DataTime: a Framework to smoothly Integrate Past, Present and Future into Models

Gauthier Lyan, Jean-Marc Jézéquel, David Gross-Amblard, Benoit Combemale

DataTime: a Framework to smoothly Integrate Past, Present and Future into Models

Gauthier LYAN
Keolis Rennes, Rennes, FRANCE
gauthier.lyan@keolis.com

Jean-Marc JÉZÉQUEL, David GROSS-AMBLARD, Benoit COMBEMALE
Univ Rennes, CNRS, IRISA, Rennes, FRANCE
first.last@irisa.fr

***Abstract*—Models at runtime have been initially investigated for adaptive systems. Models are used as a reflective layer of the current state of the system to support the implementation of a feedback loop. More recently, models at runtime have also been identified as key for supporting the development of full-fledged digital twins. However, this use of models at runtime raises new challenges, such as the ability to seamlessly interact with the past, present and future states of the system. In this paper, we propose a framework called DataTime to implement models at runtime which capture the state of the system according to the dimensions of both time and space, here modeled as a directed graph where both nodes and edges bear local states (ie. values of properties of interest). DataTime provides a unifying interface to query the past, present and future (predicted) states of the system. This unifying interface provides i) an optimized structure of the time series that capture the past states of the system, possibly evolving over time, ii) the ability to get the last available value provided by the system's sensors, and iii) a continuous micro-learning over graph edges of a predictive model to make it possible to query future states, either locally or more globally, thanks to a composition law. The framework has been developed and evaluated in the context of the Intelligent Public Transportation Systems of the city of Rennes (France). This experimentation has demonstrated how DataTime can deprecate the use of heterogeneous tools for managing data from the past, the present and the future, and facilitate the development of digital twins.**

I. Introduction

So called Intelligent Public Transportation Systems (IPTS) are complex socio-technical systems, involving people (*e.g.,* the users of the networks) as well as supporting infrastructures, from the transportation means themselves (*e.g.,* buses) to the IT supporting them [1], [2]. One key feature of IPTSs are their information systems allowing a network operator to plan, analyze and manage the network with respect to metrics such as transportation time (or commercial speed), energy consumption or accident rates. When an IPTS at least partially relies on buses, these metrics are difficult to predict. Traffic indeed varies widely during the day, with rush hours further slowing down bus loading and unloading and compromising planned connections, with a significant impact on travel time. Furthermore, when some roadworks (or other unforeseen condition such as flood) happen on a street, the IPTS should be reconfigured by diverting the impacted lines using the best possible new routes, which is not an easy task for many European cities, built around centuries old, crowded and tortuous city centers. Supporting IT systems for IPTSs are typically made of two relatively independent parts, one organized around *space* and the second one around *time*. The spatial one is an a priori model of the network: what are the network topology (modeled as a directed graph), the transportation means (e.g., bus, tramways, metros), their itineraries, the infrastructure (e.g., kinds of roads), the scheduled departures, etc. The temporal one is made of the (huge) time series of data gathered from the many sensors available in the IPTS. The existing time series give the opportunity to not only provide a *model at runtime* [3], but also a full-fledged *digital twin* [4] of the IPTS to analyze, plan and manage the operations using machine learning techniques. However these space and time parts of the supporting IT are most often not well integrated, making it hard in practice for network operators to leverage the avalanche of data gathered from the IPTS. For instance diverted lines (i.e., space modification) have no historical data to learn from, so even machine learning techniques that have successfully been applied on graph-based structures typical of an IPTS [5]–[7] cannot work out of the box. Moreover, as for many other network problems (e.g., electricity or water networks), properties of interest (e.g., commercial speed of a bus along a line) are compositions of smaller independent parts (e.g. the speed on each bus inter-stops along the line). This enables the prediction on the behaviour of composite objects (a path in the graph), based on the predictions of their sub-parts (i.e. on edges) and relevant composition laws. This opens what-if analytics scenarios, where new objects never observed before can be predicted based on their simpler parts, and can then be used e.g., to optimize diversions in case of unforeseen events such as roadworks or floods. A possible direction for integrating these models (spatial, temporal and predictive) would be to articulate them around the notion of digital twin and the concept of time, i.e., digital twins extending themselves towards Past, Present, and Future [8]. For that purpose, we propose a new framework, called DataTime, to implement models at runtime which capture the state of the system according to both time and space, here modeled as a directed graph. In this graph, both nodes and edges bear local and independent states (ie. values of properties of interest). DataTime offers a unifying interface to query the past, present and future (predicted) states of the system. This unifying interface provides i) an optimized

structure of the time series that capture the past states of the system, possibly evolving over time, ii) the ability to get the last available value provided by the system's sensors, and iii) a continuous micro-learning over graph edges of a predictive model to make it possible to query future states, either locally or more globally, thanks to a composition law. We apply our framework in the context of an urban transportation system, and concretely deploy and evaluate it on the IPTS of the city of Rennes (France), that is operated by the Keolis company.

The rest of this paper is organized as follows. We first introduce the DATATIME framework (Section 2), and then we present its use in the context of an IPTS (Section 3). Then in Section 4 we describe how it was deployed at Keolis Rennes in the context of the real bus network of the city of Rennes (France), and what lessons we have learned from this experimentation. Section 5 discusses related work, before we conclude and raise several perspectives in Section 6.

II. THE DATATIME FRAMEWORK

Figure 1 shows a simplified class diagram of DATATIME. The objective of this framework is to enable graph-based seamless space and time exploration, through the analysis of historical data, real time data and predicted data. It is composed of two main parts: i) the spatial model and predictors configuration (part II of Fig. 1) that is dedicated to the designers who implement adaptations of I and II for the end-users and ii) the digital twin/shadow to reason over time about the spatial model (part I of Fig. 1), thus the digital twin built by the designers is the end-users' entry point to manipulate DATATIME. A third component has been added in the part III of Fig. 1 to represent the required endeavour to use DATATIME for the development of a particular IPTS (see Section III).

A. Spatial model and prediction configuration

1) Graph: The spatial model is defined as a classical directed graph structure as seen in the part II of Fig. 1, focusing on the `graph` package: a set of vertices that are nodes with individual characteristics, and a set of edges that are one way connections between two vertices (a bidirectional edge is simply represented with two directed edges). Graphs have a built-in `time_frame` attribute that represents the time-frame for which corresponding referential and historical data exist. Hence it is to the discretion of the designer to consider that changes amongst a graph structure yield a new graph, or if it just changes its `time_frame` by expanding or reducing it, or leaving it unchanged (e.g. when minor changes occur).

`Edge` and `vertex` characteristics (features) are made abstract in order to use any kind of data. The package `data` contains the different features and metrics definitions. In order to make those generic, we separated their identification from their actual data and type (following the type-object design pattern). This package contains three abstract classes that have to be specialized for a specific system: `Features`, `VolatileFeatures` and `Metrics`. `Features` represent characteristics of the edges and vertices. `VolatileFeatures` are characteristics that are not represented in the data, but that can be computed on the go (e.g bus line type extrapolated from its identifier, electrical cable resistance computed from its length, etc.). `Metrics` are data that are not characteristics, such as measures (speed, volume of water per hour, etc.) or time related information. Hence, data instances that extend those classes must be able to query data from the `Digital Shadow` (cf. II-B). The package `path` represents the abstraction of a path through the graph, that is, a sequence of at least one edge for the thinnest grain. `Paths` are abstract, hence one can easily create a path hierarchy if relevant for the targeted application domain (e.g transportation networks, supply chains, smart-grids, drinking water networks, ...). Paths allow the building of hierarchical structures within the graphs while offering analysis and predictions scalability using micro-learning for the predictive aspects. Recently, micro-learning approaches have been successfully applied to graphs structures to provide what-if scenarios (e.g. in the context of power grid management [9]). In these approaches, specific models of local data are built instead of one large model on the overall data set. Micro-learning is typically useful for incremental predictive models, where one has to perform step-by-step decisions based on local properties, while yielding quality predictions. For instance for an IPTS, prediction of travel time or road reliability can be made using micro-learning over each edge and then aggregating the results using a specific composition law (sum for travel time, product for reliability, etc.). In DATATIME, data analysis and predictions are made at the level of edges (provided a predictive model has been configured for this purpose). For graphs on which data can be aggregated from edge level to different implementations of parent paths, the prediction or analysis at edge level will be automatically aggregated to higher levels with specific user defined composition laws (e.g speed of a bus between two bus stops, aggregated to the whole bus line, water leaks at single pipes sections, aggregated to the whole pipe, etc.).

2) Predictors: The `prediction` package contains the abstract class `Predictor` that embodies the predictive system of DataTime. It encapsulates predictive models and miscellaneous tools that define the expected behaviors of predictive models, from training to predicting. A specificity of this abstract class is that we included a way of managing the predictive models behavior through time. It proposes the following services:

- Choice of the predictive model, in order to choose the best suited predictive model depending on the prediction issue (using the strategy design pattern).
- Optimization of the model by searching for the best hyper-parameters . It embeds a grid search system that can either use a set of properties to seek from, or a random one (through the parameter `optimize` of `Predictor.train()` in Fig. 1).
- Training of the model, by feeding it with a training dataset, hyper-parameters and choosing whether to optimize the predictive model using grid search (through the parameters of `Predictor.train()` in Fig. 1).

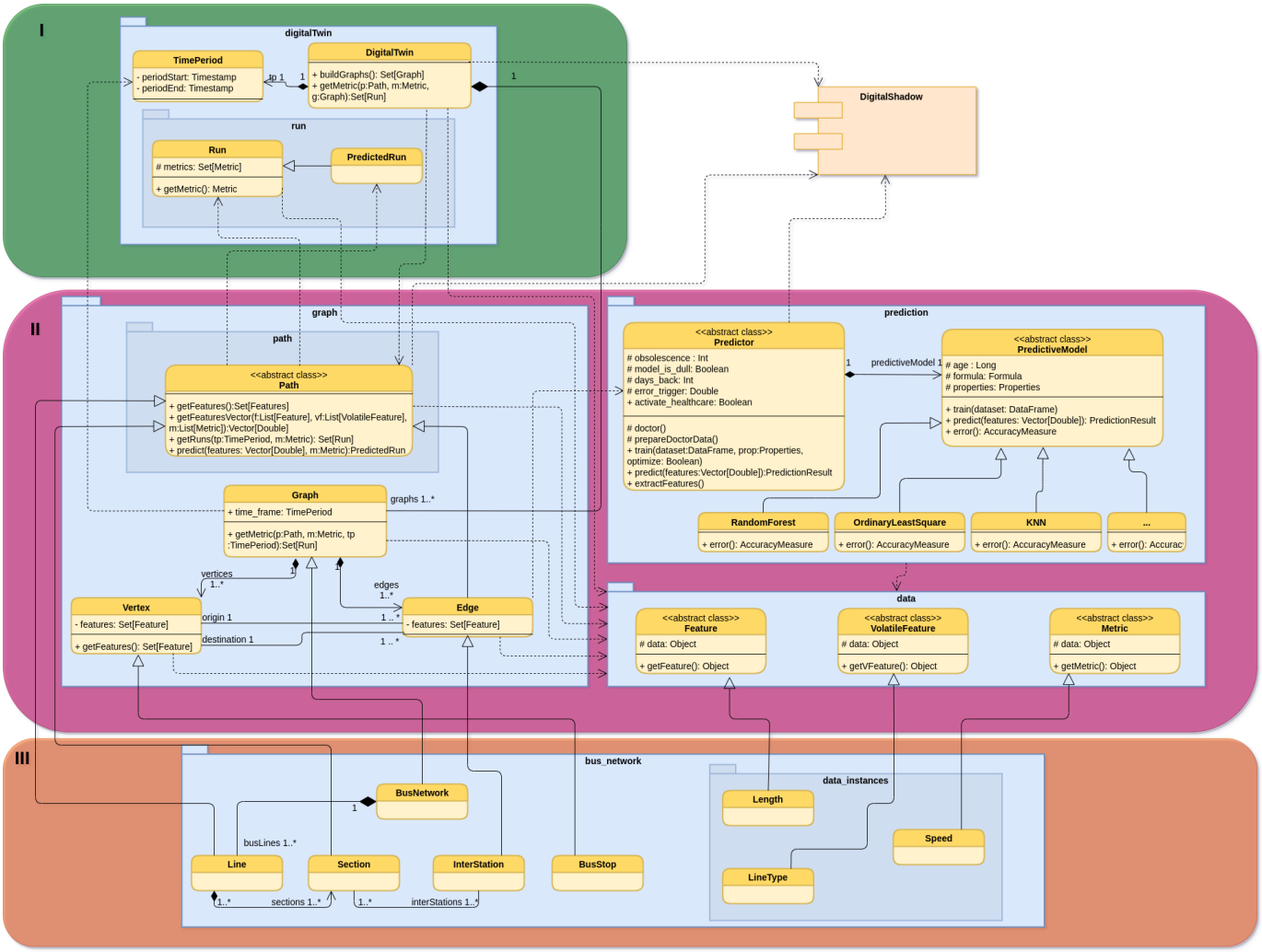

Fig. 1. The DataTime framework (excerpt)

- Saving and loading the predictors by transferring their data to the `Digital Shadow` interfaces (through the `Predictor.train()` method in Fig. 1).
- Model's health monitoring, that takes care of the predictive model ability to predict in a satisfying manner depending on its age and a prediction error threshold over which a new model is trained. This service can be disabled if needed (using the attribute `Predictor.activate_healthcare` in Fig. 1).
- Features extraction for prediction, that explores the edges features to make predictions (`Predictor.extractFeatures()` in Fig. 1).
- Predictions, by returning a `PredictedRun` when called (`Predictor.predict()` in Fig. 1).

To configure a predictor, a designer has to extend `Predictor` and implement the `prepareDoctorData()` method, whose work is to prepare data for the health check-up of the predictive model and the training of a new one when needed. The `doctor` is called when one wants to predict something and that the predictive model is more than `obsolescence` days old compared to the current system date. The objective behind that is to keep predictors up to date and efficient. We achieve this by comparing the residuals obtained predictin from a sample of the last available data from now to `back_days` back in time with their base error (the deviation of the predictive model obtained during the training phase). If the predictive models yield an average residual that is over `error_trigger` times the base error, then a new model is trained with the data yielded by `prepareDoctorData()`. The `doctor` can be deactivated if needed (for example when the models are not subject to derivation because, e.g. the data is quite consistent through time), by setting `activate_healthcare` to false.

`Predictors` can use different predictive models thanks to the abstract class `PredictiveModel`. If one uses a single machine learning API in which predictive models are children of a single interface, then any model of such an API should be made available by encapsulating them in an extension of `PredictiveModel`, by tweaking the `error()` method for each model (because the error computation varies, and

the result is not always kept into a trained model), that is necessary for the `doctor()`. Of course, if one wants to code her own predictive model, the simplest way is to extend `PredictiveModel` and override the relevant methods.

Predictors are refered by edges only, due to the fact that our system relies on compositional prediction. In order to make any prediction with any machine learning model, predictors should have their own set of features, and prediction targets. Hence the edges that call the predictors are responsible for giving them the right features by using their inner method `getFeaturesVector` that transforms their features into a vector of doubles for the predictor (with an ad hoc encoding for categorical data). Note that if some external features that have not been designed in the model have to be passed to the predictor, a container should be built to contain them. Predictors should then take the data they need directly within this object. The training of predictors is made by the `Edge` class, that calls the method `train(...)` of `Predictor`. This method has to be fed with a dataframe containing the appropriate training data (i.e. in accordance with the predictor feature set, thereby the predictive model formula), obtained by querying the digital shadow, and a set of hyper-parameters through the parameters `prop`. The parameter `optimize` triggers a randomized grid search algorithm when set to true in order to find the best configuration for the predictive models hyper-parameters. Thanks to this, any machine learning model with any features set and prediction target can be trained and used at any path level in the framework. Finally, the class `Graph` is responsible for the manipulation in a single graph instance such as seamless data analysis (through the method `getMetric(...)`), getting the corresponding historical or real-time data from the `Digital Shadow`), what-if scenarii (creating new edges, vertices, path and analysis against those new objects), and the orchestration of the data between the different parts of the model such as instantiating and feeding predictors with data when they need to be trained (through its edges). The main goal of this class is to provide a way to obtain a set of `runs` when one calls the method `getMetric(...)` by seamlessly returning historical or real time or predicted results from historical runs or predicted runs over the given `path` and time period passed to the method `getMetric(...)`. Note that it does not matter if there is one or more instances of graphs (e.g. a set of independent graphs distributed through time, with contiguous timeframes). In both cases it is the `getMetric(..)` method of `Digital Twin` that is used by the end-user to explore data in time and space.

B. Digital twin/shadow description

The `Digital Shadow` is either the representation of an existing data environment on which DATATIME can be plugged in a read-only manner in order not to have any side effect on the existing information system, or an actual fully managed data environment dedicated to DATATIME, making it responsible for the management of all the data flows between the digital twin and the real system. The digital shadow should also be responsible for the saving of the graphs instances and predictors instances. Moreover, it has the responsibility of creating time-series used for data analysis while keeping them optimized and consistent through time. In short, any data that is yielded either by the `Digital Twin` of DATATIME or the real system sensors, etc. should transit and be governed by `Digital Shadow`. Fig. 2 shows an example of how data transits through the system when `getMetric(...)` is called. The `Digital Twin` of DATATIME represents the set of `graph` instances of the model, over the time. It contains the time representation through the class `TimePeriod` that embeds two timestamps. This class helps the representation of time frames and instants (when `periodStart == periodEnd`), and time manipulation in the framework. The time periods are used to request the set of graphs that are part of the class `Digital Twin`, when calling the method `getMetric(...)`, that returns a set of `runs` which represent the result of the queries made by the `Digital Twin` over its set of `graphs`. In other words, `runs` model the action of traveling through a `path` at a given time frame, or a specific event on a `path` for a given time frame, e.g. a bus traveling along a bus line, an amount of water traveling through a section of a pipe network, the loss of electrical current between two poles, etc. Runs are built using historical data. Hence they consist of different measures & metrics made on the network represented by an instance of the `graph`. Runs should be atomic, hence they should be unique and they should be bound to `edges` only, with runs of longer paths built by aggregating runs of their `edges`. Runs can be historical / pseudo real-time (historical data) or predicted (using machine learning). Depending on the size of the historical data, runs should be lazily loaded when analyzed. However, saving the runs can be a bad idea if the historical data is huge: it would lead to storage starving in addition to duplicated data. However, this decision depends on the `Digital Shadow` structure. The `Digital Twin` class is also responsible for the building of graphs instances by querying the `Digital Shadow` for referential data, yielding `Graph` instances. Hence the `Digital Twin` is the end-user entry point on the framework.

III. APPLICATION TO AN INTELLIGENT PUBLIC TRANSPORTATION SYSTEM

To assess the applicability of DataTime, we developed a SCALA implementation of the framework adapted to urban bus networks.

IPTS such as modern urban bus networks are complex socio-technical systems made of hundreds of stakeholders, including humans, sensors, vehicles, information system, etc. Hence, a major concern with such an infrastructure is to gather, organize and normalize data, analysis and decisions in integrated tools. IPTS are slowly evolving networks, yet if an important part of the bus lines are non changing for very long and continuous periods, there exists some variation within their structure, such as the changes on bus lines when long-term roadworks are planned, the creation of new bus

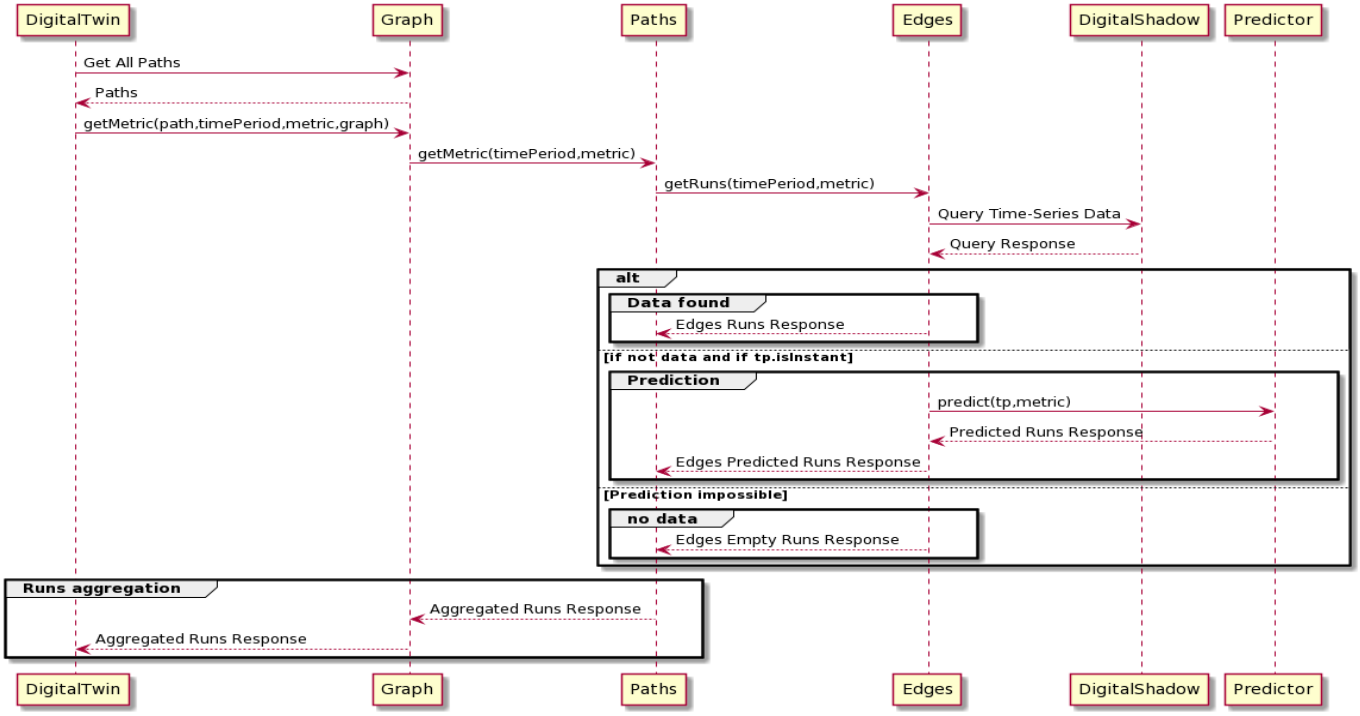

Fig. 2. Sequence diagram for getMetric

lines, etc. Hence we considered that the `time_frame` of a single instance of the bus network is corresponding to the period that was defined by the operator (usually 1 to 2 weeks), during which there are no major modifications except for some day to day bus line deviations. In our applicative environment, the information systems were already provided, also used by others. Hence, we plugged our framework on top of the existing `Digital Shadow` in order to avoid any side effect on it. In our implementation, the `Digital Twin` is responsible for the creation of as many graph instances corresponding to the different referential files that exist in the data. Referential files contain actual definitions of the bus network structure, as defined by the operator. i.e, if there are four sets of referential files, each of them defining a bus network that is valid for a given period of time, four different `graphs` instances will be created.

A. Graph and paths adaptation

We specialized DATATIME for bus networks issues as shown in the bottom part of Fig. 1. The bus network is an extension of the class `Graph`, for which the longest path within it are `BusLines`, made of `Sections` that are an ordered collection of `InterStations` (Edges). The class `Vertex` is representing the bus stops (stations) in this context. All of the elements of the graphs are immutable, in order to keep data consistency when one wants to make, e.g, an analysis over a line/section/inter-station that exists in all the members of a set of graphs.

B. Features adaptation

`InterStations` contain `Features` such as length of the inter-station (in meters), type of road (one-way, two ways, reserved for buses, etc.), number of traffic lights, number of pedestrian crossings, etc. All those features are physical features, extracted from OpenStreetMaps[1] Section and line features are obtained by aggregating their respective subpart features. In order to manage features in the bus network implementation, we created `data instances` that extend the abstract classes from the package `data` (following the type-object design pattern). We then created several data features classes (e.g. `Length`, `TrafficSignalsCount`, etc.), one `VolatileFeature` class, called `LineType` and as many `Metrics` classes as needed amongst which `Speed`, `TravelTime`, etc.

C. Runs adaptation

If we take a look at `Run`, the data they correspond to in the `Digital Shadow` is the data gathered from bus trips all over the network. We describe trips as follows:

- Trip in edges: Bus trips from bus stop `origin` to bus stop `destination`;
- Trip in sections: Bus trips `origin` to bus stop `destination` within its ordered collection of edges;
- Trip in line: Bus trips from origin to destination (terminals).

[1]https://gitlab.inria.fr/glyan/osm_bus_extractor

Trips contain metrics such as start time, arrival time, travel time, speed, dwell time (for sub paths only). One could easily add any external feature such as smart card data, weather, traffic, etc.

D. Data mass and lazy loading

Since an IPTS typically produces several GigaBytes of data per month, we soon end up with more data than can be managed on a local file system. We thus need a kind of lazy loading mechanism when querying trip data. Accordingly, runs are "built" on demand when a call is made through the method `getMetric()` from the `!DigitalTwin`. The resulting trips can be kept in memory up to a certain extent, but not saved. An advantage of lazy loading is that this allows the ingestion of data in pseudo-real time (e.g, when there is a need to observe what is going on in the bus network with a reasonable delay).

IV. Experimentation at Keolis Rennes

A. The Keolis Rennes Bus Network

Keolis Rennes is the company that manages the Urban Public Transportation Network (UPTN) of the city of Rennes, France. This network, named STAR[2], is based on a central subway line, and a wide bus network that serves both the city of Rennes and all the suburban areas. In total, the bus network covers more than $550km^2$ The company manages a total of 116 bus lines over which more than 600 buses can be traveling during rush hours. The bus network information system is made of several sub systems including an Automatic Vehicle Location (AVL) that yields large amount of fine-grain data (inter-station) both in real and delayed time. This data contains timestamped and located information such as buses speed, travel time, dwell time, etc. The data we gathered for 2 years weighs more than 40GB.

B. Evaluation of the Predictive Model

The first stage in the deployment of DataTime at Keolis Rennes was to evaluate the precision of the predictive model, in order to build confidence into the whole framework. For that, we ran an experiment described as follows:

1) We choose a set of 13 typical and different bus lines for which data could be gathered all along their path. This includes express, urban, inter-district and suburban bus lines;
2) We ran 13 experiments for each line, removing the respective line data from the training dataset that gathers data over year 2019;
3) We predicted the speed of each bus line using micro and macro-learning (i.e the training data consists of the set of full line trips data for macro-learning, while it is made of inter-station trips data for micro-learning);
4) We compared the aggregated micro-predictions and the macro-predictions to assess whether micro-learning is suitable for our model, and thus could be also used for unforeseen lines (e.g, in cases of traffic diversions).

[2]https://www.star.fr/

Table I shows the averaged results of the experiments. It presents three prediction error measures for both macro and aggregated predictions. Those measures are the Root Mean Squared Error (RMSE) in km/h (eq. 1), Mean Absolute Error (MAE) in km/h (eq. 2) and Mean Absolute Percentage Error (MAPE) (eq. 3) in percent. All of these measures are the result of the comparison of the predictions with their actual counterparts, hence, the lower the error, the better the prediction. Note that the RMSE is more sensitive to outliers than MAE and MAPE, which are not quadratic.

$$RMSE = \sqrt{\frac{1}{N}\sum_{1}^{N}(Y_t - \hat{Y}_t)^2} \quad Y_t = actual, \hat{Y}_t = predicted \quad (1)$$

$$MAE = \frac{\sum_{1}^{N}(Y_t - \hat{Y}_t)}{N} \quad Y_t = actual, \hat{Y}_t = predicted \quad (2)$$

$$MAPE = \frac{1}{N}\sum_{1}^{N}\left|\frac{Y_t - \hat{Y}_t}{Y_t}\right| * 100 \quad Y_t = actual, \hat{Y}_t = predicted \quad (3)$$

For all measures, the precision obtained for the prediction is considered as good enough from the point of view of the bus operator (i.e Keolis Rennes). Further, the micro-learning approach is at least as good as macro-learning approach[3] in terms of accuracy and offers state of the art performance [10]. The three micro-learning approach predictions error indicators are lower than those of the macro-learning approach. Hence this experiment validated the use of micro-learning predictions for bus networks such as the ones operated by Keolis in Rennes. Finally, the utmost interest of micro-learning is the possibility to predict at the finest possible grain within the bus network, hence build what-if scenarii such as the prediction of bus travel time on new or diverted bus lines.

TABLE I
Prediction accuracy

Aggregated-micro			**Macro**		
RMSE	**MAE**	**MAPE**	**RMSE**	**MAE**	**MAPE**
3.0	2.4	12.1%	3.2	2.9	12.2%

C. DataTime in Practice at Keolis

Our DataTime implementation was designed to be used by bus networks operators. Thanks to the framework we could develop the following features:

- The creation of new elements over the bus network, i.e. bus lines (or new bus lines sections), bus stops, inter-stations.
- The analysis of any event on the network at any place and anytime in the bus network. For instance lines 1 to 20 in example 1 access past data using the method `getMetric()` with a time period located in the past

[3]The full details of the validation experiments ia available at: https://gitlab.inria.fr/glyan/compred

- The prediction of any metric on the network at any place and anytime in the bus network, e.g, Algorithm 1 lines 21 to 29 where the method `getMetric()` is call with a parameter meaning *now*
- Providing the operators hints on which detour should be applied on a bus line depending on, e.g, the expected speed of this detour. Even in a complex bus network with multiple possible paths between two bus stops, finding alternative possible routes is quite standard. It is typically solved with a search based greedy algorithm, parameterized with a maximum number of paths of a maximum length. The complex part is of course *evaluating* the suitability of the possible routes that the search algorithm would yield. Depending on the goal of the network operator (e.g, smallest travel time, best reliability, etc.), the system would then just have to pick the best path among the existing ones. As long as the system is able to predict a metric (speed, reliability) on any known or unknown edge, along any path, compositional prediction makes it possible to aggregate those predictions on any number of different paths allowing it to choose the best way to propose a detour, following one or more specific heuristics.

  Algorithms 2 and 3 summarize the main steps of putting everything together to automatically apply a bus detour if the bus operator asks the system to.Algorithm 2 explains how to find all possible paths between two vertices in the graph, with some constraints on the number of iterations and paths found size. Algorithm 3 shows how to call algorithm 2 to yield an optimized deviation for a bus line, searching to maximize the speed of the deviation.

Algorithm 1 Different call examples over the getMetric method

```
1: val busLine:Line = Graph.getLine("152", Direction.B)
2:
3: // TimePeriod for which historical data exists
4: var tp:TimePeriod = TimePeriod("2019-01-01 08:00:00","2021-
   02-01 00:00:00")
5: var metric:Metric = Speed
6:
7: // Will return all the runs found for line 152 over tp
8: var runs:Set[Run] = getMetric(busLine,tp,metric)
9: runs.foreach.displayMetric()
10:
11: // TimePeriod for real time Data
12: val now:Long = System.currentTimeMillis
13: tp.setPeriodStart(now)
14: tp.setPeriodEnd(now)
15:
16: /** Will return runs corresponding to the last data available
17: * in the digitalshadow, comparing with now
18: */
19: runs = getMetric(busLine.getSections.head,tp,metric)
20: runs.foreach.displayMetric()
21:
22: // TimePeriod for future date
23: tp.setPeriodStart("2022-03-19 17:30:00")
24: tp.setPeriodEnd("2022-03-19 17:30:00")
25: metric = TravelTime
26:
27: // Will seamlessly return a predictedRun
28: runs = getMetric(busLine.getInterStation(5),tp,metric)
29: runs.foreach.displayMetric()
```

D. Lessons Learned

1) Technical considerations: In practice, real world data is dirty [11]. In our case, we gathered data on a 6 months period between early July 2018 and late January 2019. The data collected is based on the bus network AVL system which contains 116 bus lines. We made some data quality analysis of this data in order to build data-cleaning processes if needed. Table II shows the summary of the raw data. As one can see, the outliers in the dataset are quite preposterous: -996 km/h and 130 km/h are impossible speeds for buses. Moreover, readings for which bus speed was lower than 1 km/h or higher than the legal speed limit of 70 km/h, which respectively are low speed limit we defined and legal maximum speed for buses in France, can represent up to 5.4% of the dataset, which is non-negligible.

The main dataset came along with a twin that contains more reliable data, nonetheless at a coarser time grain (20 seconds instead of 1 second accuracy). Hence, we curated the data by using both filtering (removing outliers) and ad-hoc merge-join techniques. We managed to reduce the error rate to 0% while getting rid of every outlier. We published a report detailing this work in [12].

2) Performance and scalability: We fed our implementation of DATATIME using Apache Spark v3.1.1. It was behaving as a datasink, making it possible to query a datalake containing more than 2 years of data, totalizing nearly 40GB. As an example, the primo-execution of a query like the one visible at lines 1-9 in Algo.1, which consists of querying over all the trips of a bus line for a 2 years period, takes around 40 seconds on a computer equipped with a middle end 8 cores x86 CPU. If one executes this query a second time, the result is almost instantaneous provided the last request results were kept in memory. In a nutshell, the bigger the period and the longer the path, the slower the querying will be. Hence, the performance and design of the `Digital Shadow` services are paramount for querying to be efficient.

The predictors, that are able of continuous learning (by automatically training new predictive models when needed), must be trained before being able to predict. The training time depends on many factors but there are 3 of them that will have a significant impact. Those are the size of the training dataset, the number of features and the hyper-parameters tuning. The latter can be the worst one if, e.g. one uses grid search to find the best set of hyper-paremeters (using `optimize=true` in the method `train(...)`). We decided to train models with a training dataset that contains 1 or 2 month of data, with a set of 8 features and the default set of hyper-parameters. The training of such a model takes no longer than 10 minutes on a middle end computer (8 cores, 16GB RAM). Hence, the continuous training if the predictive models eventually turn dull would be in the same time range. This is rather acceptable knowing

Algorithm 2 Detours finding (Scala inspired pseudo-code)

```
/** Call function **/
def detours(orig, dest, maxLength, maxDetours):
val detoursSet = new Set[Set[Edge]]()
/** Get the edges starting from orig **/
val possibleEdges:Set[Edge] = GRAPH.edges.filter(_.orig == orig)
val edgesSet = new Set[Edge]()
for (e: Edge in possibleEdges) do
    _detours(e, dest, 1, maxLength, maxDetours, detoursSet)
end for
return detoursSet
end detours

/** Recursive function **/
private def _detours(currentEdge, dest, order, maxLength, maxDetours, edgesSet, detoursSet):
if (detoursSet.length < maxDetours and order < maxLength) then
    /** Non duplication check **/
    val vertexInSet:Boolean = if (edgesSet.isEmpty) false
    else edgesSet.map(_.orig == currentEdge.dest).reduce(_||_)
    if (currentEdge.dest == dest and not vertexInSet and edgesSet.add(currentEdge)) then
        /** FINAL CASE **/
        detoursSet.add(edgesSet)
    else if (not edgesSet.contains(currentEdge) and not vertexInSet) then
        /** GREEDY CASE **/
        val possibleEdges:Set[Edge] = GRAPH.edges.filter(_.orig == currentEdge.dest)
        /** This loop should be parallel for better performance **/
        for (e:Edge in possibleEdges) do
            edgesSet.add(currentEdge)
            if (not edgesSet.contains(e)) then
                _detours(e, dest, order + 1, maxLength, maxDetours, edgesSet.clone, detoursSet)
            end if
        end for
    end if
end if
end _detours
```

TABLE II
DATA SETS PROPERTIES

Population	Average speed	Speed standard deviation	Minimum speed	Maximum speed	Speed error rate
16793293	20.31 km/h	11.6 km/h	-996 km/h	130 km/h	5.4%

Algorithm 3 Example of how to get the best detour on line A2 between A and B for the morning rush hour of Tuesdays during working periods (Scala inspired pseudo-code)

```
val line:Line = GRAPH.lines.A2
val detourStart:Vertex = A
val detourEnd:Vertex = B
val originalRoute:Set[Edge] = line
.getSubRoute(detourStart,detourEnd)
val day = Days.TUESDAY
val holidays = Holidays.NONE
val period = Periods.MORNING_RUSH_HOUR
val metric = Metrics.SPEED
val possibleDetours:Set[Set[Edge]] =
detours(detourStart, detourEnd, 20, 10)
.except(originalRoute)
val metrics:Set[Double] = possibleDetours
.map(_.predict(metric,TimePeriod(holidays, day, period))
val bestDetour:(Set[Edge],Double) = possibleDetours
.zip(metrics).sortBy(_._2).last
GRAPH.enact(line,bestDetour)
```

that a single model is able to yield predictions for the whole system.

Finally, our implementation is scalable in these ways:

1) The digital shadow can rely on scalable databases, hence the storage and querying can be distributed amongst different machines if needed.
2) The implementation of the framework can be used on any machine and does not need a tremendous amount of computing power to analyze data or predict data: the data collection for analysis relies on the digital shadow ability to scale, and the predictive models training time is short even on a single machine.

3) Impact at Keolis: Industrial fields that are not computer-science focused often face an issue that we could call "the data overwhelming problem". This consists of having more and more complex information systems that generate more and more data, with a small team of IT engineers who are already too busy keeping the information system healthy. Thus the use of data that the company owns becomes quite impossible. In

companies that rely on stable and well known technologies such as standard relational databases and spreadsheets to analyze them, such an amount of data yields the impossibility to study large samples of data. Specifically if those companies core workforce is made of domain experts for which computers are tools and services providers only. As an example the bus network of Rennes generates nearly 2GB of data per month for the sole AVL system, and more than 10GB if we consider all the stakeholders of the bus network information system. Thus, for the domain experts of the bus network who are provided low to middle end computers, the analysis of large samples of data is quite a challenge when it is not merely out of reach. Thereby, a framework such as DATATIME favors the data centralizing with a focus on scalability for data analysis, making it possible for domain experts to integrate past, present and future within a traditional information system containing a priori models. Moreover, DATATIME could be even more user friendly with the providing of future DSLs, for, e.g, facilitating the integration of new data sources, the edition of line topologies or make data analysis more fluid, etc.In addition, DATATIME allowed us to highlight specificities of the bus networks that were yet not visible for the operator. Indeed, the use of predictive models allowed the analysis of feature importance, hence to do sensitivity tests for, e.g. speed and vehicle engine type. It appeared that the electrical buses that were test running on the bus line 12 for a few months were systematically slower than their combustion counterparts, which was later explained by their higher gravity center, due to the presence of the traction batteries on the buses roofs.

V. RELATED WORK

There are three main ways for obtaining predictive models for transportation networks: build an analytical model (e.g. using mathematical modeling), get it from simulations (e.g, multi-actors), or through machine learning. This holds for any kind of transportation networks, be it water, electricity or buses, but in this section we mostly consider those related to IPTS.

A. Analytical models

First, one can try to build an analytical model that, when adequately configured, produces quality predictions for e.g. bus speed in bus corridors (constraint environment).

Fernandez and Valenzuela [13] proposed an efficient analytical model to predict bus commercial speed anywhere on a bus network, for any kind of bus line. They upgraded a state of the art function based on exponential decay to take in account more influencing parameters such as dwell time at bus stop, passenger density or even time periods or bus technology. Their model must be precisely parameterized by tweaking weights in order to produce satisfying result regarding the reality. Thus, the use this model requires the gathering of a lot of data and domain knowledge in order to obtain satisfying results. In the same way, Valencia and Fernandez [14] developed a similar approach dedicated to bus corridors (bus lanes). Their work presents similar characteristics, hence pros and cons, as the model described before. Analytical models are built by and for specific issues. In these particular contexts they can be very powerful when adequately tweaked. The other side of the coin is that these models are very static, hence non applicable to others issues without a total rethinking of their behavior. Moreover each model is tweaked to fit a specific situation, making its portability to other instances of the same problem difficult.

B. Simulation models

Various scale simulations can be used to mimic an environment and then observe how e.g. a new traffic-lights system impacts the bus travel time at a crossroads (fine grain), or simulate the traffic flow on a whole city (coarse grain).

Simulations are dynamic models that aim at providing estimations obtained from a simulated environment. The closer the environment is to the actual environment, the better the estimations are expected to be. This kind of tool is usually dedicated to a specific task at a given granularity. Indeed, simulations are usually classified in 3 different categories [15]:

- Macroscopic: Simulation of a whole city (or even more), emulating traffic flow dynamics (inspired by fluids physics);
- Mesoscopic: Simulation of a district, involving explicit treatments of intersections;
- Microscopic: Simulation of a crossroad or a few roads using multi-agents simulation with complex rules and interactions among agents.

However these kind of simulations ask for a lot of resources to be modelled and ran properly: The cities and agents must be manually modeled, and the computing of those simulations often involves multiple CPUs and GPUs. On the other hand, DATATIME may yield quality predictions and can be deployed by a few engineers, provided they have access to accurate data.

C. Machine learning

Machine learning can be leveraged using different methods to predict e.g. travel time of bus lines or bus stop arrival time. Machine learning models are mostly used for travel time prediction in the literature: Altinkaya and Zontul [16] reviewed the computational models used in Urban Bus Arrival Time Prediction as of 2013. Their work covers the use of historical data and statistical models, time-series, regressions, neural networks and hybrids models. They explain that the existence of that many models in this field of research is due to the fact that predicting travel time of buses is a complex task for which no model in particular has proven its superiority yet. However, they claim that hybrid models (e.g combine neural networks with Kalman filters) are on the roll. Moreover, they add that predictions might be better if one splits the datasets into sub-datasets in which data represents similar conditions, restricting the models prediction scope. Thereby, our micro-learning approach can be considered as a divide and conquer method following this idea.

Mendes-Moreira and Barachi [17] proposed a prediction model for networks by predicting sub parts of the networks

and re-conciliate the aggregated predictions of the sub-parts with the path they are part of. They do this using a method they called Reconciliation For Regression (R4R) by weighting each sub-prediction using a constraint least square algorithm. Their results show that they reach state of the art performance for bus travel time prediction. However, it is unclear on how far from the reality their model perform without MAPE. One could also raise the following statement: the added complexity of R4R is questionable because it shows that it seems to never offer a better improvement than 3% in prediction precision when compared to other models, including simple ones such as Multivariate Linear Regression (MLR). Our method does not correct data on aggregation, yet the prediction of our predictive system are satisfying. However, the enhancing of the data quality and the concept of prediction error reconciliation must be considered when building a predictive system based on aggregated prediction.

D. Digital twins

Bordeleau et al. [4] suggested that models at runtime are key for implementing digital twins. Our work feeds this claim in addition to adressing the following open issues, raised by their work: 1) Models and Data, and 2) Architectural Framework for Digital Twins for the specific case of spatio-temporal models. Kirchof et al. [8] worked on the interconnectivity of digital twins, their related information system and the cyber physical system they are the virtual image of. Their work could be used to enhance the inter-connectivity of our digital twin and digital shadow, diminishing the endeavour the designers would provide to efficiently implement DataTime.

E. Integrating a priori models with models learnt from data

Combemale et al. [18] proposed the conceptual models and data (MODA) framework. They aim at providing a reference for model-driven and data-driven modelling issues. Their work proposes a data-centric and model-driven approach to integrate heterogeneous models and data into a single framework for the entire life-cyle of socio-technical systems. DATATIME can be considered as a practical implementation fitting the MODA conceptual framework.

Hartmann et al. [5], [9] worked on temporal graphs to analyze data in spatio-temporal dimensions, which embed historical analysis and predictions. Their tool, Greycat, includes a scalable graph-oriented data model that allows the building and analysis of multiple parallel worlds (forks of graphs) in a single framework. Their model is meant to be used for fast evolving networks such as smart-grids, cyber-physical systems or IoT systems that yield a lot of data and that can physically evolve quickly. The strengths of their model are the following: their model is totally and quickly scalable with a reasonable resources needs; they embed machine learning seamlessly in order to predict events on the graph and even build what if scenarii by forking graphs, yielding new instances with inherited properties. Moreover, they took a look at the interest of applying the "divide and conquer" paradigm in order to make predictions over the smaller parts of the graph (Nodes). Their findings is that for complex data models that are made of an aggregation of smaller parts, machine learning models that are trained using fine-grained data outperforms models trained with coarse-grained data (i.e parts or whole graph). Greycat has been successfully tested on a smart-grid application in Luxembourg, and is an interesting way of thinking the interactions between data and models at large scale on evolving environments. DATATIME clearly reuses the same graph-based, micro-learning strategy as GreyCat, while being optimized towards slowly evolving graphs that are typical of IPTS, and bringing a full integration of the temporal dimension (past, present, future) on top of the existing a priori model of the network.

VI. CONCLUSION AND PERSPECTIVES

In this paper, we propose a framework called DATATIME to implement models at runtime which capture the state of a socio-technical system according to the dimensions of both time and space. Space is modeled as a slowly evolving directed graph where both nodes and edges bear local and independent states (i.e, values of properties of interest). DATATIME provides a unifying interface to query the past, present and future (predicted) states of such socio-technical system, hiding the complexity and scalability issues of dealing with huge time series and machine learning. We applied our framework in the context of an urban transportation system, and concretely deployed and evaluated it on the IPTS of the city of Rennes (France). DATATIME makes it possible for domain experts to integrate past, present and future within a traditional information system containing a priori models. Still up to now, using DATATIME requires experts to work at the level of the chosen programming language, here Scala, which is seldom known to the IT department of an IPTS operator such as Keolis. We thus plan to make it easier to use the DATATIME framework by providing a set of DSL embodying its main abstractions, thus facilitating the integration of new data sources, the edition of the network topology, or make data analysis more fluid. As future work, we envision to apply our framework for other application domains that bear structural and temporal similarities with IPTS, such as smart grids, water abduction systems, or supply chains. As long as they fit the directed graph abstraction at the core of DATATIME, and as long as global properties of interest could be obtained by composing atomic values associated to edges, we do not foresee any issue in specializing DATATIME towards these systems.

REFERENCES

[1] O. Aloquili, A. Elbanna, and A. Al-Azizi, "Automatic vehicle location tracking system based on GIS environment," *IET Software*, vol. 3, no. 4, p. 255, 2009.

[2] S. Riter and J. McCoy, "Automatic Vehicle Location - An Overview," *IEEE Transactions on vehicular technology*, 1977.

[3] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg, "Models at Runtime to Support Dynamic Adaptation," *Computer*, pp. 46–53, 2009. [Online]. Available: https://hal.inria.fr/inria-00477529

[4] F. Bordeleau, B. Combemale, R. Eramo, M. Van Den Brand, and M. Wimmer, "Towards Model-Driven Digital Twin Engineering: Current Opportunities and Future Challenges," in *ICSMM 2020 - International Conference on Systems Modelling and Management*, Bergen, Norway, Jun. 2020. [Online]. Available: https://hal.inria.fr/hal-02946949

[5] T. Hartmann, A. Moawad, F. Fouquet, and Y. L. Traon, "The Next Evolution of MDE: A Seamless Integration of Machine Learning into Domain Modeling," *Software & Systems Modeling volume*, p. 17, 2019.

[6] H. Amirat, N. Lagraa, P. Fournier-Viger, and Y. Ouinten, "Myroute: A graph-dependency based model for real-time route prediction," *JCM*, vol. 12, p. 668, 2017.

[7] B. Taskar, M.-F. Wong, P. Abbeel, and D. Koller, "Link prediction in relational data," *Advances in neural information processing systems*, vol. 16, pp. 659–666, 2003.

[8] J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems," p. 12, 2020.

[9] T. Hartmann, F. Fouquet, A. Moawad, R. Rouvoy, and Y. Le Traon, "GreyCat: Efficient what-if analytics for data in motion at scale," *Information Systems*, vol. 83, pp. 101–117, 7 2019.

[10] T. Berger-Wolf and N. Chawla, Eds., *Proceedings of the 2019 SIAM International Conference on Data Mining*. Philadelphia, PA: Society for Industrial and Applied Mathematics, May 2019.

[11] M. A. H. Ndez and S. J. Stolfo, "Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem," p. 29, 1998.

[12] G. Lyan, D. Gross-Amblard, J.-M. Jézéquel, and S. Malinowski, "Impact of Data Cleansing for Urban Bus Commercial Speed Prediction," Université de Rennes ; IRISA ; Keolis Rennes, Research Report, May 2021. [Online]. Available: https://hal.inria.fr/hal-03220449

[13] R. Fernandez and E. Valenzuela, "A MODEL TO PREDICT BUS COMMERCIAL SPEED," *Traffic Engineering & Control*, vol. 44, no. 2, 2 2003.

[14] A. Valencia and R. Fernandez, "A method to calculate commercial speed on bus corridors," *Traffic Engineering & Control*, p. 8, 6 2012.

[15] J. Barceló, J. Casas, D. García, and J. Perarnau, "A METHODOLOGICAL APPROACH COMBINING MACRO, MESO AND MICRO SIMULATION MODELS FOR TRANSPORTATION ANALYSYS," p. 24, 2005.

[16] M. Altinkaya and M. Zontul, "Urban Bus Arrival Time Prediction: A Review of Computational Models," vol. 2, no. 4, p. 7, 2013.

[17] J. Mendes-Moreira and M. Baratchi, "Reconciling Predictions in the Regression Setting: An Application to Bus Travel Time Prediction," in *Advances in Intelligent Data Analysis XVIII*, M. R. Berthold, A. Feelders, and G. Krempl, Eds. Cham: Springer International Publishing, 2020, vol. 12080, pp. 313–325, series Title: Lecture Notes in Computer Science.

[18] B. Combemale, J. A. Kienzle, G. Mussbacher, H. Ali, D. Amyot, M. Bagherzadeh, E. Batot, N. Bencomo, B. Benni, J.-M. Bruel, J. Cabot, B. H. C. Cheng, P. Collet, G. Engels, R. Heinrich, J.-M. Jezequel, A. Koziolek, S. Mosser, R. Reussner, H. Sahraoui, R. Saini, J. Sallou, S. Stinckwich, E. Syriani, and M. Wimmer, "A Hitchhiker's Guide to Model-Driven Engineering for Data-Centric Systems," *IEEE Software*, 2020.