TraCI4Matlab: Enabling the Integration of the SUMO Road Traffic Simulator and Matlab® Through a Software Re-engineering Process

Andrés F. Acosta, Jorge E. Espinosa and Jairo Espinosa

Abstract SUMO (Simulation of Urban Mobility) has become one of the preferred open-source platforms for researchers to perform microscopic road traffic simulation. Thanks to the Traffic Control Interface (TraCI), SUMO offers a high level of flexibility, allowing a client to retrieve and modify the objects in the simulation. Two implementations of TraCI have been released to date for Python (TraCI-Python) and Java (TraCI4j). On the other hand, Matlab® is a software tool with a programming language with a broad user's community of researchers. Matlab is used in many tasks on simulation, control, optimization and it is a preferred tool for rapid prototyping. Both platforms share strengths that benefit the development of control strategies for road traffic. The desire of combining both strengths motivated the interest to develop a TraCI implementation for Matlab. This chapter describes an adaptive software re-engineering process of TraCI-Python used to implement TraCI4Matlab (TraCI for Matlab).

1 Introduction

SUMO (Simulation of Urban Mobility) is an open-source software project that incorporates a set of tools to create and execute microscopic road traffic simulation scenarios [16]. These tools are grouped in three categories:

J.E. Espinosa
Politécnico Colombiano Jaime Isaza Cadavid, Cra 48 No 7-151, Medellin, Colombia
e-mail: jeespinosa@elpoli.edu.co

A.F. Acosta · J. Espinosa (✉)
Universidad Nacional de Colombia, Cra 80 No. 66-223, Medellin, Colombia
e-mail: jespinov@unal.edu.co

A.F. Acosta
e-mail: afacostag@unal.edu.co

- Mapping tools. For creating the "map" (network), where the simulation will be performed, comprised by intersections, streets, traffic light definitions, polygons that represent buildings and other structures, and a variety of sensors for output delivery. The network can be created from scratch or imported from a wide range of sources. This network is represented as a directed graph, where nodes define the intersections and edges define the streets.
- Demand modeling tools. For creating vehicle demands from several sources or even randomly, allowing to define vehicle types according to their physical characteristics and specify entry times, origins and destinations.
- Simulation tools. The sumo application itself that receives the network, the demand and some optional information as inputs to execute the simulation and output results in XML format, a feature that demonstrates the high integration capacity of the simulator.

SUMO includes the Traffic Control Interface (TraCI), which simplifies the retrieval and modification of the SUMO objects through an application protocol, allowing applications like vehicular communications, dynamic routing and traffic light control algorithms [8]. Furthermore, TraCI subscription and context subscription commands allow to retrieve several attributes of an object, or those of its surrounding objects, on every simulation step.

The SUMO community has developed two remarkable TraCI clients: one made in Python by the SUMO developers, which we will call TraCI-Python; and TraCI4J, made in Java by researchers from Politecnico di Torino (Italy) [6].

Depending on the application of interest, an implementation of TraCI can benefit from the programming language in which it is developed. In the case of applications involving control and optimization, Matlab® has proven to be an excellent alternative, featuring toolboxes for optimization, robust control and model predictive control, among others [9]. This motivated the development of an implementation of the TraCI protocol for the Matlab® programming language, namely TraCI4Matlab.

Particularly, TraCI4Matlab was proposed as a requirement in the MOYCOT project [11], where optimization-based traffic lights coordination strategies are being developed using Matlab®.

However, developing a new implementation of TraCI could be more expensive than doing it based on an existing one, especially taking into account the open-source nature of the later. In this regard, a re-engineering approach has many advantages over direct code translation either by hand or using semi-automated tools [7].

This chapter describes a re-engineering process of TraCI-Python used to implement TraCI4Matlab. This chapter is organized as follows: Sect. 2 describes the software re-engineering process related to software maintenance, reverse engineering, refactoring and forward engineering; Sect. 3 describes the reverse engineering sub-process of TraCI-Python and shows the extracted architectural and design component models obtained; Sect. 4 describes the refactoring tasks needed to adapt the obtained models to the constraints imposed by the Matlab® language and the subsequent forward engineering sub-process resulting in TraCI4Matlab; Sect. 5 shows results and discussion; finally, Sect. 6 shows conclusions and future work.

2 The Re-engineering Process

Chikofsky and Cross [3] define software re-engineering as:

The examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form

They also state that the re-engineering process involves some form of reverse engineering, followed by some form of forward engineering and may include modifications related to new requirements.

It is important to note that software re-engineering is not only applied in legacy software, but also in cases where new requirements arise, such as [2] improving performance, exploiting new technologies and porting the subject software to a new platform.

In general, Chikofsky and Cross [3] related the re-engineering process with the software lifecycle and introduced formal definitions regarding the software transformations at different levels of abstraction. In the case of TraCI4Matlab, only the design level is considered, and the relationships with the software lifecycle take the form showed in Fig. 1, where the subject software corresponds to TraCI-Python and the new implementation, to TraCI4Matlab. Levels of abstraction refer to the representation of the software in the different phases of the cycle. For example, software is usually represented in terms of UML diagrams at the design phase, while in the implementation phase, the representation corresponds to the source code.

The taxonomy proposed by Chikofsky and Cross includes the following definitions, represented as sub-processes in Fig. 1:

- **Reverse-engineering**: In many cases, the subject software needs to be reverse-engineered because “usually, the system’s maintainers were not its designers”. Moreover, open-source software can be developed and extended in a distributed fashion by different developer teams. Therefore, reverse engineering enables

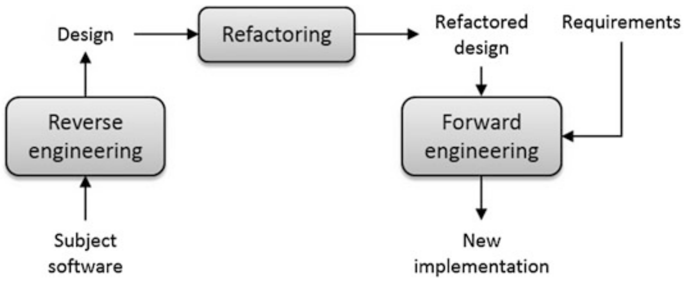


Fig. 1 Re-engineering process used in TraCI4Matlab

software comprehension by extracting artifacts from different levels of abstraction. In other words, reverse engineering allows to extract the knowledge of the software previous to its implementation, in a language that is easier to understand.

- **Restructuring**: It involves the "transformation from one representation to another at the same relative abstraction level, while preserving the subject system's external behavior". Thus, restructuring is related to the modification of the software without altering or extending its functionality, with the goal of improving its quality (structure, performance, etc.). In the case of Object-Oriented Programming (OOP), restructuring is known as refactoring [10].

Another important concept related to the software lifecycle is the use of software patterns, which have been defined by Vincke et al. as "The description of a general solution to a recurring problem" [19]. Hence, patterns allow to reuse the experience and best practices in the solution of common problems found in the development of a software product. Demeyer et al. proposed a set of Object Oriented Re-engineering Patterns [5], which they describe, as follows:

> Re-engineering patterns codify and record knowledge about modifying legacy software … We see re-engineering patterns as stable units of expertise which can be consulted in any re-engineering effort

Some of these re-engineering patterns were applied in the implementation of TraCI4Matlab, and will be explained in the subsequent sections that correspond to the sub-processes mentioned earlier.

Regarding the requirements considered for the implementation of TraCI4Matlab, there were two-fold:

- The migration to the Matlab® language: Naturally, the main requirement was to implement the TraCI API in Matlab®, taking into account its features and limitations.
- Preserving the TraCI-Python's end-user functions' structure: Since TraCI4Matlab was conceived to be open-source, it is important to simplify its use as much as possible. Therefore, the TraCI4Matlab's end-user functions' structure should be very similar to the TraCI-Python's.

It is important to note that there were not requirements related to performance, which favored the rapid release of TraCI4Matlab, this was done at the cost of a lower performance compared to TraCI-Python's, as it will be discussed later. Additionally, the implementation of TraCI4Matlab assumed that TraCI-Python is well structured, which means that the focus was not put on detecting code duplication or code smells [12].

The following sections explain the sub-processes involved in the implementation of TraCI4Matlab.

3 Reverse Engineering of the TraCI-Python Implementation

Re-engineering patterns used

Since the size of the subject software (TraCI-Python) is relatively small (around 4,300 lines of code) and the number of requirements was low, it was not necessary to apply many Object-Oriented Re-engineering Patterns. Particularly, the following patterns were used:

- **Chat with the maintainers**. According to Demeyer et al. [5] the intent of this pattern is to "Learn about the historical and political context of your project through discussions with the people maintaining the system". However, in TraCI4Matlab it was not important to learn about the historical and political context of TraCI-Python because it is part of an active software project (SUMO) and the re-engineering effort is not related to quality improvements, but to the achievement of the requirements described previously, which are related to an extension of SUMO. Instead, in TraCI4Matlab, the pattern chat with the maintainers was applied to learn about the technical aspects of TraCI-Python that are not clear enough in the documentation or are part of exceptional cases. Therefore, in this case, the intent of this pattern could be stated more generally as: "Learn about the context and the technical aspects of the project through discussions with the people maintaining the system". It's important to note that this pattern couldn't be possible without the SUMO mailing list system.
- **Read all the code in one hour and skim the documentation**, whose intent is to "Assess the state of a software system by means of a brief, but intensive code review". These patterns were enough to approach to the recovery of the subject system's design (i.e. to identify the components of TraCI-Python, their responsibilities and how they collaborate) taking into account its small size.
- **Step through the execution**, whose intent is to "Understand how objects in the system collaborate by stepping through examples in a debugger". Two open-source tools were used to apply this pattern: Winpdb [20], which is a graphical Python debugger, and StarUML [15], which is a program to draw UML diagrams. Thus, the TraCI4Traffic Lights tutorial, provided with the SUMO installation, was debugged with some modifications to understand all the components of the subject software.

These tasks helped to conclude that TraCI-Python comprises three main components: The TraCI package, the modules representing the SUMO objects (edge, junction, lane and so on) and the TraCI constants definition. Those components take the form of namespaces, which, in Python, are accessed through the *dot* operator. Thus, TraCI-Python takes advantage of the fact that Python allows to associate variables, functions and classes to namespaces [1]. Figure 2 shows two UML

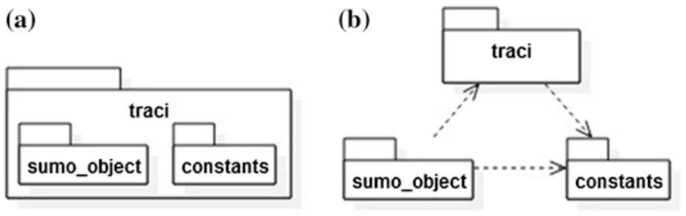


Fig. 2 TraCI-Python's components: **a** Deployment diagram, **b** Dependency diagram

package diagrams that represent the TraCI-Python's namespaces in terms of their deployment and their dependence relationships. Note that the *sumo_object* abstract package was defined to generalize the namespaces representing the SUMO objects, which have some variables and functions in common. Additionally, since in UML namespaces can be represented as packages, these terms will be used interchangeably in the remainder of this chapter.

In the following subsections, the components of the TraCI-Python implementation are described.

3.1 The TraCI Package

This is the top-level package. It contains the namespaces corresponding to the SUMO objects (the modules variable) plus five public functions and others with, at most, package visibility. Through these functions, the responsibilities of the TraCI package could be extracted, being:

- Initialize and close the connection to the SUMO server through the functions `init()` and `close()`.
- Allow several SUMO instances to be controlled by the same client and switching among the corresponding connections, thanks to the port argument in the `init()` function and the `switch()` function.
- Perform a simulation step through the `simulationStep()` function, including a `step` argument, which allow to increase or decrease the simulation step in milliseconds.
- Populate the subscription results related to each SUMO object, using the `readSubscription()` function.
- Construct and send the outgoing messages according to the TraCI protocol, through a set of functions beginning with the word `send`, which have been grouped in an abstract function called *sender* for illustration purposes. The *sender* functions prepare the `message` variable according to the desired data type to send to the SUMO server.
- Read the responses from the SUMO server and check them for errors throwing the corresponding exceptions, using the `recvExact()` function.

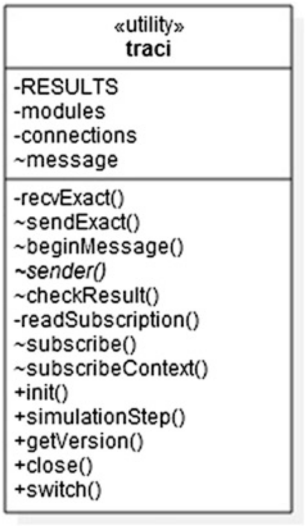


Fig. 3 Variables and functions in the TraCI package

Figure 3 shows an UML class diagram including the functions of the TraCI package. Note that the utility stereotype has been used, since those functions do not belong to a class but to a namespace.

3.2 Packages Corresponding to the SUMO Objects

These packages can be briefly summarized through the so called getters and setters which allow the end user to retrieve and modify the properties of the objects in the SUMO simulation. The get and set processes follow a sequence of functions in the TraCI components that collaborate by appending the proper command, requested attribute, and desired value (in the set case) from the TraCI constants to build the outgoing get/set message according to the TraCI protocol. Here, the `get_wrapper()` and `set_wrapper()` abstract functions are defined to represent the set of public functions designed for the end user in such a way that he/she only needs to provide the ID of the SUMO object of interest and the desired attribute value (in the set case). Finally, the `sumo_object` packages include another four wrapper functions related to the TraCI subscriptions: two for subscribing to the desired object and variable, and other two for retrieving the subscription results. Figure 4 shows an UML sequence diagram, which is an example of the above process in the case of a `getter`. Note how the different components collaborate: the end user calls the `get_wrapper()` which calls the universal getter of the `sumo_object` component, which in turn calls the proper TraCI function to build the outgoing message, read the response from the SUMO server and check it for errors. Figure 5 shows a class diagram corresponding to the abstract class `sumo_object`. It is worth to notice, that this abstract class was not

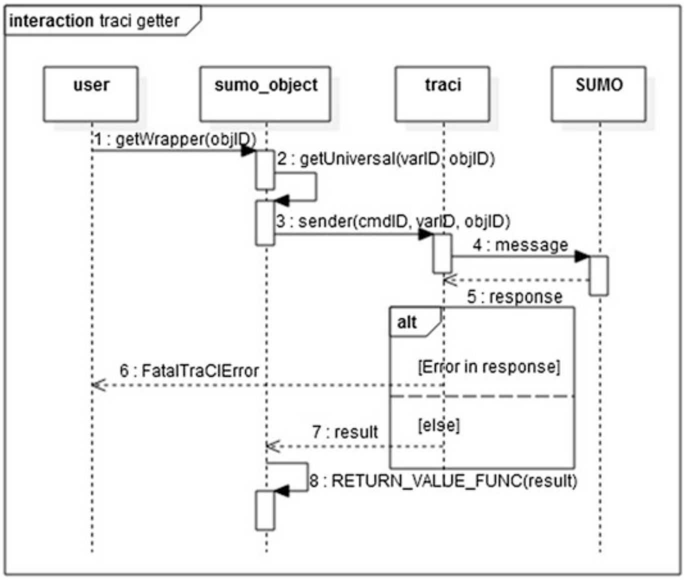


Fig. 4 UML sequence diagram for the get process in TraCI-Python

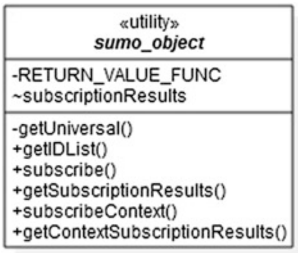


Fig. 5 Abstract package *sumo_object* representing the namespaces corresponding to SUMO objects

physically implemented, but serves as a way to explain the packages corresponding to the SUMO objects and their variables and functions in common.

3.3 TraCI Constants

This is a namespace containing the command, variable-type and data-type codes as constants from the TraCI protocol specification. TraCI-Python's components use these constants as parameters for their functions. For example, referring to Fig. 4, the parameters `varID` and `cmdID` are taken from the TraCI constants module.

4 Forward Engineering Sub-process

4.1 Re-engineering Patterns Used

As stated in the reverse engineering sub-process, because of the relatively small size of the subject system and the low number of requirements, some re-engineering patterns were not necessary. Moreover, since the official Matlab® unit testing framework is quite new [13] and taking into account that the majority of TraCI-Python's functions are not associated to a class, as explained in the previous section, tests were created in a single m-file that reproduces the TraCI4 Traffic Lights tutorial. Therefore, patterns related to *use a testing framework* were not applied. Additionally, recall that the focus of the re-engineering process is on the migration of TraCI-Python, not on the improvement of its quality. Hence, the patterns related to *write tests to understand* were not used. Taking into account these considerations, the following patterns were applied:

- **Write tests to enable evolution**: The test created in the implementation of TraCI4Matlab allowed to identify the limitations of the Matlab® programming language that prevented a direct implementation of the subject software's recovered design and, consequently, the necessary refactoring tasks to perform. Thus, according to Demeyer et al. [5], the risk of "failing to accommodate future change" was mitigated.
- **Grow your tests base incrementally**: Every time a TraCI4Matlab's component was implemented, the corresponding test was modified to incorporate the new functionalities, which enabled to *always have a running version*.
- **Conserve familiarity**: The requirement related to the preservation of the TraCI-Python's end-user functions' structure enabled to *conserve familiarity*.
- **Use profiler before optimizing**: During the implementation of TraCI4Matlab, it was noticed that its performance was lower than TraCI-Python's. Later, the stakeholders concluded that a new requirement was necessary to address this issue. In this regard, the Matlab® profiler helped to identify the bottleneck that caused the low performance. Consequently, the refactoring tasks needed to satisfy the related requirement could be identified, as will be explained in brief.

It was found in the implementation and testing phase of TraCI4Matlab that the Matlab® language specification has some limitations forcing the reverse-engineered design of TraCI-Python to be re-factored. The most important, is that Matlab® imposes only one function definition per m-file, at most, including nested functions, the same holds for class definitions. Moreover, the Matlab's import statement allows adding only package-based functions and classes to the current import list. In contrast, Python allows to have namespace's variables with the following properties:

1. They are not associated with a specific object instance.
2. They can be imported.
3. Their values can be changed by functions in other namespaces.

In order to achieve the same behavior in Matlab®, three options were considered:

- Implement TraCI-Python's namespaces as classes with static members: This solution was discarded because, although Matlab® allows to define constant attributes in a class, the same cannot be done for static ones, i.e. those that do not need the class to be instantiated and whose values can be changed [4]. Note that this option conflicts with property 3.
- Execute m-files that load the required variables into the workspace: This solution would require the Matlab's package functions to access those variables. In the Matlab® documentation, it has been stated that the best practice is to pass the variables as arguments [14]. In this way, not only the workspace would be filled with variables that should be transparent to the user, but he/she would need to pass those variables as arguments, which results impractical. Another strategy listed in the Matlab® documentation, is the use of persistent variables in a function. However, persistent variables can be changed only by the function that defined them, which conflicts with property 3. Finally, one could evaluate a given expression in another workspace, but it has limited flexibility in the sense that it does not allow the variable to contain indexes. For these reasons, this solution was discarded.
- Finally, the use of global variables was chosen because it can deal with properties 1 and 3. Global variables are defined in the functions that require them and can be accessed by any other function.

There were some special cases where there was no need to use global variables. For example, it was found that some variables were used only by one function. Therefore, those variables were defined inside the functions that use them. Another case is related to the `RETURN_VALUE_FUNC` dictionary of the *sumo_object* packages, which has constant values. In this case, a corresponding new class with only constant attributes was defined. Finally, it was found that the `modules` variable of the TraCI package only was used in two functions of the same package: `readSubscription()` and `simulationStep()`. The `modules` variable is a dictionary that associates responses from the SUMO server to the corresponding *sumo_object* module, allowing to detect errors and populate the TraCI subscription results. In the `readSubscription()` function, the `modules` variable is used to populate the TraCI subscription results based on the response of the SUMO server. For this reason, a new dictionary called `subscriptionResults` was defined inside the `readSubscription()` function. On the other hand, the `modules` variable is used in the `simulationStep` module only to reset its values, i.e. the subscription results of each *sumo_object* namespace. Note that, in this case, it is not necessary to define a map. Therefore, a new array called `modules` was defined in the `readSubscription()` function.

Figure 6 shows the re-structured architecture for the implementation of TraCI4Matlab, including the addition of the new package of constants `RETURN_VALUE_FUNC`.

Figure 7 shows the global variables used in the TraCI4Matlab implementation. Note that there are 14 global instances of the class `SubscriptionResults`, namely `edgeSubscriptionResults`, `guiSubscriptionResults` and so on (including the areal detector introduced in the version 7 of TraCI). If no subscription was made to a particular SUMO object, Matlab® sets the corresponding global variable to a null object by default. Recall, that the rest of the variables associated to namespaces are defined in the functions that use them, e.g. the `RESULTS` and `modules` attributes of the TraCI package.

However, as it was explained before, TraCI4Matlab's performance resulted to be worse than TraCI-Python's. Figure 8 shows performance results of both implementations using the cProfile module in the case of TraCI-Python and the Matlab® profiler in the case of TraCI4Matlab. It can be seen that TraCI4Matlab spends much time in sending and receiving messages through the TCP-IP implementation, which is part of the instrument control toolbox. Particularly, in TraCI-Python the `_sendExact()` and `_recvExact()` functions sum 4.903 s while in TraCI4Matab, they sum 119.147 s.

Taking advantage of the high integration capacity of Matlab® and Java, the proposed solution was to develop a new TCP-IP implementation for TraCI4Matlab using Java sockets. The solution involved the creation of a `Socket` class in Matlab® that wraps a Java socket and uses a `DataReader` class [17] which enables to read the entire buffer of the input stream. Figure 9 shows the performance of TraCI4Matlab including the implementation of the `Socket` class. It can be seen that, using Java sockets, the TraCI4Matlab's performance improved substantially. In this case, the `_sendExact()` and `_recvExact()` functions sum 16.711 s, which represent a performance improvement of 85.97 %.

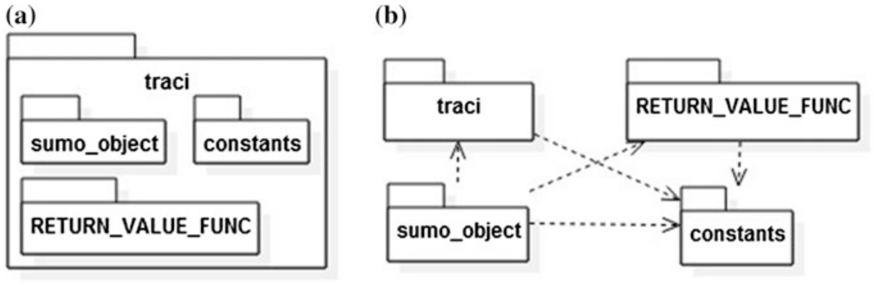


Fig. 6 TraCI4Matlab's components: **a** Deployment diagram, **b** Dependency diagram

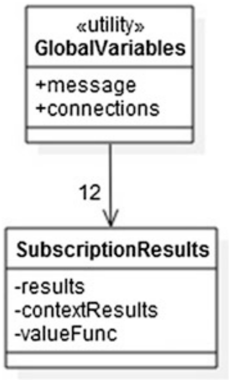
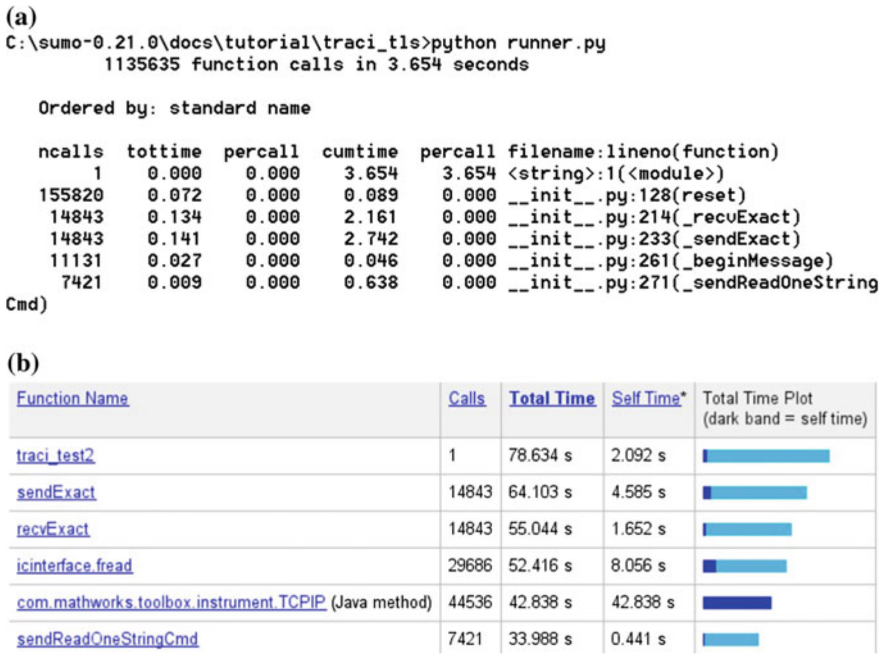


Fig. 7 Global variables defined in TraCI4Matlab



(a)

```
C:\sumo-0.21.0\docs\tutorial\traci_tls>python runner.py
         1135635 function calls in 3.654 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.654    3.654 <string>:1(<module>)
   155820    0.072    0.000    0.089    0.000 __init__.py:128(reset)
    14843    0.134    0.000    2.161    0.000 __init__.py:214(_recvExact)
    14843    0.141    0.000    2.742    0.000 __init__.py:233(_sendExact)
    11131    0.027    0.000    0.046    0.000 __init__.py:261(_beginMessage)
     7421    0.009    0.000    0.638    0.000 __init__.py:271(_sendReadOneString
Cmd)
```

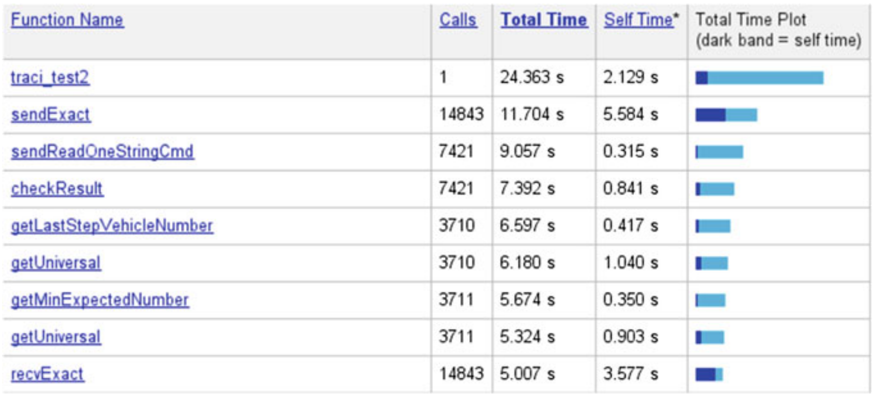
(b)

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
traci_test2	1	78.634 s	2.092 s	
sendExact	14843	64.103 s	4.585 s	
recvExact	14843	55.044 s	1.652 s	
icinterface.fread	29686	52.416 s	8.056 s	
com.mathworks.toolbox.instrument.TCPIP (Java method)	44536	42.838 s	42.838 s	
sendReadOneStringCmd	7421	33.988 s	0.441 s	

Fig. 8 Performance results of **a** TraCI-Python and **b** TraCI4Matlab

5 Results and Discussion

TraCI4Matlab was released on December 24, 2013 under the BSD license. It is free software and is available for the community at Matlab Central [18], or as part of the SUMO contributed tools since SUMO 0.20.0.



Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
traci_test2	1	24.363 s	2.129 s	
sendExact	14843	11.704 s	5.584 s	
sendReadOneStringCmd	7421	9.057 s	0.315 s	
checkResult	7421	7.392 s	0.841 s	
getLastStepVehicleNumber	3710	6.597 s	0.417 s	
getUniversal	3710	6.180 s	1.040 s	
getMinExpectedNumber	3711	5.674 s	0.350 s	
getUniversal	3711	5.324 s	0.903 s	
recvExact	14843	5.007 s	3.577 s	

Fig. 9 Performance results of TraCI4Matlab including a new TCP-IP implementation using Java sockets

Currently, TraCI4Matlab is being used in the project "Modelling and Control of Urban Traffic in the City of Medellin (MOYCOT)" [11]. One of the objectives of the MOYCOT project is to design a MPC (Model Predictive Control) traffic lights control system for the urban traffic network in the city of Medellin. Some parameters needed by this system include the length of the queues in vehicles on each signalized lane and the traffic flow in the edge. Thanks to TraCI4Matlab, preliminary results were obtained in a scenario consisting of an isolated junction, showed in Fig. 10. Using induction loops and lane area detectors, the number of vehicles entering the North-South as well as the length of the queues (jam length in TraCI) on each lane in vehicles were obtained, as shown in Fig. 11.

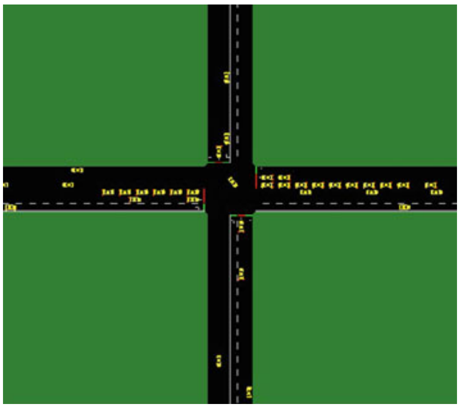


Fig. 10 The isolated junction scenario used in the MOYCOT [11] project to obtain parameters needed for a MPC traffic lights controller

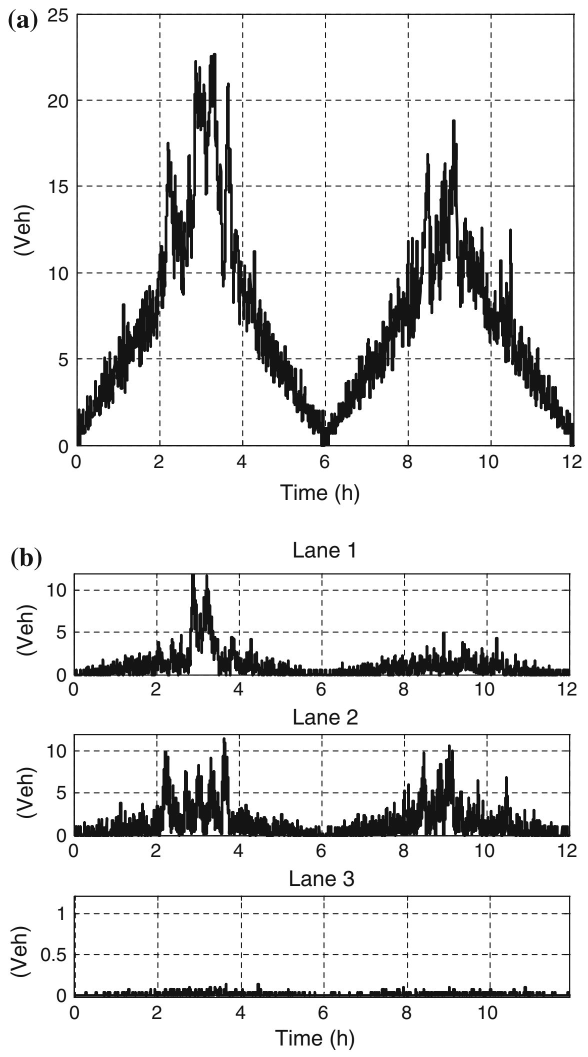


Fig. 11 Data obtained in the north-south edge using TraCI4Matlab: **a** Number of vehicles entering the edge, **b** Length of the queue on each lane in vehicles

6 Conclusions

In this chapter, the re-engineering process of the TraCI API's Python implementation (TraCI-Python) used to develop a Matlab® implementation was presented. Static and dynamic models related to the architectural and component design were obtained. The authors consider that those models can be used to implement TraCI in any object-oriented programming language.

The re-engineering process was supported with *object-oriented re-engineering patterns*, which, in some cases, had to be adapted to the specific case of TraC-I4Matlab. These patterns provide useful guidelines for a re-engineering project, including small-sized projects like TraCI4Matlab.

One of the requirements formulated for TraCI4Matlab was to preserve the same structure of the TraCI-Python's end-user's functions. Although it could be accomplished through the approach described in the forward engineering process, performance implications were not considered. As a result, it was found that performance of TraCI4Matlab was much lower than TraCI-Python's. In order to overcome this issue, a TCP/IP implementation using Java sockets was proposed, which resulted in a substantial performance improvement.

However, the re-engineering process was focused on the migration of TraCI-Python to Matlab®, without taking into account the quality of the subject software in terms of its structure (namespaces as classes, code duplication and code smells). Future work should concentrate on this topic by using (semi) automated tools and possibly including a benchmark with TraCI4J. Further performance improvements should be also considered.

Finally, the design obtained through reverse engineering suggests some private functions and some others with package visibility. Although Matlab® allows to define private functions, it has not defined, to date, a similar approach for the case of functions with package visibility.

Acknowledgments This work was supported by Proyecto Colciencias 111856934640 contrato 941-2012: Modelamiento y Control de tráfico urbano en la ciudad de Medellín. Convocatoria 569.

References

1. Beazley DM (2009) Python essential reference, 4th edn. Addison-Wesley Professional, Upper Saddle River
2. CanforaHarman G, Di Penta M (2007) New frontiers of reverse engineering. In: Future of software engineering, FOSE '07. IEEE Computer Society, Washington, DC, USA, pp 326–341. doi:10.1109/FOSE.2007.15
3. Chikofsky EJ, Cross I JH (1990) Reverse engineering and design recovery: a taxonomy. IEEE Softw 7:13–17. doi:10.1109/52.43044
4. Comparing MATLAB with other oo languages—MATLAB and simulink, n.d. URL http://www.mathworks.com/help/matlab/matlab_oop/matlab-vs-other-oo-languages.html. Accessed 02 April 2014
5. Demeyer S, Ducasse S, Nierstrasz O (2002) Object-oriented reengineering patterns. Morgan Kaufmann, San Francisco
6. egueli/TraCI4J GitHub, n.d. URL https://github.com/egueli/TraCI4J. Accessed 01 April 2014
7. Ewer J, Knight B, Cowell D (1995) Case study: an incremental approach to re-engineering a legacy {FORTRAN} computational fluid dynamics code in C ++. Adv Eng Softw 22:153–168. doi:http://dx.doi.org/10.1016/0965-9978(95)00021-N
8. Krajzewicz D, Erdmann J, Behrisch M, Bieker L (2012) Recent development and applications of SUMO—Simulation of Urban mobility. Int J Adv Syst Meas 5:128–138

9. MATLAB—the language of technical Computing—B, n.d. URL http://www.mathworks.com/products/matlab/. Accessed 09 Sept 2014
10. Mens T, Tourwe T (2004) A survey of software refactoring. IEEE Trans Softw Eng 30:126–139. doi:10.1109/TSE.2004.1265817
11. MOYCOT | MOYCOT, n.d. URL http://www.moycot.org/. Accessed 09 Sept 2014
12. Olbrich S, Cruzes DS, Basili V, Zazworka N (2009) The evolution and impact of code smells: a case study of two open source systems. In: Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement, ESEM '09. IEEE Computer Society, Washington, DC, USA, pp 390–400. doi:10.1109/ESEM.2009.5314231
13. Release notes for MATLAB—MATLAB and simulink, n.d. URL http://www.mathworks.com/help/matlab/release-notes.html. Accessed 09 Sept 2014
14. Share data between workspaces—MATLAB and simulink, n.d. URL http://www.mathworks.com/help/matlab/matlab_prog/share-data-between-workspaces.html. Accessed 02 April 2014
15. StarUML—The open source UML/MDA platform, n.d. URL http://staruml.sourceforge.net/en/. Accessed 30 Jan 2014
16. SUMO_User_Documentation—SUMO—simulation of urban mobility, n.d. URL http://sumo-sim.org/userdoc/. Accessed 30 Jan 2014
17. TCP/IP socket communications in MATLAB using java classes—file exchange—MATLAB central, n.d. URL http://www.mathworks.com/matlabcentral/fileexchange/file_infos/25249-tcp-ip-socket-communications-in-matlab-using-java-classes. Accessed 09 Sep 2014
18. TraCI4Matlab—file exchange—MATLAB central, n.d. URL http://www.mathworks.com/matlabcentral/fileexchange/file_infos/44805-traci4matlab. Accessed 01 April 2014
19. Vincke R, Van Landschoot S, Steegmans E, Boydens J (2012) Refactoring sequential embedded software for concurrent execution using design patterns. Annu J Electron 6:157–160
20. Winpdb—A platform independent python debugger, n.d. URL http://winpdb.org/. Accessed 30 Jan 2014