving.addFeature(new Priority(0));
passengerArrival.addFeature(new EnablingFunction("Queue<QueueCapacity"));
passengerArrival.addFeature(StochasticTransitionFeature.newExponentialInstance(
        configuration.rate()));
serviceArrival.addFeature(StochasticTransitionFeature.newExpolynomial(
        "0.075467664 * Exp[-0.1 x] + 0.0025155888 * x^1 * Exp[-0.1 x]",
        new OmegaBigDecimal("0"), new OmegaBigDecimal("60")));
serviceArrivalNominal.addFeature(StochasticTransitionFeature.newDeterministicInstance(
        configuration.serviceArrivalNominalTime()));
serviceArrivalNominal.addFeature(new Priority(0));
```

Listing 1: SIRIO implementation of the model described in Fig. 4.

```
BigDecimal bound = new BigDecimal("2200.0");
BigDecimal step = new BigDecimal("10.0");
BigDecimal epsilon = new BigDecimal("0.001");

// analyze
RegTransient.Builder builder = RegTransient.builder();
builder.timeBound(bound);
builder.timeStep(step);
builder.greedyPolicy(bound, epsilon);
builder.markingFilter(MarkingCondition.fromString("Queue"));

RegTransient analysis = builder.build();
TransientSolution<DeterministicEnablingState, Marking> probs =
        analysis.compute(net, marking);

// evaluate reward
TransientSolution<DeterministicEnablingState, RewardRate> reward =
        TransientSolution.computeRewards(false, probs,
                                         RewardRate.fromString("Queue"));
```

Listing 2: SIRIO implementation of the regenerative transient analysis required to evaluate the reward `Queue`.

An instance of the meta-model referring to a precise train network can be created from JSON input, or retrieved from the database and scanned with a dedicated *Visitor* [6] to analyze its structure: the Visitor retrieves all the required information associated to a precise `Route` instance, then each stop is treated separately by creating different `PetriNet` objects and analyzed independently. Each model is constructed following the structure introduced in Fig. 4 and then populated with the parameters that characterize a particular stop:

- The rate of the EXP transition `passengerArrival` is automatically calculated as the sum of incoming rates of all passenger types;
- `TramCapacity` and `QueueCapacity` places are populated according the access policy of the stop and tram instances (with controlled access or no limits);
- the PDF of transitions `serviceArrivalNominal` and `serviceArrival` is obtained from the `frequency` and `jitter` attributes of the tram stop.

This procedure completely automates the transformation between models and is implemented through the programming API and classes provided by the SIRIO library. This approach avoids manual and error-prone construction of a model for each stop of a tram network.

Table 1. Queue capacity and arrival rate (passengers/s) of each stop.

Parameter	Stop							
	1	2	3	4	5	6	7	8
Queue capacity	25	50	25	50	25	50	25	50
Arrival rate	0.01	0.01	0.02	0.02	0.04	0.04	0.08	0.08

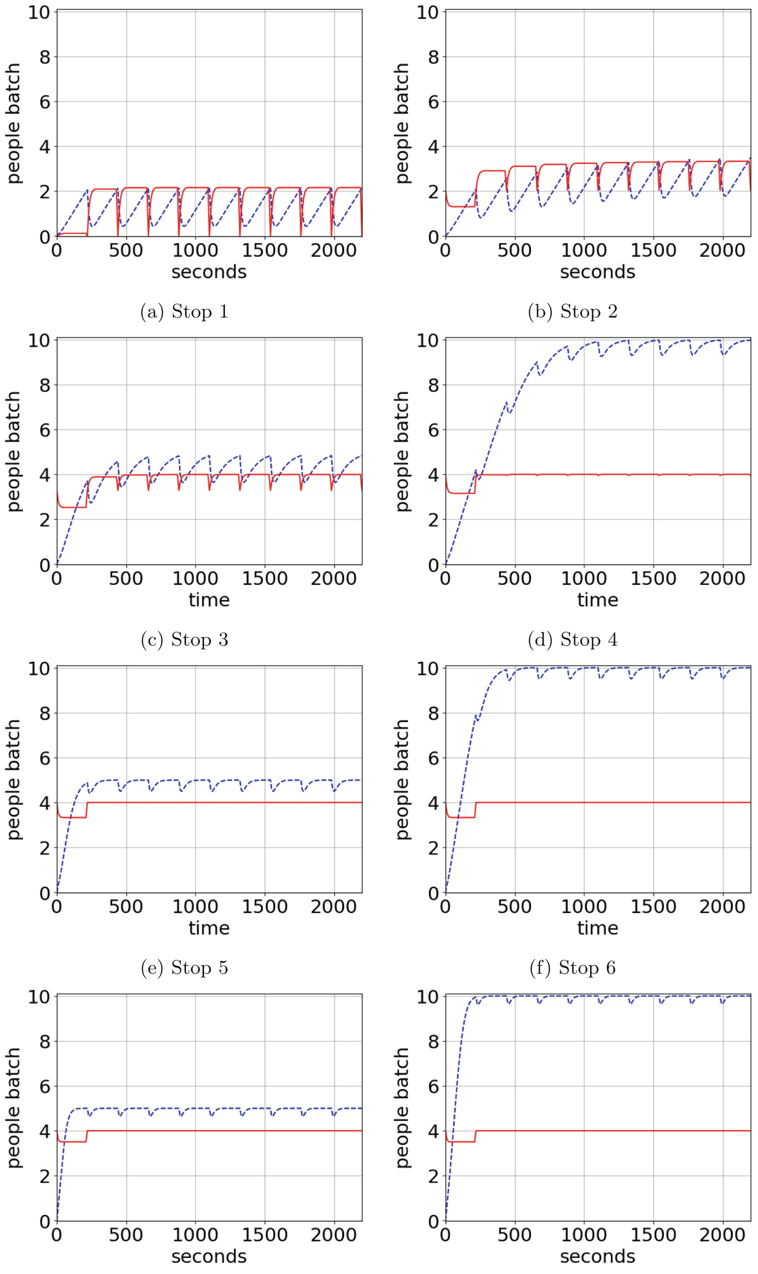
5 Experimentation

In this section, we use the SIRIO library to evaluate quantitative metrics for the case of an 8-stop tramway line. In particular, we describe our setup (Sect. 5.1) and then we discuss the obtained results (Sect. 5.2).

5.1 Experimentation Setup

We consider the case of a tramway with 8 stops, each modeled as the system of Fig. 4 with a tram interarrival time equal to 220 s, but with different queue capacities and passenger arrival rates. To reduce the complexity of the analysis, we consider the arrival process of people at a stop, and the boarding and the leaving processes on/from the tram as batch processes. In particular, a token in places `Queue`, `QueueCapacity`, `TramCapacity`, `PassengersBeforeTrainArrival`, `PassengersAfterLeaving` and `PassengersAfterBoarding` represents a group of exactly 5 people. In so doing, the tram capacity is set to 20 people (4 tokens), while queue capacity depends on the considered stop, and can have 25 or 50 people (5 or 10 tokens). Different stops are analyzed with SIRIO in sequence, using the steady state number of people that already are on the tram obtained from the analysis of the previous stop. To this end, we evaluate the expected steady state value of the rewards `PassengersAfterBoarding==0`, `PassengersAfterBoarding==1`, `PassengersAfterBoarding==2`, `PassengersAfterBoarding==3`, and (for a full tram) `PassengersAfterBoarding==4`. The obtained probabilities are used as the weights of a random switch that draws the number of people on the tram arriving at the next stop, i.e., which updates the number of tokens in place `PassengersBeforeTrainArrival` with a number of tokens between 0 and 4 (0, 5, 10, 15, or 20 people). Finally, we assume that, when the tram arrives, exactly 5 people leave it (1 token); then, as many waiting people as possible board the tram.

The analysis of a stop is performed by evaluating transient rewards `Queue` and `PassengersAfterBoarding` using regenerative analysis based on the method of stochastic state classes [8], with a time limit equal to 2200 s. Parameters of models of different tram stops (reported in Table 1) are selected with the intent of evaluating how changes can influence these metrics.



(a) Stop 1 (b) Stop 2

(c) Stop 3 (d) Stop 4

(e) Stop 5 (f) Stop 6

(g) Stop 7 (h) Stop 8

Fig. 6. Expected number of groups of 5 people waiting for a tram at the stop (blue dashed line) and traveling on a tram (red solid line), for each stop, as a function of time (in s). (Color figure online)

5.2 Experimentation Results

The graphs in Fig. 6 show the transient behaviors of the expected number of queued people at the stop (blue dashed line) and people traveling on a tram (red solid line), for each stop. In Sect. 5.2 and Sect. 5.2, the arrival rate at the stop is equal to 0.01 passengers/s, and it can be observed that the queue does not saturate or saturates very slowly, respectively. In particular, since empty trams periodically arrive at the stop, in Sect. 5.2 the queue is always emptied before reaching its capacity. Moreover, the expected number of people in the queue at the beginning of each tram period reaches a steady state. As the rate increases (Sect. 5.2 to Sect. 5.2), queue saturation occurs earlier. This can be mitigated by increasing the queue capacity (compare, for example, Sect. 5.2 and Sect. 5.2); however, already with arrival rate equal to 0.04 s (Sect. 5.2 and Sect. 5.2), the queue saturates within the first two periods of tram arrival.

The red curve in each figure shows an initial period in which queued people wait for the first tram to pass. With the exception of the first stop, where the first tram arrives empty (the red line starts at 0), at the other stops, the tram arrives with a number of passengers that depends on the number of people who left the previous stop. After the first period, the distribution of the expected number of people on the tram exhibits a periodic behavior. Since it is assumed that as many passengers as possible board the tram, as soon as the queue saturates, the tram becomes fully occupied, and the expected number of people on the tram reaches its maximum value. In addition, this behavior occurs very quickly in the final stops, which happens because trams arrive at the final stops already full. For example, since stop 3 tram becomes full within the third period, the next stop distributions reach the steady state almost immediately.

6 Conclusions

In this work, we illustrated how to use the ORIS tool and the SIRIO library as a software platform to develop quantitative predictive analytics. To this end, we combined a practical perspective, aimed at avoiding crowding at tramway stops to limit the spread of infection in a pandemic, with a formalized abstraction in the framework of the theory of polling systems, highlighting how the two perspectives are effectively connected through practices of MDE. Note that, while the scenario of tramway lines is used as an application example to show that ORIS and SIRIO are easily usable and provide effective support to develop efficient analytics, stochastic models for distributed concurrent systems have been widely used to represent crowd scenarios, typically exploiting fluid flow analysis based on the solution of ordinary differential equations. As an example, in [2], stochastic process algebra and stability analysis are used to evaluate coordination of agents in large collective systems. Stochastic process algebra and fluid flow approximation are used also in [10] to study emergent crowd behavior.

Experimental results achieved in Sect. 5 suggest that, though the expected number of queued people at a tram stop is recurrently perturbed by the process of tram arrivals and the expected number of tram passengers is recurrently perturbed by the process of leaving and boarding at the tram stops, the distribution of the number of queued people and the distribution of number of tram passengers at the beginning of each tram period actually reach a steady state, in just a few periods. Therefore, the steady state distribution of the number of tram passengers computed at a tram stop can be considered as the initial distribution of the number of tram passengers at the subsequent tram stop, enabling the formulation of an optimization problem where the system parameters are selected so as to balance the queue size at the various stops along the line.

Future work includes defining a theoretical framework to characterize the obtained experimental results, extending the model with additional parameters such as different limits on the number of people that can ride the tram at each stop of a line, and performing a broader experimentation by varying a larger set of parameters for a larger number of parameter values.

References

1. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer **42**(10), 22–27 (2009)
2. Bortolussi, L., Latella, D., Massink, M.: Stochastic process algebra and stability analysis of collective systems. In: De Nicola, R., Julien, C. (eds.) COORDINATION 2013. LNCS, vol. 7890, pp. 1–15. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38493-6_1
3. Carnevali, L., Grassi, L., Vicario, E.: State-density functions over DBM domains in the analysis of non-Markovian models. IEEE Trans. Softw. Eng. **35**(2), 178–194 (2009)
4. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, vol. 45, pp. 1–17 (2003)
5. Da Silva, A.R.: Model-driven engineering: a survey supported by the unified conceptual model. Comput. Lang. Syst. Struct. **43**, 139–155 (2015)
6. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Boston (1995)
7. German, R., Logothetis, D., Trivedi, K.S.: Transient analysis of Markov regenerative stochastic Petri nets: a comparison of approaches. In: Proceedings 6th International Workshop on Petri Nets and Performance Models, pp. 103–112. IEEE (1995)
8. Horváth, A., Paolieri, M., Ridi, L., Vicario, E.: Transient analysis of non-Markovian models using stochastic state classes. Perform. Eval. **69**(7–8), 315–335 (2012)
9. Ibe, O.C., Trivedi, K.S.: Stochastic Petri net models of polling systems. IEEE J. Sel. Areas Commun. **8**(9), 1649–1657 (1990)
10. Massink, M., Latella, D., Bracciali, A., Hillston, J.: A combined process algebraic, agent and fluid flow approach to emergent crowd behaviour. Tech. rep., CNR-ISTI (2010)
11. Paolieri, M., Biagi, M., Carnevali, L., Vicario, E.: The ORIS tool: quantitative evaluation of non-Markovian systems. IEEE Trans. Softw. Eng. **47**(6), 1211–1225 (2019)
12. Schmidt, D.C.: Model-driven engineering. Comput.-IEEE Comput. Soc. **39**(2), 25 (2006)

13. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5), 19–25 (2003). https://doi.org/10.1109/MS.2003.1231146
14. Sirio: the Sirio library for the analysis of stochastic time Petri nets. https://github.com/oris-tool/sirio
15. Trivedi, K.S., Sahner, R.: SHARPE at the age of twenty two. ACM SIGMETRICS Perform. Eval. Rev. **36**(4), 52–57 (2009)
16. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Softw. **31**(3), 79–85 (2013)