IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 14, 2020, accepted August 26, 2020, date of publication August 31, 2020, date of current version September 11, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3020745

Large-Scale Simulations Manager Tool for OMNeT++: Expediting Simulations and Post-Processing Analysis

PABLO ANDRÉS BARBECHO BAUTISTA [1], (Student Member, IEEE), LUIS FELIPE URQUIZA-AGUIAR [2], (Member, IEEE), LETICIA LEMUS CÁRDENAS [1], AND MÓNICA AGUILAR IGARTUA [1]

[1]Department of Network Engineering, Universitat Politècnica de Catalunya (UPC), 08034 Barcelona, Spain

[2]Departamento de Electrónica, Telecomunicaciones y Redes de Información, Facultad de Ingeniería Eléctrica y Electrónica, Escuela Politécnica Nacional, Quito 170525, Ecuador

Corresponding author: Pablo Andrés Barbecho Bautista (pablo.barbecho@upc.edu)

This work was supported by the Spanish Government through the Research Project sMArt Grid Using Open Source Intelligence (MAGOS) under Grant TEC2017-84197-C4-3-R. The work of Pablo Andrés Barbecho Bautista was supported by a grant from the Secretaría Nacional de Educación Superior, Ciencia y Tecnología (SENESCYT). The work of Leticia Lemus CÁrdenas was supported by a Ph.D. grant from the Academic Coordination of the University of Guadalajara, Mexico.

ABSTRACT Usually, simulations are the first approach to evaluate wireless and mobile networks due to the difficulties involved in deploying real test scenarios. Working with simulations, testing, and validating the target network model often requires a large number of simulation runs. Consequently, there are a significant amount of outcomes to be analyzed to finally plot results. One of the most extensively used simulators for wireless and mobile networks is OMNeT++. This simulation environment provides useful tools to automate the execution of simulation campaigns, yet single-scenario simulations are also supported where the assignation of resources (i.e., CPUs) has to be declared manually. However, conducting a large number of simulations is still cumbersome and can be improved to make it easier, faster, and more comfortable to analyze. In this work, we propose a large-scale simulations framework called *simulations manager for OMNeT++* (SMO). SMO allows OMNeT++ users to quickly and easily execute large-scale network simulations, hiding the tedious process of conducting big simulation campaigns. Our framework automates simulations executions, resources assignment, and post-simulation data analysis through the use of Python's wide established statistical analysis tools. Besides, our tool is flexible and easy to adapt to many different network scenarios. Our framework is accompanied by a command-line environment allowing a fast and easy manipulation that allows users to significantly reduce the total processing time to carry out large sets of simulations about 25% of the original time. Our code and its documentation are publicly available at GitHub and on our website.

INDEX TERMS Large-scale simulations, OMNeT++, results post-processing.

I. INTRODUCTION

In order to assess new developments of communication networks (e.g., infrastructure-based, ad hoc, wireless, mobile), different methodologies such as analytical, experimental, or simulation are usually used. While analytical methods are unable to characterize wireless communications fully, the use of experimental techniques (i.e., deploying a real test scenario) is often unfeasible due to involved cost and complexity. Therefore, computer networks' researchers commonly use simulation methods. Simulators allow researchers to evaluate the design and performance of a target system under different configurations. In this context, the first step researchers usually have to do is writing code to characterize their target model, for later executing simulations, and finally analyzing experiment outcomes. In the process of implementing the target model (e.g., routing protocols), testing and validating, the experiment should be executed several times

The associate editor coordinating the review of this manuscript and approving it for publication was Maurice J. Khabbaz.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

under different configurations. The complexity of simulation configurations grows, led by the number of simulation factors to thoroughly study the model. Besides, to increase the statistical soundness of the results (e.g., to bound confidence intervals), it is common to execute several repetitions of the experiment with different seeds (to be statistically independent). This leads to complex, large-scale simulations, expensive in terms of processing time due to the campaign configuration, computation, and outcome analysis.

Nowadays, high-end workstations increase their performance principally by adding more and more cores rather than increasing their working frequency [1]. Thus, minimizing execution times are commonly obtained by parallelizing the workload. As a simple and straightforward approach, several experiment runs can be instantiated manually, using different consoles. Here, the operating system should assign each process to a different resource (i.e., CPU). Nevertheless, managing this process manually is not suitable for large-scale simulations since it would require continuous supervision to execute a new instance as soon as a process ends.

In this sense, the first alternative users commonly consider is performing simulation and data analysis directly using the tools included within the simulation platform (e.g., NS-2 [2], NS-3 [3], OMNeT++ [4], OPNET [5], NCTUns [6]). Nevertheless, this approach binds the user always to perform simulation execution and outcome analysis together, which limits the re-usability of simulation outcomes (i.e., output files). Simulators usually include tools to analyze outputs that are not flexible enough to easily carry out results, especially when a large amount of simulations is required. For instance, this happens when we need to include confidence intervals (CI) in our graphs.

Other simulation platforms (e.g., NetSim [7]), instead, do not provide facilities to perform multiple simulation executions within the simulator interface. In that case, the target experiment should be executed manually by running different instances of the program, which is not scalable, as commented above.

A different approach is the use of customized solutions organized in two phases: (i) Firstly, using external tools to execute a bundle of simulations (e.g., bash or python scripts); (ii) Secondly, using another set of tools to process and analyze outputs (e.g., R, Matlab, or Python). The main disadvantage of this strategy is that simulation script execution, and further result analysis are performed in two or more separate instances. Once simulation outputs are obtained, those have to be manually imported into a separate component to be later parsed and analyzed. Also, proper execution of the simulation campaigns should include optimal use of resources to minimize execution time, but also to minimize the impact of possible inaccuracies introduced by human operations during the simulation workflow.

To deal with the problems described above, in this article, we propose a *simulation manager for OMNeT++* (SMO) tool for the execution of multi-scenario large-scale simulations and data analysis for OMNeT++. We want to highlight that our proposed methodology could easily be adapted to other similar simulators. We have chosen OMNeT++ since it is one of the most extended simulators used by the research community, especially in wireless networks. The aim is to provide a single unified framework, simple and easy to use, focusing on minimizing the time to assess network systems. SMO is intended to be used by either novice and advance OMNeT++ users. Our code and its documentation are publicly available at GitHub [8] and in the online documentation repository [9].

Our proposed framework uses Python libraries that allow users to execute and manage complex simulation campaigns. Also, Python allows users to deal with results' analysis in a more straightforward manner leveraging facilities of Python data structures. In this context, Python is a general-purpose programming language that has a significant momentum for data science. Besides, both efficient execution of simulation campaigns and data parsing processes can be easily parallelized, which leads to substantial time savings and optimal use of resources.

Due to the difficulties of carrying out real-life experiments, researchers commonly use simulations to validate new developments. In the design of new proposals to improve systems, researchers must face a lot of tests to validate the target model through the assessment of multiple factors. To guarantee a high level of confidence in the results, it is necessary to fulfill some requirements: (i) use a well-known and reputed simulator extensively validated and trusted by the research community; (ii) design and configure a realistic scenario as much close to reality as possible; and (iii) carry out a large amount of representative simulations used to include confidence intervals to show the confidence level of the results.

The outline for the remainder of this work is as follows: in Section II related work is discussed; then available tools for large-scale simulations are analyzed in Section III. Section IV presents the workflow for large-scale simulations. In Section V, we detail the architecture followed by the simulation framework. Here, a tool for building and execute large-scale simulations is presented. After that, Section VI describes proposed tools for parsing output files. Next, Section VII and Section VIII explains the usage of the command-line interface and a case study. The performance of the proposed tool is evaluated in Section IX. Finally, in Section X, conclusions, and some future work are drawn.

II. STATE OF THE ART

Currently, several simulation frameworks and tools support the automation of multiple simulations. We can differentiate solutions according to their design in two categories:

(i) Simulation frameworks with embedded tools so that the whole simulation workflow is covered within the same platform.

(ii) External applications build on top of simulation frameworks to automate simulations or to extend their facilities.

(i) An example of the former case is OMNeT++ [4]. OMNeT++ is a mature simulation framework provided with

several tools (graphical and command line). OMNeT++ graphical user interface (GUI) allows users to automate and monitor the execution of simulations graphically. Besides, to automate the execution of multiple simulations, the *opp_runall* tool can be executed throughout the command line. To analyze results, OMNET++ offers a post-simulation analysis tool that allows users to visualize results and analyze metrics within the integrated development environment (IDE) directly. Another simulator that includes embedded tools is NS-3 [3]. NS-3 can be implemented as a simulator as well as an emulator so that users can create a live network. It includes some functionalities for multiple simulations execution and data collection. Besides, users can perform the data analysis directly from within their simulation script, using NS-3's Statistical Framework [10].

(ii) The latter case corresponds to the use of external frameworks on top of simulator platforms to extend the simulator facilities or improve their limitations. In this context, the standard OMNeT++ implementation for writing output files uses a single-thread. In [11], support for multi-thread writing files is proposed. By applying parallelism to the writing of data, blocking operations can be minimized. Therefore, the run-time of simulations can be reduced. Another tool built on top of the OMNeT++ simulator is STARS [12]. It uses parallelization in order to perform large-scale simulations. It is an implementation of the multiple replications in parallel (MRIP) principle. STARS implements a centralized architecture with numerous workers so that it distributes the simulations between the simulation network. Once output files are populated, the assessing variable is controlled via the integration of the framework with Matlab. In case the current experiment fails to produce sufficient ergodic data, a new experiment is automatically run. Another example of implementing the MRIP principle is the Akaroa framework [13]. It allows launching multiple instances of the target experiment in parallel (simulation executions are instantiated on different processors) until results satisfy the desired confidence level and precision. While the Akaroa framework can be interfaced with several simulation platforms (e.g., OMNET++, NS-2), currently it seems to be abandoned. Another example of an external tool developed for NS-3 is SAFE [14]. It automates the simulation workflow, going through the initialization of model parameters, parallelized execution of experiments, post-processing of outcomes, and results plotting. An essential feature of SAFE is the dual interface to address the needs of power users and novices alike. In [15], a Python API and a console tool for the management of NS-3 simulations, called simulation execution manager (SEM), are presented. This framework allows users to create simulations, run simulations, and export results though the command-line tool (CLT) tool without the need for writing code. For the post-analysis part, an SEM script is created, consisting of a Python code that includes facilities of the SEM library. This Python library is publicly available at [16].

As commented above, OMNeT++ provides a set of effective tools to automate simulations and post-processing analysis. However, in the context of large-scale simulations, such a set of tools present some limitations as it is discussed along with this work. At the time we are writing this document, the last version of OMNeT++ 6.0 is not already released, and just a preview version is available on the official website [17]. The last release of OMNet++ includes an improved analysis tool based on Python 3, whose libraries are supported in the IDE. The new analysis tool allows users to create plots of the simulation results based on Matplotlib, a library for creating static, animated, and interactive visualizations in Python. Also, the last version of the OMNeT++ analysis tool supports Pandas DataFrames structures, consisting of a 2-dimensional data structure with labeled axes (rows and columns). Those new features can notably facilitate the analysis task, although no new specific tools for large-scale simulations management are included. Also, the data parsing tool (*scavetool*) still uses a single-thread approach.

TABLE 1. Comparison of NS-2, NS-3 and OMNeT++ tools to ease the management of large-scale simulations. Tool support level scored as low, medium or high.

	Development tools	Simulation tools	Data collection and analysis tools
NS-2	Low	Low	Low
NS-3	High	Low	Medium
OMNeT++	High	High	Medium

Despite the fact that simulators like the ones commented above include some tools to assist the researcher's work, they are not specifically intended for large-scale simulations. This makes researchers dedicate much time to prepare the set of successive simulations, launch them, gather results, process them, and finally represent them graphically.

To tackle this issue, this article aims to facilitate the creation, execution, and post-analysis of complex large-scale simulation campaigns. In the context of large-scale simulations, and after identifying limitations with the tools that are currently used to run multiple OMNeT++ simulations, we have proposed our so-called *simulation manager for OMNeT++* (SMO) tool based on a flexible software architecture.

III. ANALYSIS OF AVAILABLE TOOLS TO HANDLE LARGE-SCALE SIMULATIONS

In Table 1, we compare the main tools provided by three widely used network simulators: NS-2, NS-3, and OMNeT++. We focus on the available tools to facilitate (i) development tasks, (ii) simulation tasks, and (iii) data collection and analysis tasks, with respect to large-scale simulations.

(i) Regarding tools for software development, NS-2 uses a dual-language architecture (C++/OTcl), which allows users to write and run simulations without the need for additional compilation time. However, this duality adds extra complexity to the tasks of model development, simulation settings,

and data collection (done via *trace captures* or using the *flow monitor* tool). Hence, due to the NS-2 structure, development tools to assist large scale simulations have a Low score in Table 1. Instead, NS-3 simplifies the NS-2 architecture and uses C++ as the single development language to develop models and carry out simulations. Also, NS-3 supports the use of *Python bindings* (aka *wrappers*) to easily use Python code in the model development. That is why we score NS-3 with a high support level of development tools Table 1. Besides, NS-3 shows better performance (less memory and CPU is required) when compared with NS-2 [18], [19]. On the other side, OMNeT++, which is mainly used to build network simulations, is not actually defined as a network simulator but rather as a general-purpose discrete event-based simulation framework. OMNeT++ has a hierarchical architecture where a single module (written in C++) can consist of several modules called *compound modules*. To combine modules, OMNeT++ uses the network description language NED, which is transparently rendered into C++ code when the project is compiled. Due to the OMNeT++ hierarchical architecture, which scales well in large projects, we set a high score in the level of development tools, see Table 1.

(ii) With regard to simulation tools, both NS-2 and NS-3 have no default IDE. Besides, to obtain a graphical interface (simulation animation tool), external applications can be used (e.g., PyViz, NetAnim). However, advanced configurations (e.g., automation of simulations) are not supported. In contrast, OMNeT++ IDE (based on the Eclipse platform) implements additional simulation tools such as a graphical configuration of network models and automation of simulations (batch executions). Also, in OMNeT++ simulation results can be analyzed within the IDE. For this reason, we score OMNeT++ with high support of simulation tools, see Table 1.

(iii) Respecting data collection and analysis tools, NS-2 provides two mechanisms to capture results named *traces* and *monitors*. The *traces* mechanism captures events related to packets transmission (e.g., dropped packets), while *monitors* can provide basic statistics regarding queues or behavior of packet flows. Nevertheless, data collection tools are neither flexible nor efficient enough for collecting large-scale simulation outputs (NS-2 is scored as Low in Table 1). In the NS-3 simulator, the tracing system is the main mechanism that allows users to collect data and export result files. Also, the output of known events can be recorded in a text file or when a particular packet transmitted/received event can be exported to a packet capture (PCAP) file. Besides, the *flow monitor* module [20] can be used to assess the performance of network protocols where collected statistics are exported to XML format. Nonetheless, as it happens in NS-2, only a limited number of pre-defined metrics (i.e., a limited number of *probes*) can be recorded [21]. Here, a *probe* is an object connected to a simulation variable to record values during execution. In terms of large-scale simulations, by enabling flow monitor modules, significant additional overhead in terms of memory usage and run-time is required. The reason is that XML files are slow to read when large data-sets are handled. Finally, by using the data collection framework (DCF), simple plots can be generated (based on Gnuplot). Despite several data capture and analysis tools are provided with the NS-3 simulator, those are better suited to small-scale simulations (with small size files). That is why we score NS-3 as Medium in Table 1.

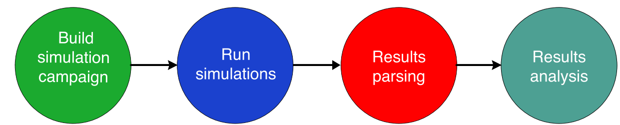


FIGURE 1. Typical workflow for large-scale simulations.

IV. LARGE-SCALE SIMULATIONS WORKFLOW

In this section, we describe the common processes involved in large-scale simulations. Multiple simulations are often required to evaluate different parameter configurations and/or to repeat the experiment with different seeds to reach statistical soundness. A typical workflow in large-scale simulations is presented in Fig 1.

A. STEPS TO PREPARE A LARGE-SCALE SIMULATION CAMPAIGN

The details and considerations taken to prepare the simulation scale are summarized in the following steps:

1) BUILD THE SIMULATION CAMPAIGN

It is built through the combination of the parameters' space, the set of scenarios to simulate, and the number of repetitions per experiment. The parameter space involves static and iterative variables used to tune the model behavior. While static variables remains constant during the whole simulation campaign (e.g., $nodes'\ number = 50$ means all simulations in the campaign will be done with 50 nodes), iterative variables vary according to a set of pre-defined values (e.g., $nodes'\ number = [50, 100, 150]$, meaning the campaign will run simulations with 50, 100 and 150 nodes, respectively). Within the OMNeT++ framework, the parameter space is indicated in the configuration file, detailed in Section V-A.

2) RUN SIMULATIONS

One of the main problems of large-scale simulations is the cost in terms of time, due to the high number of experiment executions required. Moreover, many network simulators usually perform event-based stochastic simulations, which use variables that can change stochastically (i.e., randomly) with individual probabilities. Outputs of the model are recorded, and then the process is repeated with a new set of random values. Here, large-scale simulations can result in a very time-consuming task. Even small-scale simulations could take several hours or even days in case of single-thread was used [22].

At this time, computers are equipped with multi-core processors, which can be exploited to execute parallel experiment simulations. In this context, to efficiently run simulations, our proposal includes a parallel batch processing of simulations, detailed in Section V-C. This way, simulations are distributed in batches and then scheduled throughout the available computational resources (i.e., CPUs).

3) RESULTS PARSING

Once the simulation campaign ends, output files are available for parsing. At this point, the number or outcomes may be considerable. To identify output files, those should be tagged with their correspondent configuration parameters, as it is detailed in Section V-E. It is essential to consider not only the number of files, but also their size. Besides, to minimize the processing time to obtain the results, we can parallelize the data processing, as it is detailed in Section VI-A. Furthermore, in case of files with a considerable size, different techniques (e.g., files partition, data chunking) can be applied to handle files successfully.

4) RESULTS ANALYSIS

At this point, basic and advanced statistics (e.g., mean values, component analysis) should be obtained from the collected results in the form of a standard format file. The objective of this process is to produce data structures that are ready for representation. Here, ordered data structures facilitate the creation of desired plots (e.g., scatterplots, box plots). At this point, by using Python pandas DataFrame structures and visualization libraries (Matplotlib [23], Seaborn [24]), results can be plot easily by using just a few lines of code, as it is detailed in Section VI-B.

B. COST RELATED TO THE MANAGEMENT OF LARGE-SCALE SIMULATION CAMPAIGNS

Even though there are already several tools to facilitate the execution of network simulations (e.g., OMNET++ tools), there is an intrinsic cost when evaluating a large number of simulations. In terms of large-scale simulation campaigns, the main factors that can affect the scalability of the system are the following:

Time: Since the whole simulation campaign may involve a high number of experiment executions, the overall process required to build the campaign and afterwards to run the experiment may result to be a really time-consuming task. Besides, the time needed to process output files may aggregate a significant overhead (e.g., parsing time).

Disk space: The size of the output files generated from the simulation campaign, may grow considerably and might become very large (sizes can grow to several GB). Here, enough storage space should be available to save such amount of data. Therefore, it is important that output files avoid unnecessary information which represents an extra overhead.

Memory (RAM usage): Given the high amount of data resulting from simulation campaigns, it may result expensive in terms of the RAM memory required to store and analyze those big amounts of data. In this context, an improved capturing process (i.e., recording of results) should be used during the simulation runs, avoiding unnecessary data and consequently minimizing memory overhead.

Energy (CPU usage): The execution of large-scale simulation campaigns usually requires a noteworthy amount of energy. For the sake of energy reduction, which has an impact in the climate change, it is important to optimize the use of resources by distributing load and also by parallelizing processing tasks.

To minimize the intrinsic additional cost produced by deploying large-scale simulation campaigns, a rigorous methodology that considers all the factors above mentioned is required. At the end of this manuscript we include a performance evaluation where we assess those factors in our proposal, see Section IX.

In terms of large-scale simulations, current tools included in the OMNeT++ framework have important scalability limitations described in the following sections. To tackle those limitations, we propose a large-scale simulation tool called *simulations manager for OMNeT++* (SMO). SMO is intended to complement OMNeT++ tools to facilitate the management of large-scale simulations, speed up the whole simulation process and reduce intrinsic cost related to large-scale simulations.

SMO is developed to support complex, large-scale simulations and post-analysis of results. Typical simulation-workflow depicted in Fig. 1 guides the design and development of our framework. The proposed SMO architecture is described in the next section.

V. SOFTWARE ARCHITECTURE FOR LARGE-SCALE SIMULATIONS

In this section, we detail the software architecture of our proposal SMO for automating large-scale simulations. We define two blocks, see Fig. 2: (a) The first block is the configuration file, which corresponds to the network model configuration declared using the OMNeT++ syntax; (b) The second block wraps up the SMO functionalities. In SMO, the main processes of the simulation workflow described in Section IV are covered in three modules: *Launcher, Summarizer, and Analyzer*, as it is depicted in Fig. 2. The main features of SMO tools are summarized in Table 2.

(i) First, the *Launcher* module combines the parametric space and user settings to create the simulation campaign. Once a simulation campaign is created, this module will ask the user to run the simulations. Alternatively, the user can still move back to tune any simulation parameter before running the simulations. Fig. 3 shows an example of simulation settings in a simulation campaign.

(ii) Once simulations are completed, *output files* are parsed and exported (supported output files are.npy,.mat,.csv) using the *Summarizer* module.

(iii) Finally, the *Analyzer* module allows users to customize results plotting them in a straightforward manner, leveraging facilities of Python data structures.

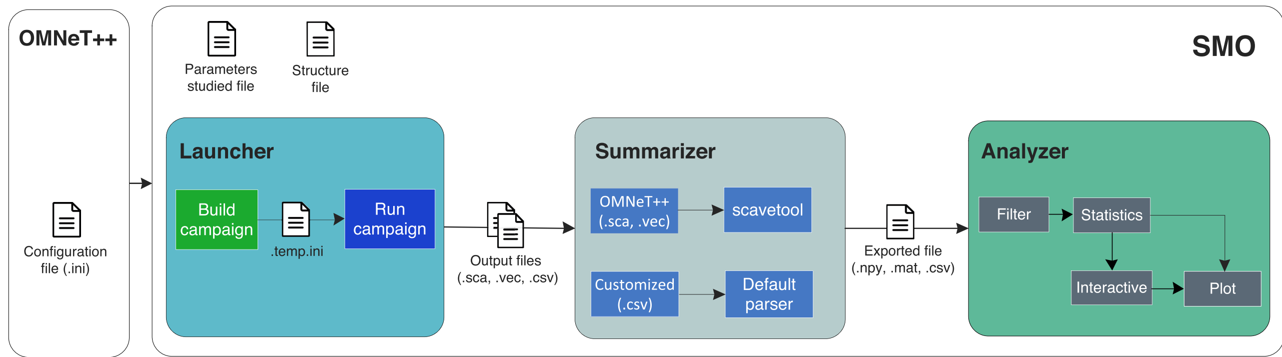


FIGURE 2. Software architecture of our proposal for large-scale simulations called *simulations manager for OMNeT++* (SMO).

TABLE 2. Main features of the tools provided in our proposed SMO framework.

Tools	Intended for	Features
Launcher	- Build and execute complex simulation campaigns. - Large-scale multi-scenario simulations.	- Automate multi-scenario simulations. - Automate resources configuration (CPUs). - Improve simulation campaign building and execution times.
Summarizer	- Export simulation results. - Large-scale multi-scenario simulations.	- Automatically export OMNeT++ and/or custom simulation output files. - Scalable in case of custom output files (multi-thread processing and chunked files). - Improve exportation times (parallelized process). - Support different output formats suited for external programs.
Analyzer	- Customizable filtering, sorting, and plotting.	- Customizable analysis in a few lines of code by using python facilities. - Interactive plotting. - Support advanced statistics.

```
Campaign Info
--------------

Scenarios to simulate: ['Barcelona', 'Berlin', 'Tokyo']
Iteration variables: 1 = [8]
Repetitions per scenario: 30
Simulation time: 500s
Total Runs: 720
--------------
```

FIGURE 3. Example of a campaign information summary.

The SMO facilities can be executed separately without the necessity of going through all the simulation-workflow processes (e.g., for troubleshooting or analysis modifications). This scheme also allows users the interoperability with other programs with which users might feel more comfortable (e.g., R, MATLAB). Although modules interact using well-defined interfaces, each module's internal structure can also be customized if needed.

In this context, our SMO framework attempts to be a solution that balances the two common approaches for simulation frameworks, which were pointed out in Section II. (i) On the one hand, there is the monolithic approach of performing simulation executions and the post-analysis within the same simulation framework (e.g., as it happens in OMNeT++). These kinds of simulators usually include tools not flexible enough for large-scale simulations. (ii) On the other hand, some approaches use customized solutions in several phases. In this kind of solution, outputs have to be manually imported into separate components to be parsed and analyzed, which is usually annoying for users and also error-prone. SMO address their drawbacks properly to attain a faster and easier process for large-scale simulation campaigns.

In Table 3, we summarize the main tools available both in OMNet++ and SMO. Tools are classified regarding they are intended for small-scale (SC) or large-scale (LS) simulations. In that table, we point out the limitations present in OMNet++ tools and developed alternative solutions in SMO to ease them. We tick in green when the framework provides a tool supporting the specific activity.

Finally, we want to remark that the proposed SMO framework has been designed to be as decoupled as possible from OMNeT++ and the network model. This way, we minimize breaking issues in case of updates to the latest versions of OMNeT++. Besides, this way SMO can easily been adapted to any OMNeT++ based network simulator (e.g., INET [25], VEINs [26], LTE [27]).

A. THE SIMULATION CAMPAIGN

The first step in preparing the simulation campaign is the generation of scenarios according to the parameters' space. With the default implementation shipped with OMNeT++, this is done through the definition of a configuration file, usually called *omnetpp.ini* [28]. It contains settings that control how the simulation will be executed and values for the parameters of the model (e.g., simulation time, number of nodes). The configuration file is a flexible ASCII text file, line-oriented,

TABLE 3. Tools available in OMNeT++ and SMO to facilitate the execution of simulations. Support for small-scale (SC) and large-scale (LS) simulations. Both CLT and IDE tools are analyzed.

Support		OMNeT++ IDE	OMNeT++ CLT	SMO CLT
Graphic simulation (Tkenv, Qtenv)		✓	✗	✗
Model troubleshooting		✓	✗	✗
Multi-scenario simulations		✓ Group launcher (Cmdenv)	✗	✓
Automate execution of simulations		SC	LS (Limited)	LS
Simulations on multiple CPUs		✓	✓ opp_runall	✓
Automate resource allocation (i.e., CPUs)		✗	✗	✓
Export simulation results		SC	LS (Limited)	LS (custom parsing)
Parse simulation results	GUI	SC	✗	✗
	scavetool	✗	LS (Limited)	LS (Limited)
	custom parsing	SC	SC	LS (custom parsing)
Statistics from simulation results		SC	✗	LS (Advanced)
Create plots from simulation results		SC	✗	LS

and structured in sections. Commonly, a general section of the configuration file is defined where general settings are declared (e.g., the simulation time). Additional sections can be included by the definition of unique IDs of the form [Config < *configname* >]. Those additional sections are used to set up and evaluate the target model under different configurations (e.g., propagation model, transmission range, number of nodes, road maps used for vehicular networks). We assume the user has the required expertise to write valid OMNeT++ simulation scripts. Parameters included in the OMNeT++ configuration file should be declared inside the model script. Their values will determine the model behavior (e.g., nodes' number = 10 will launch a simulation with ten nodes).

Using Fig. 2 as a reference, in the SMO block, the launcher module builds the simulation campaign based on the OMNeT++ configuration file. For this, a temporal file (named *temp.ini*) is created, which includes general configurations of the model and other simulation parameters in additional sections. This temporary file offers an abstraction interface to OMNeT++ that facilitates the adaptation of the user's settings (e.g., number of repetitions, simulation time), and the parameters of study (e.g., iteration variables).

Within SMO, the following files are defined:

1) *Parameters studied file*: It is an ASCII text file like the OMNeT++ configuration file written with OMNeT++ syntax. It allocates iterative variables used in the model.
2) *Structure file*: This is an optional file with a simple file format (.csv). It is required in case the user decides to customize data capturing. Each entry of the file (comma-separated) corresponds to the column name of data recorded in the output files.

By defining parameters and structure files as external files (i.e., outside OMNeT++ environment), assessed parameters will be made available to other modules for further operations, e.g., for custom parsing.

B. MULTI-SCENARIO SIMULATIONS

OMNeT++ already offers several tools, IDE and command-line tools (CLT) (see Table 3), to automate the execution of multiple simulations. Even though simulations can be executed in parallel, only one configuration (i.e., one single-scenario) can be executed at a time. Supporting complex simulations, where several system configurations should be evaluated, an alternative included in the OMNeT++ IDE is the use of the launch group tool, see Table 3. However, simulation campaigns and resources still have to be manually configured. Besides, by using the *cmdenv* tool (i.e., without graphical interface) available in the IDE, see Table 3, the execution of a whole big simulation campaign is not recommended [28]. The reason is that it is more susceptible to fall into C++ programming errors in the model, since if any of the runs crashes the whole batch will stop.

An alternative is the use of the OMNeT++ *opp_runall* CLT, see Table 3. While it is intended for executing batch simulations in parallel, it could also be used to run just one simulation with its particular configuration file at a time. However, with this tool, resources must be manually configured. In the case of multiple simulations (i.e., multiple configuration scenarios to be run simultaneously), those have to be performed by running two or more different instances. Although, possible inaccuracies (e.g., overloading the system with simulations) can be introduced by human operations.

To overcome the limitations described above, we propose an alternative tool to automate the execution of multi-scenario simulations. In this context, the SMO *launcher* module, see Fig. 2, can execute simultaneously the set or a sub-set of scenarios defined in the configuration file, see Fig. 4.

The following elements have to be defined to calculate the total number of executions included in the simulation campaign:

1) *Scenarios to simulate*: Set or subset of simulation scenarios to execute. Several scenarios can be configured within OMNeT++ through the definition of additional sections in the configuration file [28]. The selection of a subset of scenarios allows tuning or troubleshooting a particular configuration.
2) *Number of iterations*: Corresponds to the combination of the set of values of each iterative variable.
3) *Repetitions*: This is an essential parameter of study and is defined as the number of independent replicas per each experiment's configuration. Each reproduction will be executed with a different seed for random number generation to ensure statistical confidence in the results. The default random generator (RNG) of OMNeT++ is used.

Once all the above elements are defined, the *launcher* module automatically renders the experiment settings into a simulation campaign comprising N runs. The total number of runs is computed as follows:

$$N = \prod_{i} p_k \cdot s \cdot r \tag{1}$$

TABLE 4. Example of parameters' mapping to generate a simulation campaign.

ID	Param_A	Param_B	S	R
0	10	true	conf_A	0
1	10	true	conf_A	1
2	20	true	conf_A	0
3	20	true	conf_A	1
4	10	false	conf_A	0
5	10	false	conf_A	1
6	20	false	conf_A	0
7	20	false	conf_A	1
8	10	true	conf_B	0
9	10	true	conf_B	1
10	20	true	conf_B	0
11	20	true	conf_B	1
12	10	false	conf_B	0
13	10	false	conf_B	1
14	20	false	conf_B	0
15	20	false	conf_B	1

where, i stands for the number of values to iterate for each p_k simulation parameter, s is the number of scenarios, and r is the number of repetitions to be done for each experiment (each one with a different seed to have statistically independent simulations).

Table 4 shows an example of parameters' mapping to generate a simulation campaign. First, iterative variables expand to two parameters (Param_A, Param_B), each with two possible values (Param_A = [10, 20], Param_B = [true, false]) giving a total of 4 iterations. Next, the experiment considers two scenarios S defined in two different named sections, see Section V-A, so S = [Config A, Config B]. Finally, each experiment should be repeated twice with different seeds, so the number of repetitions is $r = [0, 1]$. Therefore, the total number of executions in the simulation campaign will be $N = 16$. Notice that in addition to the different evaluation variables configured in each simulation run, each scenario will also have its own particular configuration (e.g., map, coverage range, message interval) defined in their corresponding section of the configuration file. This will modify the behavior of the different evaluated scenarios.

C. PARALLEL BATCH PROCESSING

At this point, the simulation campaign is already configured containing a list of N simulations, see (1), that should be automatically scheduled across the available local resources (CPUs). To exploit the parallelism capabilities of computers equipped with multi-core processors, we faced the problem as a parallel batch processing problem, which is broadly studied in the semiconductor wafer fabrication. As discussed in the literature, this problem can be decomposed into two stages: (i) the batch formation, and (ii) the scheduling of batches in the processors [29], [30].

In Algorithm 1, the logic of the proposed strategy to manage a whole simulation campaign is presented. First, the batch construction is done in lines 1-4. Here, the batch of simulations is constructed considering each parameter combination per scenario, as it was detailed in Section V-B. Each scenario s_m $(1 \leq m \leq NS)$ includes a set of j jobs $(1 \leq j \leq j_m)$ to be run, where j_m is the number of jobs to be run in scenario m and NS is the total number of scenarios to simulate. The total number of runs N that compose the simulation campaign, are organized in batches $b_{m,w}$, $(1 \leq w \leq j_m)$, regarding each scenario s_m and the total number of jobs j_m in each scenario m, see line 3.

Then, the batch scheduling is covered with the next lines 5-21 in Algorithm 1. The batch size bs_m per each scenario m is calculated, taking into account the total number of simulations N, the workload per scenario (i.e., number of jobs j_m), and the number of available processing cores P, see lines 5-14. In case of the total number of simulations, N is less than or equal to the number of available processors P, the batch size for all the scenarios is set to $bs_m = 1$ (i.e., each job will be scheduled in a different processor), see line 6. Conversely, the number of available processors are distributed among all the scenarios taking into account the number of jobs j_m to be run at each scenario m. This way, jobs are distributed, balancing the workload among the available processors. Here, scenarios with a higher number of jobs (runs) j_m, get a higher number of processors in such a way that the total processing capacity is not violated, see lines 8-14. The number of jobs per scenario m that will be assigned to a processor is obtained inline 13. Finally, jobs are scheduled in groups of bs_m runs per each scenario to a random free processor, in lines 16-21.

As an example, Fig. 4 shows a simulation campaign comprising NS scenarios and N jobs to be run in total. For scenario 1, the number of jobs to be run is $j_1 = 12$, where simulation runs are distributed in $P_1 = 3$ processors each with a batch size of $bs_1 = 4$. For scenario 2, the number of jobs to be run is $j_3 = 9$, where simulation runs are distributed in $P_2 = 3$ processors each with a batch size of $bs_2 = 3$. Finally, for scenario NS, the number of jobs to be run is $j_{NS} = 6$, runs are scheduled in $P_3 = 2$ processors, each with a batch size of $bs_3 = 3$. Notice that the last batch of jobs b_{m,j_m} in a particular scenario m can have a lower amount of jobs to be run compared to the previous batches of that scenario (see line 13 in Algorithm 1).

D. RESULTS RECORDING FILES

One of the main problems with large-scale simulations is the high number of output files and their possible large size. The OMNeT++ framework provides two built-in mechanisms to write simulation results to the hard disk. (i) The first mechanism records statistics directly from modules, which implies that modules have to be modified (to gather the intended data) with a high level of complexity. (ii) The second mechanism is called a signal-based statistics recording (SBSR) [28]. It requires configuring statistics in NED files (Network Description files) to capture data to record, and the moment to transmit the signal. In OMNeT++, NED files

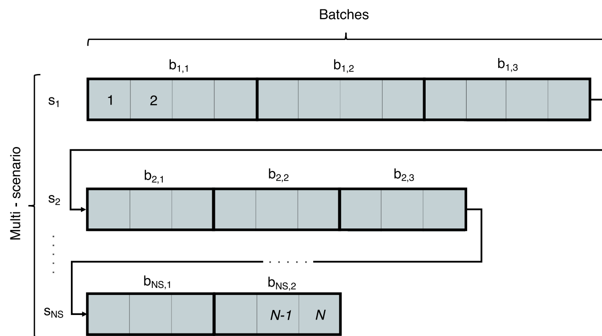


FIGURE 4. Multi-scenario batch construction.

allow us to declare the structure of a simulation model using NED language. This last option is more flexible and with a low-level of complexity.

Output files often grow considerably when evaluating a high amount of factors. However, OMNeT++ built-in data analysis tools are intended to process small-scale simulation outcomes. In case of having large size output files, processing might not be successfully completed. That is why in Table 3, we point out that OMNeT++ IDE tools are mainly intended for small-scale simulations. Besides, OMNeT++ CLT *scavetool* is intended to be used with large-scale simulations. However, it takes much more time to process output files (data is stored in memory), and file exportation might not be successfully completed. That is why in Table 3 we point out OMNeT++ CLT *scavetool* is limited for large-scale simulations. Here, integrated data analysis tools are mainly intended for casual inspection of simulations in the case of the IDE tools and are limited when handling a high number of output files with representative sizes in the case of CLT [28]. To provide a better support to large vector files, a third file (named *Index file*) has been introduced in OMNeT++ 4.0 and improved in OMNeT++ 5.6. Basically, this additional file is an index file that enables faster access to vector files.

To cope with the aforementioned limitations, an alternative approach in the literature is the definition of customized solutions where third-party tools are commonly used for reading and processing results (e.g., MATLAB, Python [31]). A more efficient solution can be obtained with the definition of a new format for the output files (e.g., binary format, text files, etc.) [32].

In this context, to maintain compatibility with OMNeT++ tools, we propose a *summarizer module*, see Fig. 2, which is capable of processing *output files*, see Fig. 2, from either (i) OMNeT++ captures or (ii) users' customized recordings. In the first case, using the OMNeT++ SBSR capturing mechanism, every output file contains data from one run only. In the second case, the user should define the data to be captured within the model. Each simulation run in the campaign generates a different output file containing recorded values writing to a simple comma-separated.csv file. Here, the structure of *output files* is defined in an external file

Algorithm 1 Parallel Batch Processing

- P = total number of available processing cores.
- N = total number of runs.
- NS = number of scenarios to simulate.
- $s_m = m_{th}$ scenario to simulate, $1 \leq m \leq NS$.
- j_m = number of jobs to be run in scenario m (e.g., $j_1 = 12, j_2 = 9, j_{NS} = 6$ in Fig. 4).
- P_m = number of processing cores devoted to run each scenario m composed of j_m jobs.
- bs_m = batch size for scenario m (e.g., $bs_1 = 4$, $bs_2 = 3$, $bs_{NS} = 3$ in Fig. 4).

Input: s_m, P, N, NS, j_m
Output: bs_m
Batch construction:

```
1  for each s_m scenario do
2      GET the job number j in scenario s_m, 1 ≤ j ≤ j_m
3      Form the batch array b_{m,w}, 1 ≤ w ≤ j_m
4  end
   Batch scheduling:
5  if N ≤ P then
       /* There are enough processors for all the N
          jobs at a time */
6      bs_m = 1, ∀ m Each job will be
       scheduled in a different processor,
       for every scenario m.
7  else
8      for each s_m scenario do
9          P_m = floor((j_m/N) × P)        // Number of
           processors devoted to run scenario m.
10         if P_m < 1 then
11             P_m = 1          // At least 1 processor
               devoted to run scenario m.
12         end
           /* Batch size for scenario m */
13         bs_m = ceil(j_m/P_m);        // Number of jobs in
           scenario m assigned per processor.
14     end
15 end
16 for each scenario s_m do
17     while there are jobs not yet scheduled in b_{m,w} do
18         Find the first available processor k, 1 ≤ k ≤ P
19         Schedule bs_m runs to processor k
20     end
21 end
```

called *structure file*, as detailed in Section V-A. The *structure file* will be available to the other modules for further data analysis.

E. OUTPUT FILES LABELING

In order to distinguish each result file, output files are tagged with a unique ID. Each ID results from the combination of the simulation ID, the correspondent value among the set of iterative parameters, and the correspondent experiment

repetition number. This mechanism allows users to identify result files for each simulation round easily.

We can have two possibilities: (i) In the case of using OMNeT++ captures, result files are labeled automatically through OMNeT++ predefined variables [28]. (ii) In the case of custom users recording, the SMO launcher module will label *output files*, see Fig. 2, using OMNeT++ predefined variables in the configuration file. Here, the user should declare in the configuration file the format of the output file name. There are several predefined variables with obvious meanings that can be used for labeling output files. For instance, the third row in Table 4, with a set of parameters: *Param_A* = *20*, *Param_B* = *true*, with repetition number *R* = *0*, for scenario *S* = *conf_A*, and with *run ID* = 2, will have an output filename labeled as *"2_conf_A_20_true_0.csv"*.

VI. POST-SIMULATION ANALYSIS

Once output files have been successfully generated from the execution of the simulation campaign (composed of N jobs to be run), N output files will be available for data analysis. Due to the high number of outcomes in large-scale simulations, some of those files can grow notably and become very large, so that parsing outcomes can be troublesome. In this context, the OMNeT++ framework includes a useful post-simulation analysis tool with the IDE. It has a graphical interface that facilitates to extract a small set of data from simulation outcomes. While the graphical tool is intended for small volumes of data, the command-line tool *scavetool* performs better when the size of data considerably grows. This tool combines the whole set of result files into a single output file. Nonetheless, *scavetool* is also limited in the sense that often cannot successfully extract results in case of bulk files [31], see Table 3. Therefore, in the literature, the adoption of custom or third-party tools for analyzing data is generally preferred [32], [33].

In the following sections, we describe two proposed tools (*summarizer module* and *analyzer module*) included with our SMO framework to overcome the limitations mentioned above. The main advantage of our proposed tools is their flexibility (i.e., they are easily customizable) through the use of Python's wide-established statistical analysis tools, which makes it possible to perform advanced analysis in just a few lines of code.

A. SMO SUMMARIZER MODULE

As it was mentioned in Section V-D, users might choose to capture simulation results using either (i) OMNeT++ mechanisms to manage output files, or (ii) customized data collection for an optimal analysis (i.e., in case just main metrics are captured). At this point, the SMO *summarizer module* is in charge of translating the set of result files into one output file with the format expected by the *analyzer* module, see Fig. 2.

(i) In case of OMNeT++ files with extensions.sca and/or.vec are detected within the results folder. In this case, the *summarizer module* is implemented as a wrapper for *scavetool*, see Fig. 2.

(ii) In case the user customizes the results collection, the *summarizer module* tool provides a default parser, see Fig. 2, which will automatically read the set of result files into a numerical matrix. In this case, an additional file with the structure of result files is required. This additional file called *structure file* was detailed in Section V-A, including an example in Fig. 7. The *parser* tool reads both the *output files* and the *structure file* directly from the results directory and from the additional files directory, respectively. The goal is to consolidate all the output data of the whole simulation campaign in a single *exported file*. In the same manner as it is done by *scavetool*, result files are exported into a common output file with the format expected by the *analyzer module*, see Fig. 2. The main functionalities of the *summarizer module* tool are the following:

1) Map data files into an ordered data frame composed of the simulation campaign parameters and the additional *structure file*.
2) Convert data into numerical results automatically.
3) Export result files into one single *exported file*. The output format is deduced from the name of the *output file*. Supported formats of the *output files* are.npy,.mat, or .csv.

While the OMNeT++ default parsing tool intended for large-scale simulations (see *scavetool* in Table 3) uses a single-thread for processing simulation output files, here we propose a parsing tool where the whole set of result files are processed in parallel, exploiting local multi-core capabilities. By default, the maximum number of available processors is used, although the user could set a lower number of processors through the command line tool detailed in Section VII. When working with large-size output files, it may be desirable to process large files as a sequence of chunked segments of the file. Within the proposed SMO parsing tool, *output files* are automatically partitioned into $n = 20$ indexed segments, by default. Notice that n can be set through the command line tool depending on the size of the *output files*, as it is explained in Section VII. In this manner, our tool provides a scalable solution intended for the analysis of large-scale simulations *output files* (pointed out as *custom parsing* in Table 3).

SMO provides a default parser, including facilities above mentioned, which will read the contents of *output files* and the *structure file* in order to generate a single *exported file* where data will be converted automatically into numerical values for further analysis (e.g., plotting). In case a more sophisticated parsing is required (e.g., simulation output files need to be processed before they can result in a clean and purely numerical data structure), users are alternatively asked to define a customized function that defines the expected data type of result from output files.

Fig. 5 shows a comparison between the parsing execution time of OMNeT++ files using the *scavetool*, and using the SMO *summarizer* tool. The experiment corresponds to a case

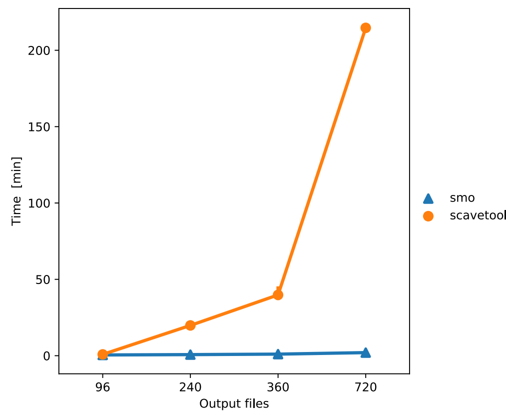


FIGURE 5. Comparison of parsing execution time using the OMNeT++ *scavetool* (single-thread) tool, and the SMO *summarizer* (multi-thread) tool. By default a single output file (format.csv) is exported. Different simulation campaign sizes, each with a different number of output files, have been evaluated.

TABLE 5. Workstation parameters to attain the results in Fig. 5 and Fig. 11.

Machine	CPU	Memory	Cores
Ubuntu	Intel Core i9-9920X @ 3.5GHz	62GB	20

study described in section VIII, in which some vehicles suffer an accident and send warning messages to both access points and to vehicles around to warn them.

The exported file corresponds to a.csv file. Specifications of the workstation used for the experiment are depicted in Table 5. We can clearly see that our SMO summarizer tool scales much better when the size of the simulation campaign grows. With 720 *output files*, the SMO summarizer tool takes less than 3 minutes to completely generate the *exported file*, while the OMNeT++ *scavetool* takes almost 215 minutes.

B. SMO ANALYZER MODULE

As final phase of a performance evaluation, we have designed a tool to easily generate graphs to ulterior analyse the simulation campaign results. OMNeT++ IDE has embedded a useful graphical analysis tool which is very helpful in the testing face or for troubleshooting. As pointed in [31], this tool is mainly intended for casual inspection of simulations. However, OMNeT++ analysis tools lack of functions essentially interesting for researchers e.g., to obtain confidence intervals for mean values, or to generate box plots. Besides, in the case of large-scale simulations, the tool execution time grows considerably due to the large size of the output files.

As an alternative approach and seeking to provide an easy to use solution, in our framework SMO we propose a novel *analyzer module*, see Fig. 2. This tool receives parsed files as inputs to finally produce results that are ready for presentation. The *analyzer module* tool leverages a key data structure of the Pandas library called DataFrames [34]. Once result files have been parsed into an easily accessible data structure (i.e., DataFrames), advanced statistical process techniques can be applied. In this context, users are asked to define custom plots through the use of common Python data structures (e.g., DataFrames) and plot libraries (e.g., matplotlib, seaborn). As creating desired plots is not always easy, the *analyzer module* tool supports a graphical environment based on the open-source Javascript pivot table tool [35], which uses Jupyter [36] web-based tool to create plots easily. When the *analyzer module* tool receives the interactive plots option passed through the CLT, described in Section VII, a web browser prompts to create plots by drag and drop columns.

The *analyzer module* tool focus on mainly three operations:

1) Customized data filtering.
2) Create ordered statistics (e.g., count, mean, CI) based on a common data structure (DataFrames).
3) Create and export customized figures (e.g., scatterplots, boxplots, CDFs).

First, to generate customized filtering, the *analyzer module* has access to the *parameters* file (i.e., the file containing iteration variables). This allows users to operate over a subset of data. Then, with clean and ordered data, statistics can be generated using a common Python *numpy* library. Numpy is equipped with robust statistical functions such as mean, median, standard deviation, between others. Finally, users can either create plots writing a Python script based on a common data structure (i.e., Pandas DataFrame) or interactively through pivot tables in a web browser.

VII. THE COMMAND LINE TOOL

To easily leverage the SMO's capabilities, it is provided with a command-line tool that can be used to create, run simulations, and export results without being necessary to interact with any Python code. The command-line interface is built with a hierarchical structure where each command can be invoked with the −−help flag (e.g., *smo − −help*) to get in deep information about the command and sub-command usage at different levels (e.g., *smo launcher –help*).

The SMO command-line tool can be installed through pip (a tool to install and manage Python packages), including *launcher*, *summarizer*, and *analyzer* SMO modules, where all the requirements (i.e., Python libraries) are automatically installed. For users who only wish to leverage parallel simulations or post-analysis, each module can be executed independently. The defined console line has been simplified as much as possible to be easy to comprehend, maintaining core functionalities of different modules. Further information about the SMO command line usage is available in [9].

VIII. A CASE STUDY. VEINS-BASED EXPERIMENT

As an example of a case study to use our SMO proposed tools, we have carried out a performance evaluation of a straightforward scenario, which is depicted in Fig. 6. This experiment is carried out with the Veins network simulator

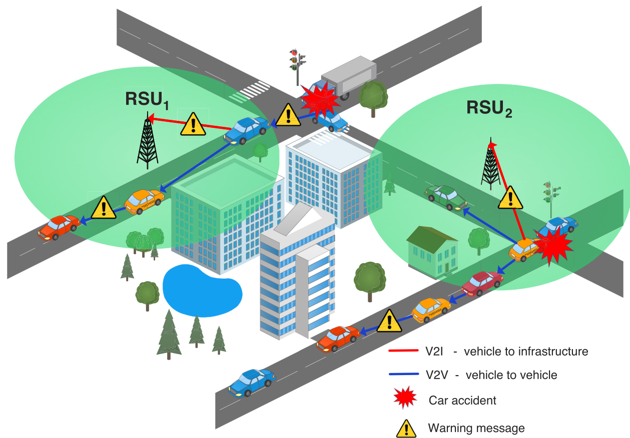


FIGURE 6. **Case study to show the benefits of our proposal SMO to save simulation and processing time in large-scale simulation campaigns. Example scenario of a simple vehicular network implemented in the Veins simulator [26]. There are 250 vehicles and 2 road side units (RSUs).**

for vehicular networks [26], which is based on OMNeT++ and SUMO [37] (road traffic simulator).

The scenario is composed of two access points (APs) randomly located along the map. In our performance evaluation, three different scenarios are assessed: Berlin, Barcelona, and Tokyo, where specifically we have selected a large-sparse district, a medium-size-sparse district, and a small-dense district, respectively. The simulation areas are 6, 4 and 1 km^2, respectively. Real maps are used, obtained from OpenStreetMaps [38].

There are 250 mobile nodes (vehicles) moving around the considered map. In the scenario, an accident takes place at a given time (150s). The crashed vehicle broadcasts warning messages to their neighborhood (i.e., nodes within the transmission range) and also towards the APs. In the same way, vehicles that receive a warning message will forward it to their neighbors. In case of an access point (AP) receives the warning message, it will forward the message to the emergency services (e.g., 112 or 911). The warning message includes the road ID where the accident situation occurs. In case a vehicle heading towards the accident receives a warning message, it will modify its route to avoid the congestion situation (i.e., the traffic jam). This simulation scenario corresponds to the default configuration of the vehicular ad hoc network (VANET) example provided in the Veins network simulator [26]. We have used this simple scenario as a case study to show the benefits of our proposal SMO when dealing with large-scale simulation campaigns.

In this simple experiment, we will analyze how the VANET is affected when the vehicles' transmission range varies from 100 to 450 meters. Fig. 7 shows an example of the configuration files to design the simulation campaign:

(i) The *Structure file* has a simple file format (.csv), where each entry (comma-separated) corresponds to the column name of data recorded in the *output files*. For instance, *Type* and *NodeID* entries in the structure file, see Fig. 7, depict the type of the element (node/RSU) and their identifiers, respectively.

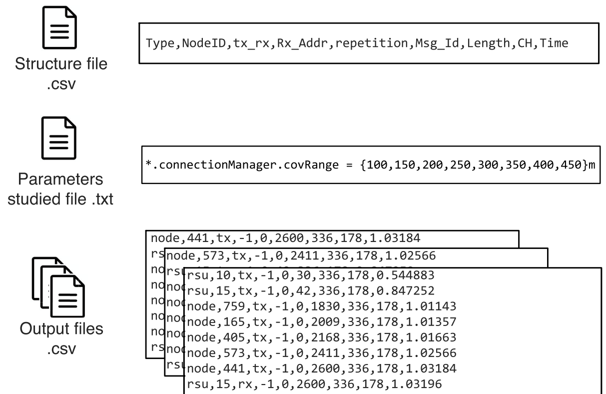


FIGURE 7. **Example of configuration files used to design a large size simulation campaign.**

(ii) The *Parameters studied* file includes the iterative variables of the model to be evaluated (*nodes' transmission range*, in our example), which is defined in the same manner as in the configuration file (i.e., with OMNeT++ syntax).

(iii) The *Output files* are used to store tabular data in.csv format. Within the Veins network simulator, the recording of data can be triggered by a *send message* or *receive message* status at the application layer of the model. In Fig. 7, each line of the *Output files* corresponds to a data record (e.g., node type, node ID, transmitted/received message, node destination address, etc) according to the *structure file*.

First, the SMO *launcher module* described in Section V-A creates the simulation campaign by means of the combination of configuration parameters: (i) *transmission range values* considered (from 100 to 450 m), (ii) evaluated scenarios (*Barcelona, Berlin, Tokio*), and (iii) *number of repetitions per experiment* (30 repetitions per simulation point). This makes a total of 720 simulations according to (1). Then, simulation results are exported with the SMO *summarizer module* tool, detailed in Section VI-A, which generates a single exported file (.csv format). Finally, Fig. 8 shows an example of how complex plots can be created with the SMO *analyzer module* tool, detailed in Section VI-B, by using just a few lines of Python code, see Fig. 9.

Fig. 8 shows the number of nodes that received at least one warning message about the accident. As we expected, as the vehicles' transmission range increases, more vehicles receive at least one warning message. This behavior is more noticeable for the Barcelona and Berlin scenarios, where only 40% and 25% of vehicles, respectively, receive flooded warning messages when the nodes' transmission range is 100m, and near to 100% when the transmission range is above 300m. The reason is that the Barcelona and Berlin scenarios correspond to sparse VANETs (62 and 42 vehicles/km^2, respectively) so vehicles are far from each other making it difficult to spread the emergency message. In contrast,

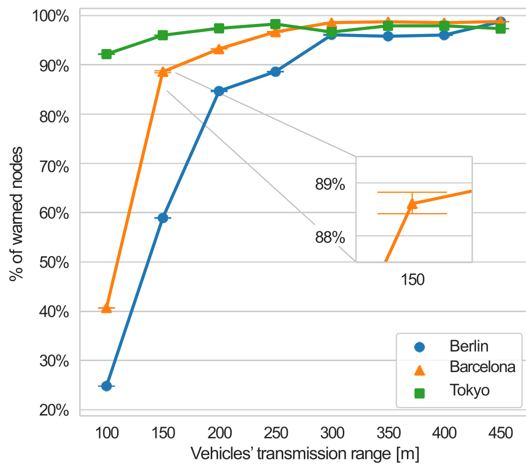


FIGURE 8. Percentage of nodes that received at least one warning message about the accident, as a function of the nodes' transmission range. The simulation scenario corresponds to the VANET case study represented in Fig. 6. Plot generated with the SMO *analyzer* tool taken from 720 output files composed of 3 scenarios (Barcelona, Berlin, Tokyo), a set of 8 values for the node's transmission range parameter, and 30 repetitions per simulation point. 95% confidence intervals are shown.

```
# Create plot
fig = sns.pointplot(x='CoverageRange', y='#Nodes',
                    hue='scenario', errwidth=1, capsize=0.1,
                    markers=["o","^","s"], data=results)


# Modify axes settings
fig.despine(right=False, top=False)
fig.set_axis_labels("Nodes' coverage range [m]", '# of warned
nodes')


# Save figure to default output_directory
plt.savefig(os.path.join(output_directory, filename))
```

FIGURE 9. Python code used to generate the graphs shown in Fig. 8.

the Tokyo scenario corresponds to a small-dense district (250 vehicles/km^2), so that vehicles are closer to each other. In the Tokio scenario even with a low vehicles' transmission range (100m), 92% of the vehicles received at least one warning message. Fig. 8 presents confidence intervals (CI) of 95%, obtained from 30 repetitions per simulation point generated with independent seeds.

It is important to highlight that the goal of this article is not to present a complete performance evaluation of the warning message scheme included in this case study, but to highlight the benefits of our proposal SMO in handling large-scale simulation campaigns.

To have an idea of the saving time for the researcher, we spent less than one hour to complete the total performance evaluation using our framework SMO, from the design of the simulation campaign file (see Fig. 3) till obtaining the final plots with the results (see Fig. 8). Using the traditional OMNeT++ tools, we spent almost five hours, mainly due to the post-processing time, see Fig. 10. The reason is that OMNeT++ generates a considerable amount of result files, some of which can be very large (almost 3GB). Here, filtering results can be cumbersome and also error-prone. Therefore,

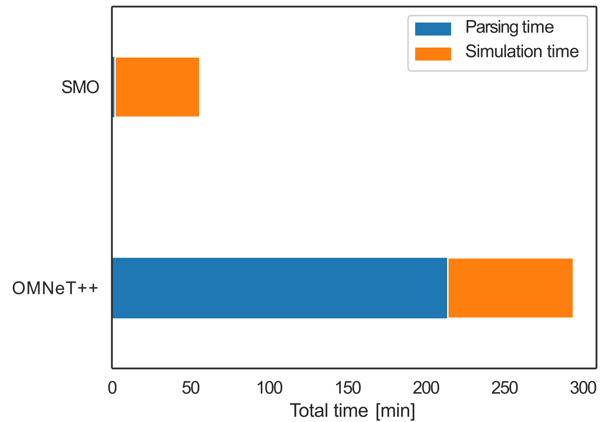


FIGURE 10. Total time devoted to carry out the large simulation campaign (simulation time + parsing time).

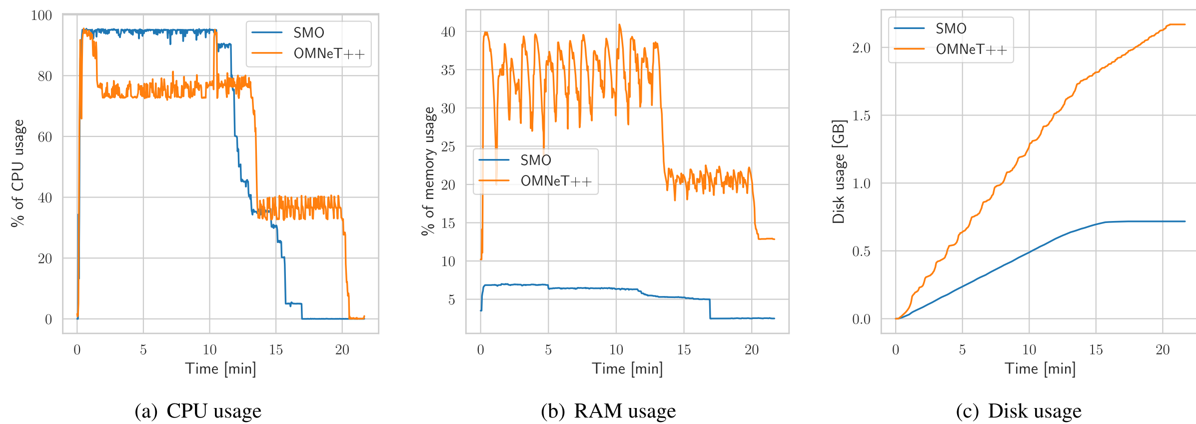
using our framework is beneficial for the sake of faster performance evaluation of proposals, especially notably for large-scale simulation campaigns.

We would like to point out that in this article, we have just used an example already available in the Veins network simulator to explain the procedure and the benefits of our proposal. Utterly, any other network model defined in other OMNeT++ based simulators can also be evaluated with our proposed SMO framework (e.g., INET [39], Artery [40]).

IX. PERFORMANCE EVALUATION. COST OF A LARGE-SCALE SIMULATION CAMPAIGN

Fig. 11 shows a performance evaluation of the simulation scenario depicted in Fig. 6 using SMO or OMNeT++. This evaluation is done in terms of CPU usage, RAM usage, and disk usage when running a simulation campaign composed of 240 runs. In this case, we have used the three same maps (Barcelona, Berlin, Tokio), eight different values for the transmission range (from 100 to 450 m) and ten repetitions per simulation point, which makes 240 runs according to (1). The features of the working station used for this evaluation is detailed in Table 5.

With OMNeT++ and SMO simulation tools evaluated separately, the simulation time to carry out the whole simulation campaign took 22 and 17 minutes, respectively. OMNeT++ takes longer to finish simulations (5 minutes more) since OMNeT++ does not use all the available CPU so effectively as SMO, as we can see in Fig. 11(a). Fig. 11(a) shows the average CPU used by all the processes involved in the simulation campaign. While the SMO CLT uses all the maximum available resources (i.e., 94% of the CPU during the first 11 minutes), OMNeT++ limits resources used by the simulation processes as it requires additional resources to maintain the OMNeT++ IDE process. We can see that at the minute 14, OMNeT++ drastically reduces the CPU usage (to 40%) because several simulation processes end at that moment. Similar behavior occurs with SMO at minute 12 when most of the simulation batches already ended, although



(a) CPU usage (b) RAM usage (c) Disk usage

FIGURE 11. Comparison of OMNeT++ and SMO simulation tools in terms of (a) CPU usage, (b) RAM usage and (c) disk usage. The large simulation campaign includes 240 runs. The main features of the workstation used for this analysis are described in Table 5.

according to a smoothest curve since the load (i.e., number of simulations running) is automatically distributed through all the available processors, as it was detailed in Section V-C.

Additionally, we have also assessed the energy consumed by the large-scale simulation campaign considered using SMO or OMNeT++. Here, we can estimate the power consumed by the simulation campaign using the % of CPU usage depicted in Fig. 11(a). For this, we use the power consumption of i9-9920X processors listed at the product specifications [41]. For the i9-9920X processors family, at full load (100% of CPUs usage) processors consume on average $P_{FL} = 165W$. Hence, we estimate the power consumption as $P(W) = (\%\ of\ CPU\ usage) \cdot P_{FL}$. Accordingly, OMNeT++ consumes on average 96.8 Watts whereas SMO consumes 94.7 Watts. Notice that those values correspond to the power consumed only by the simulation processes. In the case of OMNeT++, the IDE process requires additional power. Taking into account the simulation time (i.e., 22 min for OMNeT++ and 17 min for SMO), those power values correspond to a consumed energy of 35.49Wh for OMNeT++ and 26.83Wh for SMO, which means a saving of 24.40% with SMO.

Fig. 11(b) shows the percentage of RAM used by SMO and OMNeT++. On the one hand, we can see that OMNeT++ requires a large amount of memory, almost 40% of available memory till minute 14, and 20% till the end of the simulation campaign. This is mainly due to the OMNeT++ statistics recording mechanism, which was detailed in Section V-D. By default, the OMNeT++ statistical module processes a high amount of metrics (which implies to record values and compute them) during simulations. Once the simulation campaign ends at minute 22, OMNeT++ output files (named *scalar files*,.sca) still are being filled (statistics are still being computed and.sca files are being filled), which requires 13% of the available memory. On the other hand, SMO uses just around 7% of the available memory. This is due to the SMO implementation of a more efficient solution obtained with the definition of simple output files (i.e., ASCII text file, line-oriented) leaving to the *summarizer module* all the additional tasks for statistical computations, see Section VI-A.

Finally, due to the large amount of data stored by the OMNeT++ statistical module, a big space on the disk is required, see Fig. 11(c). We can see that the OMNeT++ output files use almost three times more disk space than the SMO output files. We can see that the improved SMO mechanism for recording the output data of interest (custom recording) allows us to minimize the overhead of large-scale simulation campaigns in terms of disk space, see Section V-D.

X. CONCLUSION AND FUTURE WORK

In this work, we have presented the SMO framework for the management of large-scale OMNeT++ simulations. First, we have developed the software architecture to automatize large-scale simulations, taking into account the limitations of the tools provided by the OMNeT++ framework. Then, we have developed a modular framework with a new approach to (i) efficiently run parallelized multi-scenario simulations, followed by the (ii) summarizing of simulation results, and finally to (iii) obtain graphs to analyze the simulation results.

Our main goal is to provide novice and advanced OMNeT++ users a useful tool to quickly and efficiently execute large-scale network simulations hiding the tedious process of conducting multiple simulations and thereby offloading this significant burden work to the researcher. Additionally, we have shown the good scalability level of our SMO tool for large-scale simulations when compared with traditional built-in OMNeT++ tools (*scavetool*).

The simulation framework SMO is publicly available at [8]. SMO is easy to install and it includes a command line tool [9] allowing the fast and easy of its manipulation. With a modular and extensible architecture, we hope this new tool will serve as a potential framework for managing complex and large OMNeT++ based simulation campaigns.

In a future work we will consider a dynamic jobs scheduler based on the simulation time or on other metrics that might be still increasing the time of simulations in the campaign (e.g., frequency of messages, number of nodes, size of the map). Considering those events could even decrease more the global simulation time.

APPENDIX A ACRONYMS

CI	Confidence intervals
CDF	cumulative distribution function
CLT	Command line tool
CPU	Central processing unit
DCF	Data collection framework
GUI	Graphical user interface
ID	Identifier
IDE	Integrated development environment
LS	Large-scale simulations
OMNeT++	Objective Modular Network Testbed in C++
PCAP	Packet capture
RAM	Random access memory
RSU	Road side unit
SBSR	Signal-based statistics recording
SC	Small-scale simulations
SEM	Simulation execution manager
SMO	Simulation manager for OMNeT++
VANET	Vehicular ad hoc network

REFERENCES

[1] A. Geist and D. A. Reed, "A survey of high-performance computing scaling challenges," *Int. J. High Perform. Comput. Appl.*, vol. 31, no. 1, pp. 104–113, 2017.

[2] *The Network Simulator—ns-2*. Accessed: Mar. 10, 2020. [Online]. Available: https://www.isi.edu/nsnam/ns/

[3] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM Demonstration*, vol. 14, no. 14, p. 527, 2008.

[4] *OMNeT++ Discrete Event Simulator*. Accessed: Feb. 14, 2020. [Online]. Available: https://omnetpp.org/

[5] *Riverbed Modeler*. [Online]. Available: https://www.riverbed.com/mx/products/steelcentral/steelcentral-riverbed-modeler.html

[6] *EstiNet Network Simulator and Emulator (NCTUns)*. [Online]. Available: http://nsl.cs.nctu.edu.tw/NSL/nctuns.html

[7] *NetSim-Network Simulator & Emulator*. Accessed: Mar. 10, 2020. [Online]. Available: https://www.tetcos.com/

[8] P. Barbecho. (2020). Automating OMNeT++ large-scale simulations. GitHub Repository. [Online]. Available: https://github.com/Pbarbecho/osm

[9] P. Barbecho. (2020). *Simulations Manager for OMNeT++ (SMO)-Documentation*. [Online]. Available: http://osm.rtfd.io/

[10] *Ns-3 Statistical Framework*. Accessed: Feb. 17, 2020. [Online]. Available: https://www.nsnam.org/docs/manual/html/statistics.html

[11] S. Neumeier, C. Obermaier, and C. Facchi, "Speeding up OMNeT++ simulations by parallel output-vector implementations," in *Proc. 5th GI/ITG KuVS Fachgespräch Inter-Vehicle Commun.*, 2017, p. 22.

[12] E. Millman, D. Arora, and S. W. Neville, "STARS: A framework for statistically rigorous simulation-based network research," in *Proc. IEEE Workshops Int. Conf. Adv. Inf. Netw. Appl.*, Mar. 2011, pp. 733–739.

[13] D. McNickle, K. Pawlikowski, and G. Ewing, "AKAROA2: A controller of discrete-event simulation which exploits the distributed computing resources of networks," in *Proc. 24th Eur. Conf. Modeling Simulation (ECMS)*, 2010, pp. 104–109.

[14] L. F. Perrone, C. S. Main, and B. C. Ward, "SAFE: Simulation automation framework for experiments," in *Proc. Winter Simul. Conf. (WSC)*, Dec. 2012, pp. 1–12.

[15] D. Magrin, D. Zhou, and M. Zorzi, "A simulation execution manager for ns-3: Encouraging reproducibility and simplifying statistical analysis of ns-3 simulations," in *Proc. 22nd Int. ACM Conf. Modeling, Anal. Simulation Wireless Mobile Syst.* New York, NY, USA: Association Computing Machinery, 2019, pp. 121–125.

[16] D. Magrin. (2019). A simulation execution manager for ns-3. GitHub repository. [Online]. Available: https://github.com/signetlabdei/sem

[17] *OMNeT++ Preview Versions*. Accessed: Apr. 11, 2020. [Online]. Available: https://omnetpp.org/download/preview

[18] R. Patel, N. Patel, and S. Patel, "An approach to analyze behavior of network events in NS2 and NS3 using AWK and xgraph," in *Information and Communication Technology for Competitive Strategies*, S. Fong, S. Akashe, and P. N. Mahalle, Eds. Singapore: Springer, 2019, pp. 137–147.

[19] A. Zarrad and I. Alsmadi, "Evaluating network test scenarios for network simulators systems," *Int. J. Distrib. Sensor Netw.*, vol. 13, no. 10, pp. 1–17, 2017.

[20] G. Carneiro, P. Fortuna, and M. Ricardo, "FlowMonitor–a network monitoring framework for the network simulator 3 (ns-3)," in *Proc. 4th Int. ICST Conf. Perform. Eval. Methodologies Tools*. Brussels, Belgium: ICST, 2009, pp. 1–10.

[21] *Ns-3 Manual Scope/Limitations*. Accessed: Jul. 9, 2020. [Online]. Available: https://www.nsnam.org/docs/release/3.31/manual/html/scope-and-limitations.html

[22] I. Mavromatis, A. Tassi, R. J. Piechocki, and A. Nix, "Poster: Parallel implementation of the OMNeT++ INET framework for V2X communications," in *Proc. IEEE Veh. Netw. Conf. (VNC)*, Dec. 2018, pp. 1–2.

[23] *Matplotlib: Python plotting—Matplotlib 3.2.1 Documentation*. Accessed: Apr. 13, 2020. [Online]. Available: https://matplotlib.org/

[24] *Seaborn: Statistical Data Visualization—Seaborn 0.10.0 Documentation*. Accessed: Apr. 13, 2020. [Online]. Available: https://seaborn.pydata.org/

[25] *INET Framework—What Is INET Framework?* Accessed: Mar. 9, 2020. [Online]. Available: https://inet.omnetpp.org/Introduction.html

[26] *Veins*. Accessed: Feb. 14, 2020. [Online]. Available: https://veins.car2x.org/

[27] A. Virdis, G. Stea, and G. Nardini, "SimuLTE—A modular system-level simulator for LTE/LTE—A networks based on OMNeT++," in *Proc. 4th Int. Conf. Simulation Modeling Methodologies, Technol. Appl.*, 2014, pp. 59–70.

[28] A. Varga. OMNeT++ simulation manual. OpenSim Ltd. Accessed: Feb. 13, 2020. [Online]. Available: https://doc.omnetpp.org/omnetpp/manual

[29] I. C. Perez, J. W. Fowler, and W. M. Carlyle, "Minimizing total weighted tardiness on a single batch process machine with incompatible job families," *Comput. Oper. Res.*, vol. 32, no. 2, pp. 327–341, Feb. 2005.

[30] H. Balasubramanian, L. Mönch, J. Fowler, and M. Pfund, "Genetic algorithm based scheduling of parallel batch machines with incompatible job families to minimize total weighted tardiness," *Int. J. Prod. Res.*, vol. 42, no. 8, pp. 1621–1638, Apr. 2004.

[31] *OMNeT++ Pandas Tutorial*. Accessed: Feb. 13, 2020. [Online]. Available: https://doc.omnetpp.org/pandas-tutorial/

[32] A. Virdis and M. Kirsche, Eds., *Recent Advances in Network Simulation* (EAI/Springer Innovations in Communication and Computing). Cham, Switzerland: Springer, 2019.

[33] C. Sommer. (2010). *INET Scripts*. [Online]. Available: https://github.com/sommer/inet-sommer/tree/analysis/etc

[34] *Pandas. DataFrame—Pandas 1.0.3 Documentation*. Accessed: Apr. 6, 2020. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

[35] N. Krunchten. (2019). *Javascript Pivot Table Library*. [Online]. Available: https://github.com/nicolaskruchten/pivottable

[36] *Project Jupyter | Home*. Accessed: Apr. 6, 2020. [Online]. Available: https://jupyter.org/

[37] *SUMO—Simulation of Urban Mobility*. Accessed: Feb. 17, 2020. [Online]. Available: http://sumo.sourceforge.net/

[38] S. Coast. *OpenStreetMap*. Accessed: Apr. 13, 2020. [Online]. Available: https://www.openstreetmap.org

[39] *INET Framework—INET Framework*. Accessed: Apr. 6, 2020. [Online]. Available: https://inet.omnetpp.org/

[40] R. Riebl. (2020). *Artery: V2X Simulations Based on ETSI Its-G5 Protocols*. [Online]. Available: https://github.com/riebl/artery

[41] *Procesador Intel Core i9-9820X Serie X. Product Specifications*. Accessed: Jul. 6, 2020. [Online]. Available: https://ark.intel.com/content/www/es/es/ark/products/189121/intel-core-i9-9820x-x-series-processor-16-5m-cache-up-to-4-20-ghz.html

PABLO ANDRÉS BARBECHO BAUTISTA (Student Member, IEEE) received the B.Sc. degree in electronic engineering from the University of Azuay, Ecuador, in 2012, the M.Sc. degree in communication networks from the Pontificia Universidad Católica del Ecuador (PUCE), and the second M.Sc. degree in telecommunication engineering from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 2017, where he is currently pursuing the Ph.D. degree in network engineering. His research interest includes design and performance evaluation of routing protocols for vehicular ad hoc networks (VANETs). His research activity focuses on the integration of electric vehicles and autonomous vehicles in VANETs and their interaction with smart grids and smart cities.

LUIS FELIPE URQUIZA-AGUIAR (Member, IEEE) received the B.Sc. degree in electronic and networking engineering from Escuela Politécnica Nacional (EPN), Ecuador, in 2008, the M.Sc. and Ph.D. degrees in telecommunication engineering from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 2012 and 2016, respectively, and the second M.Sc. degree in statistics and operational research from UPC, in 2018. He is currently an Assistant Professor with the Departamento de Electrónica, Telecomunicaciones y Redes de Información (EPN). His research interests include wireless networks, mathematical modeling, and the optimization of large-scale telecommunication problems.

LETICIA LEMUS CÁRDENAS received the Telecommunication Engineering degree from the University of Guadalajara (UdG), Mexico, in 2007, and the M.Sc. degree in telecommunication engineering from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 2011, where she is currently pursuing the Ph.D. degree with the Department of Network Engineering with the support of a scholarship of the Coordinación General Académica (CGA), UdG. Her research interests include vehicular ad hoc networks, machine learning, electric vehicles, and autonomous vehicles in urban environments.

MÓNICA AGUILAR IGARTUA received the M.Sc. and Ph.D. degrees in telecommunication engineering from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 1995 and 2000, respectively. She is currently an Associate Professor with the Department of Network Engineering, UPC. She is the author of more than 20 journal articles and has chaired several conferences. Her research interest includes design and performance evaluation of routing protocols to distribute multimedia services over vehicular ad hoc networks (VANETs). Her research activity focuses on the integration of electric vehicles and autonomous vehicles in VANETs and their interaction with smart grids and smart cities. She is a member of the Editorial Board of *Ad Hoc Networks* Journal.