SHared automation Operating models for Worldwide adoption

SHOW

Grant Agreement Number: 875530

Simulation Suite: Connections between Simulation Levels

This report is part of a project that has received funding by the European Union's Horizon 2020 research and innovation programme under Grant Agreement number 875530

1 Simulation Levels

Simulation levels are the different ways of representing traffic scenarios using computer software and underlying mathematical models. They can vary in terms of the level of detail, complexity, and realism that they capture. Some of the reasons for using different simulation levels are to study different aspects of traffic systems, such as vehicle dynamics, driver behaviour, traffic flow and emissions. Among the factors that influence the choice of simulation level are the availability and quality of input data, the computational resources and software tools and of course the purpose and scope of the study (e.g. testing different scenarios and alternatives, such as new infrastructure, policies, or technologies).

In Graz two simulation levels are used and combined to validate the pilot site. In the area around the bus terminal vulnerable road users (VRUs), such as pedestrians, cyclists, and partially animals, and their interactions with automated vehicles (AVs) are of high interest to assess safety risks. Along the remaining track, individual vehicles and their interactions with other road users and infrastructure (traffic lights) are of interest. Here street level simulation is used to captures vehicle dynamics and traffic flow.

The street level simulations are performed using SUMO. Here the focus is set on traffic flow, congestion and round trip times. Although SUMO allows to define a range – or rather visibility – of the vehicles sensors, they are not a major concern at this abstraction level. They are assumed ideal, i.e. no occlusions from other vehicles or the environment are taken into account. Moreover, the environment is also treated ideally, i.e. all road users move lawfully along predefined lanes.

VRUs level simulation are partially covered using AWSIM. Here the sensors of the vehicle are simulated based on a 3D model of the environment, with all known effects and occlusions of objects. Furthermore, with AWSIM it is possible to use the same algorithms for path planning and obstacle avoidance, that are used in the car, directly in the simulation. Thus, the gap between simulation and reality is kept low. Furthermore, objects can be moved along any trajectory and are not bound by any lanes. On the other hand, the movement of other road users and the definition of traffic is much more complicated on this level.

Thus, connecting simulation levels seems to be a good choice for combining the advantages of both simulation tools in a meaningful way. For this purpose, the following chapter presents the means by which SUMO can be coupled to other simulators during runtime.

2 Possibilities for connections using SUMO

There are various possibilities for extending the traffic simulator SUMO. Since SUMO is an open-source simulator, new features or existing ones can be improved by modifying the source code directly. Another possibility is the usage of SUMO as a library or a tool for other applications, such as self-driving vehicles, traffic management, or logistics. The driver model in SUMO is implemented as a car-following model that determines the acceleration of a vehicle based on its own speed, the speed and distance of the leader vehicle, and other factors. SUMO provides several built-in car-following models, such as Krauss, IDM, EIDM, etc. It is possible to choose one of them as a base model or create a new one from scratch.

TraCI [1] (Traffic Control Interface) allows the access and manipulation of a running SUMO simulation from an external script or application. Various programming languages, such as Matlab, Python, Java, or C++ can be used for TraCI. Unlike the method before, using TraCI does not require the compilation of the SUMO itself. Once connected, TraCI allows the retrieval of information about simulated objects, such as vehicles, pedestrians, lanes, traffic lights, etc. In the opposite direction, the behaviour and state of simulated objects, such as speed, route, colour, or traffic light signals can be controlled. Furthermore, the simulation control flow, such as advancing steps, loading scenarios, or stopping the simulation, can be influenced via TraCI.

In this document we'll discuss the TraCI interface implementation for C++. The basic principles and even the names of most functions should be transferrable to other programming languages. Of course, the syntax has to be adopted when using another supported programming language. In the annex the complete source code for the test application is given, which requires Microsoft Visual Studio Community Edition 2017 or newer running on Windows 10.

2.1 Preparation of SUMO simulation

In order to make a SUMO simulation available for TraCI, a communication port must be defined in the *.sumocfg* file. Theoretically any free TCP/IP port can be used. However, it is recommended to select a port above 1024, because some operating systems restrict the usage of ports below 1024. As soon as this change has been made, SUMO waits after loading the project for a TraCI client to attach. Without the interaction with a TraCI client, the projects seems to be frozen in graphical frontend of SUMO. The following lines need to be added to sumocfg:

```
<traci_server>
   <remote-port value="12345"/>
   <num-clients type="1"/>
</traci_server>
```

2.2 Connect to SUMO Simulation

Basically, a TCP/IP connection to SUMO must be established to start the simulation. This could be done via direct socket programming and sending data packets in

TraCI format manually. However, a better and the preferred method, is to use the TraCI library. When using the library, the connection is established with a single command:

```
client.connect("localhost", 12345);
```

The above command connects to SUMO running on the local machine using port 12345. Problems and errors during communication are handled by exception handlers.

In general, to use TraCI commands it is necessary to encapsulate the functionality in a *TraCIAPI* derived class:

```
class Client : public TraCIAPI
```

2.3 Initialization of TraCI objects

Normally, you want to set various variables and possibly place vehicles manually before starting the simulation. Such activities should happen immediately after the connection is established. For example, a vehicle can be added using the following line of code:

```
vehicle.add("shuttle", "show1", "shuttle", t0, "0", "base", v0);
```

This creates a new vehicle with the name 'shuttle' from vehicle type 'shuttle' which is following the predefined route 'show1'. It spawns at simulation time 't_0' with speed 'v_0' on any available lane immediately at the beginning ('base') of the route. Of course, the vehicle type 'shuttle' and the route 'show1' must be defined before, either in the XML configuration file or as TraCI command.

2.4 Synchronization with TraCI

When controlling the SUMO simulation via TraCI, the question of synchronization arises. In the simple case, the timing is determined via the external client, and the simulation is stopped as soon as the client performs computations. Such a behavior can be realized programmatically in the following way:

```
client.simulationStep(0);
client.initializeVehicle();

for (int i = 1; i < N; i++)
{
     client.simulationStep(i*STEP_SIZE);
     if (client.updateVehicleStatus(i) == false)
       break;
}
client.close();
```

With the first line the simulation time is initialized. Next follows the initialization and configuration of relevant vehicles and other SUMO objects. Within the loop, the simulation time is incremented step by step and for every time step the vehicle update function is called. The simulation stops either after 'N' time steps or if the update function requests for some reason the simulation to stop.

2.5 Retrieve vehicle data via TraCI

During the simulation, a large number of variables can be read out for each SUMO object. The SUMO objects are typically addressed using their id – which is basically a string. For example, the current speed of our ‘shuttle’ can be obtained with the following command:

```
v = vehicle.getSpeed("shuttle");
```

Among others, the following metrics are available: position, speed, acceleration, angle, road id, lane id, distance, energy consumption, CO_2 emissions and color. As simple as these commands may seem, they have a big disadvantage. For every query a request is sent via TCP to SUMO and the response is waited for. When several status variables of several vehicles are to be read out, the communication overhead grows quite high and, thus, the simulation runs slowly. In this case it is better to choose subscriptions. Instead of sending each request individually, a block with the subscribed data is automatically transmitted to the client after every simulation time step. However, it sacrifices flexibility, because the subscription must be set up in advance – either during the initialization phase, but at the latest before the time step where the data is needed.

The syntax for subscriptions requires accustoming and the error-proneness is rather high, but the performance improvement is substantial. The subscription command itself is quite simple:

```
vehicle.subscribe("shuttle", m_vars, 0.0, 650.0);
```

We would like to get for the time span 0 … 650s from the vehicle ‘shuttle’ the list of measures defined in ‘m_vars’. The vector ’m_vars‘ keeps a list of quantities that should be reported. For example, to get position, speed, acceleration, road and lane of the vehicle, m_vars = {0x56, 0x40, 0x72, 0x52, 0x51}. These numbers are part of the SUMO documentation. Beware, an exception is thrown at runtime if invalid identification numbers are specified. To actually get the values transmitted within a subscription, the following statement is needed:

```
TraCIResults sub = vehicle.getSubscriptionResults("shuttle");
```

The individual measurements can then be accessed like the following. As before, the use of wrong identifiers leads to an exception during runtime:

```
v0 = ((TraCIDouble*)(sub.at(0x40).get()))->value;
```

As mentioned in the previous chapter, the simulation of vehicle is not really supported by SUMO. However, the vehicles surrounding (called context), with a specified detection range, can be subscribed like the vehicle parameters would be subscribed:

```
vehicle.subscribeContext("shuttle", 0xa4, 100, m_vars, 0, 650);
```

With the statement above, the surrounding of the vehicle in a range of ±100m is reported after every simulation step. In order to retrieve the list of vehicles, the following instruction must be executed:

```
                                    SubscriptionResults         c         =
vehicle.getContextSubscriptionResults("shuttle");
```

For further details, please refer to the example code in the annex or download the TraCI example project.

2.6 Influence selected vehicle via TraCI

Just as data can be obtained from vehicles during simulation, TraCI also allows changing various states of a vehicle. This includes things like setting the desired vehicle speed, changing to the desired driving lane, changing of the vehicle route, inserting stops, selecting a vehicle color and much more. For example, to change the color of a given vehicle the following statement can be used:

```
    vehicle.setColor("shuttle", shuttlecol);
```

It should be mentioned here that these values are guiding values in the default vehicle configuration. For example, the vehicle may drive slower than the desired speed, according to the safety rules of the car-follow model. Likewise, a lane change can be requested via TraCI. If it is safe to execute, the lane change is actually performed. If the requested lane change is not possible due to traffic, it is simply ignored by SUMO.

Having said that, these safety mechanisms can be overruled programmatically. Warnings, emergency braking maneuvers and even collisions can occur when deactivating the safety checks. This way the intelligence – and lack thereof – of autonomous vehicles could be implemented and tested partially within SUMO. To lift safety checks for a given vehicle, these commands must be issued:

```
    vehicle.setLaneChangeMode("shuttle", 0b000100000000);
    vehicle.setSpeedMode("shuttle", 32);
```

3 Realization for SHOW pilot site

As can be seen in Figure 1, the Graz Pilot Site has been split in two simulation domains: the bus terminal area (red) and the track to and through the shopping center (green). The green part is actually modelled and covered with SUMO while Autoware simulation is responsible for the red part.

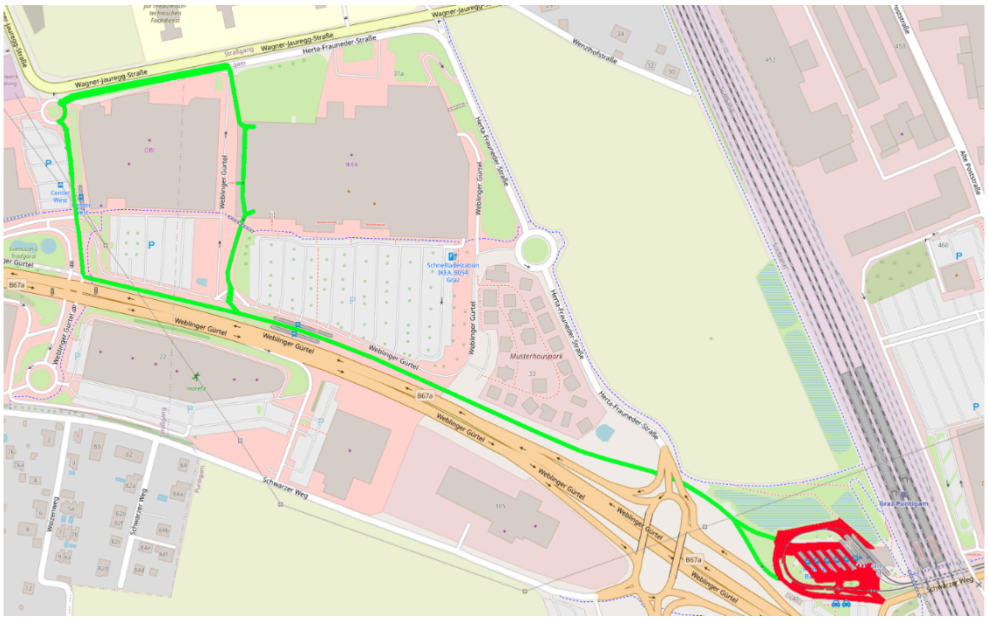

Figure 1: Spatial split of simulation domains

At the predefined handover points both simulations are coupled spatially. This means, the current vehicle state (speed, acceleration) at this location is taken from one simulator and fed into the other simulator and vice versa. This can be done rather easily with the TraCI commands presented in chapter 2.

More precisely, SUMO prepares a base traffic along the major roads in this scene. After a delay of 60 seconds the base traffic has settled. Then the shuttle is spawned with the speed and acceleration provided by Autoware simulation. Since starting and handling the Autoware simulation environment is not trivial, this handover process is done manually. Basically, one TraCI command is needed on the SUMO side for this type of coupling:

```
    vehicle.add("shuttle", "show1", "shuttle", t0, "0", "base", v0);
```

The route ‘show1’ is a list of road sections defining the green track in Figure 1. The vehicle type ‘shuttle’ determines the characteristics and vehicle dynamics of the autonomous shuttle. The shuttle spawns at the start of the track (‘base’) with a delay of t_0 = 60 seconds with the speed v_0, reported from Autoware.

References

[1] (DLR), German Aerospace Center, „Introduction to TraCI,“ 30 3 2023. [Online]. Available: https://sumo.dlr.de/docs/TraCI.html.

Appendix

A small example program is presented to show how TraCI can be used to create a vehicle in SUMO, read out the data during the simulation and influence its behaviour. Besides, it is shown how to start SUMO automatically and how the connection to SUMO is actually established. This program is intended for the use of SUMO under Windows. Visual Studio Community Edition 2017 was used as IDE.

```
#include <Windows.h>
#include <iostream>
#include <stdio.h>
#include <conio.h>
#include <utils/traci/TraCIAPI.h>
#include "parameter.h"

class Client : public TraCIAPI
{
   private:
        FILE *m_csvfile;
        FILE *m_logfile;
        std::vector<int> m_vars;

        double m_a_old;
        int m_a_count;

   public:
        Client()
        {
               //Subscribe following data from shuttle vehicle
               m_vars.push_back(0x56); //lane position
               m_vars.push_back(0x40); //speed
               m_vars.push_back(0x72); //acceleration
               m_vars.push_back(0x52); //lane-idx
               m_vars.push_back(0x51); //lane-id

               m_csvfile = fopen("D:\\Projects\\Show\\vehicle.csv", "wt");
               if (m_csvfile)
               {
                      fprintf(m_csvfile, "PosX;PosY;v;a;Road\n");
               }

               m_logfile = fopen("D:\\Projects\\Show\\kpi.log", "wt");
               {
                      fprintf(m_logfile, "t;v;a\n");
               }

               m_a_old = 0.0;
               m_a_count = 0;
        };

        ~Client()
        {
               fclose(m_csvfile);
               fclose(m_logfile);
        };

        void initializeVehicle()
        {
               libsumo::TraCIColor col, egocol;
               std::vector<std::string> myroute;
               char st[32];
               char sv[32];
               //need to have a string for the start time of the shuttle
               sprintf(st, "%4.2f", (float)EGO_START_T);
               sprintf(sv, "%4.2f", (float)EGO_START_V);

               //Create my own car = the show shuttle vehicle
               vehicle.add("shuttle", "show1", "shuttle", st, "0", "base", sv);

               //Colour the new shuttle red
               egocol.r = 255;
               egocol.g = 0;
               egocol.b = 0;
               egocol.a = 255;
               vehicle.setColor("shuttle", egocol);

               //Subscriptions must be set up before the simulation actually runs
               vehicle.subscribe("shuttle", m_vars, 0.0, 650.0);

               //Substribe context (= surroundings) +/- 200m of ego vehicle
               vehicle.subscribeContext("shuttle", 0xa4, 200.0, m_vars, 0.0, 650.0);
               vehicle.addSubscriptionFilterLateralDistance(200.0, 200.0, 50.0);

               //Example how to influence the vehicle "normal.0" and
               //change color to pink
               col.r = 255;
               col.g = 128;
               col.b = 255;
               col.a = 255;
               vehicle.setColor("normal.0", col);
               vehicle.setSpeed("normal.0", 70.0 / 3.6);
               vehicle.changeSublane("normal.0", 4.0);
               vehicle.setLaneChangeMode("normal.0", 0);
               vehicle.moveTo("normal.0", "entry_1", 5.0);
        }


        bool updateVehicleStatus(int step)
        {
               double v, a, dist;
               int laneidx;
               std::string lane;
               std::string road;
               std::string route;
               libsumo::TraCIPosition pos;
               bool ego_exists = false;

               //Get list of vehicles from running simulation
               std::vector<std::string> idlist = vehicle.getIDList();

               //See if the ego vehicle is among the vehicles in the list
               for (const std::string& s : idlist)
               {
                       if (s.compare("shuttle") == 0)
                       {
                               ego_exists = true;
                               break;
                       }
               }

               //If the shuttle completes its trip then force end of simulation
               if (step > EGO_START_T / STEP_SIZE && ego_exists == false)
                       return false;

               //If the shuttle has not yet started we can't collect its data
               if (step <= EGO_START_T / STEP_SIZE)
                       return true;
               //Get information from particular vehicle
               v = vehicle.getSpeed("shuttle");
               lane = vehicle.getLaneID("shuttle");
               road = vehicle.getRoadID("shuttle");
               dist = vehicle.getDistance("shuttle");
               pos = vehicle.getPosition("shuttle");
               a = vehicle.getAcceleration("shuttle");
               laneidx = vehicle.getLaneIndex("shuttle");

               //Basically the same as above, but better performance and less
               //communication overhead. On the other hand, the subscription
               //must be registered before
               libsumo::TraCIResults const &test =
                               vehicle.getSubscriptionResults("shuttle");

               //Get the surrounding vehicles (and their data) of the ego vehicle
               libsumo::SubscriptionResults const &ctx =
                               vehicle.getContextSubscriptionResults("shuttle");

               //And add these vehicles to our list of vehicles
               for (std::pair<std::string, libsumo::TraCIResults> const& cveh : ctx)
               {
                       std::string const &cname = cveh.first;
                       libsumo::TraCIResults const &cvalues = cveh.second;

                       //ToDo: use surrounding vehicle ids and values
               }

               //Debug output on screen
               printf("pos=(%6.1lf/%6.1lf),v=%4.2lf,a=%3.2lf\n",pos.x,pos.y,v,a);

               //Write current values to CSV file
               if (m_csvfile)
               {
                       fprintf(m_csvfile,"%6.1lf;%6.1lf;%4.2lf;%4.3lf;%s\n",
                               pos.x, pos.y, v, a, road.c_str());
               }

               //Detect KPI relevant braking events
               if (m_logfile)
               {
                       //Find strong decelleration events
                       if (a < STRONG_BRAKING_DECCEL &&
                           !(m_a_old < STRONG_BRAKING_DECCEL) &&
                           v > 2.0)
                       {
                               fprintf(m_logfile, "%5.1lf;%4.2f;%4.3lf\n",
                                                     STEP_SIZE*step, v, a);
                               m_a_count++;
                       }
                       m_a_old = a;
               }

               //Just for testing:
               if (step == 1)
               {
                       std::vector<std::string> route = vehicle.getRoute("shuttle");
                       printf("route = \n");
                       for (std::string section : route)
                       {
                               printf("    %s\n", section.c_str());
                       }
               }
               return true;
        }
```

```
};


void startup(const char* lpApplicationName, const char *parameters)
{
        // additional information
        STARTUPINFOA si;
        PROCESS_INFORMATION pi;
        char cmdline[2048];

        // set the size of the structures
        memset(&si, 0, sizeof(si));
        si.cb = sizeof(si);
        memset(&pi, 0, sizeof(pi));

        strcpy(cmdline, lpApplicationName);
        strcat(cmdline, " ");
        strcat(cmdline, parameters);

        // start the program up
        CreateProcessA
        (
                lpApplicationName,   // the path
                cmdline,             // Command line
                NULL,                // Process handle not inheritable
                NULL,                // Thread handle not inheritable
                FALSE,               // Set handle inheritance to FALSE
                CREATE_NEW_CONSOLE,  // Opens file in a separate console
                NULL,                // Use parent's environment block
                NULL,                // Use parent's starting directory
                &si,                 // Pointer to STARTUPINFO structure
                &pi                  // Pointer to PROCESS_INFORMATION structure
        );

        // Close process and thread handles.
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
}

int main(int argc, char* argv[])
{
        Client client;
        int i, res;
        bool ret;
        bool remotecontrol;

        //Start Sumo with project configuration file
        printf("Starting SUMO now\n");
        startup("D:\\Projects\\sumo-1.7.0\\bin\\sumo-gui.exe",
                 "--start --quit-on-end -c D:\\Projects\\Show\\osm.sumocfg");
        printf("SUMO started\n");

        //Wait until SUMO is ready to accept a TCP connection
        Sleep(1500);

        try
        {
                printf("Connecting to SUMO\n");
                client.connect("localhost", 12345);
                printf("Client connected.\n");
                std::cout << "time in s: " << client.simulation.getTime() << "\n";

                client.simulationStep(0);
                client.initializeVehicle();
```

```
            for (i = 1; i < 6000; i++)
            {
                   client.simulationStep(i*STEP_SIZE);
                   printf("step %d, time is %lf s\n",
                          1+i, client.simulation.getTime());

                   ret = client.updateVehicleStatus(i);

                   if (ret == false)
                          break;
            }

            client.close();
     }
     catch (tcpip::SocketException e)
     {
            printf("SUMO communication error occurred!\n");
     }

    return 0;
}
```