OPUS: AN OPEN PLATFORM FOR URBAN SIMULATION

Paul WADDELL
Director
Center for Urban Simulation and Policy Analysis
University of Washington
Box 353055
Seattle, Washington 98195
USA
Tel: +206 221 4161
Fax: +206 616 6625
E-mail: pwaddell@u.washington.edu

Hana ŠEVČÍKOVÁ
Researcher
Center for Urban Simulation and Policy Analysis
University of Washington
Box 353055
Seattle, Washington 98195
USA
Tel: +206 616 4495
Fax: +206 616 6625
E-mail: hanas@stat.washington.edu

David SOCHA
Software Manager
Center for Urban Simulation and Policy Analysis
University of Washington
Box 353055
Seattle, Washington 98195
USA
Tel: +206 616 4495
Fax: +206 616 6625
E-mail: socha@cs.washington.edu

Eric MILLER
Professor
Department of Civil Engineering
University of Toronto
35 St. George St.
Toronto, Ontario, M5S 1A4
Canada
E-mail: miller@civ.utoronto.ca

Kai NAGEL
Professor
Institut für Land- und Seeverkehr
Fachgebiet Verkehrssystemplanung und Verkehrstelematik, Sekr. SG 12
Salzufer 17-19, D-10587 Berlin
Germany
E-mail: nagel@vsp.tu-berlin.de

Abstract: Several research teams working on integrated land use, transportation and environmental modelling have begun an international collaboration to develop an Open Platform for Urban Simulation (OPUS). The initiative is creating an Open Source platform that simulates land use, activity-based travel demand, and dynamic traffic assignment, and that can be extended by users and adapted to alternative modelling applications. This paper summarizes the objectives and design of OPUS, and describes a framework for extending the system through packages contributed by the user community.

Keywords: microsimulation, open source, land use, activity-based travel, network assignment

OPUS: AN OPEN PLATFORM FOR URBAN SIMULATION

1 INTRODUCTION

Research in transportation modelling and planning has made significant advances over the past decade along four axes of innovation, within a unifying theme of modelling land use and transportation choices at the individual level.

The first of these axes is in *integrated land use and transportation modelling*, motivated by the need to assess major transportation investments and their effects on urban development and on the environment. Over the past decade, several research projects have been developing new platforms for integrated land use and transportation modelling, including ALBATROSS (Arenze and Timmermans, 2000), ILUTE (Miller and Salvini, 2003), and UrbanSim (Waddell, 2002; Waddell *et al.* 2003).

The second axis of innovation is in the development of *activity-based travel modelling*, motivated by shortcomings in the design of traditional four-step travel models and the need to improve the behavioural realism of travel demand models. Within this body of work, several activity-based travel demand model systems have been developed, including CEMDAP (Bhat *et al.* 2003), and FAMOS (Pendyala et al, 2004).

The third axis of innovation is in *dynamic traffic assignment models*, using both mesosopic and microscopic approaches to better reflect dynamic traffic conditions. Recently developed dynamic assignment platforms exemplifying these innovations include MATSIM (MATSIM, 2005) and METROPOLIS (de Palma *et al*., 1997).

Finally, the fourth axis of innovation is in the development of *increasingly sophisticated discrete choice models and the tools with which to estimate them with*. Advances in choice modelling include numerous generalizations of the multinomial logit model and the development of simulation-assisted estimation methods (Train,2003), and the generalization of discrete choice models to incorporate more explicitly the generation of the choice set, the integration of revealed and stated-preference information, and the use of latent constructs in choice model formulation. New software packages such as Biogeme (Bierlaire *et al*., 2004) that implement these advances and make them accessible to modelers have been emerging, and commercial econometric software have been rapidly incorporating these methods.

Given the rapid pace of innovation in these four areas, it is no surprise that few standards have emerged. Due to a combination of absence of common standards and architectures for efficient interoperation, creating complete systems for use in research and planning applications by coupling models together is often a very difficult and inefficient process. Yet researchers and users of these models need to do precisely this in order to pursue their respective research and planning agendas. Some of the inefficiency is due to the need to couple models ‘loosely, by inefficient data exchange methods. Other inefficiencies arise due to incompatibilities of software languages, data

formats, or to proprietary restrictions on access to the internal data structures and algorithms of coupled components.

Several research teams spanning these four innovation areas have launched an international collaboration to develop an Open Platform for Urban Simulation (Opus). The broad vision for the effort is to develop a robust, modular and extensible open source framework and process for developing and using model components and integrated model systems, and to facilitate increased collaboration among developers and users in the evolution of the platform and its applications. This paper provides the first description of this initiative, its key design elements, its current development status, and plans for its further development.

The remainder of the paper is organized as follows. Section 2 describes the objectives guiding the design of Opus, and the initial design and implementation of the Opus architecture. Section 3 addresses issues related to the specification and implementation of models in the Opus framework. Section 4 concludes with an assessment of the current status and future research and development priorities, and an invitation to participate in the development of Opus.

2 Opus Design

Opus is motivated by lessons learned from urban simulation projects, in particular from the ILUTE, UrbanSim and MATSIM projects, and by a desire to collaborate on a single platform so that projects can more easily leverage each other's work and focus on experimenting with and applying models, instead of spending their resources creating and maintaining model infrastructures.

A similar project that provides inspiration for the Opus project is the R project (www.r-project.org). R (Ihaka and Gentleman 1996) is an Open Source software system that supports the statistical computing community. It provides a language, basic system shell, and many core and user-contributed packages to estimate, analyze and visualize an extremely broad range of statistical models. Much of the cause for the rapidly growing success of this system, and its extensive and actively-contributing user community, is due to the excellent design of its core architecture and language. It provides a very small core, a minimal interactive shell that can be bypassed completely if a user wants to run a batch script, and a set of well-tested and documented core packages. Equally importantly, it provides a very standard and easy way to share user-contributed components. Much of the Opus architecture is based upon the R architecture.

In addition, the success of the R project motivated our driving vision for Opus:

> *To realize the transformative potential for the land use, transportation and environmental modelling and planning communities from collaboratively developing a shared platform that is well-designed for the relevant tasks, easy to use, easy to extend, and easy to share models, data, and results.*

The design of the core Opus architecture draws heavily on the experience of the UrbanSim project in software engineering and management of complex open source software development projects, and in the usability of these systems by stakeholders ranging from software developers, to modelers, to end-users.

The high-level design goals for Opus are to create a system that is very:

- Productive – to transform what users can do
- Flexible – to support experimentation
- Fast and scalable – to support production runs
- Straightforward – so users will create many more Opus packages
- Sharable – to benefit from others' work

In other words, to fulfil our vision we need to design a system whose parts have a particular set of qualities, some of which may seem a bit at odds. The Opus system must (in no particular order):

- Have a low cost for developing new models. It should be easy for modelers around the world to code, test, document, package, and distribute new models. And it should be easy for modelers to download, use, read, understand, and extend packages created by others.
- Make it easy to engage in experimentation and prototyping, and then to move efficiently to production mode and have models that run very quickly.
- Have an interactive command-line interface so that it is easy to explore the code, do quick experiments, inspect data, etc.
- Be flexible so that it is easy to experiment with different combinations of parts, algorithms, data, and visualizations.
- Make it easy to inspect intermediate data, in order to aid the often complex diagnosis of problems found in large-scale production runs.
- Be extensible so that users can modify the behaviour of existing models without modifying the parts being extended, or build new models from existing parts, or replace existing parts with others that provide the same services in a different way.
- Be easy to integrate with other systems that provide complementary facilities, such as estimation, data visualization, data storage, GIS, etc.
- Be scriptable so that it is straight forward to move from experimentation or development into the mode of running batches of simulations.
- Run on a variety of operating systems, with a variety of data stores (e.g. databases).
- Handle data sets that are significantly larger than available main memory.

- Make it easy to take advantage of parallel processing, since much of the advances in chip processing power will come in the form of having multiple 'cores' on a single chip.
- Provide an easy mechanism for sharing packages, so that people can leverage each others work.
- Provide a mechanism for communities of people to collaborate on the creation and use of model systems specific to their interests.

The following sub-sections describe some of the key motivations for these goals and strategies for attaining them.

2.1 Python as the Base Language

One of the most important parts in the system is the choice of programming language on which to build Opus. This language must allow us to build a system with the above characteristics.

After considering several different languages (C/C++, C#, Java, Perl, Python, R, Ruby) we choose Python for the language in which to implement Opus. Python provides a mature object-oriented language with automatic garbage collection. Python's support for reflection (the ability of a program to query what classes exist and what methods each of these classes has) makes it much easier to create extensible and flexible systems. Its support for lambda functions provides powerful and flexible ways to extend functionality at run-time.

Python has a concise and clean syntax that results in programs that generally are 1/5 as long as comparable Java programs. In addition, Python has an extensive set of excellent open-source libraries. Many of these libraries are coded in C/C++ and are thus are very efficient. Numarray, for instance, is an open-source Python library containing a wide variety of useful and extremely fast array functions, which we use throughout Opus to provide high performance for large data sets. There are also several mechanisms for 'wrapping' other existing packages and thus making them available to Python code.

Python is an interpretive language, which makes it easy to do small experiments from Python's interactive command line. For instance, we often write a simple test of a numarray function to confirm that our understanding of the documentation is correct. It is much easier to try things out in Python, than in Java or C++, for instance.

At the same time, Python has excellent support for scripting and running batch jobs, since it is easy to do a lot with a few lines of Python, and Python 'plays well' with many other languages. The run-request management system created by the UrbanSim project, for instance, uses Python to service run-requests logged to a database. For each run request, the system invokes a combination of Python, Java, and emme/2 models to simulate for a 30 year simulation that takes several days of compute time (mostly in the emme/2 assignment step). We have found that Python fulfills this role much better than Perl or Java.

Python's ability to work well for quick experiments, access high-performance libraries, and script other applications means that modelers need only learn one language for these tasks.

Opus extends the abstractions available in Python with domain-specific abstractions useful for urban modelers, as described below.

2.2 Opus Architecture

Figure 1 shows the basic Opus architecture. Much of this is now implemented and running. As described above, this basic architecture is modelled on the R architecture. However, instead of building our own interpreted language, as R did, Opus code is simply Python code that follows some standards. This allows us to leverage the development activities in the dynamic and growing Python community, and avoid creating our own mechanisms for the many things that are already done well there.

The standards that define Opus relate to how to package functionality, and how to construct new functionality by assembling or modifying existing functionality.

The basic unit of distribution for Opus functionality is an Opus package, which is a Python package that conforms to a few specific rules. Each Opus package provides a particular type of functionality, and is contained in a directory of the same name as the Opus package. This directory contains a set of required directories and files (e.g., docs, examples, tests) that form explicit guidelines of good usage. The tests directory, for instance, must contain a Python file, all_tests.py, that runs all of the automated tests shipped with this Opus package. As long as these rules are adhered to, the author of the Opus package may add other directories and files as suits their needs.

Python code in Opus can refer to the contents of another Opus package via the fully-qualified Python name for that module, e.g. opus.urbansim.households, Using fully qualified names eliminates name collisions, so that two different user-defined Opus packages can both include a class of the same name without causing confusion in the system.

Figure 1 shows Opus packages containing Python classes. This figure focuses on the two Opus packages required for UrbanSim models:

- core – this package is the heart of Opus. It contains the underlying abstractions for the basic building blocks of a modelling system. These include models, variables, datasets, coefficients, specifications, and data stores. Each dotted circle connects an abstraction shown in a rounded-cornered box, such as 'model', with the family of classes that are specialized versions of that abstraction.
- urbansim – this package contains versions of the core abstractions that are specialized for running the set of models required for UrbanSim. Figure 1 shows just a few of the actual classes defined in urbansim.

All of the classes shown in the core and urbansim packages exist today.

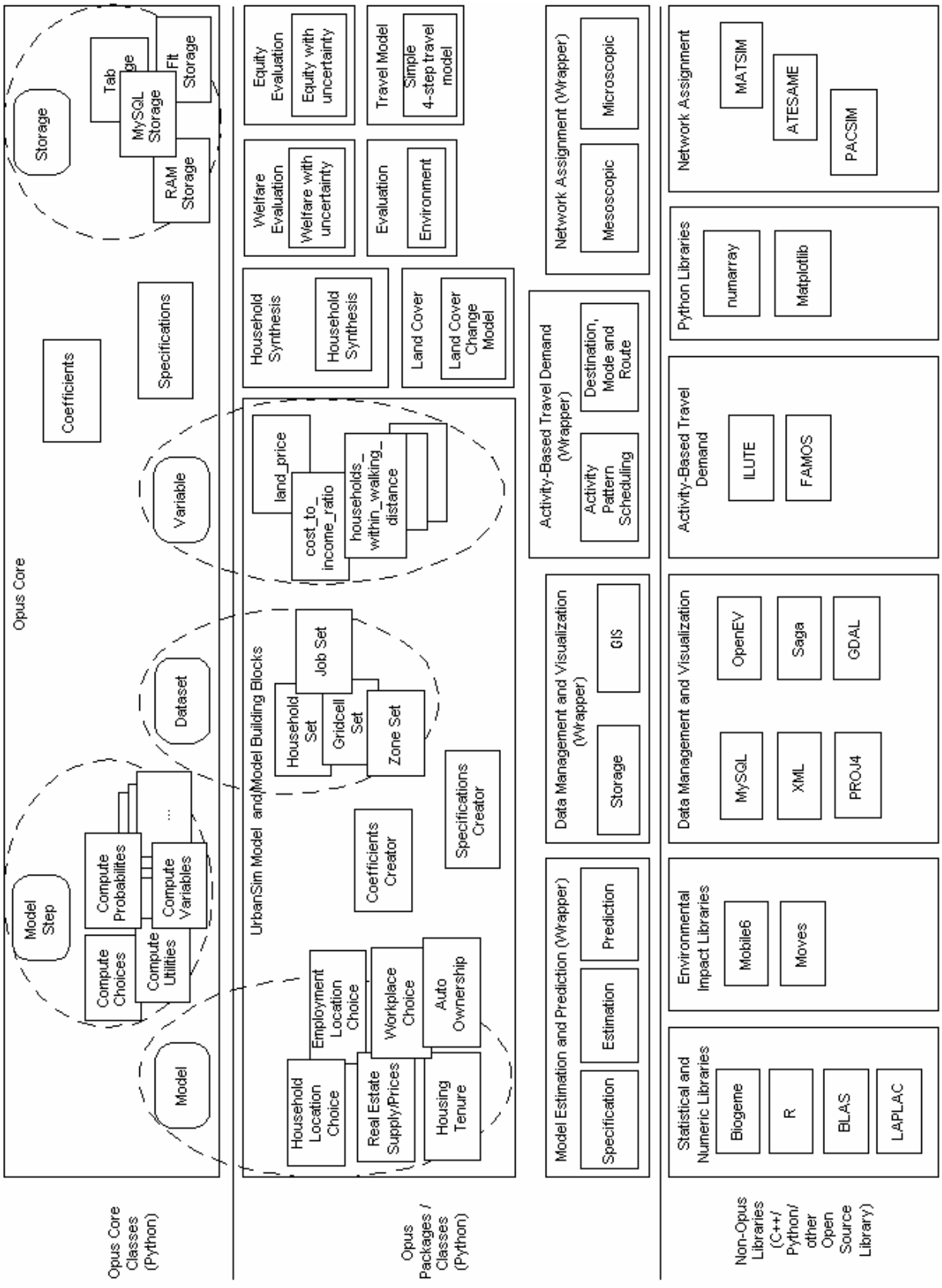

Figure 1 Opus Core, Internal and External Packages

Figure 1 also shows some other examples of desired Opus packages. For instance, we intend to integrate as an Opus package a land cover model created through collaboration between the Center for Urban Simulation and Policy Analysis (CUSPA) and the Urban Ecology Lab (UEL) at the University of Washington.

Along the bottom of Figure 1 are a set of non-Opus packages that we could envision becoming part of Opus, either by simply installing them, if they are Python packages, or by writing Python wrappers to provide an interface to them. Not only would these provide access for other Opus packages, it would allow these packages to interact with the rich and growing set of Python packages produced by the Python community. Not included in Figure 1 is Opus's underlying 'kernel'. The kernel is a minimal set of core functions to manage Opus packages. It includes the ability to download a package from a URL, to install a package, to uninstall a package, to run a package's tests, to create a new package, and to create a compressed file from which a package can be installed.

2.3 Integrated Model Estimation and Application

Model application software in the land use and transportation domain has generally been written to apply a model, provided a set of inputs that include the initial data and the model coefficients. The process of generating model coefficients is generally handled by a separate process, generally using commercial econometric software. Unfortunately, there are many problems that this process does not assist users in addressing, or which the process may actually exacerbate. There are several potential sources of inconsistency that can cause significant problems in operational use, and in the experience of the authors this is one of the most common sources of problems in modelling applications.

First, if estimation and application software applications are separate, model specifications must be made redundantly – once in the estimation software and once in the application software. This raises the risk of application errors, some of which may not be perceived immediately by the user. Second, separate application and estimation software requires that an elaborate process be created to undertake the steps of creating an estimation data set that can be used by the estimation software, again giving rise to potential for errors. Third, there are many circumstances in which model estimation is done in an iterative fashion, due to experimentation with the model specification, updates to data, or other reasons. As a result of these concerns, a design objective for Opus is the close integration of model estimation and application, and the use of a single repository for model specifications. This is addressed in the Opus design by designating a single repository for model specification, by incorporating parameter estimation as an explicit step in implementing a model, and by providing well-integrated packages to estimate model parameters.

2.4 Database Management, GIS and Visualization

The extensive use of spatial data as the common element within and between models, and the need for spatial computations and visualization, make clear

that the Opus platform requires access to these functions. Some of these are handled internally by efficient array processing and image processing capabilities of the Python Numeric library. But database management and GIS functionality will be accessed by coupling with existing Open Source database servers such as MySQL (www.mysql.org) and Postgres (www.postgresql.org), and GIS libraries such as SAGA (www.saga-gis.org) and OpenEV (http://openev.sourceforge.net). Interfaces to commercial DBMS and GIS systems will be provided mainly by user contributed packages.

2.5 Documentation, Examples and Tests

Documentation, examples and tests are three important ways to help users understand what a package can do, and how to use the package. Every Opus package must have a docs directory for documentation, an examples directory for executable examples of how to use the functionality, a data directory containing example datasets, and a tests directory containing unit- and acceptance-tests. Documentation is expected to exist in both Adobe portable document format (pdf) and web-based format (html, xml), and to include code documentation automatically created from code comments, such as done by pydoc. The pdf format makes it easy to print the document, and can produce more readable documents. Web-based documentation can be easier to navigate, and are particularly useful for automatically extracted code documentation.

Following the R lead, we plan to implement techniques to ensure that any code in the documentation works, and that any results of code as shown in the documentation are what the code actually does generate. A common way to do this is via literate programming that extracts and runs the code from the documentation and inserts the results of the code where appropriate.

2.6 Open Source License

The choice of a license is an crucial one for any software project, as it dictates the legal framework for the management of intellectual property embedded in the code. Opus will be released under the GNU General Public License (GPL). GPL is a standard license used for Open Source software. It allows users to obtain the source code as well as executables, to make modifications as desired, and to redistribute the original or modified code, provided that the distributed code also carries the same license as the original. It is a license that is intended to protect software from being converted to a proprietary license that would make the source code unavailable to users and developers.

The use of Open Source licensing is seen as a necessary precondition to the development of a collaborative software development effort such as envisioned for Opus. It ensures that the incentives for information sharing are positive and symmetrical for all participants, which is crucial to encourage contributions by users and collaborating developers. By contrast, a software project using a proprietary license has incentives not to release information that might compromise the secrecy of intellectual property that preserves competitive advantage.

There are now many Open Source licenses available (see www.opensource.org), some of which allow derived work to be commercialized. Some software projects use a dual licensing scheme, releasing one version of the software under a GPL licence, and another (functionally identical) version of the software under a commercial licence, which allows also distributing software as a commercial application. Opus developers have opted to retain the GPL license approach as it is a pure Open Source license, and does not generate asymmetries in the incentives for information sharing. Any packages contributed to OPUS by other groups must be licensed under a GPL-compatible license – we encourage them to be licensed under GPL itself, or less desirably, under LGPL (the library version of GPL).

2.7 Test, Build and Release Processes

Any software project involving more than one developer requires some infrastructure to coordinate development activities, and infrastructure is needed to test software in order to reduce the likelihood of software bugs, and a release process is needed to manage the packaging of the system for access by users. For each module written in Opus, unit tests are written that validate the functioning of the module. A testing program has also been implemented that runs all the tests in all the modules within Opus as a single batch process.

For the initial release process, a testing program is being used to involve a small number of developers and users in testing the code and documentation. Once this process is completed, a full initial release will be put on the project web site: www.opus-network.org. The current expectation is that this initial release will occur in the third quarter of 2005.

After the initial system release, two release schedules will be used to provide Opus users access to system updates. A stable release will be posted on the web on a periodic basis, approximately twice per year. This will contain major updates in core packages and updates to new versions of component systems such as Python. Since some of the component libraries include C or C++ code that must be compiled on a specific operating system, binary versions of these will be made available for Windows and Linux, and these can be compiled from sources if desired. The stable release installation file will be posted on the project web site: www.opus-network.org. The installation file will be in the form of a compressed file that a user would download to a local computer, uncompress, and install the Opus system as described in section 3.3, by running the standard setup.py module.

In addition to the periodic stable releases, nightly releases of the system will be generated once the testing process has completed without errors on all modules. A module may be added to the Opus kernel to automatically check for nightly updates and retrieve and install them, if a user wishes to use this feature. Otherwise, the installation process will be the same for nightly releases as for stable releases.

The Opus project currently uses the Concurrent Versioning System (CVS) for maintaining a shared repository for the code as it is developed by multiple

developers (though this mechanism may be changed to use Subversion in the future). Write access to the repository is maintained by a core group of developers who control the quality of the code in the system, and this group can evolve over time as others begin actively participating in the further development of the system. A repository will also be set up for users who wish to contribute packages for use in Opus, with write access.

3 Developing Models in Opus

Models can be implemented in the Opus framework using a variety of approaches, since one of the goals of Opus is to support flexibility, and the range of modelling approaches needed to implement models across the broad scope of land use, travel and environment is quite diverse. Two approaches that will be widely used to implement models in Opus are *Discrete Choice Models* that draw on the Random Utility Maximizing approach, and *Rule-based Models* such as those used in the Albatross activity-based travel model (Arentze and Timmermans, 2000), and the MATSIM microscopic traffic assignment model (MATSIM, 2005).

Opus is also being designed to specifically support *Microsimulation* at the level of the agent making choices. While it will be possible to also construct aggregate models in Opus, most of the effort in developing Opus will be to ensure the effective implementation of microsimulation models. We also want to support implementation of aggregate models, however, in order to allow flexible model systems to be developed, and to facilitate the careful comparison of alternative modelling approaches. One important question that has not been well addressed in the research and applied literature in urban models is how models with differing levels of aggregation of agents, entities and behaviour compare in terms of their results. Providing support for such comparisons is an important direction for future work on OPUS.

The representation of Agents within Opus will enumerate standard agents and their relationships. One example of such a configuration is shown in Figure 2. Some models will not need access to all of these agents and entities, but we hope to achieve some standardization in the way that certain agents and choices are represented, to allow greater flexibility in coupling model components. Note that there is some tension between standardization and flexibility in model design and implementation, but there is a case to make for adopting standard representations for certain agents, entities, and model results so that a range of different models can be developed and make common assumptions about these representations. One good example of this is in the specification of information passed from models predicting the activity-schedule of individual persons, to the model assigning the trips and tours in a person's activity schedule to the transport network. Information to support this interface must include a person identifier which maps to a household identifier, and a structured representation of the planned activity schedule and locations in a form that allows the assignment model to implement the plans on a network.

Two software approaches can be used to implement models in the Opus system. One is to code new models in Python, using and extending the core

Opus classes already implemented. The second is to create an interface for a model component that is written in some other language. If a model component is implemented in C or C++, the standard approach to developing an interface to Opus would be to use a tool like the Simple Wrapper Interface Generator (SWIG) to create Python bindings to the main functions and classes in the C or C++ code. This approach provides an efficient implementation, and allows components in these languages to interact with Opus data objects in memory, without requiring the data to be written to external files or databases. If the implementation language is Java, then the interface of model components with Opus will be through exchange of data through external files such as XML, ASCII, or a database like MySQL.

We illustrate the process of implementing models in Opus using a simple household location choice model. We begin with a set of 10 household agents, and a set of 9 locations with characteristics cost and capacity, respectively. These could be assigned at the command prompt, or loaded from a database or an ASCII file, and we skip this loading step here for clarity.

Suppose you wish to simulate a process of agents choosing locations using discrete choice model theory. As a first example, suppose your only predictor is the location attribute cost with a (hypothetical) coefficient value of -0.01 indicating a negative effect of cost on the choice preferences. Then you can create a coefficient object, a specification object and a choice model object, respectively:

```
>>> from opus.core.coefficients import Coefficients
>>> coefficients = Coefficients(names=("costcoef", ), values=(-0.01,))
>>> from opus.core.equation_specification import EquationSpecification
>>> specification = EquationSpecification(variables=("cost", ), \
                                          coefficients=("costcoef", ))
>>> from opus.urbansim.household_location_choice_model_creator import \
                       HouseholdLocationChoiceModelCreator
>>> hlcm = HouseholdLocationChoiceModelCreator().get_model( \
                       sample_locations=False, \
                       compute_capacity_flag=False, \
                       debuglevel=1)
```

The argument sample_locations in the choice model creator specifies if locations should be sampled for each agent or not. In the latter case, all locations are considered as a possible alternative for each agent. The argument compute_capacity_flag specifies if the procedure should take capacity of locations into account. The argument debuglevel controls the amount of outputs during the computation. The default value is 0 which produces no output.

We can run the household location choice model by

```
>>> hlcm.run(specification, coefficients, locations, agents)
Starting HLCM run ...
HLCM done. Time: 0.0214459896088 s
```

The results of the HLCM run determine locations that agents have chosen, and the model modifies values of the attribute 'location' of the agent set.

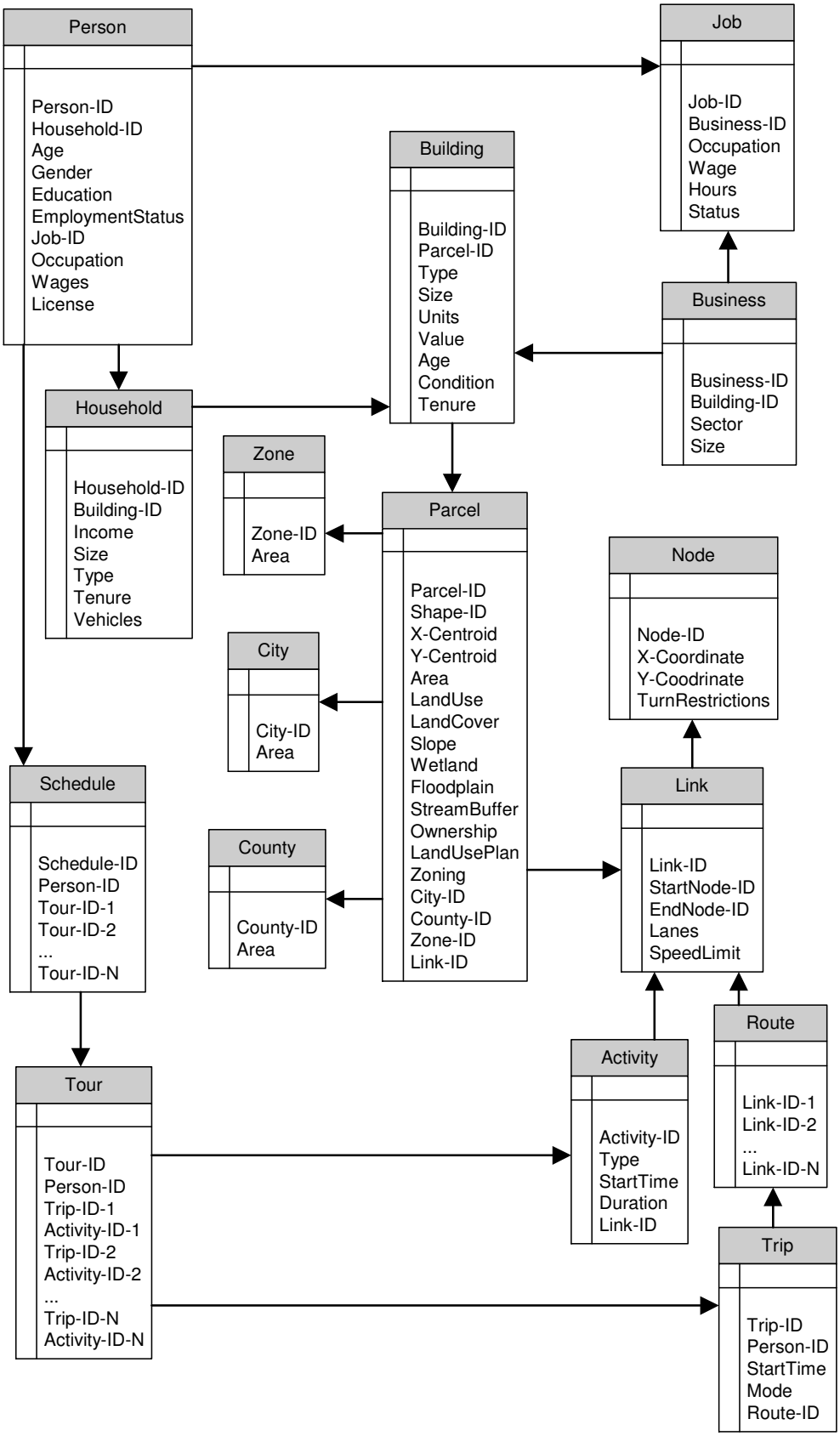

Figure 2 Example of Agent Entity-Relationships in an Opus Application

In the above example, the discrete choice model consists of steps such as computing utilities via the opus.core.linear_utilities class, computing probabilities via the opus.core.mnl_probabilities class and computing choices via the opus.core.random_choices class. These components can be easily exchanged by other implementations.

4 Conclusions and Future Research

The authors of this paper have taken the first steps in designing and implementing a shared collaborative system to meet specific needs within their respective projects. This work has been done with the broader goal of creating a system for collaboration among the research and user communities involved in land use, transportation and environmental planning in urbanizing regions throughout the world. The implementation of Opus was begun by the UrbanSim project team in early 2005, and has already reached a point that a full conversion of the UrbanSim system to Opus is almost complete as of mid-2005, and is scheduled to be used for operational planning in updating the Puget Sound region Vision 2020 plan in the coming year, providing an early indicator of the productivity of development within the Opus platform.

Much remains to be done in the broad agenda outlined in this paper, and the initial development team has identified priorities that will guide project investments in Opus over the next year or so. A more detailed working list of development priorities will be maintained on the Opus project web site (www.opus-network.org) to help establish and support a vibrant Opus user community that will engage actively in using and extending the system.

- *Initial release*. We plan to incrementally release versions of Opus, starting with an initial release in the third quarter of 2005.
- *Completing the UrbanSim transition to Opus*. The UrbanSim conversion to Opus will be completed by the initial release of Opus, and will be released concurrently.
- *Estimation and application*. Integrated model estimation for a range of discrete choice models will be supported by Opus packages, including one coded natively in Opus, and a wrapper to the Biogeme system. This would allow using a single model specification for estimation and application, and avoid cumbersome and error-prone procedures currently used to connect model estimation and application.
- *Indicators*. Model results can be voluminous and complex, and users often need a range of indicators that select, manipulate, and summarize key information used in the evaluation of outcomes or the diagnosis of the model. Support will be incorporated into Opus for streamlining the process of defining, implementing, selecting, using and visualizing indicators. A preliminary framework is already implemented, that draws heavily on the mechanism used to define variables.
- *Uncertainty analysis*. As in any other predictive process, urban simulation involves uncertainty – in data, in model parameters, and in model specifications. A failure to take uncertainty into account can lead to policy decisions based on misunderstanding of the risks. Assessing uncertainty

in land use and transportation models is an area where little research has been done. One of our goals is to include a generic framework for assessing and analyzing uncertainty into Opus. It will be based on the Bayesian melding theory (Poole and Raftery 2000). The main requirements for achieving our goal are a high modularity of the model system and an extremely good performance, since the framework will require many repeated simulation runs. The independent nature of these runs makes the framework very suitable for parallel computing.

- *Equity Analysis*. Methods recently developed to analyze relative distributions (Handcock and Janssen, 2002) will be used to analyze distributional effects of policies in a rigorous way that allows making inferences about equity. This will be developed into an Opus package.
- *Opus community*. For Opus to succeed, it needs a vibrant community of users and developers. There are many aspects of making this work, such as providing a central location for people to find and download Opus packages, providing good documentation, and making it easy for users to create and share packages with the community.
- *User and Developer Meetings*. In January, 2005 the first UrbanSim Users Workshop was held in San Antonio, and has led to increased collaboration among the UrbanSim users. We expect that regular user and developer meetings will be needed to sustain and develop the broader Opus system. Given the geographic dispersion of the user and developer community, online methods will be needed to augment in-person meetings.
- *Activity-based travel models*. Collaboration among several projects will be coordinated to implement new activity-based travel models as Opus packages.
- *Land cover change model*. CUSPA will convert a land cover change model developed as a prototype into an Opus package, providing an opportunity to incorporate feedback from land cover change on urban choice processes.
- *Web-based Stakeholder Interaction*. CUSPA has been developing a web-based system to facilitate the use of model results by modelers, policy makers, and the public. One component will allow users to configure a scenario by assembling pre-configured building blocks of transportation system and land policies and submitting these scenarios for simulation analysis. A second component will allow users to generate indicators from land use and travel models, and to visualize the results in tables, charts and maps. We plan to migrate these components into Opus packages.
- *Visualization*. Interfaces to a variety of components and libraries for visualization of model inputs, processes and results have been implemented for Opus, and more will be developed in the future. An interface to R is implemented, allowing access to all of the R statistical modelling and graphical analysis tools. Integrated charts and maps have been also implemented using the Matplotlib Python package, and the OpenEV GIS package. We plan to continue to extend the range of visualizations built into Opus systems.

ACKNOWLEDGEMENTS

This research was supported in part by the United States National Science Foundation Grant number EIA-0121326.

REFERENCES

Arentze, T.A., and Timmermans, H.J.P. (2000). Conceptual Framework. In T.A. Arentze & H.J.P. Timmermans (Eds.), ALBATROSS: A Learning Based Transportation Oriented Simulation System (pp. 71-80). Eindhoven: European Institute of Retailing and Services Studies.

Bhat, C., Sivaramakrishnan Srinivasan, Jessica Y. Guo, Activity-Based Travel Demand Modeling for Metropolitan Areas in Texas: A Micro-Simulation Framework for Forecasting, Center for Transportation Research, University of Texas, FHWA/TX-03/4080-4, 2003.

Bierlaire, M., Bolduc, D. and Godbout, M.-H. (2004) An introduction to BIOGEME (Version 1.0), URL:roso.epfl.ch/mbi/biogeme/doc/tutorial.pdf

de Palma A., F. Marchal and Y. Nesterov (1997), "METROPOLIS: A Modular System for Dynamic Traffic Simulation", Transportation Research Record 1607, pp. 178-184.

Mark S. Handcock and Paul L. Janssen. Statistical inference for the relative density. Sociological Methods & Research, 30(3):394–424, 2002.

Ihaka, R. and Gentleman, R. (1996). R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics, 5:299--314.

MATSIM (2005) www.matsim.org

Salvini, P.A. and E.J. Miller, "ILUTE: An Operational Prototype of a Comprehensive Microsimulation Model of Urban Systems", Networks and Spatial Economics, Vol. 5, 2005, pp. 217-234.

Opus (2005) www.opus-network.org

Pendyala, R., R. Kitamura and A. Kikuchi (2004). FAMOS: The Florida Activity Mobility Simulator, Presented at the Conference on “Progress in Activity-Based Analysis”, Vaeshartelt Castle, Maastricht, The Netherlands, May 28-31, 2004

Poole, D. and Raftery, A.E. (2000). Inference for deterministic simulation models: The Bayesian melding approach. Journal of the American Statistical Association, 95:1244-1255, 2000.

Train, K. (2003) Discrete Choice Methods with Simulation. Cambridge University Press.

Waddell, P., A. Borning, M. Noth, N. Freier, M. Becke, G. Ulfarsson. (2003). UrbanSim: A Simulation System for Land Use and Transportation. Networks and Spatial Economics 3 (43-67).

Waddell, P. (2002). UrbanSim: Modeling Urban Development for Land Use, Transportation and Environmental Planning. Journal of the American Planning Association, Vol. 68, No. 3, (297-314).