Systems Science

Portland State University

Occam 3.0

Design Proposal for Occam Core
4/30/00

Overview

A principal objective of the Occam 3.0 development project is to produce a flexible platform for further RA-related research. A key component of this platform is the *Occam Core*, which provides application programming interfaces (API's) which can be used in implementing further functions.

The Occam Core API's have the objective of insulating developers from details of memory representations of the various RA objects, and providing standard algorithms and operations which can be reconfigured in various ways. The Core must be efficient and robust, and it's API's must be designed for long-term stability, so that clients of the Core are not affected by future improvements and bug fixes.

The Core should also itself be extensible, so that different representation strategies can be explored. To facilitate this, it is assumed that C++ will be the development language, and that the RA objects represented by the Core will be provided to clients as C++ classes.

This document describes a proposed design for the Occam Core. The RA object model which is to be developed is described below, with details of each specific object.

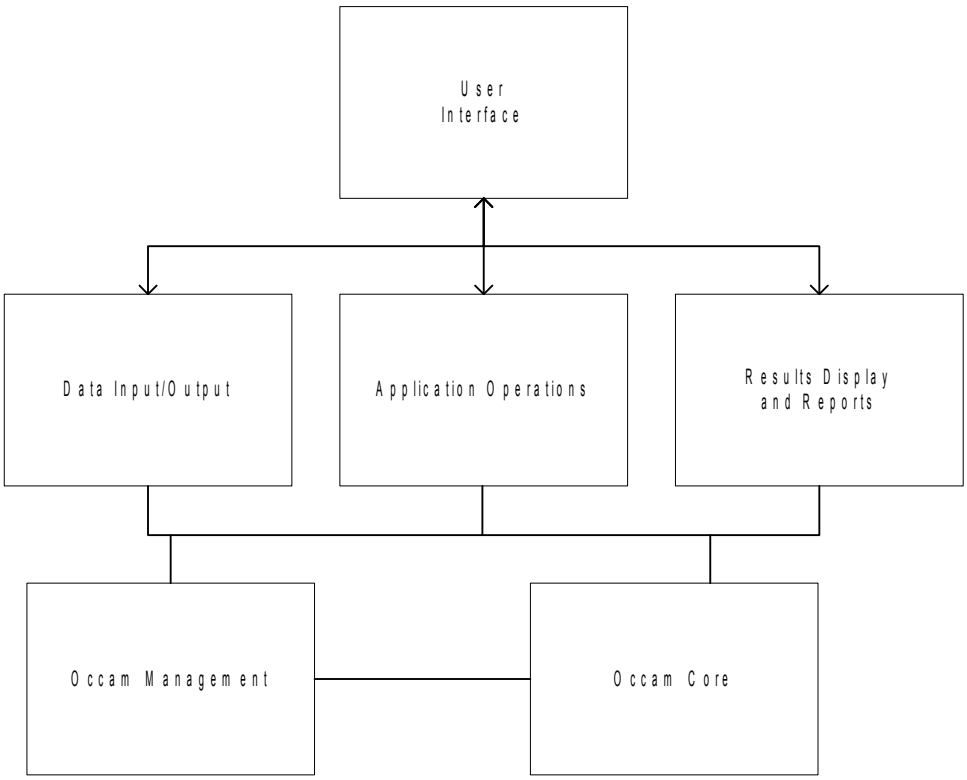
Principal Requirements

The list below gives some of the key requirements for the core, and identifies particular design features which address these requirements.

Variable Based Modeling (VBM)	Support the standard RA objects (Relations, Structures, Models, Tables, …
State Based Modeling (SBM)	Allow Tables where all table cells are not constrained. Allow Tables with different numbers of don't care variables in each cell.
Latent Variable Based Modeling (LVBM)	Allow introduction of new variables during analysis. Tag variables as to whether they are primary or latent.
Input filtering, binning, and time series analysis	Separate the input and preprocessing of observed data from the core.
Structural zeros and structural constants	Tag individual tuples as to whether they are fixed structural constraints.
Directed vs Undirected Systems	Tag variables as to whether they are dependent or independent.
Allow experimentation with fitting approaches	Separate IPF from the core.

System Architecture

The Occam Core is used as a component of a larger application, either the standard Occam application or some variant of it. The proposed structure of the Occam application is shown in the diagram below.



Under control of the user interface, data is loaded from external sources, and a representation of the data is constructed via the Occam Core and Occam Management (which manages the core objects). Then the user is able to perform various application operations, such a selecting a specific model; doing analysis of various types on the model; performing heuristic search of the lattice of models, etc. These application operations manipulate the objects in the core, perhaps producing new objects. Report and display operations access the objects in the core to produce displays of various kinds.

Object Model

The Occam Core provides facilities for construction and manipulation of various RA objects. The RA world consists of the following elements:

- Variables –a variable corresponds to some real-world observation which is part of the problem. A variable takes on one of a discrete number of values (its cardinality). In the core a variable is identified by name (mostly for reporting purposes), and also by position (for convenience variables are arranged in a fixed order, corresponding to their order of definition in the input data. Also, for some forms of display, each variable has a one-character[1] abbreviation, which is assigned by position. The values of the variable are assumed to be 0..n-1, where the cardinality is n.[2]

- Relations – a relation is defined by a set of variables, and represents the occurrence of some interaction among that set of variables. For convenience, relations are represented in canonical form with the constituent variables in positional order. A

[1] For large numbers of variables we will need a more extensible notation.

[2] For presentation purposes it is useful to have names for variable values (e.g., “male”, “female”), but this is done outside the core.

relation has a text representation which is the concatenation of the variable abbreviations (e.g., “ABC”). A specific relation, the top relation, consists of the complete set of variables.

- Structures – a structure represents a set of relations. A structure exists in a canonical form, where all embedded relations have been removed: that is, if “ABC” is a relation in the structure, then “AB”, “BC”, “AC”, “A”, “B”, and “C” are removed (since they are redundant). For convenience, structures have relations sorted lexically, so that comparison of structures for equality is simplified.

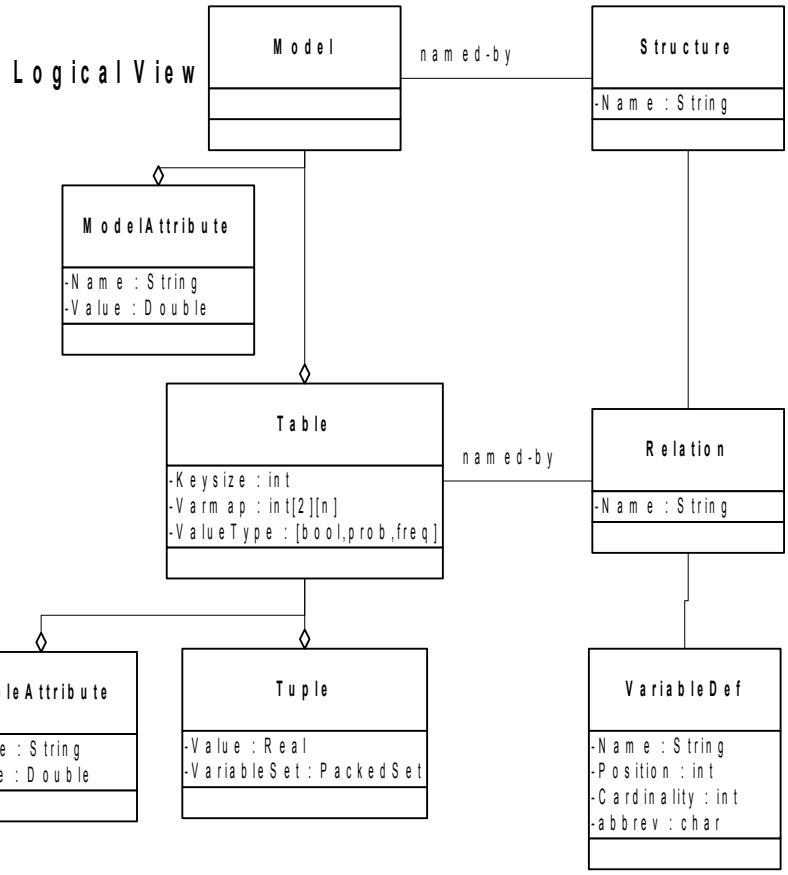
- Tuples – a tuple represents a set of (variable, value) pairs, together with a value for the tuple. Tuples are used in a number of different ways. Individual observations in a data file can be considered tuples, where the value of the tuple is 1. If the input data is in terms of frequencies, then the value of the tuple is the frequency; similarly if the data is a probability distribution, the tuple value is a probability. For set-theoretic analysis, tuple values are zero or one. The (variable, value) pairs in the tuple may or may not cover all the variables in the problem domain. A tuple must be associated with a Relation in order to be interpreted because it contains no explicit information about which variables are present or absent.

- Table – a table is a set of tuples, with a reference to the relation which defines the set of variables included in each tuple. For example, if the full variable set is “ABCDE”, and the projection is associated with the relation “ADE”, then each tuple only contains values for the first, fourth, and fifth variables (by convention, the unused positions are set to zero in each tuple).

- Model -- a model corresponds to a structure in the same way that a projection corresponds to a relation: it has a reference to a structure, and has a projection for each relation in that structure. In practice, since projections are very large and the same projections occur in many models, projections are reused among models in the analysis. Most analysis operations occur on models, so a model can hold an arbitrary number of computed attributes. Some of these are computed from the structure (such as df), and some from the underlying projections (such as informational measures).

- ModelAttributes and TableAttributes – these object hold computed values which are associated with a particular Model or Table. Different analysis approaches compute different attributes such as information, alpha, informational distance, etc.

The relationships among these objects are shown in the diagram below.



Object Descriptions

Below are the interface details for the proposed classes.

Class: Variable

Methods

Constructor – takes variable name, cardinality, position; initializes object.

Property access –getName(), getCardinality(), getPosition(). There are no set methods, because it is assumed that other data structures are dependent on cardinality and position of variables, so they must be immutable.

State Data

Name: String

Cardinality: int (max 255)

Position: int (0..n-1, where there are n variables)

Implementation Comments

None.

Class: Relation

Methods

Constructor – takes variable list, name as arguments; name is parsed as a sequence of variable abbreviations, and the positions of the corresponding variables are determined.

Property functions: getName()

getVariable(int index) returns pointer to Variable in the indicated position

State Data

List of pointers to Variables which are members of this relation.

Implementation Comments

We can assume that Variables are never destroyed after initialization, so it is safe to cache pointers to them in the Relation object.

Class: Structure

Methods

Constructor – takes no arguments; initializes an empty Relation list

addRelation() – appends a relation to the structure

makeCanonical – removes redundant relations, and sorts relations into canonical order.

getRelationCount – returns the number of relations in the structure

getRelation(int index) – returns one relation from the structure

State Data

List of relations

Count of relations

Implementation Comments

Since the number of relations in a structure is generally not too big, a simple linked storage or array storage method should be adequate.

Class: Tuple

Methods

Constructor – takes no arguments, does nothing (this is useful since Tuples are generally allocated in arrays, so it eliminates a flurry of constructor calls on array allocation).

setValue(float value) – sets the value stored in the tuple.

setKey(int keysize, Key *key) – sets the key associated with the tuple. The key is a variable length bit string, which encodes the values of all the variables relevant to this tuple. The key is copied into the tuple.

getValue() returns the value.

getKey() returns a pointer to the key.

State Data

Value – float.

Key – a bit string whose size is determined at runtime. This means the size of a Tuple is variable.

Implementation Comments

Tuples need to be variable sized (to avoid having a system limit on the number of variables and to keep the projection size as small as possible). It would be wasteful to store the size of a tuple in each tuple (since they are all the same for a given problem), so the tuple size (or equivalently, the key size) should be stored in the Projection object.

Many operations require access to tuples by the combination of variable values encoded in the key, so some access structure indexed by the key is important (perhaps a hash table). However, it is important to minimize extra space used for this indexing.

Class: Table

Methods

Constructor – takes a type, a pointer to a Relation, the keysize, and the number of tuples to allocate for the projection (an upper bound on its size). The type indicates whether the Projection stores set-theoretic probability, or frequency values.

addTuple(float value, Key *key) – adds a tuple to the projection, and indexes it based on the key.

getValue(Key *key) – returns the value of the tuple matched by the key, or zero if the tuple is not present in the Projection.

scanTuples() – initiates a scan of all the Tuples in the projection.

nextTuple() – steps to the next Tuple, during a scan.

buildProjection(Dataset *dataset, Key *keymask) – builds a projection from a dataset. The keymask is constructed from a Relation, by encoding 1's in the positions of variables to be retained in the projection, and 0's for variables to be removed (i.e., summed over). This method scans the dataset, and sums the appropriate tuples from the dataset. The nature of the sum operation depends on the type.

changeType(newType) – this method scans the tuples in the projection and updates their values, according to the new type. Type changes supported are: frequency to probability (this function just normalizes the values); frequency or probability to boolean (this function sets the value to 1 if it was nonzero).

State Data

Tuple table – some sort of indexed or sorted storage of the tuples. These will typically be sparse.

Keysize – the storage requirements for a single key.

Tuplesize – the storage size of a tuple, computed from Keysize.

Implementation Comments

Projections need to be reused, so there should be a hash table of already-created projections (hashed by the name of the relation they correspond to).

Class: Model

Methods

Constructor – takes a pointer to the structure this model corresponds to.

getStructure() – returns a pointer to the structure object for this model.

getAttribute(String name, double &value) – returns true if the attribute exists, and copies the value of the attribute into the value argument. Returns false if the attribute does not exist.

setAttribute(String name, double value) – stores a ModelAttribute associated with the model.

scanProjections() – begins a scan of the projections in the Model.

nextProjection() – steps to the next projection, during a scan.

State Data

List of ModelAttributes. A ModelAttribute is not a public object, but is accessed only via the Model.

Pointer to Structure.

List of Projections corresponding to the Relations in the Structure.

Implementation Comments

The analysis algorithms are outside of the model object (this allows new ones to be added more easily).

With this approach, Model objects are very lightweight (unlike projections).

Class: ModelAttribute

Methods

Constructor – creates a attribute with a given name and value. A ModelAttribute object is immutable once created.

getName, getValue – retrieve attribute values.

State Data

Name: String

Value: double

Implementation Comments

Occam Manager

The Occam Management package is another standard component of the Occam system, separate from the Core, but which provides services to manage Core elements. This separates representation from manipulation, and allows different management strategies to be explored.

During a given analysis session, previously computed results which may be required later (such as Models and Tables) can be retained by the Manager. At the same time, the Manager can recover resources when it is informed that certain results are no longer of interest.

Management functions will grow over time, but the principal functions to be implements in the first phase are:

- RelationCache – this data structure caches Relations and their associated Tables (if a Table has been constructed for that Relation). Reuse of Relations is a key optimization in the analysis, because Table storage dominates the memory requirements.

- ModelCache – this data structure caches Models and ModelAttributes which have been computed, and allows for later reporting on these values.

- StructureLattice – this data structure maintains the parent-child relationships among structures which have been created so far.

- InputTableCache – this data structure stores the Tables which represent the input data for the analysis session. Input processing operations (such as binning, time series conversion, and input filtering) produce tables which are stored in this cache.