RWTH Aachen University
Software Engineering Group

Domain Specific Languages for Simulation

Seminar Paper

presented by
Svejda, Jan

1st Examiner: Dipl.-Inform. Deni Raco

2nd Examiner:

Advisor: Dipl.-Ing. Evgeny Kusmenko

The present work was submitted to the Chair of Software Engineering

Aachen, June 1, 2018

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Abstract

Testing software solutions in the real world is expensive, even more so in the context of autonomous driving. As a result, ways of modelling driving scenarios in a virtual world are a topic of research, whose results are gradually transferring into the industry. In order to describe such simulations, a number of Domain Specific Languages (DSLs) exist. These provide a machine and human readable format to determine its details. In this work, DSLs for simulations are described in detail, compared and then used to demonstrate how to take advantage of them in vehicle simulations and to better grasp their application.

Contents

Chapter 1

Introduction and motivation

The development of new products comes with extensive prototyping and testing, more so in the car industry and especially in regards to autonomous driving. Doing so in the real world, however, incurs significant costs and risks, which are oftentimes unacceptable (like health risks and life threats). Simulating the environment, car, weather and/or light conditions to create testing situations substantially mitigates these risks.

In order to efficiently leverage such simulating techniques, there exists a number of Domain Specific Languages (DSLs), which aim at simplification and streamlining of the process of creating test cases and scenarios. Such a DSL allows to better represent a problem domain's characteristics. Modelling vehicles and traffic simulations already in a software language, encourages shorter implementation times. Experts, who, for instance, work on solutions for the autonomous car, can leverage DSLs to improve their algorithms, support testing and increase productivity.

The goal of this research paper is to, firstly, present detailed features of existing simulation description languages. These languages will be then compared with each other. Afterwards, examples of how it is possible to use them are to be implemented and tested in a chosen language.

There are many more DSLs for simulations and since each and every one of them has its own features and limits, it is important to understand them, so that they are used properly and where they shine most. This paper should enable better decisions on which to use for which purpose, to motivate developers to further enhance their languages and maybe remove some of their limits.

Chapter 2

Simulation Description Languages

A Simulation Description Language (SDL) serves as a means of representing a real-world system in a computer readable form. With that, a computer is able to perform time-based simulations, i.e. apply physics engines to it and see its behaviour in time.

There exists a multitude of SDLs, each developed with a certain purpose in mind and therefore with various features and capabilities. The aim of this chapter is to describe some of the existing SDLs.

2.1 Simulation Description Format

Simulation Description Format (SDF) is a powerful open-source language created for describing all parts of a simulation, including the world environment, models like robots with joints, sensors and other properties, light conditions, materials, terrain etc. It evolved from Universal Robot Description Format (URDF) and contains all its features.

SDF can fully specify a simulation in detail in a hierarchical structure. It is an Extensible Markup Language (XML) format and comes with its own C++ parser, but its main intention was to be used as a format for *Gazebo*, which is a robot simulation program. It features a useful Graphical User Interface (GUI) for creating SDF models. SDF also has an extensive format specification available on-line [SDF18], since it can represent all kinds of simulation information.

2.1.1 SDF Structure

An SDF file starts with an `<sdf>` tag, that encompasses a *world* element. A world can contain *models*, *scene*, *physics*, *joints*, *actors*, *lights* and *plugins*. It describes the whole simulation environment. A model, actor and light may be defined as standalone outside a world.

One of the most important elements is the model. With this, SDF characterizes the robots – physical properties like inertia, size, shape, movable parts, sensors, collision elements and further information. A model is built up using so called *links*, which express the aforementioned attributes for parts of the model. The links then can be connected using several types of *joints* – revolute, prismatic, screw, gearbox, ball, fixed or universal. Since

robots would not know much about their surroundings without sensors, any link may have a number of sensors defined. There are many of them already supported – GPS, camera, ray, sonar, wireless receiver/transmitter etc. – and more can be implemented using plugins.

```
<?xml version='1.0'?>
<sdf version='1.6'>
    <world name='my-world'>
        <model name='example-car'>
            <link name='body'>
                <pose >0 0 0.25 0 -0 0</pose>
                <visual name='visual'>
                    <geometry><box>
                        <size>1 0.7 0.35</size>
                    </box></geometry>
                </visual>
                <!-- ... -->
            </link>
            <link name='wheel_fl'>
                <pose >0.35 0.4 0.2 0 1.5707 1.5707</pose>
                <visual name='visual'>
                    <geometry>
                        <cylinder>
                            <length>0.1</length>
                            <radius>0.2</radius>
                        </cylinder>
                    </geometry>
                </visual>
                <!-- ... -->
            </link>
            <link name='wheel_fr'> <!-- ... -->
            <joint type="revolute" name="fl_wh_hinge">
                <pose>0 0 -0.03 0 0 0</pose>
                <child>wheel_fl</child>
                <parent>body</parent>
                <axis><xyz>0 1 0</xyz></axis>
            </joint>
            <joint type="revolute" name="fr_wh_hinge"> <!-- ... -->
        </model>
    </world>
</sdf>
```

Listing 2.1: A simple SDF example of a model car – a body-box and 4 wheels. Some parts of the file are omitted for brevity, like the collision elements and rest of the wheels and joints.

In a scene, the world's appearances are specified. This includes ambient lighting, background colour, sky (sunset, sunrise, time of day, clouds), shadows or fog. Next, the physics tag is used for controlling the dynamics engine, thereby influencing the simulation itself. Another feature, the actors, are a way of incorporating animated objects with scripted movements. This could be used, for instance, to simulate moving pedestrians or any other secondary moving elements in a car driving simulation. For these particular models, the possibility to define light sources and their properties in SDF is convenient, too.

Plugins in SDF provide means of taking advantage of the Gazebo's extensibility. A plugin

is code, either pre-installed or written by the user, that gets compiled as a shared library with the simulation. A user can use the Gazebo's C++ Application Programming Interface (API) to implement desired or missing functionality and thereby extend and customize the possibilities of simulations. After implementation, an SDF element (like world, model, link etc.) specifies the usage of a plugin, which will be then called during the simulation [KH04].

As mentioned earlier, SDF is an XML format. For a user, directly writing up models in XML is not particularly engaging. Gazebo's user interface does a good job at giving the user means of creating SDF documents. Nonetheless, in a team, version control systems might prove to introduce some unwanted issues. Since, for example, *Git* is known to have problems with merging changes of XML documents. Large simulations, however, cannot be expected to be implemented solely by individuals.

Furthermore, the language could benefit from more mechanisms for code reuse. In general-purpose languages, like C++ or *Java*, the notion of inheritance has become an integral part of how developers build systems. SDF as a domain specific language, of course, aims for a distinct purpose, even so integrating and embracing the idea would allow for more manageable simulation descriptions. There is a way to go around this issue – use a template engine to generate the specification. This will help to significantly decrease code repetition, but means the loss of straightforward implementation and additionally, requires further tools [QGS15].

2.2 Universal Robot Description Language and Xacro

The primary origin of URDF resides in the world of Robot Operating System (ROS) – a full-fledged tool-set for the development of robot applications, algorithms and other various projects. URDF is very similar to SDF, because it basically constitutes its predecessor. URDF has features which are the subset of those in SDF. The XML basis and even terminology stayed mostly unmodified in SDF. The main difference, however, is, that URDF defines only one robot's kinematic and physical properties, no world, no scene etc.

Like before, the structure of the robot needs to be definable in a tree-like fashion of subcomponents – links, joints. Links describe the individual components (e.g. visual and collision shape, position, material, colour, inertial information etc.), joints then the interaction between them (continuous, rotational, planar, prismatic or even no motion at all). With these two conceptually simple elements, URDF enables to express complex simulation models.

2.2.1 Xacro

A favourable way how to generate URDF is to use an XML Macro language – *Xacro*, especially for larger projects. Xacro introduces the following useful concepts and language constructs [XCR18]:

1. properties and property blocks, basically variables containing a value or a whole block of code;
2. math expressions to compute values, instead of hard-coding them;

3. conditional branching for blocks;
4. importing Xacro source files;
5. YAML syntax support;
6. adding other XML elements or attributes;
7. the highlight – macros – parametrized constructs, that generate code;
8. default values for macro parameters.

By using Xacro, it is possible to significantly reduce the amount of hand-coded XML. Unfortunately, loops are not inherently supported by this language, which presents a strong drawback, even though this decision (taken for the sake of simplicity) is understandable.

```
<xacro:property name="wheel_visual">
    <visual>
        ...
    </visual>
</xacro:property>

<xacro:macro name="default_inertial" params="name">
    <link name="wheel_${name}">
        <xacro:wheel_visual />
    </link>
</xacro:macro>

<xacro:default_inertial name="front_left" />
```

Listing 2.2: A brief example of how a property and macro construct in *Xacro* might be used to streamline production of URDF description files. First a property for the visual shape of a wheel is defined (some code omitted) and then a macro calls this property to create a link with a parametrized name.

2.3 MuJoCo Format

Multi-Joint dynamics with Contact (MuJoCo) is a physics engine, that is capable of running simulations in generalized coordinates with contact dynamics (which rely on solving optimization problems). MuJoCo engine boasts many features and great performance for simulations. It often outperforms the physics engines employed in Gazebo in robotics-related computations. Multi-body simulations proved difficult for it, even though it maintained the best precision out of all the engines [ETT15].

Now more relevantly to this paper, the definition of simulations is carried out in MuJoCo Format (MJCF), again an XML based format. Furthermore, MuJoCo provides a C++ API, so that models can be created programmatically. Due to the fact, that some models become extensively complex, the C++ API can serve greatly in expressing what a simulation engineer wants to model [TET12]. In addition, MuJoCo is capable of reading URDF model files, even with a few extra extensions.

MJCF strives to tackle the issue of growing models with its *default setting mechanism*, which demonstrates another example of how MuJoCo addresses these problems. This

```
<mujoco model="test"><script/>
    <compiler coordinate="global"/>
    <default>
        <geom rgba=".9 .7 .1 1" size="0.01"/>
        <site type="sphere" rgba=".9 .9 .9 1" size="0.005"/>
        <joint type="hinge" axis="0 1 0" limited="true"
            range="0 60" solimplimit="0.95 0.95 0.1"/>
    </default>
    <visual>
        <headlight diffuse=".7 .7 .7"/>
    </visual>
    <worldbody>
        <body>
            <geom type="cylinder" fromto="-0.03 0 0.2  -0.03 0 0.15"
                size="0.03" rgba=".2 .2 .5 1" density="5000"/>
            <joint type="slide" pos="-0.03 0 0.2" axis="0 0 1"
                limited="false"/>
            <site name="s1" pos="-0.03 0 0.2"/>
        </body>
        <site name="s2" pos="-0.03 0 0.32"/>
      <body>
            ... <!-- Some further bodies. -->
      </body>
    </worldbody>
    <tendon>
      <spatial width="0.002" rgba=".95 .3 .3 1" limited="true"
            range="0 0.33">
            <site site="s1"/>
            <site site="s2"/>
            <geom geom="g1"/>
            ...
        </spatial>
    </tendon>
</mujoco>
```

Listing 2.3: An example of using MJCF for describing a simple robot with a tendon-like feature for representing a rope. The entire example can be found in [MJC18]. This concrete example yields a scene depicted in Figure 2.1.

mechanism resembles *Cascading Style Sheets (CSS)* from the *Web*. The big amount of properties and settings MJCF provides are largely reduced as a result of assuming that the settings' value does not change until altered in a child element [MJC18].

In MJCF, there are a number of elements available for constructing models. *Bodies*, which have mass and inertia, make up the kinematic tree. (The simulation world is a body.) *Joints*, which allow motion between bodies – MJCF has 4 compoundable types: *slide*, *hinge*, *ball* and *free*. *DOFs*, or Degrees of Freedom, for damping, limiting velocity, joints or tendons and for armature inertia. Then there are *geoms* that serve for collisions and tendon wrapping. *Sites*, that can be used for placement of sensors, applying force, adjusting tendons etc. They stay at the same position relative to a body. Next, there are *constraints* – predefined or user-defined. With those, all kinds of conditions, that need to be upheld, can be specified. Then, *tendons*, special objects, that represent the shortest path from one point to the other while going through sites or going around geoms. Lastly, *actuators* and *sensors*, which can, for instance, control modelled mechanisms by pulling,

pushing or moving joints or, in case of sensors, receive information [TET12].

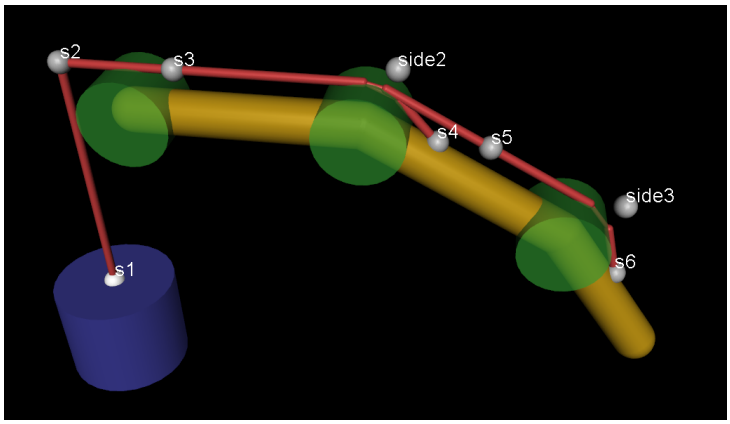

Figure 2.1: A scene as visualized by MuJoCo from a simulation description file in MJCF using a tendon element (in red), cf. Listing 2.3. Source: [MJC18].

2.4 SimLang – Simulation Language

Simulation Language (SimLang) has been created specifically for vehicle simulations to describe objects and scenarios. It is a Domain Specific Language (DSL) that utilizes *MontiCore*, a so-called "language workbench". MontiCore helps in developing DSLs by generating a whole infrastructure in Java, that ranges from model parsers, validity checkers until model transformations [HR17].

SimLang arose from the development of a simulator called *MontiCAR Simulator*. This simulator addresses some of the shortcomings of modern simulators (like Gazebo or Simulink) and endeavours to bring the best of each together into one. In particular, multi-platform support, large environment simulations with a sufficient degree of detail, real-world models of cities or locations, integration of sensors, actuators and controllers and last but not least, automated unit testing. In [GKR+17], they show that none of the existing simulators embraces all of the mentioned issues in a satisfactory manner. MontiCAR does. Furthermore, it is capable of extending its simulator engine by external helper simulators, incorporate Simulink models, use OpenStreetMaps and visualize simulated data for testing, development and demonstrations.

It is a rather new language currently under development. The attributes from a SimLang description file can be most easily accessed via *SimLangContainer*. It holds a data structure with Java Optionals, which have getters for access. With that, SimLang can be integrated into simulation tools as needed [SL18]. This, however easily integrated with Java tool, presents a certain inconvenience for the usage in others. For now, though, its main function stays with MontiCAR.

The description in SimLang can be divided into three logical parts. Namely, simulation rules, communication/channel rules and weather rules. Not all of the rules are mentioned here, for a complete list and allowed values consult the GitHub of SimLang [SL18].

The aforementioned simulation rules refer to specific settings of the simulation (as indicated by the name). With them, it can be expected to specify things like render frequency,

duration, current map to be loaded and its properties, time of the day, gravity and/or the vehicle. The file can describe an arbitrary number of cars, that may have a specific path to follow (with a start and destination), that are randomly spawned in quantities in a region and follow a path or that are randomly spawned around the whole world. Pedestrians are also definable, either with random or concrete paths.

```
sim Highway {
  sim_duration 4h;
  time 12:00;
  map_name Highway.osm;

  weather fixed {
    temperature -10°C;
    windstrength 30m/s;
    winddirection 7°;
    precipitation_type hail;
    precipitation_amount 50mm;
    weather_phenomena fog;
  };

  car1(-22.0,-34.0,90.0)->(-1.0,0.0,10.0);
  // ...
}
```

Listing 2.4: Vehicle simulation scenario of a car driving on a highway with some serious weather conditions. For more scenarios visit [SL18].

Alongside the individual simulation elements, communication/channel and weather rules may be defined. The former sets properties for communication paths like transfer rate, latency, the probability of outage and area of effect. With weather rules, it is possible to indicate the temperature, humidity, pressure, wind conditions, clouds and various phenomena like fog, storms, smog, tornadoes or even a rainbow. This way, almost any thinkable weather conditions are describable, a necessity for road scenarios.

Chapter 3

Comparison of description languages

In the previous chapter, various Simulation Description Languages (SDLs) have been mentioned. Needless to say, every one of them has evolved with a bit different goal in mind. For vehicle simulations, it is important to choose the correct tool for the development of models and scenarios. Hopefully, this chapter will bring some light into the decision. Some key aspects of every mentioned language are considered in relation to others – how easy/hard working with them is, compatibility with Operating Systems (OSs), documentation, differences in features, but of course also the simulators themselves.

3.1 Ease of use

Most of the studied languages are Extensible Markup Language (XML) dialects – namely Simulation Description Format (SDF), Universal Robot Description Format (URDF) and even MuJoCo Format (MJCF). This means a lot of good editor support – syntax highlighting, indentations, simple code completions etc. XML has a strong foundation in the IT world, which facilitates the development using these dialects. On the other hand, XML files tend to grow to large and illegible monuments due to its verbosity as time goes on and models advance. Furthermore, problems with team development, which have been mentioned in Chapter 2, might present a substantial letdown for some users. The issues with growing models are significantly reduced by *Xacro*, which conveniently transforms macros into XML. Its use has been intended mainly for URDF, which is why most of the tutorials and examples generate URDF, nonetheless, other XML based languages can benefit from it as well. MJCF shows a more elaborate solution – default settings mechanism, by which unnecessary duplicate code gets eliminated. Simulation Language (SimLang) differs quite a lot from the others, since it does not derive from XML, but has its own syntax. As a result, the descriptions are concise and express precisely the intention. This shows how Domain Specific Languages tailor the language to the needs of a concrete use-case.

Of course, any SDL can be hand-coded in XML, but a Graphical User Interface (GUI) helps tremendously in that regard. For SDF, *Gazebo* has its own integrated GUI. It is possible even to alter the underlying SDF file while running Gazebo and afterwards import

it into the world to visualize the model.[1] URDF is partly compatible with SDF so with certain caveats, Gazebo's GUI can be used for it, too.[2] Most of the 3D modelling work, though, can be done in a Computer Aided Design (CAD) software, like *SolidWorks*, which even offers a plugin for exporting to URDF. Its companion Xacro, however, as a macro language does not have a designated GUI, neither does MJCF or SimLang.

Proper documentation of a language highly elevates its user-friendliness. For many users a must. SDF just as well as URDF have on-line specifications along with several tutorials on usage and guidelines for first-comers. Every XML element's meaning is explained and for many there are even examples. Documentation of SDF is especially useful, since it shows all available child elements, that link to their own documentation. For MJCF there are extensive resources on-line, though quite thorough, they do not reach the comprehensibility of SDF's. Concerning SimLang, its GitHub pages[3] introduce the language in a rather simple manner, but every attribute is described well enough. Some examples can be found there, too. Compared to the others though, the documentation is meagre.

Out of all, SDF and URDF evince the largest community that keeps moving the simulators and language forward. This means, that any calls for help with difficult issues are usually answered and recurring problems already have been. This cannot be said about SimLang, because it is a very young language and for now is used mostly in the academic sphere of RWTH. Finally, MJCF does not have such a big community like SDF or URDF, nevertheless, on the on-line forum of its website, many issues can be resolved and in case of a new question, the developers are active in assisting with any troubles.

3.2 Compatibility

Ability to develop simulations in a multi-platform environment might be a decisive point for some users. Of course, the languages themselves can be written in any environment (since they are essentially text files), but running them and using specific tools for them might prove bothersome or downright impossible. With that said, SimLang has its tools implemented in Java, so they run in a Java Virtual Machine (JVM), which can be installed on basically any modern OS. As for Gazebo, which runs SDF, only Debian packages can be downloaded, so support for multiple platforms is limited to Debian derivatives (and Debian itself obviously). A manual on how to compile its sources on Windows is available, but the process is tedious and error-prone. Robot Operating System (ROS) has a similar shortcoming – it supports only Debian, Ubuntu and Arch Linux – so URDF development takes place there, too. Regarding MuJoCo, the basic library has support on Windows, Linux as well as macOS, but for more advanced features, like Virtual Reality interactions and Unity integration, the user has to stick with Windows.

3.3 Features

While many other aspects of SDLs are important, its features, arguably, play the main role. In the range of the described SDLs, they differ quite a bit. While URDF is capable

[1] Though after some time, the world gets overcrowded by tens or more slightly different models, so it can become confusing.

[2] See, for example, the tutorial on `http://gazebosim.org/tutorials/?tut=ros_urdf`

[3] `https://github.com/MontiSim/SimulationLanguage`

of describing a robot including its geometrical and physical properties, SDF extends the format to not only the descriptions of multiple such robots, but also to the simulation environment itself. Therefore, a user can define a whole scenario with arbitrary scene objects, light conditions, wind, sound, temperature and more. For vehicle simulations not a negligible advantage. For the part of robot description, though, their features stay identical.

MJCF remains the preferred format for MuJoCo, even though the simulator supports loading models in URDF, too. As a matter of fact, URDF can be transpiled into MJCF. Not the other way around, since MJCF contains features not available in URDF, like tendons, custom elements and a multitude of element attributes. All in all, it can do a more thorough job of robot/vehicle specification, nevertheless, compared to SDF, it has limited possibilities of describing the environment, that amount, basically, to mostly just objects with simple traits like shape and material.

With regards to SimLang, this language's intention reflects its features. With vehicle simulations in mind, the language contains everything a scenario engineer will need. It does not focus on the robot description itself, rather on how to combine models and set up the environment – general simulation values, maps, weather conditions and vehicle to vehicle communications. It is unique in its own right with these possibilities, especially in regards to the communication rules – none of the other languages have this feature so explicit.[4] Model specifications, however, are not viable with SimLang.

[4]For SDF and MJCF it would probably be possible through custom implemented plugins.

Chapter 4

Examples of Simulations

In this chapter a few examples in Simulation Description Format (SDF) and Simulation Language (SimLang) will be presented. This aims at showing some features and applications of these two languages. In particular in regards to vehicle simulations, SDF has been chosen for its comprehensive expressiveness in terms of the environment and vehicle specification. Concerning SimLang, the main reason comes from its conciseness and capability of creating scenarios rapidly. Furthermore, their respective simulators – Gazebo and MontiCAR – are well suited for vehicle simulation development.

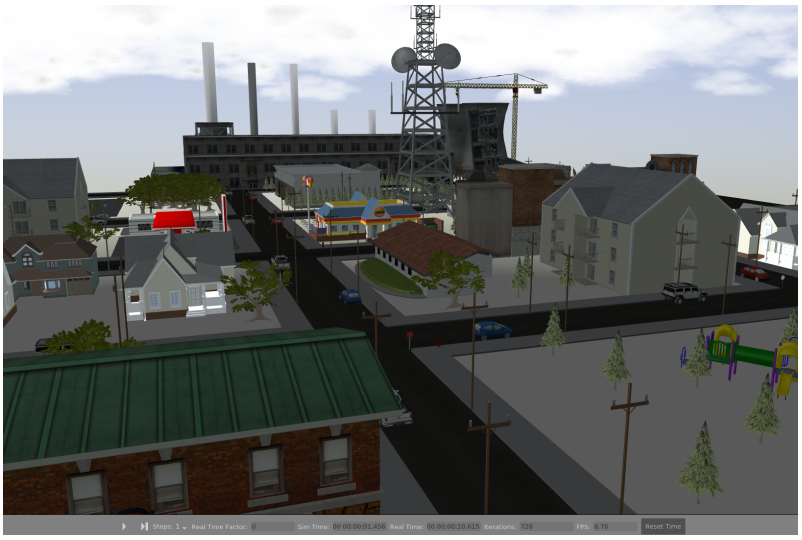

Figure 4.1: A screenshot of the city simulation world created by OSRF.

4.1 SDF

Gazebo has established itself as a particularly useful simulator. There are essentially four ways to create a scenario for it:

1. create the models and environment in SDF from scratch;
2. use Gazebo's Graphical User Interface (GUI) to build up the simulation;

3. reuse existing models from the community (or previous projects);

4. combination of all of the above.

Taking advantage of them in the most useful way can, of course, elevate the effectiveness of simulation engineers. For instance, there already exists a prepared city environment from the OSRF. It can be downloaded, compiled and used directly in a Gazebo simulation, even though, it proves to be quite resource demanding. The city contains moving and stationary cars, buildings, streets with signs, lampposts and other common objects. The main point of interest, though, is the autonomous Toyota Prius, which even has multiple sensors (like altitude, longitude, latitude, velocity etc.), whose data can be downloaded and studied for testing and development [VC17].

Not all vehicle simulations have to be done in a city simulation, though. For agriculture or industry (like mining, drilling etc.), movement on a general surface might be of much higher importance. Simulations could find many use-cases in these scenarios, for example, harvesting, sowing, weeding, heavy transportation of material from one point to another and much more. In such cases, having a precise altitude map is of great importance. With some work, such a Digital Elevation Model (DEM) can be downloaded and processed into *GeoTiff* format and imported into Gazebo.

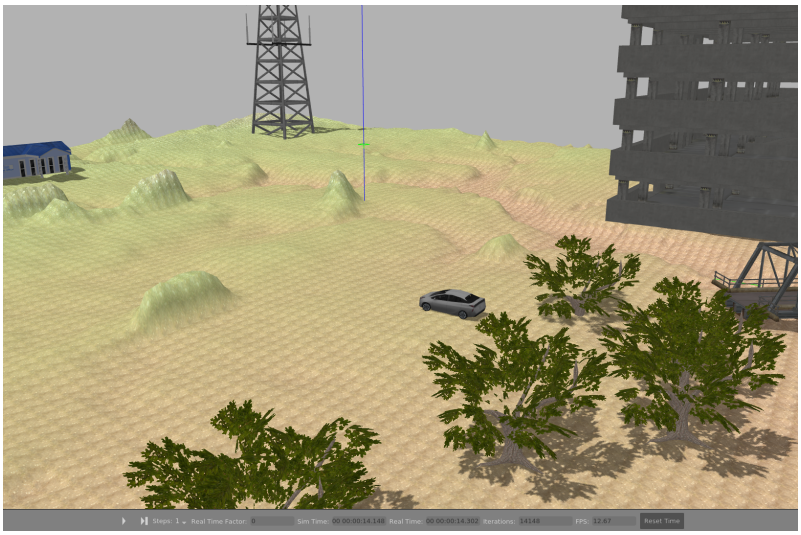

Figure 4.2: A screenshot of the simulated environment based on real (scaled-down) elevation data of Kohlcheid and its surroundings. Some objects are placed into the world, including a car, for demonstration.

The procedure of how to create such a world will be described on the *Kohlscheid* example, which can be found on the author's GitHub.[1] First of all, elevation data have to be downloaded – many free sources are available on the Internet, even with very high resolution (like LiDAR maps). In this example, a region around Kohlscheid has been picked from GMTED2010 data [DG11]. Afterwards, in a program like *QGIS*, the data can processed into a desirably cropped image, that can be exported in *GeoTIFF* format and successively down-scaled[2] so that Gazebo does not get choked on map data (the resolution is usually

[1] `https://github.com/JanSvejda/kohlscheid-sdf`
[2] With a program called `gdalwarp`.

sufficient). With this, an SDF scenario can already import the map into a simulation with an `include` tag, Gazebo will automatically recognize it as an elevation map and show it as such. For better experience, some material textures can be added on top of the height map alongside with any desired world objects. This can be viewed in the repository's file `kohlscheid.sdf` while the resulting scene is depicted in Figure 4.2. Some more information and explanation on how to accomplish importing elevation data can be found in [DEM14], however, not everything works like described, since some pieces are outdated.

4.2 SimLang

For creating a SimLang scenario, two files are necessary. A `sim` and `osm` file. The former contains a SimLang description, while the latter is OpenStreetMap world data. The maps can be exported on OpenStreetMap's portal. [3] After selecting these files they get uploaded to a simulation server via a web user interface, where the simulations can be visualized, too. The server is equipped with a default auto-pilot, so simulation scenarios can be tested right away.

```
sim MorningRain {
  sim_duration 1h;
  weather fixed {
    temperature 280K;
    wind_strength 10m/s;
    precipitation_type rain;
    precipitation_amount 30mm;
  };

  time 7:00;
  pedestrian_density 10;
  map_name mapsample.osm;

  my_car(212.0,286.0,0.0)->(160.0,120.0,0.0);
  <p> (160, 120.5, 0)->(160, 120.5, 0); // pedestrian
}
```

Listing 4.1: An example of a scenario with strong rains, wind and some pedestrians in the morning.

SimLang does have a lot of features, unfortunately, not all of them are implemented or at least visualized in MontiCAR. Helpful would be, for instance, to have an option to see the simulation's settings, to be able to delete and create scenarios more easily – without the need to specify both files at the same time, but to have a list of available maps and then simply upload SimLang files – and to see the defined weather and lighting conditions. Nevertheless, the project is still under development, so improvements will surely follow.

As can be seen in Listing 4.1, a very complex scenario can be described in just a few lines of code. SimLang also has a big advantage, because its syntax is close to the domain language of a simulation expert. Therefore, the usage is straightforward and the learning curve gentle. For example, suppose you need to simulate a scenario with many vehicles, but only two of them you actually care about and merely some of them should be on a similar path. Such a fairly complex scenario can be easily defined in SimLang, cf. Listing 4.2.

[3] Quite often though, the simulator seems to fail at parsing the maps.

```
sim ManyCars {
  ...

  important_car1(a,b,c)->(x,y,z);
  important_car2(a',b',c')->(x',y',z');

  <v> (i,j,k)->(l,m,n) 20;

  <v> 100;
}
```

Listing 4.2: Code excerpt of how to describe a scenario with many cars. Two of them are of interest, twenty more will randomly follow a certain path and a hundred will be randomly driving around the map.

For vehicle to vehicle communication, SimLang offers a number of features, that enable fast scenario build-up, though again, it is not easily demonstrable in the simulator. A channel can be either local (with coordinates and radius), global (available throughout the entire map) or bound to a vehicle (though the way of specifying which vehicle is unclear); either foolproof or with outage probability and can also concretely specify the latency and transfer rate. Therefore, it is possible to actually construct something like Internet communication, cf. Listing 4.3.

4.3 Discussion

Even though the verbosity of SDF can become rather annoying, the maturity of the language and Gazebo offers a stable simulation setting, with rich support for scenario description, including, as shown on an example, topography data, object definitions, reuse of models and more. The simulations can be further enhanced with Robot Operating System (ROS) tools for controlling the vehicle (e.g. with autonomous driving algorithms), sensors, actuators etc. Moreover, there is a large community ready to help with any problems and many on-line sources to learn the workflow. Some more work could be done on how the models are created from the GUI, but maybe a different program would be more suitable for it, since the primary purpose of Gazebo is the simulation itself. Be it as it may, Gazebo and SDF offer such a multitude and variety of possibilities how to set up a simulation environment, that almost anything can be put to test on a computer instead of the real world.

```
sim InternetComms {
  ...

  fixed channel InternetMock {
    transfer_rate 10 Mbit/s;
    latency 30ms;
    outage 0.1;
    area global;
  }
}
```

Listing 4.3: Code excerpt of how to describe communication on the map scale, simulating, for instance, how the Internet would behave.

All in all, SimLang is a feature-full language, which does have the potential of becoming

very useful for vehicle simulations. It is rather new and still has some ways to improve, but its concept overall aptly fits the simulation domain. Especially of interest are the communication channels, which could become a unique sell-point for scenarios of vehicle to vehicle communications, in the future maybe even vehicle to infrastructure communications. The simulator and its GUI also have to mature into an industrially viable product first, nevertheless, the software addresses so many of the drawbacks of other simulators, that it really might become an attractive solution for companies in the automotive industry.

If one thing should be said though, it is that all of the mentioned languages do some things better than other, which means they all can be improved and developed further. To increase their usability, applicability and their capabilities.

Chapter 5

Conclusion

This short research paper looked in detail at various different simulation description languages. A better understanding in the decision of which language to choose for vehicle simulations has been acquired by studying the capabilities of each, looking at its features, purpose or available simulators and comparing them to each other.

Universal Robot Description Format (URDF) has established itself for the description of robots in the big toolset of Robot Operating System (ROS). With ROS, the user gets an entire framework of libraries and tools for in-the-loop development of simulation-first solutions. An enhancement of URDF development can be found in Xacro – an Extensible Markup Language (XML) macro language, that takes some of the code bulk away. Next, Simulation Description Format (SDF) resolved many of the limitations URDF has, like describing multiple robots and a whole world environment in one SDF file. On top of that, the robot specifications syntax was adopted from URDF, so in that part they are almost compatible. Gazebo can be even hooked up to ROS for more control.

The paper also mentions a language from a different corner, MuJoCo Format (MJCF). It comes with interesting language constructs – like its default setting mechanism, tendons and custom elements. The MuJoCo simulator is powerful and well optimized, nevertheless, for vehicle simulations it lacks some key features, e.g. seamless map imports. Lastly, a bit of a newcomer, Simulation Language (SimLang) represents a Domain Specific Language (DSL) specially created for use in vehicle simulations. It comes as no surprise then, that it features support for maps, weather, communication channels, pedestrians, path specification and multiple car definitions. Though in development, both the language and the simulator (MontiCAR) promise a bright future, because they solve many shortcomings of other simulation tools. 

In the examples, it has been shown, that in SDF as well as SimLang, complex vehicle scenarios can be implemented. The applications of SDF, for instance, are not limited only to road transportation, but include agricultural and industrial settings for testing any relevant algorithms. SimLang, on the other hand, is very well suited for regular autonomous driving settings, because it provides essentially all of the attributes needed for possible road situations. Even communication routes are definable.

Future autonomous driving systems will have to be prepared for anything on the streets (and land). Rigid, safe and dependable systems can be reached only by rigorous testing and verification. Cars will have to react rapidly and correctly to avoid collisions, damage and death. They will have to communicate on a local and regional scale. If done properly,

simulations will save a lot lives, money and work in this whole endeavour – and the right language will, too.

Bibliography

[DEM14] Digital Elevation Models, Gazebo `http://gazebosim.org/tutorials?tut=dem` , January 2014.

[DG11] J.J. Danielson and D.B. Gesch. Global multi-resolution terrain elevation data 2010 (GMTED2010). Technical report, U.S. Geological Survey, 2011.

[ETT15] T. Erez, Y. Tassa, and E. Todorov. Simulation Tools for Model-Based Robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX. In *Proceeding of IEEE International Conference on Robotics and Automation*, pages 4397–4404. IEEE, June 2015.

[GKR+17] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation framework for executing component and connector models of self-driving vehicles. In *Proceedings of MODELS 2017*, pages 109–115, 2017.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench*, volume Edition 2017. Shaker, 2017.

[KH04] N. Koenig and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, Sept 2004.

[MJC18] MuJoCo `http://www.mujoco.org/`, May 2018.

[QGS15] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS*. O'Reilly, 2015.

[SDF18] Simulation Description Format `http://sdformat.org/`, May 2018.

[SL18] Simulation Langugage GitHub `https://github.com/MontiSim/SimulationLanguage`, May 2018.

[TET12] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, Oct 2012.

[VC17] Vehicle and city simulation `http://gazebosim.org/blog/car_sim` , October 2017.

[XCR18] Xacro `http://wiki.ros.org/xacro`, May 2018.

Appendix A

Acronyms

API Application Programming Interface.

CAD Computer Aided Design.

DEM Digital Elevation Model.

DSL Domain Specific Language.

GUI Graphical User Interface.

JVM Java Virtual Machine.

MJCF MuJoCo Format.

MuJoCo Multi-Joint dynamics with Contact.

OS Operating System.

OSRF Open Source Robotics Foundation.

ROS Robot Operating System.

SDF Simulation Description Format.

SDL Simulation Description Language.

SimLang Simulation Language.

URDF Universal Robot Description Format.

XML Extensible Markup Language.