Working Paper

Coupling MATSim and UrbanSim: Software design issues

Thomas W. Nicolai, Kai Nagel

TU Berlin, Germany Revision: 1

FP7-244557 20/12/2010

Contents

Coupling MATSim and UrbanSim

Thomas W. Nicolai, Kai Nagel
TU Berlin
Salzufer 17-19
10587 Berlin Germany

Teleph.: +49 30 314 29592
Telefax: +49 30 314 26269
nicolai@vsp.tu-berlin.de

20/12/2010

Abstract

This paper proposes different methods to couple UrbanSim and MATSim. The integration of MATSim into UrbanSim aims to provide improved accessibility indicators from the (agent-based) transport model that for instance incorporates congested home to work commute trips in order to improve the urban simulation model.

The paper is organized as follows. The next section introduces the current integration approach. In section 2 pitfalls of the integration process and new methods and concepts are presented. The proposed methods and concepts are discussed in the following section. Finally section 4 summarizes the results and provides a conclusion.

Keywords

UrbanSim, MATSim, Integration, Python, Java, XML, XSD, Data-Binding, PyXB, JAXB, JNI, JEPP, CPython, Jython, JPype, Ja.Net, J#

1 Introduction

This section aims at describing an already existing prototype approach integrating MATSim into UrbanSim and its drawbacks.

UrbanSim simulates inter-relations between land use, transportation, the economy and the environment; it is a tool for the integration of several models aimed at the simulation of urban development. In order to improve simulation results, MATSim, a microscopic, multi-agent based traffic model, is integrated with UrbanSim. For more detailed information about UrbanSim and MATSim, please see UrbanSim Manual (2009) and MATSIM www pages (accessed March 2010).

At this point an overview of the prototype integration approach and the consequent interaction between UrbanSim and MATSim as an external travel model is presented. The following steps are executed iteratively for each UrbanSim simulation year; in the following this is called ”(UrbanSim) simulation year” (see also Figure 1):

1. **UrbanSim initialization**: For each simulation year UrbanSim needs a “current” data set in order to perform the update. Either it gathers the data set from a given base year – this is the case for the first (=initial) iteration – or from the previous iteration. In the data set the initial state of an urban area for the current iteration is stored. It incorporates several tables like a person-, household-, building and job-table for instance. These tables define the relations between persons, households and jobs as well as coordinates of residences and workplaces for example.

2. **Demand generation**: In order to run MATSim as an external model UrbanSim creates two input files that include the distribution of persons and their workplaces on a parcel level extracted from the data set. Additional resources like the location of the traffic network data are provided via a separate MATSim configuration file.

3. **MATSim run**: After executing the traffic simulation MATSim writes the computed accessibility indicators into an output file. UrbanSim reads this file and joins the accessibility indicators into its own data set for the next iteration.

4. **Next UrbanSim iteration:** Resume with step 1, otherwise shutdown.

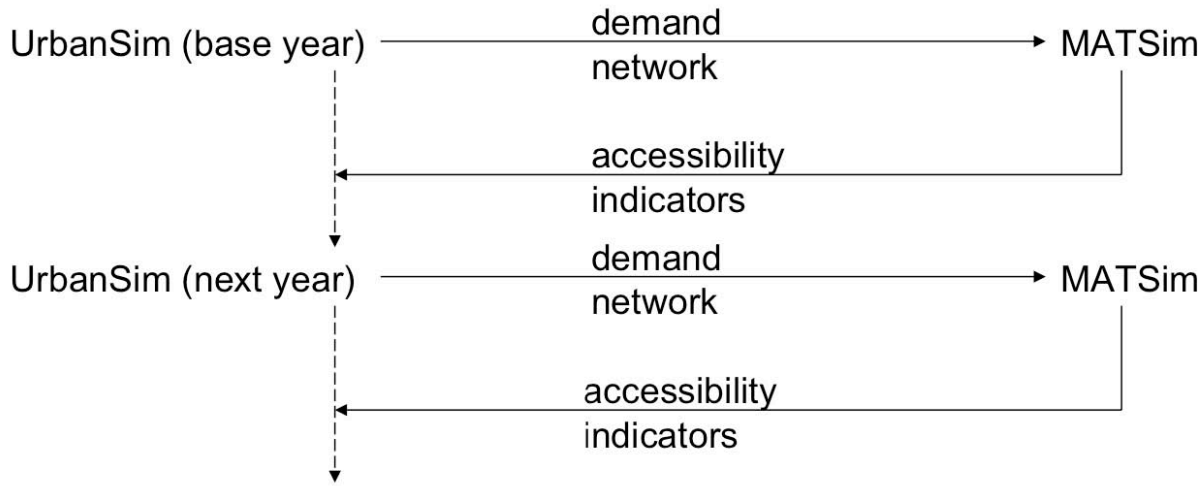

Figure 1: Interaction between UrbanSim and MATSim.

The prototype approach uses a file-based communication between UrbanSim and MATSim, through generating the travel demand stored as files for MATSim and importing the resulting accessibility indicators in UrbanSim. The locations of input and output files, the attributes or indicators of these files (like person id and respective home and work location) as well as the MATSim configuration file are hard coded in both simulation frameworks. Besides, both frameworks have their own configuration file that both need to be readjusted in order to run different scenarios or case studies. Hence the following drawbacks of this prototype approach can be stated:

1. **Hard coded**: Hard coded parts like the in- and output locations or the fixed defined indicators to calculate make it impossible for users to change or adopt the settings for their own needs without programming in Java or python or both. This results in an inconvenient and inflexible setup of both frameworks in order to run them correctly on different computers and to use them for various case studies.

2. **Separate configuration files**: Maintaining or adopting two separate configurations at different locations leads to an inconvenient and error-prone setup in order to run both frameworks with a meaningful and correct configuration.

3. **Extensibility**: Finally it is difficult to add new functionality or to improve this integration approach.

Because of these drawbacks this paper attempts to find an alternative, more robust and flexible approach coupling UrbanSim and MATSim. Promising integration approaches are presented in the next section.

2 Introducing new methods and concepts to integrate UrbanSim with MATSim

As mentioned in section 1, the fragility of the current software integration is caused by the inconvenient handling of the set of configuration files that are distributed on different locations and the hard coded references in both simulation frameworks, e.g. to the input and output files. That leads easily to inconsistent configurations when references change.

Therefore a first step is to centralize the configuration for a more convenient adjustment, to reduce the user's maintenance burden. The UrbanSim side is a perfect place to manage such a centralized configuration, because it embeds MATSim as a plug-in. A straightforward way to achieve a centralized configuration is to embed necessary MATSim parameter settings into the travel model configuration section of the UrbanSim configuration (see Figure 2) and to extend the UrbanSim data processing in order to handle its new configuration part. Necessary MATSim parameters are for example the location of input-files like the traffic network, or settings like the sampling rate. It is needless to say that the embedded configuration must adopt the syntax of the UrbanSim configuration.

```
<sampling_rate type="float">0.01</sampling_rate>
<matsim_config_filename type="file">matsim_config/seattle_matsim_0.xml</matsim_config_filename>
<years_to_run key_name="year" type="category_with_special_keys">
  <run_description type="dictionary">
    <year type="integer">2001</year>
  </run_description>
  <run_description type="dictionary">
    <year type="integer">2002</year>
  </run_description>
</years_to_run>
```

Figure 2: Travel model configuration section that contains e.g. the sample rate and the actual years to run.

In order to run MATSim, UrbanSim needs to pass the MATSim configuration settings and the input files like the distribution of persons and facilities to MATSim. There are several ways to realize that. This paper identifies two main concepts:

1. **File-based coupling**: Like in the current file-based approach, UrbanSim still generates input data, but moreover it creates the MATSim configuration file. It is helpful to validate the configuration file in order to achieve a robust and reliable file-based communication, for example to avoid unintended side effects like a system crash. The validation incorporates a syntax and parameter type (string, int, float) check. A detailed explanation is given below in section 1.1.

2. **Object-based coupling**: A file-based communication is rather old fashioned. It is desirable to communicate directly and bi-directionally between UrbanSim and MAT-

Sim, so that UrbanSim will be able to access MATSim classes and its methods directly like other UrbanSim classes. A description how to apply such an object-based communication is presented below in section 1.1.

The next parts of this section are organized as follows: First the file-based approach is introduced. Then a couple of methods and projects to realize the object-based approach are shown.

2.1 File-based coupling via data binding

Once we have a centralized configuration-file, written by UrbanSim and readable by MATSim, that incorporates all parameter settings from the transport model section of the UrbanSim configuration, we are able to configure and run MATSim properly.

Technically it is not very difficult to generate such a configuration in XML[1] and to extend UrbanSim and MATSim to process an XML document using APIs[2] like SAX[3] and DOM[4], but it is error-prone and costly to build and maintain such extensions, especially when the parameters or the structure of the configuration changes. In the context of a robust and reliable communication, this consideration leads to two main requirements for a centralized configuration:

1. A mutual consent between UrbanSim and MATSim regarding the structure of the configuration file is crucial. In other words a validation of the XML is essential (see Figure 3).

2. In case of changes regarding the structure of the configuration, e. g. when a new parameter is added, the XML processing modules and classes in UrbanSim and MATSim needed to be adopted automatically.

XML data binding fulfils both requirements. It is a technique to link different applications by transforming XML documents into an object of a desired programming language. This allows the transformation of a Python-object in UrbanSim into a Java-object on the MATSim side via XML. This is similar to pass a configuration-object from UrbanSim to MATSim.

[1] Extensible Markup Language

[2] Advanced Programming Interface

[3] Simple API for XML

[4] Document Object Model

The mutual consent about the structure of an XML-file and the data type definitions of its elements and attributes (like string, boolean, float) is achieved by an associated XSD[5]. An XSD is an abstract collection of metadata about an XML document. It is a "formalization of the constraints, expressed as rules or a model of structure [...]" (van der Vlist (2002)). In other words, an XSD acts like a validation checkpoint between different programmes exchanging XML-documents (see Figure 3). A more comprehensive description of XSDs can be found at World Wilde Web Consortium www pages (Accessed June 2010).

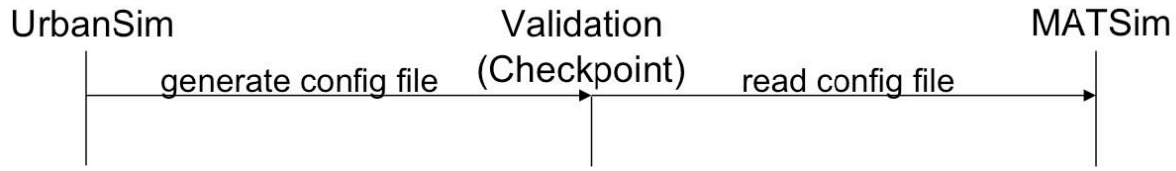

Figure 3: Validation of the generated XML MATSim configuration file in order to provide robust file-based communication.

The second element, updating XML processing modules and classes in UrbanSim and MATSim, is achieved by additional software packages PyXB and JAXB. PyXB is an acronym for Python W3C XML Schema Bindings. It is needed on the UrbanSim side to generate so called binding classes. Analogously, JAXB generates binding classes on the MATSim side; it stands for Java Architecture for XML Binding. These binding classes are like object templates, following the same XSD specification like the XML-file. Performing changes in the configuration and hence in the related binding classes is simple. For this purpose the XSD needs to be adapted to the new requirements, and PyXB and JAXB needs to generate the binding classes again.

Once UrbanSim creates an instance of this binding class and fills it with the extracted MATSim settings from the transport model section of the UrbanSim configuration, this object instance will be transformed into a XML representation and written to file. When starting MATSim, UrbanSim passes the location of the XML-file as a programme argument to MATSim. MATSim validates the XML via the XSD and instantiates its own binding classes, filled by the settings stored in the XML.

This data binding approach allows quick and convenient changes of the XML configuration structure if required. Moreover it provides a robust and reliable file-based communication be-

[5] XML Schema Document

tween UrbanSim and MATSim.

Notes: For sake of completeness “generateDS”, another software package for Python should be briefly introduced. “generateDS” stands for Generate Data Structures from XML Schema. It was also tested as an alternative for PyXB, but since the generated XML-files did not contain any XML-header it was not investigated in more detail.

For more detailed information on PyXB, JAXB and generateDS please refer to the project websites (PyXB www pages (Accessed June 2010); JAXB www pages (Accessed June 2010); generateDS www pages (Accessed June 2010)).

2.2 Object-based coupling

As mentioned above, it is desirable to communicate directly on an object-based-level between UrbanSim and MATSim. This would enable UrbanSim to generate the configuration and input data as objects and pass them as arguments to MATSim. When MATSim finishes the simulation run, it would pass the computed results back to UrbanSim as another object.

The reader may recall that UrbanSim and MATSim are implemented in different programming languages; Python and Java. At this point it is useful to highlight that both programming languages are very different. Some notable constraints for interaction at the object level are listed as follows: The most important constraint is the different and incompatible byte code representation. Java programmes are translated into Java byte code that is executed by the Java Virtual Machine (JVM). The reference implementation Python is written in C, called CPython. Analogue to JVM CPython compiles Python source code into Python byte code. (For a discussion of Java-based implementations of Python see below.) Another distinction is that Java employs static typing where type checking is performed during compile-time as opposed to run-time in dynamic typing languages like Python. As a last example Java provides basic data types like int, float, double, char and boolean in contrary to Python where even basic data types are objects.

Despite these and other unnamed constraints Python and Java can still communicate and work with each other. At this point two interfaces, the Java Native Interface (JNI) and JPype, are outlined that satisfy the following requirements:

1. Java to Python integration.
2. Interoperability, since UrbanSim and MATSim are used on different platforms (Windows, Mac, Linux).

For the sake of completeness also some other related projects that do not fit our requirements are presented afterwards.

2.2.1 Java Native Interface (JNI)

The JNI is a native programming interface for Java programmes. It allows Java code that run inside a JVM to interoperate with applications and libraries written in other programming languages (see Figure 4). In order to call native applications and libraries out of the JVM it is required to implement an additional software layer in C. This additional software layer causes some extra maintenance effort. This can be easily seen in the following steps summarizing the process to write a simple Java application that prints a message on the screen via a C programme (for more information refer to Liang (1999)):

1. Create a Java class that declares a native method.

2. Compile this class.

3. Create a C header file that defines the interface for the C implementation (generated by the javah compiler).

4. Write the C implementation of the native method. The implementation must follow the header file.

5. Compile the C implementation into a native library.

6. Run the Java application. Both the java class and the native library are loaded at runtime and can be executed.

Interfacing Java with the Python interpreter (CPython) would needs much more work implementing this software layer, e.g. to consider a meaningful handling for different data types. This implementation alone would be a large project for itself. It is partly solved by several projects presented in the following sections.

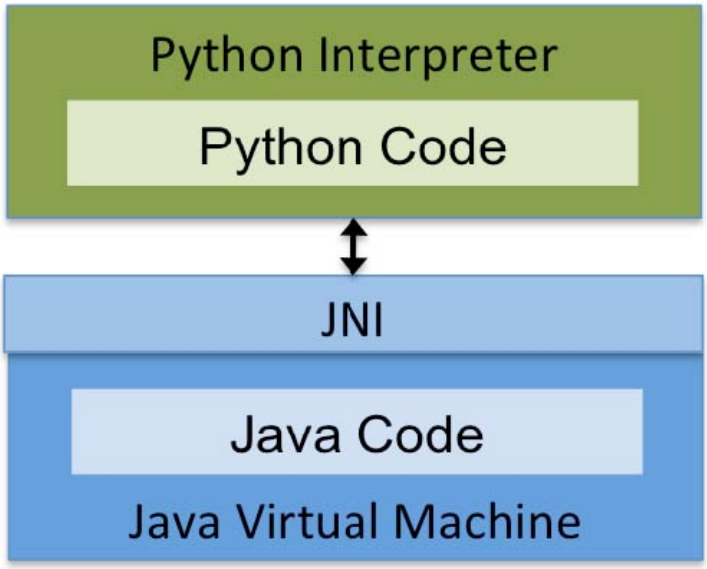

Figure 4: Communication between CPython and the Java Virtual Machine via JNI.

2.2.2 JPype

The JPype project allows Python full access to Java class libraries. This is achieved through interfacing the Python interpreter (CPython) and the JVM at the native level using JNI and PNI[6] (see Figure 5) (JPype www pages (Accessed June 2010)). So the extra development effort to implement an own software layer like for JNI is no longer necessary. Two great benefits of JPype are that it is available for prevalent platforms like Windows, Mac and Linux, and it is convenient to employ. Figure 6 and Figure 7 illustrate a simple Python application that prints a string message in Java. To do so, it is necessary to import JPype into a Python module and to open and to close the connection to the JVM via "startJVM" and "shutdownJVM". In order to call the Java methods "printOut", "setString" and "getString" of our Java test class (see Figure 7) it is necessary to set the class path where the desired Java classes are stored. Then Python accesses the "test" Java package (via "jpype.JPackage('test')") in order to create an instance of the Java "Test" class.

However, some limitations should be considered using JPype, e.g. the type conversion while translating between Python and Java. At this point two short examples regarding the limitations of the type conversion are given. A detailed overview can be found in the JPype user guide (JPype User Guide (Accessed June 2010)).

JPype converts a Python object like an "int" into the Java native types of byte, short or int if the value fits. Since Java allows overloading a method (Python does not allow that), JPype is

[6] Python Native Interface

sometimes unable to decide which method to call. To convert a Python value explicitly, JPype provides wrappers like JByte, JShort, JInt et cetera. But in result the developer is responsible to resolve those ambiguities.

Another example: The wrapper class “JArray” is used in Python to receive Java arrays, or to pass them to Java methods. While creating such a wrapper object, the array type, the number of dimensions and the actual number of elements in the array are needed. Again the developer is accountable to provide the correct number of dimensions as well as the data type of the JArray that should match the declared data type of the Java method. Effectively, this means that the important convenience of resizeable arrays that has, in recent years, entered high level programming languages, could not be used for Python–Java data interchange.

Those limitations should be kept in mind while creating even more complex objects.

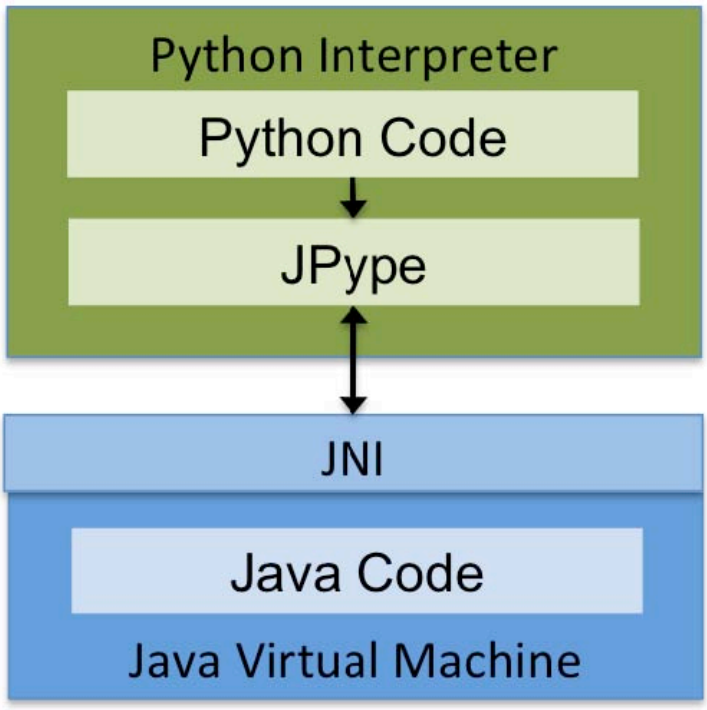

Figure 5: Java to Python integration. Adopted from Schreiber (2009).

```
import jpype
import os.path

classpath = os.path.join('/path/to/java/classes')   # set class path
jpype.startJVM(jpype.getDefaultJVMPath(), "-Djava.class.path=%s" % classpath)
testPkg = jpype.JPackage('test')                    # get the package
Test = testPkg.Test                                 # get the class
t = Test()                                          # create an instance of the class
t.printOut("This is a test message")                # call the printOut method
t.setString("Hello World")                          # set a string
s = t.getString()                                   # get the string back
print s                                             # show the received string (it contains 'Hello World")
jpype.shutdownJVM()                                 # close JVM connection
```

Figure 6: Python application using Java via JPype.

```
package test;

class Test {
    private String msg; // stores a message from the python program

    // prints a message on screen
    public void printOut(String msg) {
        System.out.println(msg);
    }
    // stores a string
    public void setString(String s) {
        msg = s;
    }
    // returns the stored string
    public String getString() {
        return msg;
    }
}
```

Figure 7: Ordinary Java class with public methods.

2.2.3 Other object-based integration methods

In this subsection we present four further methods to link Python and Java. These are JEPP, Jython, Ja.Net and J#. These methods do not fit our constraints to integrate MATSim into UrbanSim; for example, they support the wrong calling direction from Java to Python. They are summarized for sake of completeness. The utility of these methods is discussed in the following section.

JEPP is an acronym for Java Embedded Python. It embeds the CPython interpreter in Java via JNI (see Figure 8) and allows Java to control Python applications, to evaluate Python statements and to execute Python scripts. In contrast to JPype JEPP only allows Java to invoke CPython. But that is the opposite calling direction of what we need since UrbanSim (in Python) starts MATSim (in Java). For more information about JEPP refer to the JEPP www pages (Accessed June 2010).

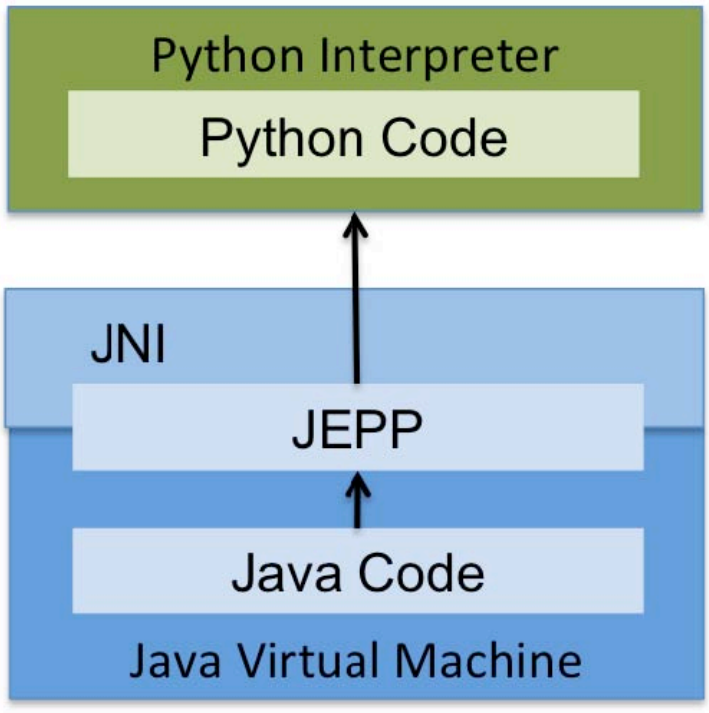

Figure 8: JEPP embeds CPython via JNI in Java. Adopted from Schreiber (2009).

Jython (formally known as JPython) is another production-quality Python implementation like CPython. Jython is a pure Java implementation of the Python interpreter. It consists of a

Python compiler that compiles Python source code to Java byte code, which runs on a JVM (see Figure 9). Jython allows to mix and match Python code and Java code, and to use Python to script Java applications. Also, Python can be used from Java applications. Moreover Jython can be used interactively like the Python command shell. Almost all modules of the standard Python language, which are originally implemented in C, are part of Jython, except for the standard modules to create user interfaces. Those must be written in Java or more precisely in Swing[7], AWT [8]or SWT[9] (Jython www pages (Accessed June 2010); Pedroni and Rappin (2002)).

Our intention while investigating Ja.Net and J# is to find out if they are applicable to couple Python and Java via coupling the .NET runtime from Microsoft with the PNI. Ja.Net is an open source project that implements the Java programming language for the .NET runtime environment. J# was released by Microsoft as a transitional language for Java programmers to introduce the .NET development environment. It can be also used to translate existing Java applications to .NET even if their original source code is not available (MSDN www pages (Accessed July 2010); Ja.NET www pages (Accessed June 2010)).

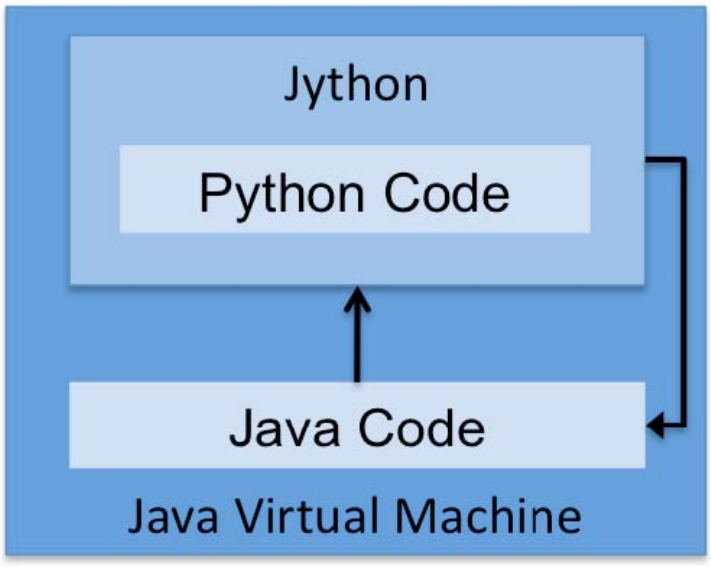

Figure 9: Jython is a pure Java implementation of the Python interpreter. Adopted from Schreiber (2009).

[7] Swing is a toolkit to create a graphical user interfaces (GUI) in Java.
[8] The Abstract Windowing Toolkit, or AWT, is the predecessor of Swing.
[9] In contrast to AWT and Swing the Standard Widget Toolkit uses native widgets to invoke platform specific features to build user interfaces.

3 Discussion

In this section the presented solutions from the previous section are discussed.

To recall our aims while integrating MATSim into UrbanSim the main requirements are repeated here:

- Achieve a more robust integration of MATSim into UrbanSim.
- Achieve a more convenient and less error-prone configuration of MATSim and UrbanSim.

Since centralizing the configuration handling as explained above solves the latter point, our focus in this section lies on the integration part.

The file-based approach combines many advantages. It provides a robust and reliable file-based communication between UrbanSim and MATSim through data binding via PyXB and JAXB and the validation via XSD. The concept of a file-based communication is easy to understand, it is platform-independent and no additional programming languages are needed. In contrast, object-based approaches suffer from an increased software complexity that implies either an increased implementation and maintenance effort or at least a lower robustness.

For example, the main handicap of JNI is to implement and maintain additional native methods that allow Java applications to call functions implemented in native applications and libraries. These native methods can be seen as an additional software layer. Since this software layer is written in native code it must be adopted and at least re-compiled in order to run on different platforms. This causes an increased effort while extending, improving and testing the cooperation between UrbanSim and MATSim compared to the status quo. Furthermore in order to implement the native methods an additional programming language besides Python and Java is needed.

Reducing the implementation and maintenance effort is one of the goals of the JPype project. Like the native methods in JNI, JPype can be considered as an additional software layer. Unlike JNI it is more convenient to use and does not lead to an extra implementation and maintenance effort, e.g. it does not need be adapted to different platforms by the developer or user. But due to the translation process on the native level between Python and Java the software complexity still increases. In general an increasing software complexity leads to lower ro-

bustness. Another disadvantage of JPype is the loose documentation that contains several spelling mistakes.

The following methods are not applicable to integrate MATSim into UrbanSim. JEPP only allows Java to invoke CPython. Unfortunately, that is the opposite calling direction. Jython would be a very interesting way to couple Java and Python, but it lags behind CPython regarding processing speed and even more importantly it does not allow access to most Python extensions like numpy or scipy. These extensions are extensively used in UrbanSim and are only available for CPython. Using Ja.Net and J# to couple Python and Java through translating Java into .NET does not seem to be feasible. To name only two problems: First .NET is designed for Windows. It is a moot question whether .NET applications run in a stable way on Mac or Linux, e.g. with mono (see Mono www pages (Accessed June 2010)). It is also questionable if MATSim can be translated without trouble with all its additional libraries into .NET since J# is not supported any more.

4 Conclusion

Regarding the presented object-based coupling methods JPype can be seen as the most useful approach for our purposes. First of all it is applicable for prevalent platforms like Windows, Mac and Linux. Compared with JNI it relieves the user from implementing native methods. In addition, it is convenient to deploy. Since we plan to exchange only simple objects between UrbanSim and MATSim, limitations concerning different object types are negligible. But the loose documentation could be a bottleneck in the implementation process, and the large number of spelling mistakes on the website and the documentation makes JPype look immature. JPype is a promising project, but regarding the status quo it cannot be recommended as the standard way to couple UrbanSim and MATSim.

Therefore we believe that right now our file-based coupling method via data binding (XSD/PyXB/JAXB) is the best way to foster a stable and reliable integration of MATSim into UrbanSim.

5 References

GenerateDS www pages (Accessed June 2010) generateds – generate data structures from xml schema, http://www.rexx.com/~dkuhlman/generateDS.html.

Ja.NET www pages (Accessed June 2010) Ja.NET SE – Java 5 JDK for .NET, http://www.janetdev.org/.

JAXB www pages (Accessed June 2010) Java Architecture for XML Binding (JAXB),http://java.sun.com/.

JEPP www pages (Accessed June 2010) Jepp – java embedded python, http://jepp.sourceforge.net/.

JPype User Guide (Accessed June 2010) JPype 0.4 – User Guide, http://jpype.sourceforge.net/doc/user-guide/userguide.html.

JPype www pages (Accessed June 2010) JPype – Java to Python integration, http://jpype.sourceforge.net/.

Jython www pages (Accessed June 2010) The Jython Project, http://www.jython.org/.

Liang, S. (1999) *The Java Native Interface. Programmer's Guide and Specification*, first printing, ADDISON-WESLEY, June 1999.

MATSIM www pages (Accessed March 2010) Multi Agent Transport SIMulation, http://matsim.org/.

Mono www pages (Accessed June 2010) Cross platform, open source .net development framework, http://www.mono-project.com/.

MSDN www pages (Accessed July 2010) Microsoft Development Network, http://msdn.microsoft.com/.

Pedroni, S. and N. Rappin (2002) *Jython Essentials*, first edition, O'REILLY, March 2002.

PyXB www pages (Accessed June 2010) PyXB: Python XML Schema Bindings, http://pyxb.sourceforge.net/.

Schreiber, A. (2009) Mixing Python and Java. How Python and Java can communicate and work together, http://elib.dlr.de/, June 2009.

UrbanSim Manual (2009) *UrbanSim Manual*, Center for Urban Simulation and Policy Analysis University of Washington, University of Washington, UrbanSim Version 4.2, 2009.

van der Vlist, E. (2002) *XML Schema*, first edition, O'REILLY, June 2002.

World Wilde Web Consortium www pages (Accessed June 2010) World Wilde Web Consortium, http://www.w3.org/.