PlaSMA
Multiagent Simulation

PlaSMA Multiagent Simulation

User Guide

Version 13.03, March 2013

Contents

1 Introduction

The software and agent platform PlaSMA, originally developed in the context of the Collaborative Research Center 637 (SFB 637) at the University of Bremen, provides means for the simulation of large-scale logistic scenarios. Agents may represent autonomously acting entities such as means of transport and cargo. Depending on their respective specification these agents constitute either the intelligence of a single entity in the simulation environment or they manage groups of entities in the simulation environment. A further class of agents is that of service agents which have secondary (logistic) functions and thus provide for instance traffic information or contract brokering services.

Besides the simulation backend and a graphical visualization client, the PlaSMA installation package comprises a selection of example logistics scenarios that have been developed for demonstration purposes.

The document at hand provides detailed information about the process of setting up the PlaSMA simulation environment and the compilation/configuration of custom application scenarios. Besides, the simulation model which rests at the core of the PlaSMA simulation backend is introduced, i. e., the synchronization model and the server world model. In the end, the document describes the steps that developers must take in order to successfully implement new agents based on the PlaSMA API.

The remainder of this user guide is structured as follows: In succession to this introduction, Chapter 2 provides a step-by-step explanation of the installation of a complete PlaSMA simulation environment under *Microsoft Windows* and *Linux*. The chapter also explains the configuration of both the simulation backend (Section 2.2) and the client (Section 2.3).

Chapter 3 is then dedicated to the description how to deploy a successfully installed PlaSMA environment both for non-distributed simulation settings (Section 3.1) and distributed settings which involve multiple hosts that jointly host the simulation (Section 3.1.2).

Chapter 4 describes the way (existing) scenarios are set up in their respective configuration. Topics which are covered here comprise amongst others the configuration of the simulation runs (Section 4.1), including synchronization and domain configuration, and the logging of scenarios (Section 4.3).

While the description of the scenario configuration so far was sufficient in order to adapt given scenarios provided for instance as standard scenarios together with the PlaSMA releases, Chapter 5 is geared towards the creation of new scenarios. Section 5.1 details the default structure of PlaSMA scenarios followed by an explanation of the Apache Ant-based standard build and deployment process for scenarios as well as guide lines when starting work on a new scenario. Section 5.4 outlines the steps to create a custom logistics world model suitable for the respective simulation task at hand where the focus is primarily on an ontology-based modeling of this world model. Section 5.5 is concerned with the development of new types of agents. The chapter is concluded with an introduction to simulation time synchronization and the mechanism used in PlaSMA.

2 PlaSMA Setup

The following section describes how to set up PlaSMA for the major branches of operating systems the development team is actively supporting, namely *Linux*[1] and Microsoft *Windows 7*. In the following OS-specific setup instructions we assume a standard installation of the respective OS. Wherever special software packages are required for successful deployment of the PlaSMA multi-agent-based simulation environment this is explicitly noted.

The PlaSMA simulation environment mainly consists of four software components:

- The simulation core (*PlaSMA Core*) that controls the execution of simulation scenarios and can be thought of as an intermediate level built on top of the functionality provided by the underlying FIPA-compliant multi-agent system JADE.
- A (optional) database system that stores all the data and performance metrics generated by a running scenario for analysis purposes.
- The *PlaSMA Client* for selection and control of simulation scenarios by the user and for their visualization, if required.
- The actual agents that represent logistic objects and populate the scenarios.

The OS-specific release version is packaged as tar.bz2 or Zip archive and is available for download at `http://plasma.informatik.uni-bremen.de/download.html`.

Please unpack the contents of the downloaded archive to the directory you wish PlaSMA to be installed in. In the following we will refer to this directory as `<plasma_dir>`.

2.1 Installation and Setup of required and optional Software

This section describes the setup of required third party software. There is a short step-by-step guide for each of the supported operating systems. In case of specific further problems, please refer to the handbook for the corresponding software.

2.1.1 PlaSMA Requirements

The following set of third-party software packages is required for the full operation of the simulation system.

[1] Ubuntu is the primary Linux distribution of our core developers.

- *Sun Java SE 6* (or newer)
  Available from http://java.com/download or *OpenJDK 6*
- (optional but recommended) *PostgreSQL 9.1* (an installation of version 8.3, 8.4 or 9.0 might work as well)
  Available from http://www.postgresql.org/download
  If the usage of PostGIS is planned (see 5.4.4) a 32 bit version of PostgreSQL is recommended.

For these software packages the following installation steps apply:

Java Installation

There are different variants of Java. PlaSMA uses Oracle Java Standard Edition (SE) or OpenJDK; other versions might work as well but are not tested by the PlaSMA-Team. These editions are available as software development version (Java Development Kit, JDK) and as plain run-time version (Java Runtime Environment, JRE).[2] It is pre-installed on several systems. But you have to check that it is at least version 6.[3] If you want to implement new agents the JDK is needed. If the version of the installed JRE or JDK is below Java 6 an update is necessary.

Java installation on Windows Hosts If there is no appropriate Java installed, download the Java installer (JDK or JRE) for your Windows platform from the Website stated above. Just execute the Java installation program and follow the instructions. There are no specific installation options required by PlaSMA.

In any case, you have to make sure that Java is contained in the Windows program search path listing (environment variable `$PATH$`). The listing (separated by semicolons) must contain an entry like `C:\Program Files\Java\jdk1.7.0_10\jre\bin` or `C:\Program Files\Java\jre7\bin`. This path must correspond to the location where your Java JRE or JDK is installed. Add the correct path if that is not the case.[4]

Java installation on Ubuntu Hosts If you are using Ubuntu you can simply install the newest version of OpenJDK (JDK or JRE) from the Ubuntu repositories.[5]

```
prompt> sudo apt-get install openjdk-7-jre
```

or if you need the JDK

```
prompt> sudo apt-get install openjdk-7-jdk
```

PostgreSQL Installation

PostgreSQL is the database used by PlaSMA to store your simulation data. If you do not intend to evaluate a simulation scenario but rather want to visualize it for demonstration purposes, you might skip the installation of PostgreSQL. Some scenarios might need a PostgreSQL/PostGIS installation, the scenario maintainer can tell you if an installation is required. An installation at a later point is also still possible. If you choose to skip the installation of PostrgreSQL note that you have to configure the PlaSMA server accordingly (see Section 2.2).

[2] If you don't intend to actually implement own agents (i. e., writing Java code) the JRE is fine for you.
[3] You can check Java version by typing `java -version` on command line. (The internal version number for Java 7 (or 6) will be indicated as 1.7 (or 1.6) though.)
[4] The procedure is explained here: http://support.microsoft.com/default.aspx?scid=kb;en-us;310519
[5] If you want to install Java from Oracle you can do so, by downloading the linux specific archive from the the Oracle download page http://www.oracle.com/technetwork/java/javase/downloads/index.html and follow the instructions from Oracle.

Installation procedure Windows Before installing PostgreSQL please ensure, that you have removed all files from previous PostgreSQL installations. When removing previous installation files you might be prompted to restart your Windows operating system.

It is recommended that most users download the binary distribution for Windows. Latest versions of PostgreSQL are available as a Windows Installer package from the PostgreSQL website at the downloads section.

There are two kinds of installer packages available for Windows operating systems. Generally it is recommended to choose the “one click”-installer package. The “one click”-installer package is designed to be as straightforward as possible and the fastest way to get up and running with PostgreSQL on Windows. The following installation instructions regard to the “one click”-installer package.

In case you are about to run PostgreSQL on a Windows operating system, which is part of a domain network, you might experience problems with the startup of the PostgreSQL database service. In this case it could be necessary to set the password of the Postgres-Service manually using the Windows control panel and restart the server again.

Run the “one click”-installer package to start the installation process. Note that you can only use a installer package as an administrator. To run the executable with the user rights of an administrator, you can right click the executable file and choose “run as administrator”. If you are not allowed to run an executable as an administrator and/or you do not have sufficient user rights on your local machine, ask your system administrator for support.

Please follow the instructions and use any given default settings throughout the installation process. Note that the installation routine fails to setup up the database automatically on a file system other than NTFS. Therefore we strongly recommend to choose an installation directory, that resides in a NTFS file system. During the installation process you are prompted for a password for the database superuser (namely *postgres*, which is also used as a Windows service account). Later you will have to configure PlaSMA with this database user information.

Now the basic components of PostgreSQL are being installed. At the end of the installation process you are being asked whether the Stack Builder application should be run. You should request no and manually start *Stackbuilder* as an Administrator from the Windows start menu.

The Stack Builder application helps you to install additional components of PostgreSQL. Please choose to install the following additional components, which are required for PlaSMA when storing simulation data is desired:

- *JDBC* database driver (*pgJDBC* in category "Database Drivers")
- *ODBC* database driver (*psqlODBC* in category "Database Drivers")[6]
- (optional) *PostGIS* (see 5.4.4) driver (*PostGIS* in category "Spatial Extensions").

Then the selected additional components are being downloaded and installed by the Stack Builder setup wizard. Please check during installation of PostGIS whether you specified the installation path of PostgreSQL correctly.

Please continue with Section 2.1.1 PostgreSQL Configuration for PlaSMA.

Installation procedure Ubuntu It is recommended to install the PostgreSQL version which is listed in the software repository of your preferred Linux distribution. In Ubuntu, *postgresql* is the name of the required package. To install the package on Ubuntu Linux, type the following command on the console.

```
prompt> sudo apt-get install postgresql
```

Make sure that *postgresql-client* is also installed on your system. Usually, it will be installed with *postgresql* automatically. You can do so by typing:

```
prompt> /etc/init.d/postgresql status
```

[6]ODBC can be used to access the database with MS-Excel

The output should look like this:

```
9.1/main (port 5432): online
```

You can also choose to install the **pgadmin**-tool, if you plan to configure the database through a graphical environment instead of the shell (see 2.1.1). This is done by typing:

```
prompt> sudo apt-get install pgadmin3
```

For full compatibility you might also want to install the following *optional* packages via apt:

- *ODBC* database driver (package `odbc-postgresql`)[6]
- (optional) *PostGIS* (see 5.4.4) driver (package `postgis`).

you can do so by typing

```
prompt> sudo apt-get install postgis odbc-postgresql
```

PostgreSQL Configuration for PlaSMA

PlaSMA utilizes PostgreSQL in order to store simulation data. Such simulation data can be used to analyze the model-based simulation runs. This in fact might help to understand in depth the computations for a particular system model, that was subject to simulation. For further details about the logging and computational metric we encourage you to continue reading in Section 4.3 and Section 5.5.10.

Generally there are three different ways to configure PostgreSQL database for using PlaSMA. One of these implies to use the *pgAdmin III* tool; the other two ways make use of a command line tool. We recommend to use the ready-to-use command line tool together with a configuration script as described in the following subsection. Note that in order to have your simulation runs being computed in a distributed fashion, you will most probably have to make additional settings as described in Section 2.1.1.

For all methods you should ensure, that the PostgreSQL service has started and is running successfully.

Windows
You can check the status of the PostgreSQL Windows service in the Windows services overview via the system panel.

Ubuntu
You can checkt the status of PostgreSQL by typing:

```
prompt> /etc/init.d/postgresql status
```

If the service is not running type:

```
prompt> sudo /etc/init.d/postgresql start
```

Base Configuration, Method 1: Initialization Script

As a first option you can use the `initdb` script file provided in `<plasma_dir>` that can create both the PlaSMA database user and the PlaSMA database and then initialize the database automatically without any user interaction (except entering a password). The script can be used as follows[7]:

Windows

```
prompt> initdb.bat <path/to/postgresql_dir> <postgres_main_user>
```

[7] You will need to start the Windows command line console or an Linux Terminal first, enter the directory `<plasma_dir>`, and then execute the script as stated.

Ubuntu

```
prompt> sudo −i −u postgres
prompt> ./initdb.sh postgres
```

On Windows the first argument for the script specifies the PostgreSQL installation directory. (Whitespaces need to be escaped with a leading ^.) The last argument specifies the main PostgreSQL user (*postgres* by default). A particular call of the script might be as follows:

```
prompt> initdb.bat "C:\Program^ Files^ (x86)\PostgreSQL\9.1" postgres
```

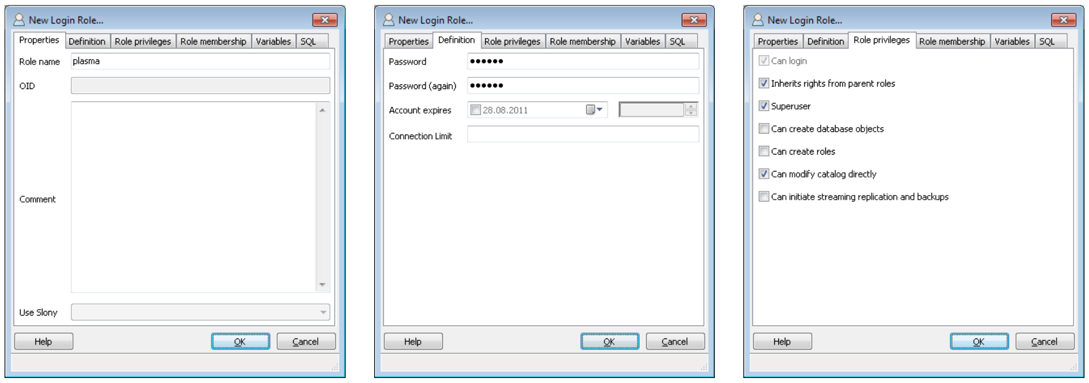
During the execution of the script, the user is prompted to enter a password for the freshly created database user *plasma*. In cases where the local network connection to the database is not trusted, the user is required to enter the password for the PostgreSQL main user at certain stages of the setup process. If the script succeeds in setting up the database PlaSMA is ready for deployment.

Base Configuration, Method 2: pgAdmin GUI

In this subsection we will guide you through PostgreSQL database setup using the graphical frontend *pgAdmin III*.

Connecting to the server After you started the Application you can see the object browser on the left hand side of the *pgAdmin III* tool window. Doubleclick an entry of the server list to establish a connection to this particular server. (Usually the local server shows up automatically. You might have to add the server through *File → Add Server...* if that is not the case.) You will be prompted to enter the server password. By default you have specified this password during the PostgreSQL database installation routine.

Creating a new user For database access we need a new database user with sufficient rights. Therefore open the context menu (right-click on mouse) over the list entry *Login Roles* of the *Object Browser*. Choose *"New Login Role..."* from the context menu. A new application dialog comes up, which shows the new login role to be created. Figure 2.1 shows a configuration, that can be used to run PlaSMA. Please confirm the login role settings in this dialog to create the new login role. As a result you can see this new login role in the *Object Browser* list view.



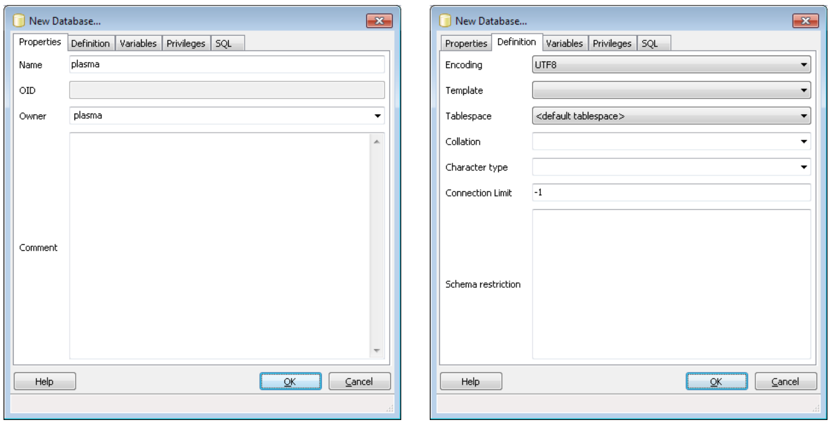
Role name plasma
OID
Comment
Use Slony

Password ••••••
Password (again) ••••••
Account expires ☐ 28.08.2011
Connection Limit

☑ Can login
☑ Inherits rights from parent roles
☑ Superuser
☐ Can create database objects
☐ Can create roles
☑ Can modify catalog directly
☐ Can initiate streaming replication and backups

Figure 2.1: Creating a new login role in *pgAdmin III*.

Creating a new database In the next step we will create a database dedicated to PlaSMA. This can be done via a context menu over the *Object Browser* list view as well. Open a context menu over the list entry *Databases*. Then choose *"New Database..."*. Again a new application dialog comes up, in which you can specify the new database according to the screenshots in Figure 2.2. It is important to set access permissions for the previously created login role. Setting this user as the owner of your dedicated database is just fine. PlaSMA will assume, that you use UTF-8 character encoding in this database, which is set by default. When you confirm these setting a new database will be created.

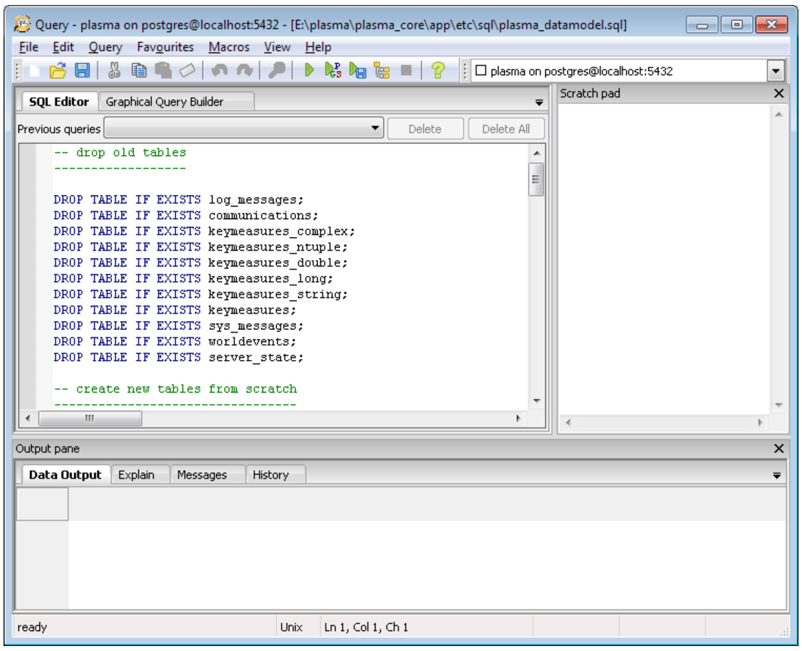


```
Name              plasma
OID
Owner             plasma
Comment
```

```
Encoding          UTF8
Template
Tablespace        <default tablespace>
Collation
Character type
Connection Limit  -1
Schema restriction
```

Figure 2.2: Creating a new database in *pgAdmin III*.

Setting up the database After creation of the database, we load a database schema, which describes the structure of the PlaSMA database. The database structure is generated by the SQL script found in `<plasma_dir>/etc/sql/plasma_datamodel.sql`. This script contains SQL statements which create tables, fields and indices needed. In order to execute the script, select the newly created database in the *Object Browser* and choose from the menu bar *Tools* → *Query tool*. A new application dialog comes up, within you can load the script file and execute it (see Figure 2.3). Choose *Query* → *Execute* from the menu bar to run the script once. Thereafter, the PlaSMA database is ready for use.



```
-- drop old tables
------------------

DROP TABLE IF EXISTS log_messages;
DROP TABLE IF EXISTS communications;
DROP TABLE IF EXISTS keymeasures_complex;
DROP TABLE IF EXISTS keymeasures_ntuple;
DROP TABLE IF EXISTS keymeasures_double;
DROP TABLE IF EXISTS keymeasures_long;
DROP TABLE IF EXISTS keymeasures_string;
DROP TABLE IF EXISTS keymeasures;
DROP TABLE IF EXISTS sys_messages;
DROP TABLE IF EXISTS worldevents;
DROP TABLE IF EXISTS server_state;

-- create new tables from scratch
---------------------------------
```

Figure 2.3: Loading database schema via *pgAdmin III*.

(Optional) Setting up network access The following configuration instructions refer to distributed computation of your simulation. In case you would not like to have computation distributed within your network you can step over the following configuration. You can edit the network setting via *Tools* → *Server Configuration* → *pg_hba.conf*. In this configuration file you have to add all hosts of you network, which in turn may access the PostgreSQL database on your local machine. The following format should be applied for IP addresses *<IP-Adresse>/32*.

Base Configuration, Method 3: Command Line

In this subsection we will guide you through PostgreSQL database setup using the command line tool. First open a command line window. On Windows change to the *bin* directory of PostgreSQL. You will find this directory as a subdirectory within the PostgreSQL installation path (e. g., `C:\Program Files\PostgreSQL\9.1\bin`).

Now, we will create a new user role for our PlaSMA database (with permissions to create other users and databases and with an encrypted password). Enter and execute the following command. You may choose an arbitrary user name.

Windows

```
prompt> createuser.exe -U postgres -s -P -E <user_name>
```

Ubuntu

```
prompt> createuser -U postgres -s -P -E <user_name>
```

Then we will create a new database dedicated to PlaSMA. Enter and execute the following command. This command takes the aforementioned user name as an argument. You may choose an arbitrary database name (by default it is *plasma*).

Windows

```
prompt> createdb.exe -U <user_name> <database_name>
```

Ubuntu

```
prompt> createdb -U <user_name> <database_name>
```

At last the database structure is being generated. To build up the database structure we use a create script. Enter and execute the following command. This command takes the full path to our create script as an argument, as well as the name of PlaSMA database.

Windows

```
prompt> psql.exe -U <user_name> -d <database_name> -f <plasma_dir>\sql\plasma_datamodel.sql
```

Ubuntu

```
prompt> psql -U <user_name> -d <database_name> -f <plasma_dir>/sql/plasma_datamodel.sql
```

Additional Configuration for Distributed Simulation

To record data during a distributed simulation session, it is necessary to modify the configuration of the PostgreSQL database system. After a successful installation, you need to edit the file `postgresql.conf` as user with administrator/root privileges. The location of this file depends on your operating system. If you use the default installation Directory this might be:

Windows

```
C:\Program Files\PostgreSQL\9.1\data\
```

Ubuntu

```
/etc/postgresql/<postgresql_version>/main/postgresql.conf
```

Most of the options are already listed in the configuration file but may be disabled by the commentary symbol (#) at the beginning of the line. To enable an option, this symbol has to be removed.

In any case you should enable password encryption:

```
password_encryption = on
```

If you wish to use more than one computer for your simulation, TCP/IP connections to the database must be allowed, you can create a wild-card for all IPs with * or you allow IPs explicitly with a whitespace seperated list.[8]

```
listen_addresses = '123.456.789.1 123.456.789.3'
```

Configuration of Access Privileges

The modification of database access privileges is performed once again as the *root* user. Per default, it will only be possible to access the database locally if the login of the database user and owner of the current Linux session match.

The following configuration will change this state of affairs such that access is allowed both locally and from remote machines on the network using a username and password identification scheme. For the sake of security both are transmitted using an MD5-Hash.

At the end of the file `pg_hba.conf`[9], change the access configuration as shown in the configuration excerpt below:

```
# Database administrative login by UNIX sockets
local  all   postgres     md5
# TYPE DATABASE USER CIDR−ADDRESS METHOD
# "local" is for Unix domain socket connections only
local  all   all        md5
# IPv4 local connections:
host  all   all   127.0.0.1/32  md5
# IPv6 local connections:
host  all   all   ::1/128    md5
```

If TCP/IP connections from other hosts are desired, it is necessary that either the particular host or the subnetwork it resides in is granted access.

```
# IPv4 connection from host 192.168.0.X
host all all 192.168.0.0/24 md5
```

In order to finally adopt the configuration changes, the PostgreSQL server must be restarted.

2.2 PlaSMA Server Configuration

The server configuration file at `<plasma_dir>/etc/serverconfig.xml`[10] contains global settings and is needed to run the PlaSMA server. The server settings cover the message and database logging configuration. The following enumeration lists all elements and whether they are required or optional and can occur once or multiple times in the configuration file. An example server configuration is shown in Figure 2.4.

<server> *required, once.* Is the top element and has one optional attribute.

  clientport *optional.* Sets the port at which clients can connect to the server. The default port is *1099*.

There are two sub elements enclosed in *<server>*.

[8]If you don't plan to run distributed simulations, you should not set this option, because opening your database this way may be a security risk.

[9]Usually located in the same directory as `postgresql.conf`.

[10] Technically interested users can find the configuration file as a XSD schema at `<plasma_dir>/etc/serverconfig.xsd`.

<scenariodir> *optional, multiple.* Describes additional directories where the PlaSMA server searches for scenarios. The default scenario directory `<plasma_dir>/scenarios` is included automatically without being configured explicitly. The path can be specified in *<scenariodir>*'s **path** attribute:

> **path** *required.* Sets the path for additional scenarios.

<logging> *required, once.* Configures the way PlaSMA server emits messages and where these are stored or displayed. The attributes of the *<logging>* element are:

> **loglevel** *required.* It sets the standard log level according to the implemented log level types, including *NONE*, *ERROR*, *WARNING*, *INFO*, *DEBUG* and *TRACE*.[11]

Generally there are two output channels for the server: the database and the console output. Consequently they are configured first in the *<logging>* element.

<database> *optional, once.* This element configures the database connection of the server. If omitted the database is turned off, as well as if the **enabled** attribute is set to *false*. The database settings should be configured according to the database setup (see also Section 2.1.1). *<database>* contains these attributes:

> **enabled** *optional.* If *false*, the database connection is turned off and there is no usage of it. The default value is *true*. Use this to turn off the database without deleting the whole element.
>
> **host** *optional.* The address of the database host. Can be a plain IP address or a name (e. g., *192.168.0.xx* or *localhost*). The default value is *localhost*.
>
> **name** *required.* The name of the database created for PlaSMA.
>
> **user** *required.* The username to connect to the database host.
>
> **password** *required.* The password credentials to connect with the formerly stated username to the database host.
>
> **dbtype** *required.* Currently PlaSMA only supports PostgreSQL databases. So the only available value is *POSTGRESQL*.

<console> *optional, once.* This element configures the console output channel. If omitted the console output is disabled.

> **enabled** *required.* Set *true* to displaying log messages on console.
>
> **colored** *optional.* Set *true* to color the output (just on supporting platforms, e.g., Linux). The default value is *true*.

<category> *optional, multiple.* The log categories may be hierarchically ordered by providing an absolute path, e.g., **name** = *"System.Communication.Input"*. If, e.g., the category *"System.Communication"* is not specified it will be implicitly created. Additionally it will inherit all settings from the category *"System"*. The ordering in which these categories are specified has no effect. The **loglevel** attribute is optional – if it is not provided it is inherited from its parent category.

> **name** *required.* Defines the hierarchical logging category for which a loglevel is to be set.
>
> **loglevel** *optional.* The special loglevel for this category. Inherits from parent category by default.

<backtracking> *optional, once.* Configures the tracing of errors during simulation runs. The system can be configured to temporarily store a certain amount of protocol messages that would normally be discarded regarding their log level. In case of an error they will be logged and may be helpful in tracing back the error. The attribute **loglevel** limits which protocol messages are temporarily kept. The number of messages has to be configured by the **bufferSizeModel** and **bufferSize** settings. If the model is *ABSOLUTE*, a maximum of **bufferSize** messages is temporarly stored whereas the *PER_AGENT* model is allowed to store **bufferSize** messages per agent.

Due to the distributed architecture of the simulation these settings apply for every connected computer and not for the simulation cluster as a whole. That is because the traceback information are managed locally on every participating computer.

[11] See section 4.3 for more details on log levels.

```
<?xml version="1.0" encoding="ISO−8859−1" ?>
<server xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance" xsi:noNamespaceSchemaLocation="serverconfig.
     xsd" clientport="1099">
  <scenariodir path="./scenarios"/>
  <logging loglevel="INFO">
   <database enabled="true" host="localhost" name="plasma" user="plasma" password="plasma" dbtype="
        POSTGRESQL"/>
    <console enabled="true" colored="true"/>
    <category name="Communication" loglevel="NONE"/>
    <backtracking loglevel="TRACE" bufferSizeModel="PER_AGENT" bufferSize="100"/>
  </logging>
</server>
```

Figure 2.4: An Example of the server configuration.

```
<?xml version="1.0" encoding="ISO−8859−1" ?>
<client xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance" xsi:noNamespaceSchemaLocation="clientconfig.
     xsd">
  <!−− PlaSMA Server Parameters −−>
  <server name="local server" address="localhost" port="1099" index="0" />
  <!−− Aurora Parameters −−>
  <visualization enabled="true" />
</client>
```

Figure 2.5: The default client configuration.

loglevel *required*. The loglevel at which messages should be kept for backtracing.

bufferSizeModel *required*. If *ABSOLUTE*, the number of messages kept is independent of the simulation state. If otherwise *PER_AGENT*, the number of messages depends on the number of simulation agents currently in the simulation run.

bufferSize *required*. The size of the buffer that holds the log messages. The actual size depends on the **bufferSizeModel** attribute, too.

2.3 PlaSMA Client Configuration

The mandatory client configuration (located at `<plasma_dir>/etc/clientconfig.xml`) provides settings which specify the preferred PlaSMA server to which the client should establish a connection. Figure 2.5 shows an example for a client configuration.

Editing the connection informations is fully supported by the visual client itself (see Section 3.2.1). Choose *Server Options* → *Connect to...* → *Edit Connections* from the main menu panel.

The following enumeration lists all available configuration settings with a short explanation.

<client> is the top element without any PlaSMA specific attributes.

The *client* element may contain different sub elements as follows.

<server> *optional, multiple*. Each entry defines a server available in the client's *Connect to...* menu. If no server is specified the client automatically generates a default server entry with the **address** *localhost* and **port** *1099*.

name *required*. The displayed name of the server.

address *required*. Declares the DNS name or the IP address of the PlaSMA server.

port *required*. Specified the port the PlaSMA server is listening on.

index *required*. An index used for internal ordering of all available server entries. The lower the number the higher up the entry will appear in the menu.

<visualization> *optional, once*. Toggles the visualization of the scenario. The visualization will be enabled by default if this setting is omitted.

enabled *required*. If set to *false* the visualization of scenarios will be disabled.

<displaymode> *optional, once*. Chooses the display mode. Currently two settings are supported: *use3DGlobe* and *usePlanar* which result in a 3D or 2D representation, respectively. If omitted the default value *use3DGlobe* will be used.

mode *required*. Sets the display mode. Available parameters are *use3DGlobe* and *usePlanar*.

<max_zoom_on_startup> *optional, once*. Specifies the maximum camera elevation when starting a scenario to avoid the camera moving too close to the ground, especially in scenarios with only one node. The default value is *50000*.

<localization> *optional, once*. Changes the localization of the client.

language *required*. Sets the language, specified by a language code, followed by a hyphen and a country code[12]. For example *en-US* or *de-DE*. The default language if this parameter is omitted is English (*en-US*).

[12]Conform to RFC 3066.

3
Running PlaSMA

This chapter is concerned with the general usage of the PlaSMA multiagent-based simulation environment in both local and physically distributed settings. Special aspects like scenario implementation and evaluation are addressed in subsequent chapters.

The general usage of PlaSMA in this chapter covers the execution of the PlaSMA server, potential remote controllers, as well as PlaSMA's command line or GUI clients. All start scripts are located directly in `<plasma_dir>`.

3.1 PlaSMA Simulation Server

The start of a PlaSMA server (depending on the host operation system) is done with the following command:

Linux: `server.master.sh [<password>] [-jade [<jade-args...>]]`

Windows: `server.master.bat [<password>] [-jade [<jade-args...>]]`

The specification of the database password is optional. You can also specify the password in the XML server configuration (`<plasma_dir>/etc/serverconfig.xml`). If the password is given as command line argument it also overrides the XML configuration. It is required that the PlaSMA server and the database run on the same machine.

PlaSMA uses JADE as agent environment. You can add JADE arguments by using the `-jade` option. *All* following arguments are passed to JADE.

3.1.1 Server States

A running PlaSMA server is in either of one of the following states:

- **Blank**: The state right after start. No scenario was loaded yet.
- **Loading scenario**: The server is loading a simulation configuartion file.
- **Scenario loaded**: The simulation has loaded a scenario configuration file.

- **Initializing**: The server is initializing a simulation scenario by creating the initial world model and setting up all participating agents.
- **Scenario ready**: Scenario initialization is done. The simulation is ready to be started.
- **Running**: The server is currently simulating.
- **Paused**: The current simulation run is paused and can be resumed.
- **Run Stopped**: The last simulation run has finished. You can either trigger a new run of the same scenario or load another scenario.
- **Error**: A critical error occurred on the server and the simulation cannot proceed. The server needs to be restarted.

The current server state is, e. g., displayed in the PlaSMA GUI client. But usually some of the intermediate states will not be noticed because they go by so quickly. These states can include, e.g., the **'Scenario loaded'** and **'Initializing'** state.

3.1.2 Parallel and Distributed Simulation

PlaSMA simulations can be parallelized with multiple processors and computers. To benefit from a multi processor (or multi-core processor) machine, the PlaSMA server will automatically make use of the processors available in its host system. Therefore, if you only use one machine there is nothing else to be done.

But in addition, PlaSMA is also capable of distributed simulation, i. e., simulation on multiple host computers in parallel. For such cases, additional settings need to be taken care of. The following subsections describe these settings as well as the execution procedure to be applied.

Configuration

PostgreSQL needs to be configured to accept incoming connections from all host machines which participate in the distributed simulation. Therefore, you have to allow those connections in the firewall (the required port is set in the `postgresql.conf`[1], the default value is `port = 5432`) and add the hosts to the `pg_hba.conf`[1]. Assuming that all hosts are located in the subnet *192.168.100.0*, the database user is named *plasma* and you are using the database *plasma*, you have to add the following lines at the end of the file (the order is type, database, user, address and authentification method):

```
# Without password authentication
host    plasma     plasma      192.168.100.0/8     trust
# Same With password authentication
# host    plasma     plasma      192.168.100.0/8     md5
```

Furthermore you have to configure the name resolution of all simulation hosts so that all host names (including the own host name and *localhost*) can be looked up. On Linux systems this is done in the file `/etc/hosts`. On windows systems this file is called `C:\Windows\system32\drivers\etc\hosts`.

On the host *simulationHost_01*, the entries may look as follows:

```
192.168.100.210 simulationHost_01 localhost
192.168.100.211 simulationHost_02
192.168.100.212 simulationHost_03
```

The configuration of the other hosts is analogous, but with *localhost* at the corresponding line.

In addition, it has to be assured that no firewall is blocking the ports 1098 and above. Otherwise the RMI connection of the JADE framework will be blocked.

Accordingly, you have to replace localhost with the host name of the database server in the file `<plasma_dir>/etc/serverconfig.xml`.

[1] The default location is `C:\Program Files\PostgreSQL\9.1\data\` for Windows or `/etc/postgresql/9.1/main/` for Linux.

Execution

For the simulation, the same version of the PlaSMA scenarios (if used) must be present on each simulation host[2]. There are two commands available for execution: `server.master.sh` and `server.slave.sh`. The first one starts the agent platform with the single top-level controller (cf. Section 3.1). The top-controller must be started first.

The second command has to be executed once on each additional simulation host. The IP address of the computer that executes the top-level controller, or its name (as listed in `/etc/hosts`), have to be set as the `server.master.IP` in the `<plasma_dir>/etc/plasma.properties` file. The script starts a remote JADE container and a PlaSMA sub-controller which registers itself with the top-level controller.

Linux: `prompt$ server.slave.sh`

Windows: `prompt$ server.slave.bat`

When all sub-controller have been registered with the top-level controller, the system is prepared for the execution of a distributed simulation.

The maximum number of agents which should work on the respective simulation computer is specified in the file `<plasma_dir>/etc/plasma.properties` in key `server.maxAgents`. The default value is 1000 and based on experience. For other agent implementations it may not fit as well. In the end, it only matters that `server.maxAgents` properly reflects the difference of computational power and working memory for all computers participating in a distributed simulation.

The top-level controller manages the distribution of the agents among all hosts participating in the simulation: A new agent will be started at the computer with the smallest ratio from currently running agents to maximal allowed agents. This will result in a suitable distribution of the workload on all computers and therefore a maximal performance of the whole simulation. The agents stay always on the computer they were started on. In case of intensive agent communication and very different agents this may lead to a suboptimal workload distribution.

3.2 PlaSMA Clients

3.2.1 Visual Client

The control, visualisation and visual feedback, respectively, is done by the PlaSMA client GUI (*Aurora*). It is started with the following command:

Linux: `aurora.sh`

Windows: `aurora.bat`

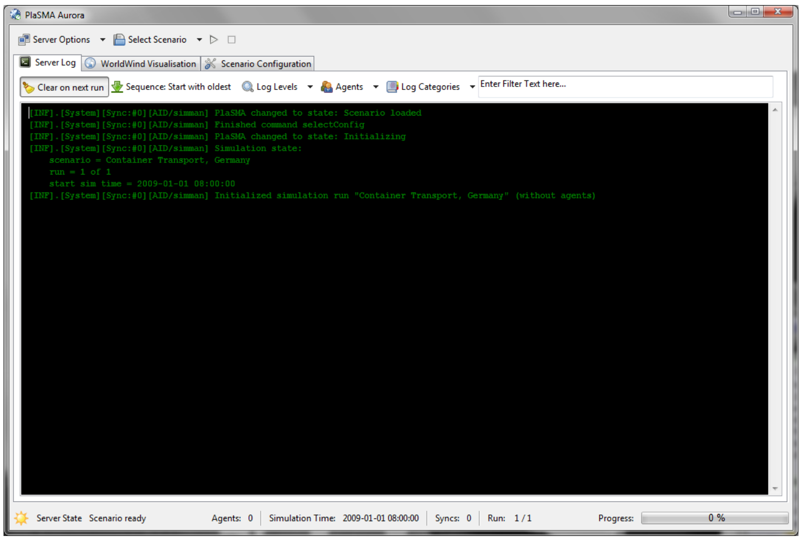
To connect the client with the server choose *Server-Options* → *Connect to...* in the window menu. From startup, only the local server is available. You may create new entries for any server you wish to connect to by selecting *Server-Options* → *Connect to...* → *Edit Connections*. When stating a host name for the PlaSMA server (instead of an IP), it is necessary that the client can resolve that host name, e. g. by adding it to the `/etc/hosts` (Linux) or `C:/Windows/System32/drivers/etc/hosts` (Windows) file.

After a successful login to the simulation server, you can select a scenario from the dropdown menu *Select Scenario*. The menu contains all valid scenario configurations in your scenario folders (usually `<plasma_dir>/scenarios` unless specified otherwise, see Section 2.2).

[2]If the scenario configuration got changed through Aurora (see Section 3.2.1) that changes are only present on the master server and the configuration needs to be redistributed manually to the other hosts.

Depending on the size of the scenario (number of agents, size of the traffic infrastructure), the loading of the scenario may take a few seconds or even longer. After the scenario has been loaded, the simulation can be started, paused and stopped with the *Play/Pause* and *Stop* button in the main toolbar.

Once the scenario is loaded, you can look at it from 3 different perspectives. The *Server Log* (see Figure 3.1) shows recent messages from agents and the simulation system itself. In this view, you may use the menus *Log Levels*, *Agents* and *Log Categories* to filter which types of messages you want to be displayed. The filter text field is another convenient way to look for specific messages. If *Clear on next run* is selected, the Log will be cleaned up at the beginning of each run for the sake of clarity. The *Sequence* button can be used to toggle if the messages should be shown in ascending or descending order.



```
[INF].[System][Sync:#0][AID/simman] PlaSMA changed to state: Scenario loaded
[INF].[System][Sync:#0][AID/simman] Finished command selectConfig
[INF].[System][Sync:#0][AID/simman] PlaSMA changed to state: Initializing
[INF].[System][Sync:#0][AID/simman] Simulation state:
    scenario = Container Transport, Germany
    run = 1 of 1
    start sim time = 2009-01-01 08:00:00
[INF].[System][Sync:#0][AID/simman] Initialized simulation run "Container Transport, Germany" (without agents)
```

Server State Scenario ready | Agents: 0 | Simulation Time: 2009-01-01 08:00:00 | Syncs: 0 | Run: 1 / 1 | Progress: 0 %

Figure 3.1: The Log Tab shows the Server Log.

WorldWind Visualization (see Figure 3.2) is based on *NASA World Wind*, an OpenGL-based 3D engine for geographical visualizations. [3] Thus, scenarios are expected to be located somewhere on the Earth surface. The menu items *WorldWind Layers* and *PlaSMA Layers* allows you to select various layers displayed on that surface. The first category includes layers for, e. g., satellite images, road maps, place names, and country borders. The real-world data for these layers is retrieved from the Internet, therefore you need an Internet connection. Anyhow, *World Wind* will cache loaded images that will be displayed even without Internet connectivity. The latter category provides visualization of physical objects and the traffic infrastructure in the respective scenario. Note: Besides these default scenario layers, available visualizations will depend on the configuration of the scenario (see Section 4.2).

The setup of your scenario can be viewed and edited in the *Scenario Configuration* tab. It contains a wide variety of options from modifying basic scenario parameters (see Figure 3.3) to add or remove scenario agents [4]. You may change the displayed settings according to your simulation preference and purpose (e. g., number of runs and length of each run). All modifications will be applied automatically as soon as you start the scenario run. You may save the modified configuration for later usage by using the buttons on top of the tab.

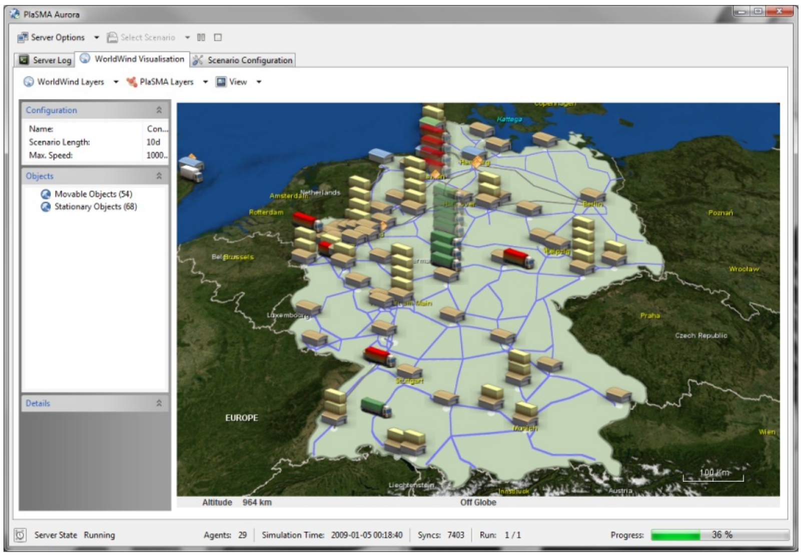
Amongst other things, the status bar at the bottom of the window will show the state of the server (if connected), progress of the simulation time (see Section 5.6) and the number of running agents.

3.2.2 Command Line Interface

There is a command line interface (abbr. *cli*) without any visualization, too. It is started with the following command:

[3]Note that your graphics adapter needs to be set to 32 bit colour depth for proper visualisation. If you encounter additional problems you should update the driver of the graphics adapter.

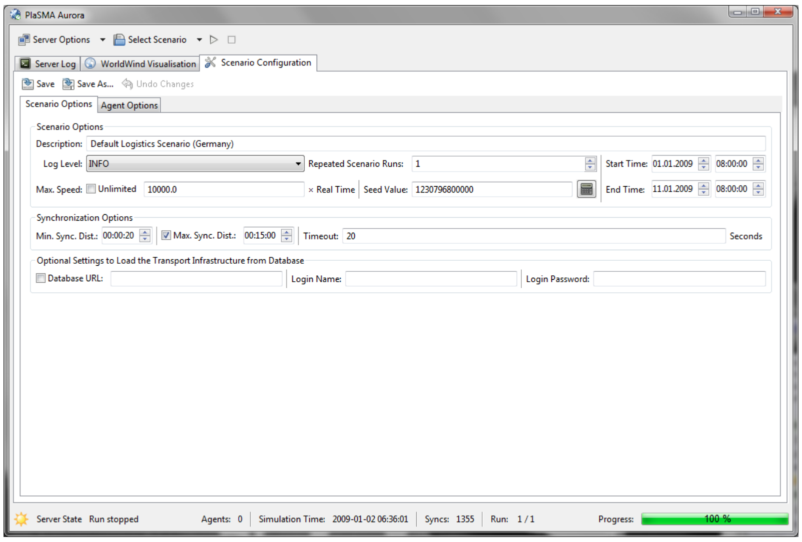
[4]See Chapter 4 (see Figure 3.4) for a full description of all available scenario options.



Name: Con...
Scenario Length: 10d
Max. Speed: 1000...

Server State Running   Agents: 29   Simulation Time: 2009-01-05 00:18:40   Syncs: 7403   Run: 1 / 1   Progress: 36 %

Figure 3.2: The WorldWind Visualization Tab shows the infrastructure and all physical objects.



Scenario Options
Description: Default Logistics Scenario (Germany)
Log Level: INFO   Repeated Scenario Runs: 1   Start Time: 01.01.2009 08:00:00
Max. Speed: ☐ Unlimited 10000.0 × Real Time   Seed Value: 1230796800000   End Time: 11.01.2009 08:00:00

Synchronization Options
Min. Sync. Dist.: 00:00:20   ☑ Max. Sync. Dist.: 00:15:00   Timeout: 20 Seconds

Optional Settings to Load the Transport Infrastructure from Database
☐ Database URL: Login Name: Login Password:

Server State Run stopped   Agents: 0   Simulation Time: 2009-01-02 06:36:01   Syncs: 1355   Run: 1 / 1   Progress: 100 %

Figure 3.3: The Config Tab supports editing the scenario options.

Linux:

```
prompt>./client.cli.sh [−{}−server <server>] [−{}−port <port>] <scenario>
```

```
prompt>client.cli.bat [−{}−server <server>] [−{}−port <port>] <scenario>}
```

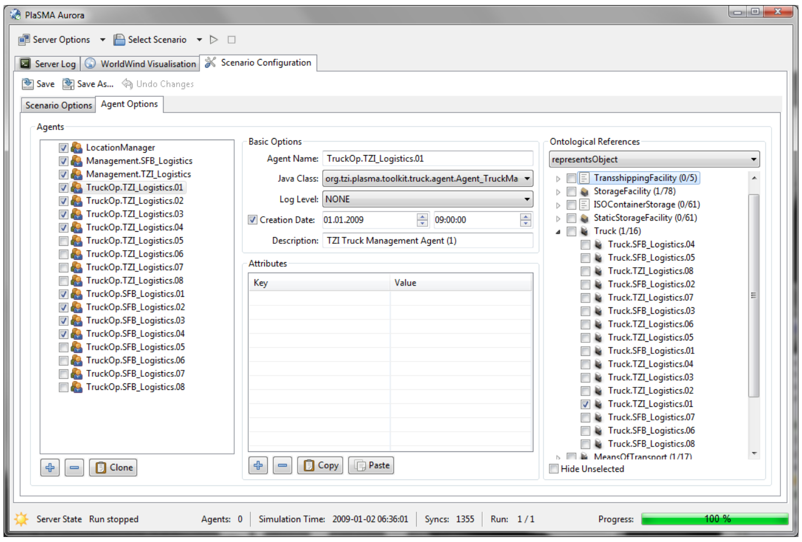
The startup parameters are as follows:

Windows: ***<server>*** *optional*. The address of the server the client should connect to. For local simulation this parameter can be skipped, since *localhost* is the default value.

<port> *optional*. The port the server listens on. The default value is *1099*.

<scenario> The name of the scenario to load. This is the attribute *name* from the scenario configuration.

If this name contains any whitespace put it in quotes. Using *?* as name will list all available scenarios.



Agents
- [✓] LocationManager
- [✓] Management.SFB_Logistics
- [✓] Management.TZI_Logistics
- [✓] TruckOp.TZI_Logistics.01
- [✓] TruckOp.TZI_Logistics.02
- [✓] TruckOp.TZI_Logistics.03
- [✓] TruckOp.TZI_Logistics.04
- [ ] TruckOp.TZI_Logistics.05
- [ ] TruckOp.TZI_Logistics.06
- [ ] TruckOp.TZI_Logistics.07
- [ ] TruckOp.TZI_Logistics.08
- [✓] TruckOp.SFB_Logistics.01
- [✓] TruckOp.SFB_Logistics.02
- [✓] TruckOp.SFB_Logistics.03
- [✓] TruckOp.SFB_Logistics.04
- [ ] TruckOp.SFB_Logistics.05
- [ ] TruckOp.SFB_Logistics.06
- [ ] TruckOp.SFB_Logistics.07
- [ ] TruckOp.SFB_Logistics.08

Basic Options

Agent Name:	TruckOp.TZI_Logistics.01
Java Class:	org.tzi.plasma.toolkit.truck.agent.Agent_TruckMa
Log Level:	NONE
[✓] Creation Date:	01.01.2009 09:00:00
Description:	TZI Truck Management Agent (1)

Attributes

Key	Value

Ontological References

representsObject
- [ ] TransshippingFacility (0/5)
- [ ] StorageFacility (1/78)
- [ ] ISOContainerStorage (0/61)
- [ ] StaticStorageFacility (0/61)
- [ ] Truck (1/16)
  - [ ] Truck.SFB_Logistics.04
  - [ ] Truck.SFB_Logistics.05
  - [ ] Truck.TZI_Logistics.08
  - [ ] Truck.SFB_Logistics.02
  - [ ] Truck.TZI_Logistics.07
  - [ ] Truck.SFB_Logistics.03
  - [ ] Truck.TZI_Logistics.06
  - [ ] Truck.TZI_Logistics.05
  - [ ] Truck.SFB_Logistics.01
  - [ ] Truck.TZI_Logistics.04
  - [ ] Truck.TZI_Logistics.03
  - [ ] Truck.TZI_Logistics.02
  - [✓] Truck.TZI_Logistics.01
  - [ ] Truck.SFB_Logistics.07
  - [ ] Truck.SFB_Logistics.06
  - [ ] Truck.SFB_Logistics.08
- [ ] MeansOfTransport (1/17)

Server State Run stopped | Agents: 0 | Simulation Time: 2009-01-02 06:36:01 | Syncs: 1355 | Run: 1 / 1 | Progress: 100 %

Figure 3.4: The Config Tab supports editing the agent options.

<batchrun> *optional*. This parameter enables an alternative batch run mode. The *scenario* parameter now referes to the parent directory of the scenario configuration, e.g., *scenario_default*. All configurations found in its `config` subdirectory will be part of the batch run.

Make sure to not exit the client until all requested scenario runs have finished, since the client will queue any additionally scenarios configurations itself.

<shutdown> *optional*. Specifies if the server should be shut down after all simulation runs are finished.

3.3 Error Handling

This section will cover errors in the programming of agents. The main aspect of this section is the behaviour of the system if an *exception* occurs. The system will automatically catch all exception and pass them to the user. After that, the user can analyze and resolve the error. Details are following. Logical errors, which would not have any consequences for the simulation system, will not be detected automatically.

If an agent of the simulation core causes an error or if an unpredicted error occurs in the scenario, this will be reported and processed at a central location. In each case this will lead to an abort of the simulation and the error will be forwarded to the graphical user interface which will show a window with the cause of the error.

Depending on the configuration of the simulation core (see Section 4.3), an error case will trigger a *backtrace* on each connected simulation computer. Detailed protocol messages of the agents promptly before the occurrence of the error will be logged. They will appear, depending on the configuration, either in the database or on the terminal on which the simulation was started. This mechanism is based on a huge amount of protocol messages which are buffered by the system although they would be actually discarded by the current log level. This is required, because most errors of the simulation are caused by concurrency. That means that they would appear infrequently or never, if the simulation is executed with a debugger or at another log level, because of the slow down of the execution. The buffering of the detailed protocol messages does not lead to any performance losses but makes the search for errors much easier.

4 Scenario Configuration

Simulations runs, their visualizations, and the logging of data of the scenario activities are configured in a scenario specific XML file. The format of this file is specified by the XSD scheme `config.xsd` in the directory `<plasma_dir>/etc`. For execution, all scenarios have to be deployed to a subdirectory in `<plasma_dir>/scenarios` in whose subdirectory `config` the XML configuration file is saved.

Besides manually editing the configuration file with a text editor the scenario configuration can also be edited with the visual client (see Section 3.2.1). If the desired scenario was selected from the *Select Scenario* menu, the configuration can be modified in the *Scenario Configuration* tab. A scenario template without any agents can be found in the `<plasma_dir>/scenarios/scenario_template/config` directory and should be available in the *Select Scenario* menu as *PlaSMA Scenario Template*.

The following describes the creation and modification of a scenario configuration XML file.

4.1 General Simulation Settings

<simulation> *required, once.* Is the top element of simulation run description and contains the following attributes:

name *required.* Defines the name of the simulation scenario. This name is shown during scenario selection in the client program.

repeats *optional.* Gives the number of iterations of the simulation run. If this attribute is missing, 1 is used as default value. This means that the simulation runs only once. While starting more then one simulation run the simulation time progress continuously in subsequent runs.

maxSimLength *required.* Specifies the simulation time after which the simulation should be stopped.

The input format of **maxSimLength** consists of a number and a time unit. The time unit can be one of the following:

- ms – milliseconds
- s – seconds
- sec – seconds
- min – minutes
- h – hours
- d – days

- y – years

If no unit is specified the number is interpreted as milliseconds. Another time unit, which is not listed above, leads to an load error when loading the scenario.

simStartTime *optional*. Specifies the simulation time at which the simulation scenario starts. This attribute can be specified either as UNIX time stamp, thus as milliseconds since January 1st 1970, or as a date in the *YYYY-MM-DD HH:mm:ss* format. If this attribute is missing, the current system time is used as starting time for the scenario.

maxSpeed *optional*. This attribute should be set if the scenario is visualized, too. This attribute specifies the maximal rate of simulation time to computation time. A value of 1000 means that the simulation time elapses not faster than 1000 times real time. A value of 1 would mean, that it will pass exactly as fast as the real time, accordingly. Values less than 1 but greater than 0 are allowed, too, which would corresponds to slow motion. Values less or equals to 0 will deactivate this attribute and the simulation time elapses as fast as possible. The default value is -1.

The speed limit is realized by a delay of the next synchronisation step. The actual simulation speed might differ depending on machine power and scenario specifics.

seed *optional*. The random seed for the scenario. This value can be used to initialize random number generators (PRNGs) for the scenario to achieve a deterministic behavior of the simulation. The value is given as long integer. If this attribute is missing, a fixed seed is used. The seed can be read out by the agent with `SimSystem.getRandomSeed()`. The usage of this random seed is very important to the reproducibility of scenarios as well as to the reproducibility of results (see Section 5.5.11).

<description> *optional, once*. A description of the scenario to be displayed in the client.

4.1.1 Synchronisation Configuration

<sync> *required, once*. This subelement defines the time control and synchronization of the simulation (see Section 5.6 for further explanations). It contains the following attributes:

minSyncDist *required*. Specifies the minimal progress of simulation time in a synchronization step. This allows for grouping multiple agent's execution in the same synchronization cycle. Otherwise it is most likely that each agent requests its own synchronization step with all other agents being idle. E.g., having two agents requesting a synchronization in 5 and 10 seconds, respectively. With a **minSyncDist** value of 1 second, two cycles are completed with only one agent being active. If set to 20 seconds, only one cycle with both agents running their tasks is executed.

A reduction of the minimal progress can increase the precision of the time laps compared to the real scenario. Contrariwise, this will decrease run time performance because synchronization overhead increases.

This attribute uses the same format as the **maxSimLength** attribute (see Section 4.1).

maxSyncDist *optional*. Specifies the maximum simulation time that might elapse between two synchronization cycles. If this limit is exceeded a synchronization step is forced automatically.

A fixation of the maximum simulation time progress causes *all* agents to be waked up in the given interval. That can be helpful to guarantee a certain granularity for monitoring and analysis of the scenario. This is especially meaningful if the scenario is visualized. Furthermore, an *incorrect* programming of the agents can cause that agents are waked up nevermore if, unintentionally, they receive no further messages[1]. An ensured wake-up using **maxSyncDist** can resolve this problem but is not advisable in general.

This attribute uses the same format as the **maxSimLength** attribute (see Section 4.1).

timeout *optional*. Sets the agent response timeout (in seconds) for synchronization cycles. The default value is 5. If an agent computes longer than this time the PlaSMA controller will

[1] If an agent is waked up nevermore, it has finished its last processing cycle passively. That is, it has not requested an explicit wake-up but is waiting for incoming messages of other agents.

log a warning message. The GUI client will show a corresponding warning and prompt to proceed or to stop simulation. A timeout event might indicate that an agent got stuck.

This attribute uses the same format as the **maxSimLength** attribute (see Section 4.1).

4.2 Domain Configuration

A simulation scenario domain is primarily defined by the agents to be executed. These simulation agents are parallel logical simulation processes. Each simulation agent may represent a set of physical objects or may provide some service not directly related to a specific physical object. This section describes how to configure the agents to be executed and optional settings for the visualization of physical objects they handle.

<domain> *required, once.* The embracing XML tag for the scenario domain configuration. The following sections describe its subelements.

4.2.1 Scenario Ontology Specification

<scenario> *required, once.* Defines the scenario. The content of the XML tag is the scenario OWL ontology URI (*Uniform Resource Identifier*). This ontology URI specifies the scenario's traffic network as well as all simulated objects and agents with their relations and attributes. The ontology is denoted by an *logical* URI. PlaSMA resolves this URI to a physical URL. In general, the scenario will be located in the `<scenario>/owl` sub-directory or in the general PlaSMA ontology directory (`etc/owl`). For instance, the URL `./owl/scenario.owl` may be resolved to the file `<scenario>/owl/scenario.owl`. An absolute Internet URI like `http://plasma.informatik.uni-bremen.de/owl/scenario.owl` may be resolved to the same file if present at `<scenario>/owl/scenario.owl`.

Here is an example for the scenario ontology specification:

```
<scenario>./owl/germany_north.owl</scenario>
```

See Section 5.4 for more information about creating a scenario ontology.

4.2.2 Simulation Agents

When a simulation scenario is loaded in PlaSMA, all agents and objects (e. g., means of transport, shipping containers, or machine tools) with their relations and properties are retrieved from the ontology (cf. Section 4.2.1). Some of these ontology instances might be represented by simulation agent in PlaSMA.

<mappings> *required, once.* Contains the mapping of agent descriptions, i.e., their attributes, to the corresponding Java class.

<agent> *optional, multiple.* Describes the simulation agent and maps it to its Java implementation. This description includes various simulation related attributes as well as custom attributes accessible by the agent at runtime.

name *required.* A unique name identifying this simulation agent.

class *required.* Denotes the implementation of the simulation agent by its fully qualified Java class name. The stated Java class has to extend the PlaSMA Java class *SimulationAgent*.

description *optional.* Specifies a short note about the agent in addition to its name.

enabled *optional.* Determines whether or not the agent should be active in the scenario. The default value is *true*. If set to *false* the agent will be ignored.

creationDate *optional.* Specifies the simulation time at which the agent should join the simulation. This attribute uses the same format as the **simStartTime** attribute (see Section 4.1). If omitted or set to a value before **simStartTime**, the agent will be created at the start of the scenario run.
Delayed agent creation is helpful, if an agent is not supposed to be active from the beginning, but to be created at a specific timestamp during a simulation run. Note that these agents are not created dynamically (i. e., by other agents to react to a specific event[2]) but are created at a predefined simulation date.

loglevel *optional.* Specifies an agent-specific log level for each agent overriding the default scenario log level (see Section 4.3). Available values are *NONE*, *ERROR*, *WARNING*, *INFO*, *DEBUG* and *TRACE*.

<attribute> *optional, multiple.* Specifies an agent property. These attributes might be useful to configure simulation agents if not possible or adequate using ontology-specified attributes. Note that the attributes are only forwarded into the agent setup as JADE user arguments and may be interpreted therein. They do not correspond to any of the simulation agent's Java class fields.
Attributes might be used to link a simulation agent to one or more specific ontology instances, e.g., by using some kind of `operatesFor` or `representsObject` property with an ontology ID as value.

key *required.* The unique key of the attribute.

value *required.* The value of the attribute in string representation.

description *optional.* A short description of this attribute.

<attributeset> *optional, multiple.* Extends *<attribute>* to support multiple values per key.

key *required.* The unique key of the attribute.

description *optional.* A short description of this attribute.

<entry> *required, multiple.* A list of one or more values associated with this key.

value *required.* One value of the attribute in string representation.

The following listing shows an example agent mapping configuration.

```
<mappings>
  <agent class="org.tzi.plasma.toolkit.forwardingagency.agent.Agent_ForwardingAgency" description="TZI Logistics
      Forwarding Agency" enabled="true" name="Management.TZI_Logistics">
    <attributeset key="available_locations">
      <entry value="http://plasma.informatik.uni−bremen.de/owl/shared_infrastruture.owl#Storage.
          ACME_Air_Assembly"/>
      <entry value="http://plasma.informatik.uni−bremen.de/owl/shared_infrastruture.owl#Storage.Bremen"/>
    </attributeset>
  </agent>
  <agent class="org.tzi.plasma.toolkit.truck.agent.Agent_TruckManagement" creationDate="1230800400000"
      description="TZI Truck Management Agent (3)" enabled="true" name="TruckOp.TZI_Logistics.03">
    <attribute key="operatesFor" value="http://plasma.informatik.uni−bremen.de/owl/default_northrange.owl#
        TZI_Logistics"/>
    <attribute key="representsObject" value="http://plasma.informatik.uni−bremen.de/owl/default_northrange.owl#
        Truck.TZI_Logistics.03"/>
  </agent>
</mappings>
```

4.2.3 World Model

<worldmodel> *optional, once.* Provides some world model related settings. The creation of the world model itself is explained in Section 5.4.

ontology *optional.* Determines if run-time changes of the world model, e. g., adding objects, should be fully backed up by the ontology (*true*) or if a slightly simpler data structure should be used (*false*). The default value is *false* which is also recommended for performance reasons.

[2]For this effect see Section 5.5

<database> *optional, once*. Provides informations to connect to a database representing the transport infrastructure.

This is only required if the infrastructure will be imported from that database (see Section 5.4.4) instead of provided directly through the ontology (see Section 5.4.1).

url *required*. The URL to the database. This might require a `jdbc:postresql` scheme, e. g., *jdbc:postgresql://localhost:5432/postgis*.

user *required*. The database user.

password *required*. The password of that user.

enabled *optional*. Setting this attribute to *false* disables the usage of the database infrastructure import without removing the whole XML tag. The default value is *true*.

<filterobject> *optional, multiple*. States that an ontology ID should be explicitly taken account of. Usually only the most specific (and some predefined) ontology concepts are considered by the world model while all other concepts are filtered out. This might be useful if a specific concept should be displayed by the visual client.

4.2.4 Online Scenario Visualization

The PlaSMA GUI (*Aurora*) can handle scenario-specific visualization settings. These settings include, e. g., special images for certain objects or object types as well as 3D scenery properties such as an initial camera position. The PlaSMA GUI uses the *NASA World Wind* framework[3] for its 3D planetary visualization[4].

<visualization> *optional, once*. The embracing XML tag for all visualization settings.

enabled *optional*. Allows to deactivate visualization temporarily without the need to remove the XML settings.

<images> *optional, once*. Denotes an XML sequence of generic image mappings.

<image> *optional, multiple*. A single image mapping.

key *required*. The unique image identifier. The key may, e. g., correspond to the ontology ID (local or with namespace) of any object or the ontology ID (with namespace) of an object type.

src *required*. The URI of the image file. A relative URI will be resolved relative to the scenario directory or, if not found there, relative to PlaSMA resource directory `etc`.

<worldwind> *optional, once*. Configures the visualization.

mode *optional*. Switches between a 2D projection (*planar*) or a 3D spheroid visualization (*sphere*). *sphere* is the default setting and strongly recommended.

<initposition> *optional, once*. Determines the view point of the camera when the scenario is loaded. If omitted NASA World Wind will set the initial position itself based on the dimensions of the scenario infrastructure graph.

latitude *required*. The latitude of the initial view point.

longitude *required*. The longitude of the initial view point.

altitude *optional*. The altitude of the initial view point in meters. If omitted the default value of *500000* is used.

<smoothiconmovement> *optional, once*. Smoothes the movement of icons. Depending on the **maxDuration** and the **maxSpeed** (see Section 4.1) attributes the smoothed movement might look odd. Therefore if omitted the smooth icon movement is disabled (default).

moveSmoothly *required*. Toggles the smooth icon movement.

[3] `http://worldwind.arc.nasa.gov`

[4] Aurora does not support the 2D visualization of previous versions. The sole visualization engine is NASA World Wind now. However, it is possible to set World Wind to a 2D projection mode of earth (see below).

maxDuration *required*. Sets the duration it will take for an icon to smoothly move to the desired position. This value should be selected in regards to the **maxSpeed** attribute.

<background> *optional, multiple*. Places an image above of the world wind background.

src *required*. A relative path from the scenario base directory to a local background picture. Transparency of PNG files is supported.

north *required*. The top latitude of the background.

south *required*. The bottom latitude of the background.

west *required*. The left longitude of the background.

east *required*. The right longitude of the background.

To summarize, a visualization configuration might look like the following example.

```
<visualization>
  <images>
    <image key="agent1" src="images/agent1.png"/>
    <image key="http://plasma.informatik.uni-bremen.de/owl/test.owl#Truck" src="images/Truck.png"/>
  </images>
  <worldwind mode="sphere">
    <initposition latitude="53.5454" longitude="9.8064" altitude="300000"/>
    <background north="55.059" west="5.861" south="47.264" east="15.046" src="images/germany.png"/>
  </worldwind>
</visualization>
```

The image mapping includes an object instance image for *agent1* and a general object type image for trucks. The initial camera position is focused to Bremen in Germany from an altitude of 300 kilometers. Also a background image with the area of Germany was added.

4.3 Simulation Logging

The logging component of PlaSMA provides the ability to write messages to the database or the console to retrace and analyze a simulation run.

4.3.1 Log Levels

A log level defines the type and level of detail of each messages (e. g., errors get the level **ERROR**). Depending on logging detail configuration a generated message will be actually logged or ignored. The log level declared in the scenario's XML configuration is the determination base. A message with the same or a log level above the specified level will be logged. Otherwise it is discarded. The lowest log level is the level **TRACE**, whose messages are only logged if the log level **TRACE** is specified. For instance, if the log level **WARNING** is specified, only messages with the log level **WARNING** and **ERROR** are logged.

Here is an overview of the log level hierarchy in increasing order of detail:

1. **NONE** can only by specified in the configuration file. With this level all messages are discarded.
2. **ERROR** is the highest log level for messages and should be used to log an error that usually causes a simulation run to fail.
3. **WARNING** denotes a warning that does not cause the simulation to fail necessarily.
4. **INFO** Messages that are helpful to keep track of important events in the scenario but not generated too frequent.
5. **DEBUG** for analysis of errors and problems in the programming of agents or their interaction.
6. **TRACE** is the lowest log level and should be used to trace the exact behaviour of an agent. The simulation will be slowed down by a high amount of these messages.

4.3.2 Logging Categories

Categories provide the ability to group messages and filter them to get a clear arrangement. Furthermore, you can define a special log level for each category. This allows you to log messages from a specific category more or less extensive or even not at all.

If a category is specified for a logging message, the log level of the category will decide about the actual logging of the messages and not anymore the overall log level.

The categories are organized hierarchically and are identified by the path. The respective levels are divided by a dot, e. g., `System.Communication.Input`. Thus the categories are organized in a tree structure. This means, that a category inherits the log level of the super class, if it does not specify its own log level. If no log level is specified for a top level category, the global default log level is used.

There exist some predefined categories in the Java class `LogCategory` as static variables, whereas `System` shall be used for system messages only, which is thus reserved for the simulation core. Note that category identifiers are handled case-sensitive!

4.3.3 Agent-specific Logging

It is possible to assign a specific log level to each agent that overrides the default scenario log level. This log level can be set at the respective *<agent>* entry in the XML configuration (see Section 4.2.2).

4.3.4 XML Configuration

<logging> *optional, once.* Sets the scenario's default log level and encloses the specific log category definitions.

- **loglevel** *required.* The scenario's default log level. Available values are *NONE*, *ERROR*, *WARNING*, *INFO*, *DEBUG* and *TRACE*.
- ***<logcategory>*** *optional, multiple.* Specifies a log category.
  - **name** *required.* The name of the log category.
  - **loglevel** *required.* The log level of this category. Available values are *NONE*, *ERROR*, *WARNING*, *INFO*, *DEBUG* and *TRACE*.

The following listing shows an example logging configuration excerpt.

```
<logging loglevel="INFO">
    <logcategory name="Agent" loglevel="NONE" />
    <logcategory name="Agent.Planning" loglevel="DEBUG" />
</logging>
```

5
Scenario Implementation

5.1 Overview of the General Scenario Structure

A PlaSMA scenario consists of three basic parts:

- The **world model** structure is specified by OWL ontology files. Therein one can define which kinds of objects and agents are part of the scenario as well as the network infrastructure they are situated in. It is also possible to import the infrastructure from OpenStreetMap[1], see Section 5.4.4 for details.
- The **simulation agents** are the actual simulation processes that implement the behaviour of physical objects and/or software agents. Simulation agents are implemented as Java classes.
- Simulation-specific **scenario settings** are made in an additional XML file. Essentially, in this file the user can map objects to simulation processes, i. e., *simulation agents*. There are additional settings for simulation time management, visualization, and logging. The settings configuration is explained in Chapter 4.

PlaSMA scenarios coarsely exhibit the content structure as shown in Figure 5.1. Each scenario is self-contained such that it is possible to build only a particular scenario of interest in place as all necessary resources are included to deploy, i. e. install, the scenario for use with the PlaSMA simulation system. Note that using identical (fully qualified) class names in different scenarios might cause problems, since only the first occurrence of that class will be loaded. This applies to external libraries used in scenarios as well.

All scenario source code is placed under a certain source folder (`src` by default), i. e. the build process should not depend upon sources external to the scenario folder. All custom libraries which are required for the compilation and deployment of a particular scenario are to be placed in the library folder (`lib` by default). The `owl` folder contains the ontology-based specification of the simulation world model. This model may contain scenario-specific schema extensions to the general OWL ontologies (tlo, trans, comm, prod, ...) which are provided with the PlaSMA system. These extensions comprise the introduction of new concepts and relations which are not covered by the general ontologies, e. g. a groceries ontology.

The other and most important part of the ontology-based world model is the concrete instance-level configuration of the scenario such as the transport network, production facilities and other physical

[1]See `http://www.openstreetmap.org`.

objects in the world. These extensions are typically bundled in a file named `scenario.owl`. Details about this configuration are provided in Section 5.4.

The `config` folder contains the scenario XML configuration(s). The structure of this configuration file is introduced in Chapter 4. It is possible to pool a set of scenarios which build upon the same overlapping code base into a single scenario folder. In such a case, for each particular scenario a dedicated XML configuration file must be placed in the config folder. Note that it is not possible to configure more than one scenario in a *single* XML file.

The `images` folder contains images for a customized visualization of certain simulation entities such as trucks, containers etc.

As part of the PlaSMA distribution a scenario stub is provided besides functional sample scenarios `<plasma_dir>/scenarios/scenario_template`. When starting the development of a new scenario, it is advisable to copy this folder and start developing by providing your own XML configuration, the ontology-based scenario specification and finally your own Java sources.

5.2 Scenario Build Configuration

Once you have successfully set up a new scenario workbench, i. e. the scenario directory, and started the development of your own scenario code base, there is the requirement for structured compilation, packaging and deployment of your code. For all these steps, the compilation of the Java sources, the compilation of the Java class files in a single *Jar* archive and the installation of this archive together with the remainder of the scenario resources (deployment), we suggest to use the Apache Ant build system. It is possible to use Ant from the command line. For IDE-based development, Ant is also well-integrated in the popular Eclipse IDE. Before describing both styles of Ant usage, there are some simple configuration steps required in order to set up the build system for your new scenario and your particular computer.

When you have started your scenario development with a copy of the scenario template mentioned above, you'll see in the scenario directory the file `build.xml` which provides generic build support. The build file needs to be configured via a standard Java properties file `configuration.properties`. In order to create such a file, just copy the template `configuration.template`. Afterwards you can customize the configuration to suit your system.

For customization, only the section named *user configuration*, needs to be edited. Thereby, the developer first needs to specify the location of the PlaSMA installation (referred to as `<plasma_dir>`). It is important to provide this location correctly such that it is possible to deploy the scenario to its correct destination.

If you develop more than a single scenario for deployment in PlaSMA, you might find it convenient to reuse parts of the classes developed in one base scenario in another dependent scenario. The Ant-based build process accommodates this particular scenario in the following way: If you configure a dependent scenario, you can uncomment the flag *scenario.hasDependencies* in the configuration file. Then during compilation, Ant makes sure the Jar archives of previously deployed PlaSMA scenarios are present in the classpath during compilation.

```
+ scenario_example
  + src
  + lib [optional]
  + owl
  \ + scenario.owl
  + config
  + images [optional]
  \
  + build.xml
  + configuration.properties
```

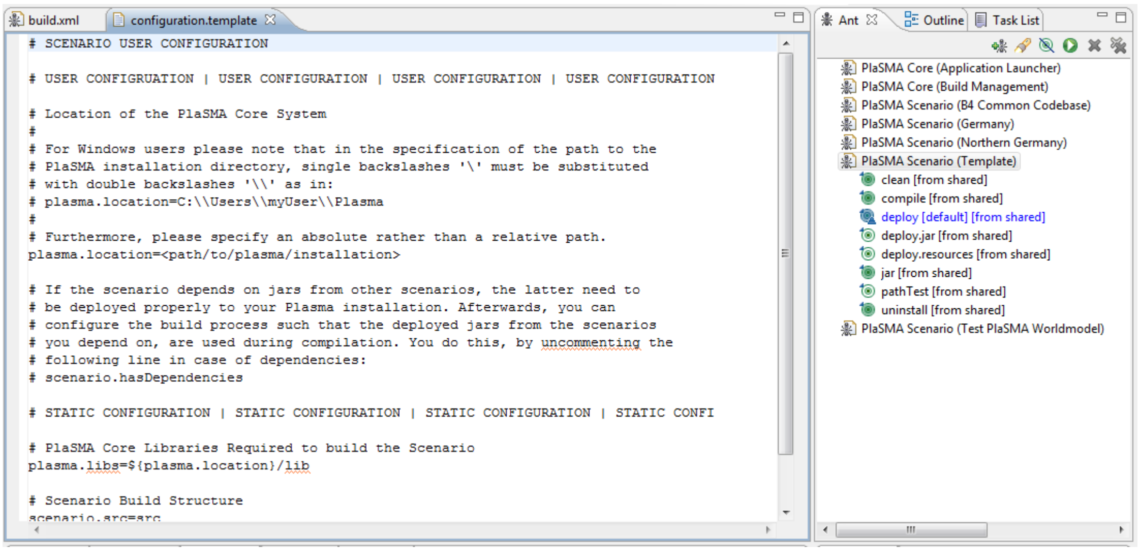
Figure 5.1: Schema of the structure of a typical PlaSMA scenario.

Once the `configuration.properties` file is prepared, the Ant-based build system is ready for use. Note, that if you try to use Ant without the configuration being present, the build process will abort with an error message asking you to provide said file first.

5.3 Scenario Build Process

The scenario build process is based on the Apache Ant tool. A standard `build.xml` file is part of each scenario which supports the following set of commands (called *targets*):

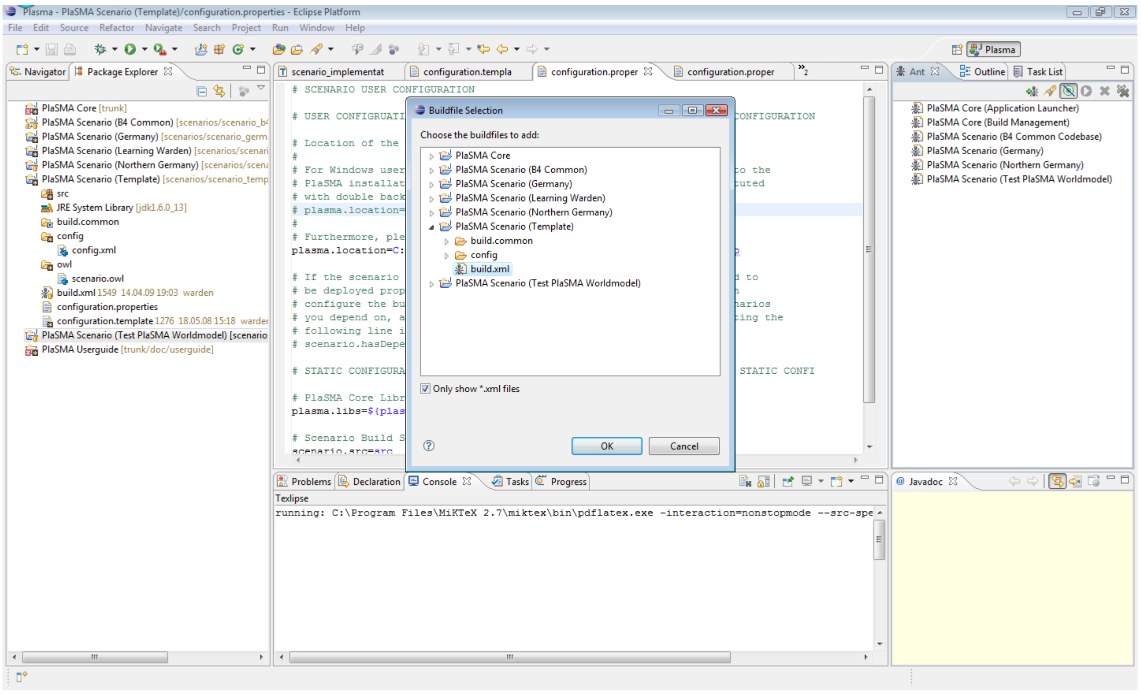
clean Cleans the scenario workbench by deleting all compiled Java class files from the `scenario.build`



```
# SCENARIO USER CONFIGURATION

# USER CONFIGURATION | USER CONFIGURATION | USER CONFIGURATION | USER CONFIGURATION

# Location of the PlaSMA Core System
#
# For Windows users please note that in the specification of the path to the
# PlaSMA installation directory, single backslashes '\' must be substituted
# with double backslashes '\\' as in:
# plasma.location=C:\\Users\\myUser\\Plasma
#
# Furthermore, please specify an absolute rather than a relative path.
plasma.location=<path/to/plasma/installation>

# If the scenario depends on jars from other scenarios, the latter need to
# be deployed properly to your Plasma installation. Afterwards, you can
# configure the build process such that the deployed jars from the scenarios
# you depend on, are used during compilation. You do this, by uncommenting the
# following line in case of dependencies:
# scenario.hasDependencies

# STATIC CONFIGURATION | STATIC CONFIGURATION | STATIC CONFIGURATION | STATIC CONFI

# PlaSMA Core Libraries Required to build the Scenario
plasma.libs=${plasma.location}/lib

# Scenario Build Structure
scenario.src=src
```

```
PlaSMA Core (Application Launcher)
PlaSMA Core (Build Management)
PlaSMA Scenario (B4 Common Codebase)
PlaSMA Scenario (Germany)
PlaSMA Scenario (Northern Germany)
PlaSMA Scenario (Template)
    clean [from shared]
    compile [from shared]
    deploy [default] [from shared]
    deploy.jar [from shared]
    deploy.resources [from shared]
    jar [from shared]
    pathTest [from shared]
    uninstall [from shared]
PlaSMA Scenario (Test PlaSMA Worldmodel)
```

Figure 5.2: Excerpt from the build configuration template file `configuration.template`.



```
PlaSMA Core [trunk]
PlaSMA Scenario (B4 Common) [scenarios/scenario_b4
PlaSMA Scenario (Germany) [scenarios/scenario_germ
PlaSMA Scenario (Learning Warden) [scenarios/scenari
PlaSMA Scenario (Northern Germany) [scenarios/scena
PlaSMA Scenario (Template) [scenarios/scenario_temp
    src
    JRE System Library [jdk1.6.0_13]
    build.common
    config
        config.xml
    owl
        scenario.owl
    build.xml 1549 14.04.09 19:03 warden
    configuration.properties
    configuration.template 1276 18.05.08 15:18 warder
PlaSMA Scenario (Test PlaSMA Worldmodel) [scenario
PlaSMA Userguide [trunk/doc/userguide]
```

Choose the buildfiles to add:

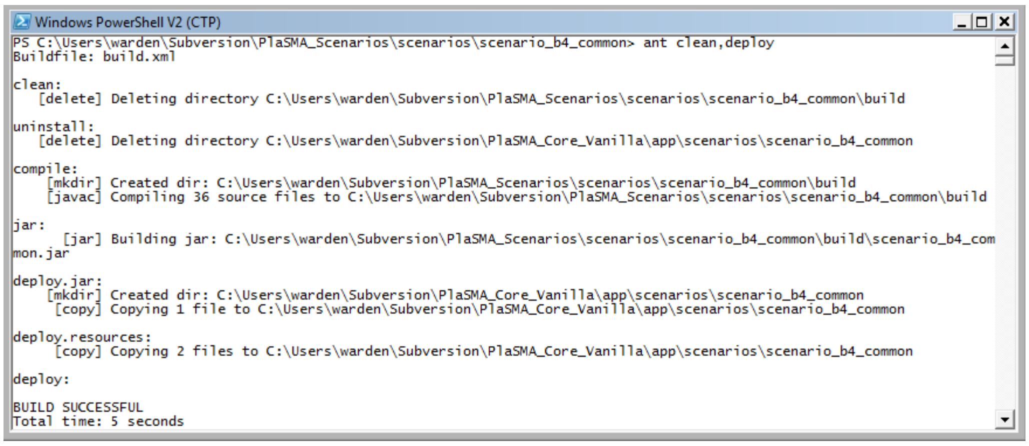
```
PlaSMA Core
PlaSMA Scenario (B4 Common)
PlaSMA Scenario (Germany)
PlaSMA Scenario (Learning Warden)
PlaSMA Scenario (Northern Germany)
PlaSMA Scenario (Template)
    build.common
    config
    build.xml
PlaSMA Scenario (Test PlaSMA Worldmodel)
```

[✓] Only show *.xml files

```
PlaSMA Core (Application Launcher)
PlaSMA Core (Build Management)
PlaSMA Scenario (B4 Common Codebase)
PlaSMA Scenario (Germany)
PlaSMA Scenario (Northern Germany)
PlaSMA Scenario (Test PlaSMA Worldmodel)
```

```
Texlipse
running: C:\Program Files\MiKTeX 2.7\miktex\bin\pdflatex.exe -interaction=nonstopmode --src-spe
```

Figure 5.3: Using the Ant build system for scenarios in the Eclipse IDE.



Windows PowerShell V2 (CTP)

```
PS C:\Users\warden\Subversion\PlaSMA_Scenarios\scenarios\scenario_b4_common> ant clean,deploy
Buildfile: build.xml

clean:
   [delete] Deleting directory C:\Users\warden\Subversion\PlaSMA_Scenarios\scenarios\scenario_b4_common\build

uninstall:
   [delete] Deleting directory C:\Users\warden\Subversion\PlaSMA_Core_Vanilla\app\scenarios\scenario_b4_common

compile:
    [mkdir] Created dir: C:\Users\warden\Subversion\PlaSMA_Scenarios\scenarios\scenario_b4_common\build
    [javac] Compiling 36 source files to C:\Users\warden\Subversion\PlaSMA_Scenarios\scenarios\scenario_b4_common\build

jar:
      [jar] Building jar: C:\Users\warden\Subversion\PlaSMA_Scenarios\scenarios\scenario_b4_common\build\scenario_b4_com
mon.jar

deploy.jar:
    [mkdir] Created dir: C:\Users\warden\Subversion\PlaSMA_Core_Vanilla\app\scenarios\scenario_b4_common
     [copy] Copying 1 file to C:\Users\warden\Subversion\PlaSMA_Core_Vanilla\app\scenarios\scenario_b4_common

deploy.resources:
     [copy] Copying 2 files to C:\Users\warden\Subversion\PlaSMA_Core_Vanilla\app\scenarios\scenario_b4_common

deploy:

BUILD SUCCESSFUL
Total time: 5 seconds
```

Figure 5.4: An example for an Ant build session.

directory.

compile Compilation of scenario's Java source files.

jar Packaging of scenario classes into a Java archive (*jar*).

deploy Deploys the scenario with all its resources to the PlaSMA scenario directory. Thereby, the scenario will be available for execution in PlaSMA.

uninstall Uninstalls a scenario by deleting the corresponding sub-directory in the PlaSMA directory for deployed scenarios.

5.3.1 Using the Ant-Based Build System from Eclipse

In the following, we assume that the scenario developer has already created a new Java project for the new scenario[2]. Thus, when starting from the scenario template, the situation is as shown in Figure 5.3. In order to benefit from the Eclipse Ant integration, first enable the Ant view in Eclipse and arrange according to individual preferences in the active perspective. Then, choose *Add Buildfiles* from the top menu of the Ant view in order to invoke the build file selection dialogue as shown in Figure 5.3.

Select the `build.xml` file provided with the scenario template and confirm the selection. You will then see a new top-level item in the Ant view which represents the build file and upon unfolding reveals the distinct build targets presented in the previous section. The *deploy* target is colored in blue as it represents the default target.

You may find that there are actually two flavors of Ant targets which can be told apart visually via inverted icons (cf. right side of Figure 5.2). These targets with a dark dot icon represent public targets which are used directly by the scenario developer. The remaining targets are internal targets which can be neglected in most cases. There is an option in the Ant view menu to hide these for a better overview.

The Ant targets are invoked via a double click. Log output will be presented in the console view which is in most cases located below the main code editor window.

5.3.2 Using the Ant-Based Build System from the Command Line

The usage of the Ant-based build system from the command line, both under Windows and Linux operating systems, presuppose a correct installation of Apache Ant on your system. Binary distributions

[2]To use code completion and remove the "cannot be resolved to a type" errors of Eclipse, you might want to add `<plasma_dir>/lib/plasma.jar`, `<plasma_dir>/lib/jade.jar`, `<plasma_dir>/lib/commons.jar`, `<plasma_dir>/lib/logger.jar`, `<plasma_dir>/lib/worldmodel.jar` , `<plasma_dir>/lib/toolkit.jar`, and `<plasma_dir>/lib/plasma.jcncl.jar` as external JARs to your project according to your needs. This step is optional, since the Ant build file has a reference to those JAR files itself.

are available from the project website (for Windows users). In most Linux distributions Ant can be installed conveniently via the distributions package management system.

In order to invoke the Ant build targets for your scenario from the command line, switch to the scenario root directory where the `build.xml` file is located. Here you tell Ant to invoke either a single target or a comma-separated target sequence using the following syntax:

```
prompt $ ant target1{,target2,..}$
```

Figure 5.4 shows an example session. First, the scenario workbench is cleaned. Afterward the scenario is re-deployed as a multi-step process. A previous, now outdated version of a given scenario is uninstalled from the PlaSMA system. Afterwards, the scenario sources are compiled, put into a Jar archive and finally re-deployed (i. e. installed) for use in the PlaSMA system.

5.4 World Model Design

A simulation usually consists of an entity (or multiple entities) that represents the (physical) environment in the simulation. In the case of PlaSMA, this environment is provided by a dedicated world model which stores position and state of every physical object. Additionally, the world model provides a graph-based infrastructure where vertices represent locations and edges represent connections between them, such as roads or railways.

The PlaSMA world model is created from an ontology specification using W3C Web Ontology Language (OWL [BvHH+04, PSHH04]) in its OWL-DL sub-language. Ontologies are specified in *.owl* files with XML encoding. While there are several predefined ontologies for object types in logistic scenarios (see Section 5.4), a new scenario needs its own instance declarations stated in a scenario-specific ontology file (see Section 5.4.3). The following subsections provide the technical background of ontologies, their usage for PlaSMA world models as well as how to create own scenario ontologies.

Technical Background

Ontology knowledge bases are mainly separated in two distinct parts: *TBox* and *ABox*. While the TBox defines the schema level, the ABox specifies instances. But this distinction is not necessarily reflected in separate ontology files.

The TBox defines terminological knowledge, i. e., it incorporates concept definitions and property definitions. Concepts (or classes) describe the schema of everything that is some concrete or abstract entity in a particular domain of interest. Properties define relations between concepts (object properties) and attributes of concepts with primitive types such as strings or numbers (datatype properties). For instance, two concepts in logistics are “means of transport" and “cargo". An attribute would be the means of transport’s maximum speed or the cargo’s weight. A relation between means of transport and cargo is, e. g., “contains". That is, means of transport contain objects of type cargo. Thus, an object relation `contains` for the concept means of transport is restricted to concepts of type cargo. In this relation the means of transport is called the property domain while the cargo is the property range. One particular concept relation is the so called `is-a` relation specifying a concept hierarchy of super-classes and sub-classes. A concept C is called *subsumed* by a concept D if (in every possible model) the set of instances denoted by C is a subset of the set of instances denoted by D. In contrast to conventional approaches, the concept hierarchy may be predefined or logically inferred. This inference is called *subsumption*. Furthermore, ontologies allow for multiple inheritance, i.e., a concept may be the sub-concept of multiple super-concepts that need not form a single path in concept hierarchy.

The ABox defines asserted knowledge, i. e., knowledge about instances (also called individuals). This incorporates assertions like some particular object being an instance of a concept (*concept assertion*), or having some property value (*role or property assertion*). While a property in the TBox defines domains and ranges, the ABox defines property values. For instance, an ABox may contain the assertions that *truck1* is an instance of concept *means of transport* and in relation *contains* an instance *cargo1*. Again ontologies allow for logical inferences here. We did not state that *cargo1* is an instance of concept *cargo* but we may infer this because the range of property *contains* for concept *means of transport* is *cargo*. The inference of determining if an individual is an instance of some concept is called *instance checking*. The inference of determining all of a concept's instances is called *retrieval*. *Realization* is the inference of finding the most specific concept(s) of some particular instance.

The above inference tasks for TBoxes and ABoxes are not complete. In particular, one important inference in TBoxes is *satisfiability* of concepts, i. e., checking whether a set denoted by a concept may be non-empty in some model. This helps you detect if your designed world model is logically consistent. For instance, the check will fail if you model a type of airplane that is also asserted or inferred to be an ISO shipping container, because no such things are allowed to exist in the ontology. Such mutual type exclusions are expressed by denoting two concepts (e. g., airplane and ISO container) as *disjoint*.

The Logistics Ontologies

The simulation system provides five OWL ontologies (as TBoxes) for world modeling which are briefly described in the following list.

- **TLO**: The top-level domain ontology for logistic scenarios. TLO specifies general types of physical objects, the basics of traffic infrastructure, organizational membership, ownership of physical objects, and software agents providing abstract services. All other ontologies import this ontology for extension.
- **Trans**: The ontology for transport-specific issues. For instance, this ontology defines several types of nodes and links in the traffic infrastructure as well as a number of different types of vehicles and transport containers.
- **Prod**: Production logistics is handled by this ontology. It specifies machines and production orders with their respective properties.
- **Com**: This communication technology ontology defines communication and computation devices with properties such as radio coverage, power supply type, and computational power (e.g., for mobile agents).
- **Goods**: The goods ontology provides a general schema to classify goods. For instance, classification may differentiate physical or non-physical good (e.g., property rights), packaged good or bulk good, food or non-food, dangerous good or not, phase of matter etc.

The above ontologies are neither considered complete nor mandatory. That means, the system user is free to use or not use one or more of these ontologies for a scenario and may also adapt or extend them for individual needs. There is only one exceptions to this:

- The traffic infrastructure and physical world model is considered to be a directed graph of nodes and edges with certain mandatory properties such as length and mandatory types (cf. Sect. 5.4.1).

The easiest way to keep the system running correctly is not to modify the TLO.

Ontology Entities

Each named resource in an OWL ontology (concept, property, or instance) has a unique resource identifier. In order to guarantee cross-ontology uniqueness resource identifiers have a namespace as prefix which is the URI of their ontology extended by a # character. Ontologies may define shortcuts for namespaces of imported ontologies. By default, the ontology's short name in lower case (e. g., "tlo") is used in PlaSMA. To indicate that a resource belongs to the top-level ontology we write, e. g., *tlo:PhysicalObject* or *tlo:SoftwareAgent*. The scenario ontology namspace has the default abbreviation "scen". These abbreviations are provided as String constants in Java class `OntologyService`.

In PlaSMA, the URI of a resource is represented by the Java class `BasicRI`, i. e., *basic resource identifier*. This class only encapsulates two character strings, namespace and local ID, and is optimized for handling in PlaSMA.

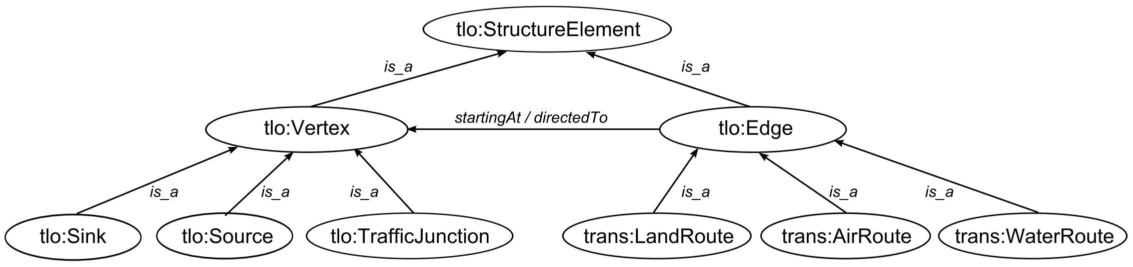


Figure 5.5: Hierarchy of the most important ontology concepts for physical world infrastructure modelling.

5.4.1 Physical World Infrastructure

As stated above the simulation system's physical world model is specified by a directed graph consisting of nodes and edges. Nodes (or vertices) are abstract locations in your world that may correspond to, e. g., traffic junctions as well as logistic sources and/or sinks. In production scenarios a machine may be modeled as a node as well. Edges are directed links between nodes (e. g. roads or railways).

Nodes are at least typed as *tlo:Vertex*. They need to have a location which is stated in geographical coordinates using the datatype properties *tlo:latitude* and *tlo:longitude*. Edges have the basic type *tlo:Edge*. Each edge must have a specific length stated in kilometers (property *tlo:edgeLength*) and exactly one start and end location respectively. These locations are defined by nodes as values for the properties *tlo:startingAt* (start) and *tlo:directedTo* (end). Note that one always has to create two edges to model a link that is trafficable in both directions.

Types of Nodes and Edges

The transportation ontology (*trans*) provides several types of nodes and edges. The basic edge types are *trans:LandRoute*, *trans:WaterRoute* and *trans:AirRoute*. *trans:LandRoute* is further divided in *trans:Road* (for wheeled or tracked vehicles) and *trans:RailRoad* (for trains). A *trans:WaterRoute* may be either a *trans:InlandWaterway* or a *trans:SeaWaterway*.

Sub-types of nodes are, e. g., *trans:TrafficJunction* like a motorway interchange, *tlo:Source* and *tlo:Sink* to denote origin and destination of goods as well as *trans:Seaport* and *trans:Airport* where edges of different types come together.

Figure 5.5 gives an overview on the most important edge types with the corresponding concept hierarchy. Refer to the ontology *trans.owl* for a comprehensive description of all predefined types.

The following extract of a sample scenario ontology shows the corresponding XML syntax of a graph specification:

```
<tlo:Vertex rdf:about="#Bremen">
    <tlo:latitude>53.0759</tlo:latitude>
    <tlo:longitude>8.8073</tlo:longitude>
</tlo:Vertex>
<tlo:Vertex rdf:about="#Hamburg">
    <tlo:latitude>53.5506</tlo:latitude>
    <tlo:longitude>9.9933</tlo:longitude>
</tlo:Vertex>
<tlo:LandRoute rdf:ID="Hamburg_Bremen">
    <tlo:startingAt rdf:resource="#Hamburg"/>
    <tlo:directedTo rdf:resource="#Bremen"/>
    <tlo:edgeLength>120.0</tlo:edgeLength>
</tlo:LandRoute>
<tlo:LandRoute rdf:ID="Bremen_Hamburg">
    <tlo:startingAt rdf:resource="#Bremen"/>
    <tlo:directedTo rdf:resource="#Hamburg"/>
    <tlo:edgeLength>120.0</tlo:edgeLength>
</tlo:LandRoute>
```

Static Objects and Graph Nodes

The PlaSMA ontology considers physical objects *disjoint* from traffic infrastructure as given by elements in the directed graph. Thus, physical objects are *located* at structure elements but are no structure elements themselves. That is, in general a structure element should have no function besides representing locations and connections between them and a static object like a storage facility is located at some node and represents the ability store items there. Exceptions from that rule are possible and up to the designer of the scenario.

5.4.2 World Model and Agents

All autonomously acting entities in a simulation scenario are represented by their corresponding Java class so it is not necessary to model simulation agents at all. However, it is possible to model those agents if that is required for some reason. Although it may seem reasonable to equate physical objects (e. g. a truck) or services with (simulation) agents, it is a matter of confusion. Agents are, e. g., humans, organisations, or artificial entities. An *agent* may represent objects and decide in the interest of the object's owner but, in general, it is no physical object itself. The exception would be humans or robotic agents, which can be viewed as physical objects as well as agents.

Furthermore, one has to distinguish an agent's attributes and the attributes of the object it acts for or the service it provides. In particular, an agent's position (if any) is not necessarily equal to the position of the physical object it represents. Also, it may represent more than one object at a time and would thus have no consistent position. Thus, the suggested way to handle autonomous entities is to distinguish agents from physical objects and services they provide and exclude them from the ontology if possible.

5.4.3 Scenario Ontologies

Usually, your own scenario consists of at least two or three ontology files: the top-level domain ontology (TLO, `tlo.owl`) and your scenario-specific ontology,[3] and possibly other ontologies for sub-domains such as transport or production (cf. Section 5.4). The scenario ontology is supposed to be located in the respective scenario's sub-directory `owl`. The predefined ontologies can be found in the PlaSMA sub-directory `etc/owl`. If you have a adapted copy of the predefined ontologies in the scenario's sub-directory `owl`, this is used instead for the scenario.

As already mentioned, you are able to modify predefined ontologies (because you can use a local scenario copy) but you should not change the top-level ontology to avoid that PlaSMA cannot load your scenario.

The actual scenario ontology particularly contains ABox information (such as objects in your scenario). You have to create this ontology by yourself. Either you use a text editor (preferably one that supports XML syntax highlighting) or a graphical ontology editor like Protégé[4].

Using a Text Editor

The following XML document defines a very simple scenario that only uses top-level ontology concepts and properties. The ontology is named `test.owl` and imports the TLO using the `<owl:imports>` tag. The `DOCTYPE` section (lines 2–11) just defines standard abbreviations as XML entity definitions that, for instance, can be used as prefix for identifiers in other XML namespaces when used in XML attribute values or tag contents. `<rdf:RDF>` is the actual root XML tag. Its attributes in lines 14–22 define abbreviations, too. But these may be used for identifiers of XML tags and attributes only – not their content or values.

The following lines (29–60) denote the actual scenario OWL description with instance and property assertions. The given scenario consists of two locations (Bremen and Hamburg), a bidirectional traffic link between them, and two trucks.

[3] The scenario ontology's default file name is `scenario.owl`. You may also use other file names depending on the XML scenario setting.

[4] Available for free from `http://protege.stanford.edu/download/download.html`.

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY owl11 "http://www.w3.org/2006/12/owl11#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY owl11xml "http://www.w3.org/2006/12/owl11-xml#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
    <!ENTITY tlo "http://plasma.informatik.uni-bremen.de/owl/tlo.owl#" >
    <!ENTITY test "http://plasma.informatik.uni-bremen.de/owl/test.owl#" >
]>

<rdf:RDF xmlns="http://plasma.informatik.uni-bremen.de/owl/test.owl#"
     xml:base="http://plasma.informatik.uni-bremen.de/owl/test.owl"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl11="http://www.w3.org/2006/12/owl11#"
     xmlns:tlo="http://plasma.informatik.uni-bremen.de/owl/tlo.owl#"
     xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
     xmlns:test="http://plasma.informatik.uni-bremen.de/owl/test.owl#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

    <owl:Ontology rdf:about="">
        <owl:imports rdf:resource=
         "http://plasma.informatik.uni-bremen.de/owl/tlo.owl"/>
    </owl:Ontology>

    <!-- Nodes / locations -->
    <tlo:Vertex rdf:about="#Bremen">
        <tlo:latitude rdf:datatype="&xsd;float">53.0759</tlo:latitude>
        <tlo:longitude rdf:datatype="&xsd;float">8.8073</tlo:longitude>
    </tlo:Vertex>
    <tlo:Vertex rdf:about="#Hamburg">
        <tlo:longitude rdf:datatype="&xsd;float">53.5506</tlo:longitude>
        <tlo:latitude rdf:datatype="&xsd;float">9.9933</tlo:latitude>
    </tlo:Vertex>

    <!-- Edges / roads -->
    <tlo:Edge rdf:about="#Bremen_Hamburg">
        <tlo:startingAt rdf:resource="#Bremen"/>
        <tlo:directedTo rdf:resource="#Hamburg"/>
        <tlo:edgeLength rdf:datatype="&xsd;float">120.0</tlo:edgeLength>
    </tlo:Edge>
    <tlo:Edge rdf:about="#Hamburg_Bremen">
        <tlo:startingAt rdf:resource="#Hamburg"/>
        <tlo:directedTo rdf:resource="#Bremen"/>
        <tlo:edgeLength rdf:datatype="&xsd;float">120.0</tlo:edgeLength>
    </tlo:Edge>

    <!-- physical objects -->
    <tlo:LandVehicle rdf:about="#truck1">
        <rdf:type rdf:resource="&tlo;MeansOfTransport"/>
        <tlo:positionedAt rdf:resource="#Hamburg"/>
    </tlo:LandVehicle>
    <tlo:LandVehicle rdf:about="#truck2">
        <rdf:type rdf:resource="&tlo;MeansOfTransport"/>
        <tlo:positionedAt rdf:resource="#Bremen"/>
    </tlo:LandVehicle>
</rdf:RDF>
```

Listing 5.1: OWL scenario ontology example.

5.4.4 How to Import a Transport Infrastructure from OpenStreetMap

It is quite easy to import new transport infrastructures from OpenStreetMap (OSM) to PlaSMA. Before starting the import process you need to install the following applications and download the OSM data:

- *PostgreSQL* (Version 8.4 or above, including *pgAdmin III*) with *PostGis* (Version 1.5) [5]
  - *PostgreSQL* download and installation guide is available under: `http://www.postgresql.org/download/`
  - Use the *Application Stack Builder* while installing *PostgreSQL* for downloading and installation of *PostGis* in Ubuntu you can install postgis via apt the package is called `postgis`.
  - The *PostGis* references is available under: `http://postgis.refractions.net/`
- *Osmosis*
  - *Osmosis* is a command line Java application for processing OSM data. It is especially useful for writing OSM data to databases. It is also possible to filter the OSM data. For example you can just import motorways, motorway junctions, inner city roads or other types of OSM highways.
  - Here you can download *Osmosis* and find an installation guide as well as a detailed documentation: `http://wiki.openstreetmap.org/wiki/Osmosis` In ubuntu you can install osmosis via apt the package is called `osmosis`
  - Optional: *OSMembrane* is a frontend to the *Osmosis* data processing tool. It helps by grouping tasks in functions and shows a visual representation of the pipeline. For more information see: `http://osmembrane.de/`
- OSM data
  - Here you can download OSM data:
    - `http://www.geofabrik.de/data/download.html`
    - or `http://downloads.cloudmade.com/`
- Optional: *Quantum GIS*
  - *Quantum GIS* (often abbreviated to *QGIS*) is a Geographic Information Systems (*GIS)* application that provides data viewing, editing, and analysis capabilities. With *QGIS* you can visualize the data from *PostGis* databases.
  - Download, references and userguide: `http://www.qgis.org/`
  - If you are worknig under *ubuntu*, you can simply install the packages `qgis` and and `qgis-api-doc` from the default repositories.

Now you have everything you need for importing OSM data to PlaSMA.

Preparing a database for the PlaSMA OpenStreetMap Importer

In this section the creation of a new database is described. We only explain how to do this with the *pgAdmin III* tool.

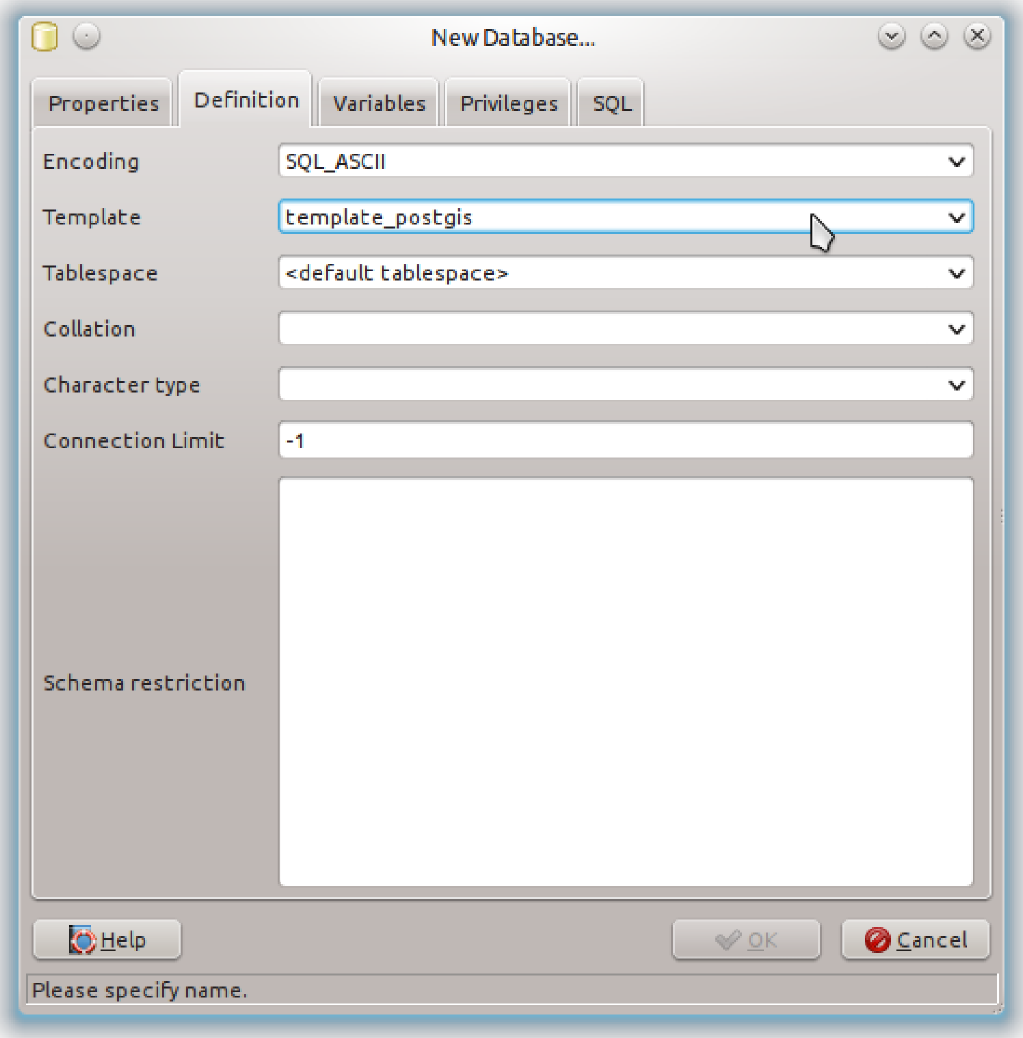
Linux
If you are using a Linux system you might want to create a template store the *postgis* schemata. The template can be used again if you wish to import other data via osmosis. You can create the template by executing the following commands:

```
prompt> sudo su postgres
prompt> createdb postgis_template
prompt> psql -d postgis_template -f /usr/share/postgresql/9.1/contrib/postgis-1.5/postgis.sql
prompt> psql -d postgis_template -f /usr/share/postgresql/9.1/contrib/postgis-1.5/spatial_ref_sys.sql
```

[5] Until now (February 2012) there exists only a stable 32 bit version of *PostGIS*. Therefore you also have to install the 32 bit *PostgreSQL* version.

Create the database

1. Open *pgAdmin III*
2. Go to *edit* → *create*
3. Insert the name of the new database
4. Go to *template* and select *postgis_template* (see Figure 5.6)
5. Hit the OK button to complete the progress



Encoding: SQL_ASCII
Template: template_postgis
Tablespace: <default tablespace>
Collation:
Character type:
Connection Limit: -1
Schema restriction:

Please specify name.

Figure 5.6: How to create a new *PostGis* database.

Using the PlaSMA OpenStreetMap Importer

The *PlaSMA OSM Importer* is a tool to transform OSM-data into a format that is easy to handle inside PlaSMA-Scenarios. At first you have to create a *postgres database*. You can do so by simply opening *pgadmin* and creating a new database (and if desired a new user), that should contain the imported data. [6]

PlaSMA provides a tool named *plasma.datamimport.osm* to convert *osm*-files into a format that can be handled by PlaSMA. You can start it by executing the provided script for your OS in the PlaSMA installation directory.

Windows

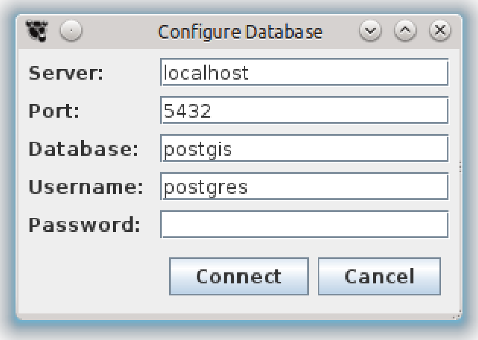
```
prompt> osm.import.bat
```

Linux

```
prompt> ./osm.import.sh
```

Connecting to the Database Once you have created the database (and a user) to store the data from an *osm file* you can use the *PlaSMA OpenStreetMap Importer* to create the needed tables and store the provided data. You can start the Importer by executing the `osm.import.sh` or `osm.import.bat` in the plasma main directory.

As soon as you has started the program a dialog appears 5.7, that prompts you to insert all needed data to connect to *postgres* database. created in the previous step.



Server: localhost
Port: 5432
Database: postgis
Username: postgres
Password:

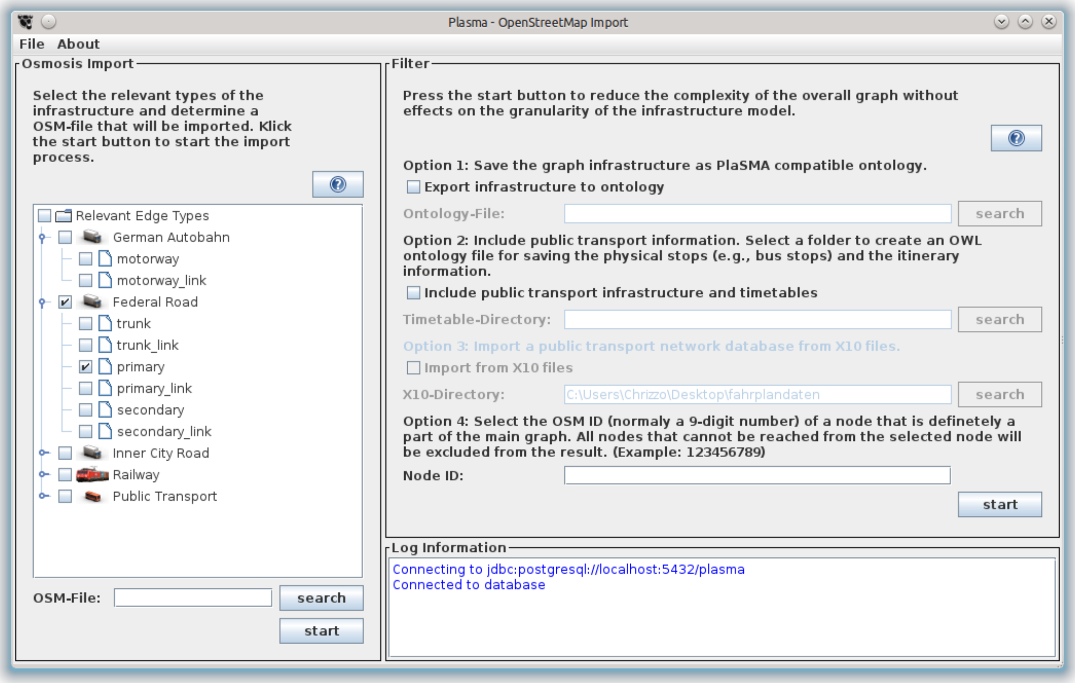
Figure 5.7: How to connect to an existing Database.

If you provided the correct data, a connection to the database will be established and the dialog will close and the main window will appear. After the first start the *PlaSMA OpenStreetMap Importer* will store all provided information excluded the provided password.

Main Interface The *main window* 5.8 can be divided in the 4 components described below.

- **Menu** The menu at the upper top of the window contains additional functionality like storing/loading configuration. Information about loading and storing data can be found in paragraph *Saving/Loading data*.
- **Osmosis Import** This panel on the left hand site of the window contains some options for importing osm data. See *Importing OSM-Data* for detailed instructions.
- **Filter** This panel on the upper right of the window contains settings about various filter configurations. Theses settings are described in detail in the paragraph *Filtering relevant data*.
- **Log** The Log panel on the lower right of the window contains information about the progress of pending operations. All errors, warnings and verbose informations can be found here.

[6]See chapter 2.1.1 about seeting up the database for more information.


Plasma - OpenStreetMap Import

File About

Osmosis Import

Select the relevant types of the infrastructure and determine a OSM-file that will be imported. Klick the start button to start the import process.

- [ ] Relevant Edge Types
  - [ ] German Autobahn
    - [ ] motorway
    - [ ] motorway_link
  - [✓] Federal Road
    - [ ] trunk
    - [ ] trunk_link
    - [✓] primary
    - [ ] primary_link
    - [ ] secondary
    - [ ] secondary_link
  - [ ] Inner City Road
  - [ ] Railway
  - [ ] Public Transport

OSM-File: search

start

Filter

Press the start button to reduce the complexity of the overall graph without effects on the granularity of the infrastructure model.

Option 1: Save the graph infrastructure as PlaSMA compatible ontology.

- [ ] Export infrastructure to ontology

Ontology-File: search

Option 2: Include public transport information. Select a folder to create an OWL ontology file for saving the physical stops (e.g., bus stops) and the itinerary information.

- [ ] Include public transport infrastructure and timetables

Timetable-Directory: search

Option 3: Import a public transport network database from X10 files.

- [ ] Import from X10 files

X10-Directory: C:\Users\Chrizzo\Desktop\fahrplandaten search

Option 4: Select the OSM ID (normaly a 9-digit number) of a node that is definetely a part of the main graph. All nodes that cannot be reached from the selected node will be excluded from the result. (Example: 123456789)

Node ID:

start

Log Information

```
Connecting to jdbc:postgresql://localhost:5432/plasma
Connected to database
```

Figure 5.8: OSM Importer main interface.

Importing OSM-Data All settings for importing osm data into a plasma compatible format can be found in the *Osmosis Import panel*. To import data to the database you created earlier you have to follow the next steps:

- Specify all *edge types* you wish to import from the tree on the left. Only selected data types will be imported.
- Choose the *osm-file* from your file-system, you want to import into the database. You can choose a file by clicking on the search button and browse your file-system for the desired file or you type in the desired URL.[7]

After you provided the information above, you can start the import by clicking on start button on the left site of the *main window*. The process might take a while. You can see information about success or failures in *Log panel*.

Filtering relevant data The *Filter panel* provides settings for filtering imported data. These settings can be used to reduce the complexity of the osm data, without having effects to granularity of the graph. You can start the progress by clicking on the *start-button* on the bottom of this panel. The *start-button* triggers a mechanism that removes all nodes that have only one incoming and outgoing edge. There are some options to change the default behavior of the mechanism.

- **Export to OWL** You can export all data to an *owl* ontology file. To do so check the box under *Export infrastructure to Ontology*. If you decide to export the ontology to owl you have to specify a path where the infrastructure should be saved in the field next to the checkbox.
- **Include public transport infrastructure and timetables** If you wish to use information about *public transport* you can store this data as a `.csv` file. For this you have to specify a target

[7]You can download osm data from various sites. For example: `http://www.geofabrik.de/data/download.html`, or `http://downloads.cloudmade.com/`

directory in the field *time-table-directory*. If you do not use the additional option *import from X10 file*, the importer will auto-generate timetables for you. In this case for each generated timetable each bus/tram starts at every full hour and will need a two minutes ride per stop and wait on each stop for one minute. It is also possible to use X10 data to specify timetables and get a better *public transport infrastructure*. [8] Currently you can only simulate two different types of vehicles(called buses and trams), and you have to clone one week day for every day of the week. To use this feature you have to specify the *X10-directory* and do some manual manipulation of the *PlaSMA OpenStreetMap Importer*-Properties file. See *Configuration of the PlaSMA OpenStreetMap Importer* for more information.

- **Remove irrelevant sub graphs** You can also remove subgraphs that are not connected to the graph you are interested in. To do so you have to specify a node ID (a 9-digit number) corresponding to on node in the graph you want to keep. All nodes that can not be reached from that node will get removed from the graph.

Saving/Loading data The *PlaSMA OpenStreetMap Importer* loads the last used configuration as default when starting the program. You can also save the current data for later use or to share configuration with other users. To do so you can simply click on the *Save Setting* or *Load Setting* Entry in the *File* menu. This will open a file chooser Dialog where you can select the file you want to save/load. By default the files are stored with as `.iosm` file. These files are readable in any common text editor and can also be changed by hand if desired (See *Configuration of the PlaSMA OpenStreetMap Importer* for more details).

Configuration of the PlaSMA OpenStreetMap Importer There are two different configuration files for the PlaSMA OpenStreetMap Importer. Both files are build of `key`, `value` pairs separated by a = sign. The part before the equality sign is the key that is used by the importer that is used to find the corresponding value. You should always only change the values of these pairs otherwise the program might not work as expected. If the Importer can't find one of the files at startup, a file filed with default `values` and blank fields will created automatically.

Database Settings The database settings are stored after every successfully established connection. There should be no need to specify any data here by hand but if wished you can do so. Be aware of that no password data will be saved for database access.

- **DEFAULT_SERVER** The Server-IP you want to connect to(default `localhost`)
- **DEFAULT_PORT** The Port on which the server listenes to incomming database connections (default 5432)
- **DEFAULT_DB** The name of the database you want to access
- **DEFAULT_USER** The name of the user with the needed rights to access the database

Filter Settings Filter Files are used to save the settings of the *PlaSMA OpenStreetMap Importer*. There are also some small additional settings for X10-import that are only stored in this files. The last settings will be stored in a file named `current_filter_settings.iosm`. Other files can be named as you which but they might be easier to find if you keep the `.iosm` file ending. The following keys can be modified.

- **OSM_FILE** The full qualified path of the osm file that was used.
- **INCLUDED_EDGES** The Edge types that where used to import the data.
- **USE_OWL** Whether or not OWL should be used (possible values `TRUE` and `FALSE`)

[8] see http://www.vdv.de/oepnv-datenmodell.aspx for a detailed german information about the X10 data format.

- **OWL_FILE** The full qualified path of the owl file the infrastructure should be exported to
- **USE_TIME_TABLE** Whether or not public transport and timetable data should be used (possible values `TRUE` and `FALSE`)
- **TIME_TABLE_DIR** The full qualified path of the directory where the public transport data should be saved.
- **USE_X10** Whether or not the X10 comparison should be used (possible values `TRUE` and `FALSE`)
- **X10_DIR** The full qualified path of the directory where the X10 data lies.
- **X10_BUS_ID** The ID that is used for buses in the provided X10 data.
- **X10_TRAM_ID** The ID that is used for trams in the provided X10 data.
- **X10_ADDITIONAL_SQL_SCRIPT_FILE** An additional sql file correct mappings depending on the place infrastructure you import.
- **X10_DAY_OF_WEEK** The day of the week that should be used for the import(currently only one day is cloned for every day of week)
- **NODE_ID** The 9-digit node ID of a node that is in the main graph(all edges that can not be reached from this node will be removed)

Optional: Visualization with QGIS

You can visualize your *PostGis* data with QGIS to do so follow the steps below:

- Open *QGIS*
- Create a new Layer (see Figure 5.9)

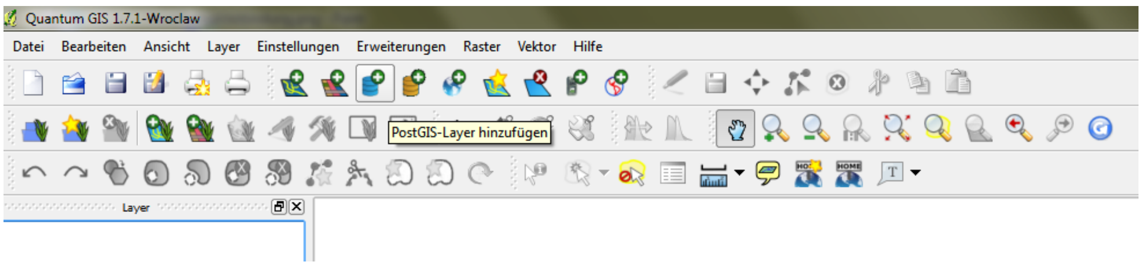
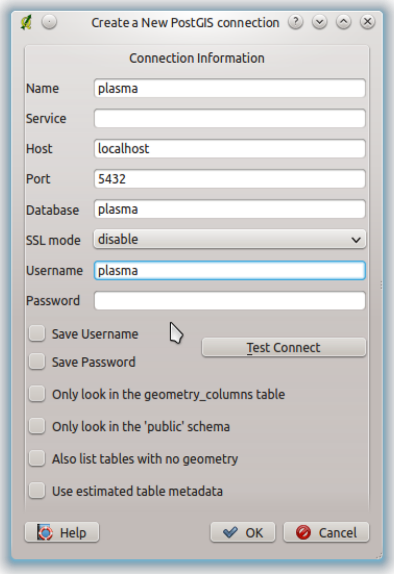


Figure 5.9: Create a new Layer.

- Connect *QGIS* to *PostGis* (see Figure 5.10)
- Select the data that you want to visualize (see Figure 5.11)
- **It is important that you change the datatype of the ID columns from bigint to integer in your databases!** We recommend to do this with *pgAdmin III* that is installed with *PostgreSQL*.

Manipulate the scenario configuration

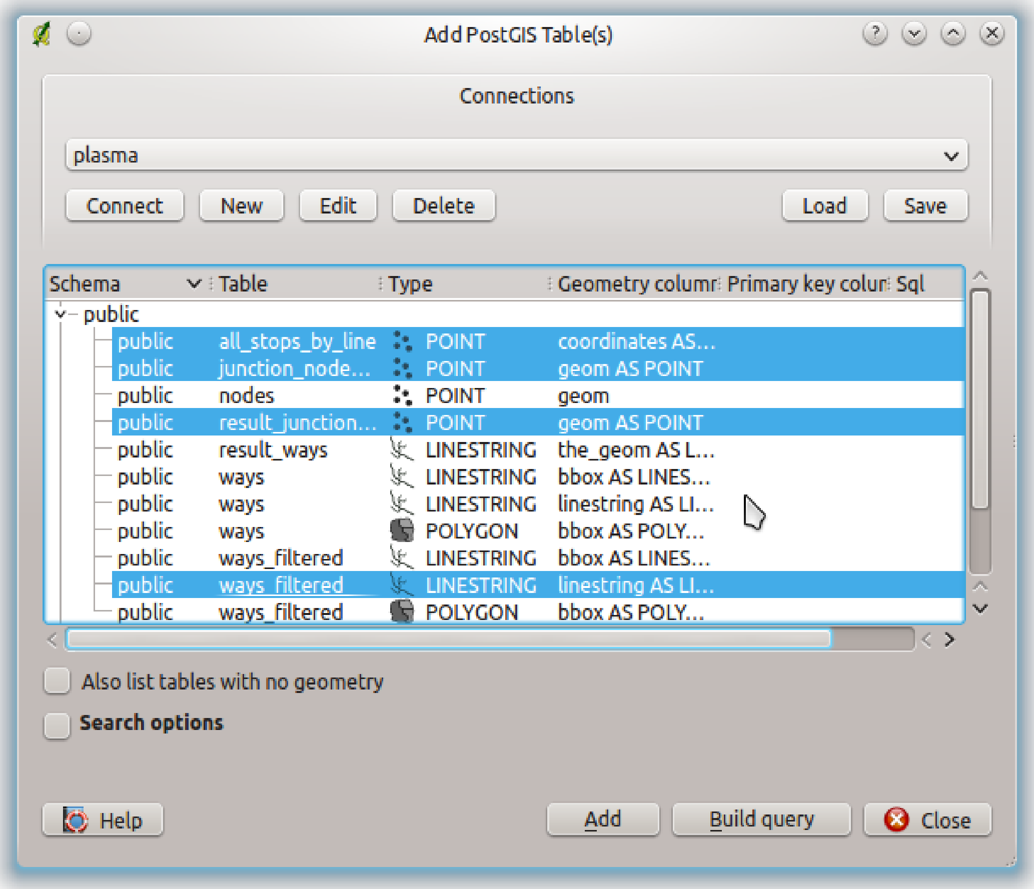
At last you have to specify in your scenario configuration *.xml* file to use an ontology file or the database for representing the transport infrastructure. In the later case you also have to configure the connection properties in the following way:



Name	plasma
Service	
Host	localhost
Port	5432
Database	plasma
SSL mode	disable
Username	plasma
Password	

- [ ] Save Username
- [ ] Save Password
- [ ] Only look in the geometry_columns table
- [ ] Only look in the 'public' schema
- [ ] Also list tables with no geometry
- [ ] Use estimated table metadata

Figure 5.10: Connect to *PostGis*.



Connection: plasma

Schema	Table	Type	Geometry column	Primary key column	Sql
public					
public	all_stops_by_line	POINT	coordinates AS...		
public	junction_node...	POINT	geom AS POINT		
public	nodes	POINT	geom		
public	result_junction...	POINT	geom AS POINT		
public	result_ways	LINESTRING	the_geom AS L...		
public	ways	LINESTRING	bbox AS LINES...		
public	ways	LINESTRING	linestring AS LI...		
public	ways	POLYGON	bbox AS POLY...		
public	ways_filtered	LINESTRING	bbox AS LINES...		
public	ways_filtered	LINESTRING	linestring AS LI...		
public	ways_filtered	POLYGON	bbox AS POLY...		

- [ ] Also list tables with no geometry
- [ ] **Search options**

Figure 5.11: Select the data that you want to visualize.

```
        <worldmodel>
            <database enabled = "true"
            url = "jdbc:postgresql://localhost:5432/<database name>"
            user = "postgres"
            password = "xxx"/>
                ...
```

Note: If you do not use an ontology representation for the transport infrastructure I would recommend to set the positions of *physical objects* in your *scenario.owl* in the following way:

```
<tlo:positionedAt>
    <tlo:Coordinates rdf:about="#name">
          <tlo:latitude rdf:datatype="&xsd;float">53.08766</tlo:latitude>
          <tlo:longitude rdf:datatype="&xsd;float">8.722644</tlo:longitude>
  </tlo:Coordinates>
</tlo:positionedAt>
```

It is inconvenient to lock for a specific node in the database.
If there do not exist a node on a specified coordinate a new node is created and connected to the nearest existing node. The length of the new edge is the linear distance times 1.3.

5.5 Agent Implementation

For simulation purposes real world objects and entities as well as environmental processes can be represented by parallel logical simulation processes, called *simulation agents*. A simulation run in PlaSMA is a forward-directed computation of these logical processes (LPs). In PlaSMA, an LP is realized by subclassing of `SimulationAgent` class. `SimulationAgent` is a base class for implementing agents in PlaSMA.

It is not necessary to model any ontological counterpart for those simulation agents (see Section 5.4.2) or linking it to any ontological instances. However, the implementation of a simulation agent usually models the control over a specific set of physical objects (e. g. trucks and containers) or environmental processes, since physical objects are neither given any control nor do they carry a model of technical-intelligent behaviour. As an exception, you could model a robotic agent that is a physical object and a control system at the same time.

5.5.1 Agent Setup

When subclassing from `SimulationAgent` one has to override the `userSetup()` method. `userSetup()` is being executed, when the agent is brought to life. It replaces the `setup()` method known from normal JADE agents.

A PlaSMA simulation agent will already be registered at the agent platform (AMS) and is therefore capable of sending and receiving messages. Anyhow, the simulation agent will be registered at one of the PlaSMA sub-controllers *after* `userSetup()` finished. Registration to the PlaSMA controller is necessary for simulation time synchronisation and agent life cycle management.

Thus, when `userSetup()` is called the simulation agent is not actually running yet. In the `userSetup()` method one can do start-up routines like setting up individual behaviours of the agent. Initial procedures like registering the agent with some service (FIPA directory facilitator or an other agent) should not be done in the `userSetup()` in order to avoid concurrency problems. For instance, the desired service might not be running yet.

Below is an example of a `userSetup()` method, in which some ontological properties of the agent are read from the ontology and two behaviours are registered. The agent itself is being held in an extra thread; its behaviours are subprocesses.

```
protected void userSetup() {
    // get represented object
    BasicRI myObject = getReferencedObjFor(OntologyObjProperty.representsObject);

    // get general transport capacity (will not change)
    PhysicalObject objStatus = getWorld().getObjectByID(this.myObject);
    this.capacity = objStatus.getState().getDoubleValue(
                new BasicRI(OntologyService.TLO_ONTOLOGY_URL, "storageCapacity"));

    transBeh = new TruckTransportBehaviour(this);
    this.addUserBehaviour(transBeh);
    procBeh = new TruckProceedingsBehaviour(this);
    this.addUserBehaviour(procBeh);
}
```

5.5.2 Reception and Processing of ACL Messages

The JADE platform provides two ways for querying the message inbox of an agent. There are blocking methods and non-blocking methods.

- `blockingReceive()`
- `receive()`

Moreover derived methods can be used to setup a message filter `MessageTemplate` or to specify a maximum waiting time. Note that this waiting period refers to the real, not the simulation time! In simulation, the non-blocking as well as the blocking method will return `null` if no message has been received at the beginning of the current agent processing cycle. The simulation inbox management will intercept all later messages to insure simulation run causality and reproducibility (cf. [GSW08]).

When retrieving messages, note the following:

- Using the blocking methods for retrieving messages from the inbox stops all other JADE behaviours of the agent, until a message could be received or the waiting period expired. Moreover, the original JADE blocking receive methods with no timeout could yield into a deadlock in simulation, because there is no guarantee that the desired message will be available in the current cycle. Thus, SimulationAgent always uses a default timeout of 5 seconds.[9]
- Querying the inbox with waiting time is generally to be avoided, as hereby simulation will be unnecessarily delayed. Such a method call should rather be repeated in a following cycle (with other simulation time). It should also be noted that the waiting time is real time, not simulation time.
- If the receive method returns a message, the message is being dequeued from an internal message queue of the JADE agent. Then this message will be no longer available in the inbox. Therefore, a single behaviour should not remove a message from the inbox, that is regarded to another behaviour. One can ensure to receive only appropriate messages with the help of filters (`MessageTemplate`). It is also possible to put message back in the inbox, if it turns out, that they are out of place (see `Agent.putBack(ACLMessage)`).

5.5.3 Agent Behaviours

The behaviour of the agents is realized through `jade.core.behaviours.Behaviour` class and its subclasses in the same package. When subclassing from these `Behaviour` classes in turn, one has to override the `action()` method. To find out more about JADE behaviours you are referred to the JADE

[9] There are changes for future versions PlaSMA in preparation, which also allow blocking wait. The specified waiting period would be considered simulation time and the agent would automatically end its cycle if no matching message is available.

manual [BCTR07, pp. 24]. It is possible to have several distinct types of JADE behaviours mixed up for one agent, so that both sequential and parallel behaviours are being processed. Up to now the parallel behaviours are not fully supported in PlaSMA.

In general, behaviors of simulation agents should be synchronized. This is achieved by the implementation of the Java interface `UserBehaviour` from within one's custom behaviour class. Instances of this behaviour should then be added to the agent with the `addUserBehaviour()` method. In extremely rare cases, it can also be useful to use unsynchronized behaviours. Those can still be added with the base class method `addBehaviour()` if needed. When using unsynchronized behaviours, please ensure that the agents do not communicate with each other off-time their synchronized cycles. This would distort the simulation results, or at worst lead to unplanned downtime of the simulation.

It is recommended to instantiate initial behaviours within the `userSetup()` method. Behaviours are being processed sequentially in the order of their registration time.

There are two ways to remove a synchronized behavior: When a behaviour is finished (`done() == true`), it is automatically removed from the scheduler. If it should be removed explicitly, please call the method `removeUserBehaviour()`.

5.5.4 Simulation Time Progression

In order to realise time progression in simulation, agents indicate that they finished their processing cycle for the current simulation time. This is accomplished by requesting a new time event from the simulation controller. This request is sent automatically by each agent when all active behaviours of that agent finished the processing cycle. In order to indicate that a behaviour has finished computation for one cycle, it calls the `myAgent.finishedCycle(UserBehaviour b, long nextSimTime)` method. The first parameter refers to the behaviour itself and the second parameter provides the simulation time at which the behaviour has to be scheduled again (as long as no other event occurs before). Note that finishing a processing cycle differs from finishing execution of a behaviour completely. The latter is indicated by the return value of a behaviour's `done()` method. Finishing a processing cycle only prevents the behaviour from being scheduled until the next event occurs or simulation time progresses.

For simulation stability and causality reasons it is important to ensure that the Java source code of a behaviour does not comprise any further commands (other than `return`) subsequent to calling a `finishedCycle()` method. Otherwise, unexpected side effects regarding scheduling and synchronization can occur!

5.5.5 Using Finite State Machines for Complex Behaviours

Advantages of a Finite State Machine

JADE provides several types of behaviours for special occasions, such as the `OneShotBehaviour` which executes its `action()` method exactly once and terminates thereafter, or the `SimpleBehaviour` which can be re-used at any time. A very powerful complex standard behaviour provided by JADE is the `FSMBehaviour`. This behaviour is a specialization of the `SequentialBehaviour`. Instead of executing registered child behaviours in a fixed sequence, the `FSMBehaviour` allows to combine child behaviours to a Finite State Machine.

By definition, a Finite State Machine is a model of behaviours, that maps every possible situation of a system to a representing state. Each situation might be changed by a finite number of actions, accordingly transitions lead from one state to another. Transitions might depend on conditions as actions might depend on situation requirements. In any case, transitions have to be deterministic[10].

The concept of Finite State Machines can be easily adopted to the design of a complex behaviour. You will often find behaviours that directly depend on the results of other behaviours. For instance, the

[10]If more than one transition leads from one state to a following, every transition needs a unique condition to determine the correct following state.

driving behaviour of a truck agent might depend on the result of a routing behaviour. Instead of mixing them up into one oversized behaviour or exchanging wake-up calls between them, it might be more accurate to combine these behaviours to a Finite State Machine.

This way, all behaviours can be developed and implemented without knowledge about each other. It is easier to add, remove or swap behaviours and allows to re-use behaviours from other agents without making major changes to the source code.

Implementation of a FSM-Behaviour

The idea is simple: Every state of the FSM represents a certain behaviour. Whenever a behaviour terminates, a result value will be returned when its `onEnd()` method is called. The result can be used as a transition condition to determine which behaviour ought to be scheduled next. To keep this result readable and traceable, it may be useful to define a Java `enum` type for each behaviour, that contains all possible results. For instance:

```
public static enum result {
    SUCCESS, FAILURE, ANY_OTHER_RESULT
};
```

Basically, the `jade.core.behaviours.FSMBehaviour` provides all necessary methods to create a Finite State Machine. As a Finite State Machine has a first state and at least one last state, `registerFirstState(...)` and `registerLastState(...)` allow to define start and stop behaviours. Any other states in between can be registered with the method `registerState(...)`. Required arguments are the behaviour that will be represented by the state and a unique `String` label for this state. To keep the code of the `FSMBehaviour` independent from the managed behaviours, it may be useful to keep the label for each state as a static variable within the respective behaviour class.

```
this.registerState(new MyFooBehaviour(this.myAgent, this.getDataStore()), MyFooBehaviour.LABEL);

this.registerState(new MyBarBehaviour(this.myAgent, this.getDataStore()), MyBarBehaviour.LABEL);
```

To implement dependencies between the states, transitions have to be registered; either as default transitions (e.g., if the following state will always be the same) or as a conditional transition. Every state may have various outgoing transitions. For consistency, every transition needs a unique condition, as discussed before. It is possible to register a default transition and conditional transitions from the same state. In such a case, the default transition will be used when none of the other explicitly specified transitions apply.

```
this.registerTransition(MyFooBehaviour.LABEL, MyBarBehaviour.LABEL,
        MyFooBehaviour.result.SUCCESS.hashCode());
```

Though `hashCode()` should be sufficient, there could, theoretically, be collisions. If that is the case a solution would be to manually set and keep track of the `integers` representing each result.

Finishing cycles

The usage of `finishedCycle()` is nearly the same as for every other behaviour, as described in Section 5.5.4. The main difference is, that only the child behaviours need to call the `finishedCycle()` method. PlaSMA keeps track of the active child behaviours and a parent behaviour will finished the current cycle only if (and as soon as) all managed behaviours have reported that they are done for this cycle. So there is no need to call `finishedCycle()` in a `FSMBehaviour` itself. This applies also if a `FSMBehaviour` contains additional `FSMBehaviours` as children.

5.5.6 Accessing the World Model

The world model provides information on the current status of the simulated physical environment with the agents and objects therein. Additionally, simulation agents can change this environment by triggering actions and events as well as by creating new simulation agents or new physical objects. An agent can access the world model using method `SimulationAgent.getWorld()`. This method returns the interface `WorldModel` which is a combination of two interfaces[11]: `WorldModelQuery` for obtaining information and `WorldModelAdmin` to alter the model.

Among others, `WorldModelQuery` includes the following methods:

- `PhysicalObject getObjectByID(BasicRI objectID)`
  Returns a descriptor of a physical object in the world model identified by its object ID. The descriptor contains object properties but is not adequate to actually change the object properties.
- `ObjectPosition getPositionByID(BasicRI objectId)`
  Returns the given object's current position.
- `DirectedGraph getGraph()`
  Returns a directed graph as the scenario's underlying traffic infrastructure.

`BasicRI` is a simple datatype representing an unique resource identifier (URI) for a resource/object in the simulation model. It consists of two strings for the namespace and the local ID respectively. Most of the methods in this interface are self-explanatory. For a complete and more detailed description of the interface please refer to the Javadoc API documentation of PlaSMA.

World Modifications with Actions

To alter objects in the world model (i. e., by moving objects) so-called `Actions` have to be defined for the scenario. A couple of base and example actions are already included in PlaSMA[12]. To trigger an action the simulation agent has to register it with the world model by calling

```
WorldModel.registerAction(action, agentID).
```

Until their completion, actions will be re-evaluated by the world model. For every progression of simulation time all actions will be evaluated and processed if necessary. For example, an action could move a truck from A to B over a road stretch (edge) in the modelled traffic infrastructure. Depending on the simulation time that passes between the creation and the following synchronization step, the truck will be shifted on the edge towards B. If the truck has not yet reached B the action will not be marked as complete and will be re-scheduled automatically at the next desired synchronization step.

In order to create your own action it is necessary to extend the abstract class `Action`. For better understanding, the following text gives a short description of the important methods:

- `WorldModelQuery getWorldModel()`
  Provides read-only access to the world model *after* action registration. The action should use this method to query needed information on the current state of the world model.
- `long getCreationTime()`
  Returns the simulation time (as UNIX millisecond timestamp) when the action was created. The creation time corresponds to the time it was registered with the world model.
- `void init()`
  This method initialized the action as intended by the action programmer. It is called by the world model upon registration. In contrast to the constructor, the action creation time and the world model will be accessible when the method is called.
- `boolean checkPreconditions() throws ActionException`
  This method will be called by the world model after registration and initialization of the action. If this method returns `false` or throws an `ActionException` the action is aborted and the executing agent is waked up. By throwing an `ActionException` you can provide an explanation why precondition are not met.

[11] Java package `org.tzi.plasma.worldmodel.interfaces`.
[12] Java package `org.tzi.plasma.worldmodel.actions`.

- `void checkInvariants() throws ActionException`
  Checks world model properties (*invariants*) that need to hold during whole action exection. Invariants are checked before each action exceution. The method should throw an `ActionException` if current world model conditions do not comply with the invariants. Implementation of this method is optional.

- `WorldModelChanges execute(long currentTime)`
  This method is executed by the world model to compute the effect (delta) of the action. The effect is supposed to be computed for the time span between last action execution and the current simulation time. If we reuse the vehicle example from above, the method should compute the average driving speed of the vehicle and the passed distance. A new `ObjectPosition` should be created and stored in the `WorldModelChanges` instance that is returned. `WorldModelChanges` are supposed to hold all changes taking effect in the current timestamp by the given action. `WorldModelChanges` is an read-only interface. Action implementations should use its implementation `ObjectModelDelta`.

- `long nextExecutionTime()`
  States the simulation time when the action wants to be executed next if no affecting external events occur. The world model will check this time in each synchronization cycle. If the desired execution time is reached, the world model will execute the action. Please note that the actual execution time may be delayed due to time progression granularity as configured by `minSyncDist` scenario property (cf. Sect. 4.1.1).

- `WorldModelChanges transitionState()`
  This optional method is called if an incomplete action is not executed but was registered or executed in the previous world model update. It gives the action the possibility to indicate a transitional world model state of the action (e. g., to move objects into `TransitionalObjectPositions`.).

- `boolean isCompleted()`
  This method indicates whether the action has been completed yet. This property has to be set by the action programmer and is evaluated by the PlaSMA world model. Complete actions will not be scheduled again.

An Action Implementation Example: The Load Action After the description of the important methods provided by the `Action` interface, this paragraph will give a small example on how to implement those methods by looking at the `LoadAction`[13]. This `Action` is supposed to store a package into a storage facility, thus it requires the information which `load` to store into which `storageFacility`. It also takes into account how long the loading process will take (`timeToLoad`).

```
public class LoadAction extends Action {
    [...]
    public LoadAction(BasicRI load, BasicRI storageFacility, long timeToLoad) {
        this.storageFacility = storageFacility;
        this.load = load;
        this.duration = timeToLoad;
        goalPosition = new RelativeObjectPosition(storageFacility);
    }
```

The `init` method of this `Action` is rather short, since only the `finishingTime` is determined by using the `getCreationTime` method.

```
    @Override
    public void init() {
        this.finishingTime = getCreationTime() + duration;
    }
```

The next method is called after the initialization and will check the given preconditions. In this case it is verified that the `load` is a freight object, the `storageFacilty` is a storage facility and that both objects are in the right position by requesting those informations from the world model.

[13]See `org.tzi.plasma.worldmodel.actions.LoadAction`.

```
@Override
public boolean checkPreconditions() {
    WorldModelQuery wmq = this.getWorldModel();
    if (!wmq.belongsToClass(load, ObjectModel.FREIGHT_OBJECT_CLASS_ID)) {
        return false;
    } else if (!wmq.belongsToClass(storageFacility,
            ObjectModel.STORAGE_FACILITY_CLASS_ID)) {
        return false;
    } else if (wmq.getPositionByID(load).isTransitional()
            && !currentPosition.equals(wmq.getPositionByID(load)
                    .asTransitionalPosition())) {
        return false;
    } else if (!wmq.getObjectCoordinates(load).equals(
            wmq.getObjectCoordinates(storageFacility))) {
        return false;
    }
    return true;
}
```

The `execute` method is most likely the core of each `Action`. It determines the necessary changes to the world model and stores them into an `ObjectModelDelta`. In this case after the scheduled loading time is passed the appropriate changes are provided to indicate that the `load` is now stored in the `storageFacility`. If the required time hasn't passed yet, nothing changed and an empty delta is returned.

```
@Override
public WorldModelChanges execute(final long currentTime) {
    final ObjectModelDelta delta = new ObjectModelDelta();

    if (currentTime >= finishingTime) {
        delta.addChangedObjectPosition(load, goalPosition);
        delta.addPhysicalObjectObjectProperty(storageFacility,
                ObjectModel.HAS_LOADED_PROPERTY_ID, load);
        this.complt = true;
    }

    return delta;
}
```

During the load process the `load`'s position is set to a transitional position since it's not placed in its old position anymore but also not stored in the storage facility yet.

```
@Override
public WorldModelChanges transitionState(final long currentTime) {
    final ObjectModelDelta delta = new ObjectModelDelta();
    final ObjectPosition pos = this.getWorldModel().getPositionByID(load);

    if (!(pos instanceof DefinitePosition)) {
        this.abort("Illegal initial position type " + pos.getType());
    } else {
        this.currentPosition = new TransitionalObjectPosition(
                (DefinitePosition) pos, goalPosition);
        delta.addChangedObjectPosition(load, currentPosition);
    }

    return delta;
}
```

The `needsExecution` method may indicate if the `Action` needs re-evaluation due to the latest changes to the world model, which is not the case in this example.

```
@Override
public boolean needsExecution(final WorldModelChanges changes) {
    return false;
}
```

The last method `nextExecutionTime` provides the simulation timestamp at which the `Action` should be executed the next time. In this example the next execution time is directly after the loading process has finished.

```
    @Override
    public long nextExecutionTime() {
            return finishingTime;
    }
}
```

For more examples take a look into the toolkit source code which can be found in the `toolkit_src` directory. Common actions are located in `toolkit_src/org/tzi/plasma/worldmodel/actions` and `toolkit_src/org/tzi/plasma/toolkit/load/agent/actions`. The code of this example can also be found in the toolkit source in `toolkit_src/org/tzi/plasma/worldmodel/actions/LoadAction.java`.

5.5.7 Agent Attributes

To be able to configure an agent for different scenarios it is possible to provide arguments in the scenario configuration[14] that the agent can access at run time. This is also a convenient way to link an agent to ontological instances. PlaSMA provides three predefined attribute keys for common ontology references: *representsObject*, *representsInfrastructure*, and *operatesFor*.

As an example take a look at the `Agent_ForwardingAgency`[15]. This class represents a forwarding agency that supports multiple planning modes and operates for a specific company. Its configuration contains the following attributes.

```
<agent description="TZI Logistics Forwarding Agency" name="Management.TZI_Logistics" class="org.tzi.plasma.
    toolkit.forwardingagency.agent.Agent_ForwardingAgency">
  <attribute key="run_mode" value="CENTRAL_PLANNING" />
  <attribute key="operatesFor" value="http://plasma.informatik.uni−bremen.de/owl/default_northrange.owl#
      TZI_Logistics"/>
</agent>
```

Recommended Attribute Access

To access those attributes it is recommended to implement the `ConfigKeyInterface`[16] as an `enum`. Given that `enum` the attribute can be accessed through the agent's `getParameter(ConfigKeyInterface, Class<T>)` method or `getParameterList(ConfigKeyInterface, Class<T>)` if the attribute contains multiple values. The first parameter is the corresponding `enum` entry of the attribute's key (spelling needs to be exactly the same) and the second parameter determines the type of the attribute's value (to avoid the usually required cast).

```
// assuming a CustomAttribute Enum
String mode = getParameter(CustomAttribute.run_mode, String.class);
```

The previously mentioned predefined keys are provided through the `OntologyObjProperty`[17] enum. Since ontological references are always represented as `BasicRI`s, the agent has special methods to skip that second type parameter, namely `getReferencedObjFor(OntologyObjProperty)` and `getReferencedObjsFor(OntologyObjProperty)`.

```
BasicRI company = getReferencedObjFor(OntologyObjProperty.operatesFor);
```

The GUI scenario configuration editor is capable of suggesting attributes created this way, but needs to associate them with an agent. Therefore the `enum` implementing the `ConfigKeyInterface` should be linked to all agent classes that support that attributes through the `@AgentConfiguration` annotation.

[14] See Section 4.2.2 for details.
[15] `org.tzi.plasma.toolkit.forwardingagency.agent.Agent_ForwardingAgency`
[16] `org.tzi.plasma.control.ConfigKeyInterface`
[17] `org.tzi.plasma.control.SimulationAgent.OntologyObjProperty`

```
@AgentConfiguration({CustomAttribute.class, ForwardingAgencyAttribute.class})
public class Agent_ForwardingAgency extends SimulationAgent {
  [...]
}
```

Implementing the ConfigKeyInterface Though it is possible to implement the `ConfigKeyInterface` in any class, it is recommended to use an `enum`. The following listing contains an excerpt of the `CustomAttribute`[18] implementation.

```
 1 package org.tzi.plasma.toolkit.forwardingagency.agent;
 2
 3 import org.tzi.plasma.control.ConfigKeyInterface;
 4 import org.tzi.plasma.control.ConfigAttribute;
 5 import org.tzi.util.BasicRI;
 6
 7 public enum CustomAttribute implements ConfigKeyInterface {
 8      run_mode (ConfigAttribute.ofType(String.class).withValues("DECENTRAL_PLANNING")),
 9      // Mandatory in configuration
10      depots (ConfigAttribute.ofType(BasicRI.class));
11
12      private ConfigAttribute data;
13
14      private CustomAttribute(ConfigAttribute data) { this.data = data; }
15
16      @Override
17      public ConfigAttribute data() { return data; }
18 }
```

Lines 12–17 enable the basic functionality of the `ConfigKeyInterface` and can be inherited to every `ConfigKeyInterface` implementation. The `data` object encapsulates parsing and storing of the attribute. The first lines (8–10) add `enum` values and make use of the constructor in line 14. Those values needs to be spelled exactly like the attribute's key in the configuration. The `ConfigAttribute` parameter can be created with the `ofType(Class<T>)` method. If the attribute contains a default value (or a list of default values) this can be set by chaining the `.withValues(T...)` function. It is also possible to add a description by chaining `.withDescription(String)`. Those default values can be used by the scenario configuration editor as suggested value.

Alternative Attribute Access

Although the previously described method is the recommended way because it supports the scenario configuration editor and makes the access to the attributes clearer, there is an alternative way to get to the attribute's value. All attributes are stored in a `Map<String, List<String»` mapping that maps the attribute's key to its value(s) in `String` representation. This map can be retrieved by calling the agent's `getArguments()` function and extracting the first object from the returned array.

```
final Object[] args = getArguments();
final Map<String, List<String>> params = (Map<String, List<String>>) args[0];
```

5.5.8 Termination of the Agent Life Cycle

The recommended way to control termination of a simulation agent is to override the method `isAgentDone()` in the class extending `SimulationAgent`. This method always returns `false` in the default implementation. The method is evaluated at the end of each agent processing cycle. The agent is properly terminated and deregistered from PlaSMA if it returns `true`. For termination `doDelete()` is executed.

[18] `org.tzi.plasma.toolkit.forwardingagency.agent.CustomAttribute`

If `isAgentDone()` is insufficient to handle situations when the agent shall be terminated you can call this method directly in your behavior code as well. When doing so, you *must not* call `finishedCycle()` for this behaviour anymore. Otherwise the agent might deregister with PlaSMa twice which will cause an error!

Either way, you need to take care though that an agent that registered with the directory facilitator agent (DF) needs to deregister before termination. In order to do such clean-up work, override the method `onTermination()` of `SimulationAgent`.

5.5.9 Agent Logging

One can use `log()` method or `syslog()` method of `SimulationAgent` class to log messages. Where `syslog()` should be used if a system message is to be logged, which is not related to the actual simulation scenario but only its implementation.

Both `log()` and `syslog()` require two parameters: One has to pass the message to be logged and a corresponding `LogLevel`. The `LogLevel` value can be retrieved from of the `LogLevel` enumeration class. The given LogLevel decides whether the message is logged or is discarded.

In the method `log()`, there is an optional parameter for a `LogCategory` object. As described above the log messages can be divided into different categories with different levels of granularity. It is recommended to use this option, since it increases the clarity of the logger output.

Important: It is not necessary to specify all log categories in the XML configuration file. If an unknown log category is used, it will depend on the system-wide log levels, whether a specific message is logged or not.

One can either choose a pre-defined category from `LogCategory` class or create a new log category on demand. The latter is done by `new LogCategory(categoryName)`. As a category name one has to qualify the full path of the category, i. e. `Communication.Input`. Specifying `Input` as a log category name will denote a different category.

The agent-specific logging is done automatically when the logging methods of the class `SimulationAgent` are used. The agent-specific logging suppresses the log level of any category. That is, if one specifies a log category whereas a separate log level is defined for the agent, only the latter is taken into account.

5.5.10 Performance Metric Logging

In addition to the message-based logging system there is the possibility of using typed performance metrics, called *performance indicators* in PlaSMA. Performance indicators can be used to log agent metrics to the database (such as driven kilometers, speed, truck utilisation, profits etc.) and analyze them when the simulation is done.

Usage

Performance indicators are identified by keys. Such a key contains some basic informations about the performance indicator.

- A **unique** name (identifier) for the performance indicator.
- A (short) description of the performance indicator.
- The unit of the values the performance indicator will store.

If only a small number of performance indicators is used each can be created separately using the `PerformanceIndicatorKey` class. If a lot of indicators are used it is recommended to store them in an `Enum` implementing the `PIKeysInterface` interface. Examples can be found in the toolkit source code by searching for classes that implement the `PIKeysInterface` interface.

```
PIKeysInterface computationTimeIndicator = new PerformanceIndicatorKey( "sp_cmp_time", "Shortest path
    computation time.", "ms");
```

To access a performance indicator with the previously created key an instance of the `PILogger` is required. Such an instance can be easily created using the `PILogger.forOwner(SimulationAgent)` method. The simulation agent parameter specifies the agent the performance metric will be logged for. For example there might be multiple trucks logging their driven kilometers (using the same performance indicator key) that can only be distinguished by their corresponding agent. A `PILogger` created this way can be further split up into different contexts. For example, the toolkit's default scenario uses one agent to manage all storage facilities. In this case the key and owner are the same for every facility. To be able to distinguish them the `.withContext(String)`[19] method can be chained to the previously created logger.

```
PILogger pilogger = PILogger.forOwner(agent);
```

To finally log a value the `PILogger.takeMeasurement(PIKeysInterface, T)` method must be called on the previously created `PILogger` instance, providing the corresponding performance indicator key and the value to be logged. The values are logged once per cycle, so assigning multiple values within one cycle only gets the latest value logged. The generic class parameter `T` is set to the class of the first value logged for a performance indicator key. So if for some reason an indicator should be able to log different classes the first value must be cast to a class that all values share (and implements the `Serializable` interface).

```
long t0 = System.currentTimeMillis();
path = algorithm.getShortestPath(start, destiantion);
long t1 = System.currentTimeMillis();
pilogger.takeMeasurement(computationTimeIndicator, t1 − t0);
```

5.5.11 Reproducibility

Usually there are some random elements in any kind of scenario. For example the number of incoming orders received by a logistics company or environmental effects affecting the route a truck may take. Those values could be fixed in the source code or provided through simulation configuration attributes. But a main advantage of using a simulation is, that the user can easily run the same scenario multiple times with different randomly generated settings to achieve a wide range of results to evaluate. It is important though to ensure the reproducibility of those settings.

Pseudorandom Number Generation

Random numbers generated by a computer are usually deterministic and the corresponding generator class[20] therefore is called a pseudorandom number generator. To achieve a deterministic behaviour the `Random(long)` constructor takes a seed argument. **There is a second constructor without that seed argument, that relies on the current system time. Don't use this for scenario random number generation!** The basic seed for a scenario is provided through the **seed** attribute in the simulation configuration (see Section 4.1). This seed is accessible from every agent through `SimSystem.getRandomSeed()` and all random number generation should be based on this seed in some way.

The seed will change throughout multiple runs of the same scenario (if more than one run is specified in the scenario configuration's **repeats** attribute). The seed of a run beyond the first is always based on the previous run's seed. Therefore it is possible to repeat a specific range of runs from a complex series. If all runs should use the same seed again, a custom attribute could be used or all runs have to be treated as single runs and started with only one repeat each.

[19] Spaces, # and / characters are not allowed.
[20] See `java.util.Random`.

Creating Own Seeds Having all `Random(long)` generators use the same seed is often inconvenient. So it is necessary to generate new seeds based on the scenario's base seed.

One very important aspect to keep in mind is, that every factor influencing the random generation needs to be deterministic. If the same scenario run concludes with different sync counts it is most likely, that somewhere non-deterministic numbers are generated.

It might be handy to provide a fixed seed generation value for an agent generating its own numbers and combine that value with the base seed.

```
Random rng = new Random(SimSystem.getRandomSeed() * agent_seed_attribute);
```

Or if multiple instances of the same agent are used, provide them with some kind of `id` and use that for calculating different seeds per instance.

```
Random rng = new Random(SimSystem.getRandomSeed() * id);
```

If using Java's `hashCode()` method make sure it is adjusted to the needs of random number generation and doesn't change with each execution. It is also possible to use the agent's **local** name (use the local name, since the full name should change based on the host's name) if no other source for varying values is conveniently accessible.

```
Random rng = new Random(SimSystem.getRandomSeed() * getLocalName().hashCode());
```

Using Random Generators When using a random number generator it should be restricted to a single instance of a class to avoid concurrency problems. Also the order of method calls accessing the generator should be deterministic.

The following listing shows some examples on how to use the previously created random number generators.

```
// random is assigned an integer value between 0 and 9 (inclusive)
int random = rng.nextInt(10);

// random is assigned an integer value between 5 and 10 (inclusive)
int random = 5 + rng.nextInt(5);

// percent is assigned a float value between 0.0 and 1.0 (inclusive)
float percent = rng.nextFloat();
```

Using Collections

If the environment is generated randomly it is imported, at least for some collections, to ensure that the order is always the same. If, for example, a collection holds all trucks a logistics company manages and the company sends a message to each truck at the same time, the order might be irrelevant. But consider a collection containing all orders the company has to deliver. The order of this collection could influence the outcome of a route planning algorithm. Therefore this collection should be sorted before applying the route planning algorithm.

To be able to sort a collection it is required that it can be sorted, i. e., it should be a list[21] or be able to sort itself[22]. This might require to convert the given collection into a suitable format.

The values of the collection also need to be comparable, i. e., they have to implement the `java.lang.Comparable` interface. Alternatively (or in case a different order instead of the default ascending order is desired) a `java.util.Comparator` can be passed as (optional) second parameter.

The collection then could be sorted using the `java.util.Collections.sort()` method (or is already sorted by itself in case of a `SortedSet`).

[21] See `java.util.List`.
[22] See `java.util.SortedSet`.

```
public void planning(Collection<Order> unsortedOrders) {
    List<Order> orders = new LinkedList<Order>(unsortedOrders);
    Collections.sort(orders, new Comparator<Order>(){
        @Override
        public int compare(Order a, Order b) {
        return a.value − b.value;
        }
    });
    [...]
}
```

Equals and Hash Code

Since agent communication might involve the serialization of objects it is advisable to adapt the `equals` method. The default implementation for `Object` only returns `true` if both objects refer to the same object (which is not the case after serialization). There might also be other cases where this could be an issue, e. g., reading a string based configuration attribute and creating an object out of it that should equal other objects created with the same string, but in another class or agent.

It is also recommended to override the `hashCode` method. Though it most likely will not be an issue, it could affect the reproducibility by changing the behavior of a `HashMap` or creation of a random seed. And those kind of errors are usually hard to track down.

5.6 Time Management and Synchronization

Time management and synchronization are an integral part of the PlaSMA simulation system. The following text which has been inherited from [Geh08, pp. 34] provides an introduction with regard to these topics.

To begin with, it is necessary to distinguish different notions of time related to multiagent-based simulation (MABS). Generally, *physical time* refers to the time at which simulated events happen in the real world. *Simulation time* (or virtual time) models physical time in simulation. *Wall-clock time* refers to the time that is consumed by the simulation program executing the simulation. As simulation speed might differ between agents, each of them has its own *local virtual (simulation) time* or in short LVT [Jef90].

In contrast to equation-based modeling (cf. [PSR98]), time progression is discrete in MABS. Equidistant steps, as applied in time-stepped simulation, are hardly efficient since time steps are even processed if no events occur [Fuj00] and inadequate if time steps are too long because events are considered simultaneous although they would not be in physical time [GSW08]. By contrast, in discrete event simulation (DES) time steps bridge the gap until the next event occurs. Thus, DES is the dominant approach in most cases. Thus, PlaSMA uses a time model with discrete steps resulting from scheduled agent actions or explicit agent wake-up requests.

5.6.1 Synchronization in Simulation

Agents in MABS can be simulated on distributed platforms, which may differ regarding their computational power. Even on one platform the local virtual time of agents may diverge, if their computational demands differ. For instance, consider an agent passing a message to another agent that is advanced in its local virtual time. The recipient of such a so-called straggler message might have taken other decisions if it were aware of that message on time. This is denoted as the causality problem [Fuj00]. In order to guarantee correct and reproducible simulations, the simulation system has to ensure that agents process events in accordance to their time-stamp order.

This problem is addressed by synchronization, which can be either optimistic or conservative. In general, progression of local virtual time is not restricted for agents in optimistic synchronization. This allows executing simulations efficiently since fast processes do not have to wait for slower ones. Whenever an agent receives a straggler message with time stamp t a rollback is conducted that resets the agent to its former state at $LVT = t$. However, optimistic synchronization is more complicated in implementation and has potentially high requirements regarding space [Fuj00]. Conducting a rollback might require re-turning many steps back in time. Hence, all preceding states of every agent must be stored.

By contrast, conservative synchronization prevents causality problems by ensuring the correct order of event processing at each time. This, in turn, restricts the speedup achievable by parallelism. There are numerous approaches to conservative synchronization which are described in [Fuj00]. The synchronization mechanism is an important choice when implementing a MABS system.

5.6.2 Synchronization in PlaSMA

The PlaSMA system applies coordinated conservative synchronization with a simulation controller hierarchy. As a consequence, all active simulation agents have the same LVT resulting in a global virtual time. This approach applied by PlaSMA constrains speedup because agents may be forced to wait for other agents. One should set the scenario `minSyncDist`[23] property not too small in order to avoid such effects. Better use values greater or equal 20 seconds. Details about the synchronisation implementation are documented in [SGW08].

[23] See Section 4.1.1.

Bibliography

[BCTR07] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco, and Giovanni Rimassa. *JADE Programmer's Guide*. Telecom Italia S.p.A., June 2007.

[BvHH$^+$04] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, L. A. Stein, and F. W. Olin. OWL Web Ontology Language Reference. Available from `http://www.w3.org/TR/owl-ref/`, February 10 2004.

[Dav00] P. Davidsson. Multi Agent Based Simulation: Beyond Social Simulation. In *MABS 2000*, pages 97–107, Boston, MA, USA, 2000. Springer-Verlag.

[Fuj00] R. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley & Sons, New York, NY, USA, 2000.

[Geh08] J. D. Gehrke. Collaborative Experimentation Using Agent-based Simulation. In *Proceedings of the 2008 Workshop on Building Computational Intelligence and Machine Learning Virtual Organizations*, pages 34–38, 2008.

[GOB07] J. D. Gehrke and C. Ober-Blöbaum. Multiagent-based Logistics Simulation with PlaSMA. In *Informatik 2007. Informatik trifft Logistik. Beiträge der 37. Jahrestagung der Gesellschaft für Informatik*, number 109 in GI Proceedings, pages 416–419, Bremen, Germany, September 24–27 2007. Gesellschaft für Informatik (GI).

[GS09] J. D. Gehrke and A. Schuldt. Incorporating Knowledge about Interaction for Uniform Agent Design for Simulation and Operation. In *8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 1175–1176, Budapest, Hungary, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[GSW08] J. D. Gehrke, A. Schuldt, and S. Werner. Quality Criteria for Multiagent-Based Simulations with Conservative Synchronisation. In *13th ASIM Dedicated Conference on Simulation in Production and Logistics (ASIM 2008)*, pages 177–186, Berlin, Germany, 2008. Fraunhofer IRB Verlag.

[Jef90] D. R. Jefferson. Virtual Time II: Storage Management in Conservative and Optimistic Systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing (PODC 1990)*, pages 75–89, Quebec, Canada., August 22–24 1990. ACM Press.

[PSHH04] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. Available from `http://www.w3.org/TR/owl-semantics/`, February 10 2004.

[PSR98] H. V. D. Parunak, R. Savit, and R. L. Riolo. Agent-Based Modeling vs. Equation-Based Modeling: A Case Study and Users' Guide. In *Multi-Agent Systems and Agent-Based Simulation, First International Workshop (MABS 1998)*, volume 1534 of *Lecture Notes in Computer Science*, pages 10–25, Paris, France, July 4–6 1998. Springer.

[SGW08] A. Schuldt, J. D. Gehrke, and S. Werner. Designing a Simulation Middleware for FIPA Multiagent Systems. In *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2008)*, pages 109–113, Sydney, Australia, 2008. IEEE Computer Society Press.