A Domain-Specific Language for Urban Simulation Variables

Alan Borning
Dept. of Computer Science and Engineering
University of Washington
Box 352350
Seattle, Washington 91895
borning@cs.washington.edu

Hana Ševčíková
Center for Urban Simulation and Policy Analysis
University of Washington
Box 353055
Seattle, Washington 91895
hana@cs.washington.edu

Paul Waddell
Evans School of Public Affairs
University of Washington
Box 353055
Seattle, Washington 91895
pwaddell@u.washington.edu

ABSTRACT
UrbanSim is a modeling system for simulating the development of urban regions over periods of 20-30 years. Its purpose is to help evaluate alternative proposed policies and transportation infrastructure projects by simulating the long-term impacts of the different alternatives. In the process of adapting and calibrating the system for use in a new region, planners and modelers must prepare input data, specify and estimate a set of component models, and assess the results before giving them to policy makers. All of these activities involve considerable investigation and experimentation using different model variables that describe attributes of actors, processes, and geographies of the simulated environment. In many cases, the original variables must be transformed or combined to create new variables that are more suitable for analysis; and in other cases, creating new variables on the fly may facilitate exploration of the results. In this paper we describe the design and implementation of domain-specific language for specifying these variables, with a syntax and semantics tailored to the domain. As a result of using this language, the code size for specifying variables is reduced by an order of magnitude, and user productivity is greatly increased.

Categories and Subject Descriptors
D.3.2 [**Programming Languages**]: Specialized Application Languages; J.4 [**Computer Applications**]: Social and Behavioral Sciences; I.6.2 [**Simulation and Modeling**]: Simulation Languages

General Terms
Design, Languages

Keywords
Modeling, urban simulation, domain-specific languages, Opus, UrbanSim

1. INTRODUCTION
In many regions throughout the United States and worldwide, there is great concern about such issues as traffic congestion, resource consumption, lack of sustainability, and sprawl. Elected officials, planners, and citizens grapple with these difficult issues as they develop and evaluate alternatives for major land use and transportation decisions, such as building a new rail line or freeway, establishing an urban growth boundary, or changing incentives or taxes. UrbanSim [2, 14, 15] is a simulation system for projecting the impacts of alternative policies and transportation infrastructure projects over periods of 20-30 years, with the purpose of better informing public decision-making about issues such as these. To date, it has been applied operationally in Houston, Texas, and is being transitioned into operational use in the Puget Sound region in Washington State (Seattle and surrounding cities), and in Salt Lake City, Utah [18]. The UrbanSim group has also worked with other agencies in applying UrbanSim in the urban areas around Detroit, Eugene, Honolulu, and San Francisco. There have also been research and pilot applications in such diverse regions as Amsterdam, Burlington, Durham, El Paso, Melbourne, Paris, Phoenix, Tel Aviv, and Zurich; and there is an active user community, including a mailing list and two UrbanSim Users Group meetings, with a third meeting planned for March 2008.

UrbanSim is implemented as a set of interacting component models that simulate different actors or processes within the urban environment. For example, the Household Location Choice Model simulates the actions of households seeking a place to live. It is a discrete choice model [6], which computes the probability of a household moving into an available vacant apartment or house. These probabilities depend on characteristics both of the household (e.g., income, number of children), and of the available location (e.g., cost, distance to major employment centers). Other component models include Employment Location Choice, Land Price (simulating the real estate market for land), Real Estate Development, and an interface to an external travel model.

The current version (UrbanSim 4) is implemented using a new framework called Opus (Open Platform for Urban Simulation) [17]. Both Opus and UrbanSim are open source, and freely available for download from the project website at `http://www.urbansim.org`. (In the remainder of this paper, we will generally refer to UrbanSim rather than Opus, although in some cases the capabilities are more general ones that are available for other packages developed using Opus as well as UrbanSim.) Opus and UrbanSim are in turn implemented in Python, and make heavy use of a set of Python libraries, notably NumPy (`http://numpy.scipy.org`), a highly efficient array and matrix manipulation library, as well as other libraries for interfacing with databases, producing graphs, charts, and maps, and so on.

2. ITERATIVE DEVELOPMENT OF DATA AND MODELS IN URBANSIM

There are numerous stakeholders in an application of UrbanSim to an urban region, including citizens, elected officials, urban planners, and many others. In this paper, we focus on better supporting one particular group of stakeholders: namely, the technical planners and modelers who adapt and calibrate the system for use in a given region. We use as a case study the UrbanSim model system developed for the Puget Sound Regional Council (PSRC), the Metropolitan Planning Organization for the region in Washington State that includes the cities of Seattle, Bellevue, Tacoma, and others, as well as adjacent suburban and rural areas [16].

The PSRC model system uses 12 component models:

- Real Estate Price Model
- Expected Sale Price Model
- Development Proposal Choice Model
- Building Construction Model
- Household Transition Model
- Employment Transition Model
- Household Relocation Model
- Household Location Choice Model
- Employment Relocation Model
- Employment Location Choice Model for home-based jobs
- Employment Location Choice Model for non-home-based jobs
- Employment Location Choice Model for governmental jobs

Four of these must be estimated using observed data (i.e., the coefficients for variables in the different choice and regression expressions must be fitted to this data). The component models requiring this estimation are the Real Estate Price Model, the Household Location Choice Model, the home-based Employment Location Choice Model, and the non-home-based Employment Location Choice Model. Further, two of these models are split into sub-models: the Real Estate Price Model contains 18 sub-models, one for each land use type; and the non-home-based Employment Location Choice Model contains 15 sub-models, one for each employment sector. For these models with sub-models, the same code runs on different partitions of the data. Thus, each sub-model must be estimated separately.

In total, prior to running the PSRC UrbanSim application, the user must estimate 18 regression models and 17 discrete choice models. The estimation process is usually preceded by data analysis and preparing data for estimation and simulation. This includes detecting outliers, dealing with missing values, and preparing a set of possible predictor variables for each model.

The term "model variable" (or just "variable") is used here to indicate an attribute of actors or geographies involved in the particular model. Transformations are often used to create new variables that are more suitable for analysis than the 'raw' attributes. For example, if the underlying statistical model assumes a normal distribution of the variables being analyzed, often a transformation is necessary to better approximate normality, such as the log transformation. Another case of requiring the creation of a new variable is when we want to use a predictor of a certain entity but have collected data on different entity. For example, for predicting the price of each geographic cell, we want to use population density on that cell as a predictor variable, but initially the population data is organized by household (location and number of persons for each household). To achieve this goal, we can create a new variable from the household data that provides the number of persons for each geographic cell. For discrete choice models we want to use interaction variables that characterize an interaction between agents and choices. For example, in the Household Location Choice Model, a strong candidate for a predictor variable is an interaction between household's income and the cost of a location. Such interactions can be also represented by creating a new variable. Finally, transformations can be also used to deal with missing data and outliers, by defining a new variable that filters out outliers or fills in missing data from the original.

As an aside, note that the term "model variable" (or "variable") does *not* have the same meaning as "variable" as used in programming languages, although they are related. (This fact can be a trap for the unwary in understanding the semantics of variables.)

The process of developing a model involves configuring the arguments used in constructing the model, and then selecting the variables to use in the specification of the discrete choice or regression equation used in the model. The specification of the model is a list of independent variables to be used as predictor variables. The process of developing a specification involves a combination of theory regarding the domain being modeled, and analysis of the statistical results from estimating the model parameters. A common process is to run the estimation with an initial specification that contains key variables motivated by theory and knowledge of the domain, then to examine the significance level of each variable, and from this decide which variables to keep and which to eliminate. This process usually takes several iterations of including and excluding variables, sometimes including creating new and more suitable ones. For some models, such as regression models, the number of iterations can be reduced by using methods for variable selection, such as Bayesian Model Averaging [10]. Nevertheless, creating a comprehensive list of good predictors is often a challenging task, requiring considerable experimenting with the model and the data.

Data analysis is typically performed again at the end of a simulation process on the model results. Before giving the results to the policy makers, modelers will often want to check that they are reasonable, that the model has not produced invalid results as a consequence of problems in the data or a bug in the code. The process is similar to the one used when preparing data, and involves investigating simulated characteristics and their transformations of the different actors and geographies, often by computing variables that help in diagnosing the reasonableness of results.

3. DATASETS IN URBANSIM

Within a simulation, data is held in instances of the class *Dataset* (or a specialized subclass of it). A dataset is much like a table in a relational database: it can be thought of as a $n \times m$ table, where n is the number of entries, and m is the number of *primary attributes* associated with each entry. For example, households are represented using such attributes as `income`, `number_of_persons`, `number_of_children`, `number_of_workers`, `age_of_head`, and `building_id`, as well as a unique ID. (A primary attribute is a variable that comes from some external observation, rather than being generated by the running simulation.) As a second example, parcels of land in the simulated region are represented using such attributes as `parcel_sqft`, `used_land_area`, `land_value`, and `improvement_value`. A dataset will have a small number of attributes, but can have a very large number of entries. For example, in the PSRC application of UrbanSim, there are $1,282,940$ households, each with 8 attributes; $1,177,140$ parcels with 26 attributes; $1,849,447$ jobs with 8 attributes; and $1,008,869$ buildings with 13 attributes. The number of entries can change during a simulation (and usually increases, for example as simulated population increases).

Efficiency — both for storage space and also execution speed — is critical. For a given dataset, UrbanSim's component models typically access only a small subset of the attributes, but for all entries (for example, just the `land_value` attribute of every parcel). To match this usage pattern, each dataset attribute is stored as a NumPy array, allowing compact storage and fast access. Further, the model computations all work from NumPy arrays, so that the attributes are already in the desired format. Attributes are loaded only on demand (i.e., lazily), and can be flushed to disk if needed to free up main memory.[1]

4. REPRESENTATION OF OPUS VARIABLES

As noted above, a dataset has a set of primary attributes, such as income or number of children, that comes from some external observations. In addition, we often require additional attributes that are computed using some transformation of existing attributes. These are *derived attributes.* Within the class Dataset, they are simply handled as additional columns of the dataset to which they belong.

A variable that represents a derived attribute is implemented as a class in a single Python file of the same name. For example, the `zone.average_income` variable shown in Figure 1 computes the average income per household within a zone. The `compute` method returns an array of numbers, one per zone, representing the mean of the incomes of all households placed in the corresponding zone. The underlying computations — on the 1,282,940 households and 938 zones in the PSRC application — are handled efficiently using NumPy functions and operators. The variable has three other variables on which it depends (defined in the method `dependencies`):

- `household.income`
- `zone.zone_id`
- `urbansim_parcel.household.zone_id`

The first two variables are primary attributes (of the household and zone datasets respectively), while `urbansim_parcel.household.zone_id` is derived. (A household is placed in a building, the building is placed on a parcel, and the parcel is located in a zone.)[2]

Variables representing derived attributes use lazy evaluation: the `compute` method is only invoked if either the variable has not been computed before, or if the values of one or more of the variables on which it depends have been updated. The system maintains a version number for each variable to keep track of the bookkeeping for this. Thus, if the value of a variable (or a particular version of a variable) is never requested, it won't be computed.

Figure 2 shows a more complex variable, which computes the square feet per residential unit for each parcel. The variables on which it depends are `parcel.parcel_sqft`, a primary attribute of parcels, and `parcel.residential_units`, which is computed for the parcels dataset as the sum of residential units of buildings located on that parcel. The `compute` method returns a ratio of parcel sqft and residential units, with zeros for entries in which `parcel.residential_units` is equal to zero. As before, this is an array-based computation, and we return an array of ratios, one per parcel.

This software architecture — including datasets and Opus variables — has proven to be efficient. By being integrated with our own estimation routines, it has provided a huge boost in the ability of modelers to experiment with alternative configurations and to better fit the model to the environment being simulated relative to standard work practice, which involves using an external econometric package.

However, in the course of estimating models and analyzing data, as described in Section 2, for every new transformation of the data, even one as simple taking a log or square root, the user must to define a new Python class that represents the new variable, including defining the computation itself and the dependencies. For even the simplest of new variables, this would be 8–10 lines of code, much quite similar to that in existing variables. For software engineering reasons, we also want to include a unit test (including test data), and often a post-check, bringing the total lines of code to 50 or more. (The unit test is invariably much longer

[1]At this point the reader may wonder why we don't simply represent datasets as a table in a real database. The reason is simply efficiency — for external storage we do often store datasets in a relational database, but using database accesses for all computations during the running simulation proved much too slow. There are in fact convenience routines to convert between datasets and database tables in an external database (typically either MySQL or PostgreSQL).

[2]Note that in Opus the full name of a variable refers to the location where the variable is implemented. Thus, the `zone_id` variable is defined in `urbansim_parcel/household/zone_id.py`.

```
from opus_core.variables.variable import Variable

class average_income(Variable):

    def dependencies(self):
        return ["household.income", "zone.zone_id",
               "urbansim_parcel.household.zone_id"]

    def compute(self, dataset_pool):
        households = dataset_pool.get_dataset("household")
        return self.get_dataset().aggregate_dataset_over_ids(households, "mean", "income")

% *** code for unit tests omitted ***
```

Figure 1: Definition of the `zone.average_income` variable as a Python class

```
from numpy import ma, float32
from opus_core.variables.variable import Variable

class parcel_sqft_per_unit(Variable):

    def dependencies(self):
        return [parcel.parcel_sqft, urbansim.parcel.residential_units]

    def compute(self, *args):
        parcels = self.get_dataset()
        res_units = parcels.get_attribute("residential_units")
        return ma.filled(parcels.get_attribute("parcel_sqft") /
                         ma.masked_where(res_units==0,res_units.astype(float32)), 0.0)

% *** code for unit tests omitted ***
```

Figure 2: Definition of the `parcel.parcel_sqft_per_unit` variable as a Python class

```
average_income = zone.aggregate(household.income, function=mean)

parcel_sqft_per_unit = numpy.ma.filled(parcel.parcel_sqft /
    numpy.ma.masked_where(urbansim.parcel.residential_units == 0,
        urbansim.parcel.residential_units.astype(float32)), 0)

is_pre_1940 = parcel.aggregate(building.year_built *
    numpy.ma.masked_where(urbansim_parcel.building.has_valid_year_built == 0, 1), function=mean) < 1940
```

Figure 3: Example expressions. The first expression defines a variable for the `zone` dataset, and the other two define variables for `parcel`.

than the variable definition itself, but not testing the definition is not recommended!) Users can of course copy existing definitions and modify them, but even this is tedious.

We initially provided some simple ways of producing new variables without writing new Python classes, by using a limited vocabulary of expressions and substitutions. This was implemented by textual substitutions and expansions of the strings that defined the variables. However, this proved limited, fragile, and difficult to extend. So we moved to a much improved solution: namely, defining a domain-specific language for defining Opus variables.

5. A DOMAIN-SPECIFIC LANGUAGE FOR DEFINING OPUS VARIABLES

There is a long tradition in the programming language community of domain-specific languages: languages tailored to a particular application domain. These are also referred to as *little languages* (particularly within the Unix community), *application languages*, or *problem-oriented languages*. Reference [7] is a comprehensive survey paper, while [1] is a classic discussion of little languages. Also see [13] for an annotated bibliography of (pre-2000) papers on domain-specific languages.

Such languages can offer significant gains in ease of use and expressiveness for certain kinds of problems. For this problem, two characteristics that strongly point toward using a domain-specific language are: first, the natural recursive structure of Opus expressions, in which an expression can be composed using operators and functions from other expressions; and second, the difficulty of succinctly specifying Opus variables in Python itself.

Three basic choices in the design of a domain-specific language are its syntax, its semantics, and the implementation strategy. These are discussed in Sections 5.2, 5.3, and 5.4 respectively. A common tradeoff in the design of domain-specific languages regards the range of programming activities the language should support. Generally, supporting a larger set of activities means more complexity of the language and its implementation. Here, our task is much simplified because we embed the Opus variable language within Python, allowing us to concentrate simply on the task of defining new Opus variables. Further, we decided to retain the old Opus variable mechanism as well, so that a variable could still be defined using a Python class for more complex cases. Thus, we prefer a language that supports 90% of the cases, and is relatively clean and simple, over a more complex one that supports some additional, infrequent cases.

5.1 Examples

Figure 3 gives several examples of expressions, all from actual use in UrbanSim models. The `zone.average_income` variable shown in Figure 1 can be replaced by the first expression — in the process becoming more succinct and also more readable. Here, `average_income` is an alias for the expression. We are aggregating the `income` attribute of households up to the zone level by taking the mean over all incomes in each zone. The next expression is the equivalent of the `parcel_sqft_per_unit` variable from Figure 2. Here, the dependent variable `urbansim.parcel.residential_units` can be implemented as (or directly replaced in the expression by) `parcel.aggregate(building.residential_units, function=sum)`. The last expression, for `is_pre_1940`, returns an array of booleans: each parcel for which the average building age is older than 1940 gets a True, and others get False. We find the average age of the buildings in each parcel using the `aggregate` method, masking out buildings for which the `year_built` field is invalid. Then we compare these ages with 1940, returning the desired array of booleans, one per parcel.

The availability of the expression syntax makes it possible for modelers to much more readily experiment with model specifications when developing new models or modifying their specification, by making it easy to add new expressions to the model specification and assigning short aliases for them. The problems associated with diagnosing simulation results are also reduced, since the expression syntax provides a much easier to use capability for modelers to generate new expressions that help diagnose specific problems they observe in the results.

5.2 Syntax

For the syntax of the language for defining Opus variables, we selected a subset of Python itself, rather than designing a new, custom syntax. This section gives an informal description of the language; a more complete specification is given in the *Opus/UrbanSim Users Guide and Reference Manual* [3].

An expression consists of the name of a variable, or a function or operation applied to other expressions. This definition is recursive, so that a unary function or binary operator can be applied to expressions composed from other expressions. The variable name is the name of some existing variable (including its associated dataset name), for example `building.year_built`, or `parcel.residential_units`. Variable names can also include the package in which they are defined, to allow selecting a specific definition (e.g. `urbansim.parcel.residential_units`). The unary functions are any of the functions available in NumPy, such as `exp` and `sqrt`. Similarly, all of the NumPy operators can be used in Opus expressions, including `+ - * / ** < > ==`. Note the NumPy semantics for these. For example, `*` does elementwise multiplication of two NumPy arrays, or with a scalar argument, scales all the elements in an array, e.g. `1.2*household.income`. Similarly, the comparison operators perform an elementwise equality comparison on two arrays, returning an array of booleans (as in the `is_pre_1940` expression). The expression can also include a cast to a different NumPy type (for example, to coerce a `float64` to a `float32` to conserve memory).

Expressions can be used directly in model specifications. They can also be collected into an `aliases.py` file placed in the package for that dataset specification; then in a specification, the user can refer to the variable simply by its alias. This supports reuse of definitions, and has worked out much better in practice.

An important class of methods that provide special operators beyond those in NymPy are used to aggregate and disaggregate variable values over two or more datasets. The `aggregate` method associates information from one dataset to another for a many-to-one relationship, while the `disaggregate` method does the same for a one-to-many relationship. The expression for `zone.average_income` in Figure 3 shows one example of aggregation, done by taking the mean. Other functions for aggregation include taking the sum, the min, or the max of the values being aggregated. Analogously, the `disaggregate` method takes information from a coarse set of entities and allocates it to a finer set of entities with a one-to-many relationship. Aggregation and disaggregation can be done over multiple levels of geography in a single expression, for example aggregating the number of jobs from buildings, up to parcels, then up to zones, then to cities.

Finally, in order to work with variables that describe the interaction between two datasets, Opus includes a subclass of `Dataset` called `InteractionDataset`. Attributes of this class are stored as two-dimensional arrays. For example, in the Household Location Choice Model, we are interested in the interaction between household income and cost per residential unit. (We would certainly expect that the combination of these two would have a strong effect on choice of residence.) A full range of operators and functions are available for these 2-d interaction arrays as well. For example, the expression

```
ln(household.income) * zone.average_housing_cost
```

returns an $n \times m$ array, where n is the number of households and m is the number of zones. Each element in the array is the log of that household's income times the average housing cost for that zone.

To support this, for expressions evaluated within the context of an $n \times m$ interaction set, the 1-d arrays resulting from evaluating an attribute of the first component of the interaction set are cast into 2-d arrays, with each row having m copies of the attribute; for attributes of the second component, the 1-d arrays are cast into 2-d arrays with each column having n copies of the attribute. (This makes it easy to produce readable and succinct expressions.)

To illustrate, consider the example of household income interacted with cost per residential unit. Suppose for simplicity that we have 3 households (rather than 1,282,940), with incomes of $45,000, $100,000, and $60,000. In presenting the array results we'll express these in thousands of dollars. Then evaluating `household.income` within the context of the interaction set returns a 2-d array

```
[ [ 45,  45],
  [100, 100],
  [ 60,  60] ]
```

If there are two zones, with average housing costs of $400,000 and $300,000, then evaluating `zone.average_housing_cost` within this context returns (again in thousands of dollars)

```
[ [400, 300],
  [400, 300],
  [400, 300] ]
```

Then the expression `ln(household.income) * zone.average_housing_cost` can be evaluated using standard NumPy operations, yielding the result

```
[ [1522.66, 1142.00],
  [1842.07, 1381.55],
  [1637.74, 1228.30] ]
```

5.3 Semantics

In this section we present an informal discussion of two aspects of its semantics that seem of particular interest.

In using an expression, there are *two* stages of evaluation (as with macros, and in contrast to the usual programming language semantics). First, an expression is evaluated to yield an Opus variable. This Opus variable is then evaluated a second time (using its `compute` method) to produce the final value, which will be a one or two dimensional array of attribute values. The first stage of evaluation is independent of any particular dataset — evaluating a given expression would always return the same variable definition. (Because of this fact, the implementation only needs evaluate an expression once; after that it uses the cached variable definition for that expression.) The second stage of evaluation is done in an environment in which the value of the variable is computed relative to a particular dataset. A local environment (the dataset pool) binds other dataset names to datasets, which is used if the definition involves references to variables of other datasets, for example for a variable being aggregated.

Another interesting aspect of the semantics is the use of lazy evaluation for all expressions. In the programming languages world, lazy evaluation for all expressions in the language is typically available only for certain functional programming languages, e.g., Haskell [9] and Miranda [12]. Otherwise it is only provided by specific programmer action (for example wrapping the expression to be evaluated lazily in a method) — the idea of providing lazy evaluation in an imperative language would be generally viewed as hopelessly inefficient. In our domain, however, in which the data dimensions are typically very large — perhaps a million-element array — and a single operation is performed on the entire array rather than element-by-element, lazy evaluation becomes not only reasonable, but more efficient than the alternative of eager evaluation. The bookkeeping overhead required is dwarfed by the computation times for the expressions.

5.4 Implementation

The implementation can be described succinctly, although the actual code is somewhat complex. The goal is to take an expression (represented as a Python string), and from it automatically compile a new, anonymous class that represents the Opus variable to which the expression evaluates. The primary workhorse for this is a class `AutogenVariableFactory`. For example, given the expression for `zone.average_income` given in Figure 3, `AutogenVariableFactory` parses the expression, analyzes the parse tree to find the dependencies, and writes a new class with a `dependencies` and a `compute` method. This new class is functionally equivalent to the one shown in Figure 1 (and has the same performance).

The aggregation/disaggregation methods present a few complications. For the second stage of evaluation (from a variable to a NumPy array), most functions and operators expect arrays for the operands. The `aggregate` and `disaggregate` methods, however, expect the first argument (the variable to be aggregated or disaggregated) unevaluated, as an Opus variable. A further complication thus arises when the first argument to `aggregate` or `disaggregate` is an expression rather than a variable. To handle this, `AutogenVariableFactory` recursively calls itself, to generate a second Opus variable that represents the expression being aggregated or disaggregated. For example, consider the expression

```
zone.aggregate(ln(parcel.land_value),function=mean)
```

Here, the system will generate two different anonymous variables, one for `ln(parcel.land_value)` and another for the expression as a whole. The `is_pre_1940` variable (Figure 3) is another example of this.

For the implementation, since we use Python syntax (although not semantics), we are able to use the built-in Python `parser` module to parse expressions into a parse tree. To analyze the result, the Python parse tree format was reverse-engineered, and parse tree patterns produced for the different cases we needed to recognize (e.g., a variable reference, an aggregation method call, and so forth). The Python parse tree format changed between Python 2.4 and 2.5 (with the addition of `if` expressions to the language), and so the system can produce two different versions of the parse tree patterns, with the appropriate version produced depending on which version of Python is being run. The system maintains a dictionary of expressions for which an Opus variable class has already been generated. When an expression is first encountered, the system looks it up in that dictionary; if found, it returns the existing class, and otherwise generates a new one.

In implementing the language, we used an agile programming methodology [4], including in particular test-first programming methodology, and extensive use of unit tests [5, 8] for the different kinds of expressions (a total of 116 unit tests for the domain-specific language in the current implementation). For example, one representative unit test verifies the compilation of code involving the `aggregate` method, by evaluating an expression that uses `aggregate`, evaluating the resulting variable on a small set of test data, and comparing the actual result with the expected result (which was computed by hand). Our automated build system (CruiseControl, `http://cruisecontrol.sourceforge.net`) runs all of these unit tests — for the domain-specific language, variable definitions, and all others — whenever new code is checked into our source code repository. A traffic light mounted in the hall of our lab displays the results: green if all tests have passed, red if there is a failure, and yellow if a build is underway. The reader can check on the current state of the traffic light by browsing to `http://www.urbansim.org/status/`. Also, the reader who is interested in browsing the implementation and unit tests can do so using our source code repository and trac system linked from `http://www.urbansim.org`, or can download the code via the same URL and run the tests.

6. EVALUATION

We first provide some quantitative results for reduction in code size. As a representative sample, the estimation for the Real Estate Price Model in the PSRC application uses 53 different variables, each defined using a one-line expression. Setting the unit tests aside for the moment, the equivalent code using variables defined in Python would require 53 different classes, with an average code size of 10+ lines per class. This is an order of magnitude reduction in code size. The system includes no special support for unit tests for expressions yet, so the number of lines of code for unit tests is unchanged. However, for the simplest expressions (involving say just taking the log of a variable), a unit test seems unnecessary at this level — the functionality of compiling the Opus variable classes for expressions involving log and other functions is extensively tested by the unit tests accompanying `AutogenVariableFactory` (Section 5.4). In this case it is more appropriate to test the component model in which the variable is used. For more complex expressions (e.g., the one for `is_pre_1940`), a unit test *is* warranted (although we haven't yet convinced our users to write them in most cases). We have, however, made a start at writing such tests for more complex expressions, which we place in a separate place — this keeps the file with the expression definitions themselves much cleaner and easily understood.

For a more comprehensive evaluation of code size reduction, we compared two different model systems: the gridcell version of UrbanSim (which uses a 150×150 meter gridcell as the basic geographic unit), and the parcel version of UrbanSim (which uses individual land parcels). The gridcell version is older, and was written before the introduction of expressions; the parcel version was written after their introduction. These provide roughly comparable functionality. The total number of non-blank lines of code for defining variables in the gridcell version is 26330; the total for variables and expressions in the parcel version is 3643 — a reduction of number of lines of code of more than a factor of 7.

However, reduction in code size is not the ultimate goal. Rather, we want to improve the productivity of users of UrbanSim, reduce the number of errors that they make, and make the system accessible to a wider range of users. Our estimate is that the domain-specific language for Opus variables has boosted the productivity of our modelers by at least an order of magnitude on tasks involving creating new variables and experimenting with them. This may actually understate the effect, since many modelers would not have been comfortable creating new variables at all if they require coding as separate classes, given the complexity of the coding. Further, the language has made a qualitative difference in the kinds of data exploration that users are able to undertake, by interactively writing and trying expressions to investigate data on the fly — formerly these tasks would have been too cumbersome and most modelers would have used some other way to explore the data, or potentially of greater concern, might not undertake as thorough a diagnosis due to the level of effort involved in coding variables.

This estimate of the boost in productivity is definitely that: just an estimate (although one based on a significant amount of real-world experience working with the language within our research group). To provide a more rigorous evaluation of productivity increase, we could do a controlled experiment, in which a set of modelers were given the task of coding a model using the old-style variables, and using expressions. This would be counterbalanced, so that one group would code first using old-style variables and then expressions, and the other using expressions first and then variables. However, we haven't done this, for two reasons. First, the productivity gains are so large that such an experiment wouldn't provide that much useful additional information (it is already clear that expressions are strongly preferred). Second, there aren't that many people in the UrbanSim user community who can code additional variables in the old way — it's just too complicated. So finding a subject pool to get statistically valid results would be hard. Instead of undertaking such an experiment at this point in the work, we prefer to defer a user study, and carry it out for a comprehensive graphical interface to UrbanSim that integrates expressions with support for the full life cycle of applying UrbanSim (Section 7).

A significant aspect of the usability of the language is the syntax itself. Is it appropriate to use Python syntax for the Opus variable language, since this language is decidedly not Python? A language design principle (due to Alan Kay) is that two constructs in a programming language should be either the same, or quite different — having them be only slightly different is a recipe for confusion. Applying this principle, we would want either a strict subset of Python syntax, or else something quite different (e.g. Scheme syntax). So far the choice of Python syntax has proven to be the right one: users of the system have found the syntax quite intuitive, and don't need to learn a new one. Further, there has not been an issue of confusing Opus variable expressions with ordinary Python, since the variables are written in quite specific and well-marked contexts in the system. Writing expressions using the GUI will make this distinction even clearer.

7. CONCLUSIONS AND FUTURE WORK

Mernik, Heering, and Sloane [7] observe that

> DSL development is hard, requiring both domain and language development expertise. Few people have both.

This particular domain-specific language is small relative to many others; further, we were able to build on Python's syntax and `parser` module, thus simplifying the task. Nevertheless, the basic point made by Mernik et. al holds.

Two further observations. First, the language semantics involving two stages of evaluation is obvious in retrospect, but was quite difficult for us to come to. One basic problem is the use of the same term ("variable") in two different ways by programming language researchers and urban modelers. But just saying that "there are two different uses of the word — they don't mean the same thing" was not enough. Instead, we needed to understand how the concepts from programming languages and urban modeling interact, and to bring in a programming language design sensibility to move forward. (More generally, one might suggest that in doing interdisciplinary research, it is essential to be aware of the pitfalls that come from differences in vocabulary; but beyond that, to look for ways that understanding these differences can inform the resulting design.) Second, conventional wisdom about what functionality is reasonable and efficient to provide in a language (e.g., lazy evaluation) should be examined critically in light of the domain.

The Opus variable language has been quite successful as far as it goes. There are several areas in which we want to extend it, in particular to provide parametrized variables (which currently are still handled by the older Python-based implementation). We also need better handling of user errors in writing expressions, so that more meaningful feedback can be given.

Another area for research is the language semantics. This is improved over what it was formerly, but there is still some additional work to be done. As noted in Section 5.3, the two stages of evaluation used for expressions are analogous to those for programming language macros. In general, macro facilities in programming languages have a reputation as being useful but very messy. However, the Scheme language [11] has a relatively clean macro system and semantics. So one direction for this part of the work will be studying macro systems, in particular the one in Scheme, to better inform the design and semantics of the Opus variable language.

The most important area for future research, however, will be integrating the language with a graphical interface to support the full life cycle of applying UrbanSim, including estimation, specification, and other activities. Currently the configuration for an estimation is written in Python, with lists of strings giving the expressions that define the variables. To experiment with these, modelers comment and uncomment strings, add new ones, and so forth. We are currently developing an integrated graphical interface to support these activities, which will have tree-structured displays of the variables used in an estimation or specification, and that will let the user conveniently add, enable, and disable variables, and re-estimate on demand. Some of the co-authors of this paper even have hopes that by providing good support for writing unit tests in the GUI, users will include them for more complex variable definitions.

8. ACKNOWLEDGMENTS

We would like to thank all of the UrbanSim research team members and collaborators. This research has been funded in part by grants from the National Science Foundation (IIS-0534094 and IIS-0705898), in part by a partnership with the Puget Sound Regional Council, and in part by gifts from IBM and Google.

9. REFERENCES

[1] J. Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, 1986.

[2] A. Borning, P. Waddell, and R. Förster. UrbanSim: Using simulation to inform public deliberation and decision-making. In H. C. et al., editor, *Digital Government: Advanced Research and Case Studies.* Springer Verlag, 2006. In press.

[3] Center for Urban Simulation and Policy Analysis, University of Washington. *Opus: The Open Platform for Urban Simulation and UrbanSim Version 4*

Reference Manual and Users Guide, 2007. Available from `http://www.urbansim.org/download`.

[4] B. Freeman-Benson and A. Borning. YP and urban simulation: Applying an agile programming methodology in a politically tempestuous domain. In *Proceedings of the 2003 Agile Development Conference*, Salt Lake City, Utah, June 2003. Available at `http://www.urbansim.org/papers`.

[5] A. Hunt and D. Thomas. *Pragmatic Unit Testing*. The Pragmatic Programmers, LLC, 2003.

[6] D. McFadden. Econometric models of probabilistic choice. In C. Manski and D. McFadden, editors, *Structural Analysis of Discrete Data with Econometric Applications*, pages 198–272. MIT Press, Cambridge, MA, 1981.

[7] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *Computing Surveys*, 37(4):316–344, 2005.

[8] R. E. Noonan and R. H. Prosl. Unit testing frameworks. In *SIGCSE '02: Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pages 232–236, New York, 2002. ACM Press.

[9] S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, Cambridge, 2003.

[10] A. Raftery, D. Madigan, and J. Hoeting. Bayesian model averaging for linear regression models. *Journal of the American Statistical Association*, 92:179–191, 1997.

[11] M. Sperber et al. *Revised^6 Report on the Algorithmic Language Scheme*, 2007. Available from `http://www.r6rs.org/`.

[12] D. A. Turner. A non-strict functional language with polymorphic types. In *Proceedings IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, 1985. Springer Lecture Notes in Computer Science 201.

[13] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[14] P. Waddell and A. Borning. A case study in digital government: Developing and applying UrbanSim, a system for simulating urban land use, transportation, and environmental impacts. *Social Science Computer Review*, 22(1):37–51, 2004.

[15] P. Waddell, A. Borning, M. Noth, N. Freier, M. Becke, and G. Ulfarsson. Microsimulation of urban development and location choices: Design and implementation of UrbanSim. *Networks and Spatial Economics*, 3(1):43–67, 2003.

[16] P. Waddell, M. Outwater, C. Bhat, and L. Blain. Design of an integrated land use and activity-based travel model system for the puget sound region. *Transportation Research Record*, (1805):108–118, 2002.

[17] P. Waddell, H. Ševčíková, D. Socha, E. Miller, and K. Nagel. Opus: An open platform for urban simulation. Presented at the Computers in Urban Planning and Urban Management Conference, London, June 2005. Available from `www.urbansim.org/papers`.

[18] P. Waddell, G. Ulfarsson, J. Franklin, and J. Lobb. Incorporating land use in metropolitan transportation planning. *Transportation Research Part A: Policy and Practice*, 41:382–410, 2007.