April 10, 2003

METADB

Metaphor Representation using a Relational Database System

MICHAEL R. MEISEL
MMEISEL@UCLINK.BERKELEY.EDU

INTRODUCTION TO METADB

MetaDB is a database and user-friendly, web-based interface intended for use in cataloging, examining, and computing metaphor. MetaDB is intended to help realize the goals of the MetaNet project – namely, to catalog as many cross-cultural metaphors as possible in a complete, widely accessible, and human- and computer-searchable manner.

Embodied Construction Grammar

The concepts behind MetaNet and the general structure of MetaDB are based on Embodied Construction Grammar (ECG), a non-language-specific formalism for representing linguistic constructs that was originally developed by Benjamin Bergen and Nancy Chang as a part of the Neural Theory of Language (NTL) project. ECG itself (and, indeed, the NTL project as a whole) is based largely on the linguistic theories of George Lakoff and Mark Johnson.

The rest of this section contains my own interpretation of the concepts in ECG that are relevant to MetaDB. As such, it is designed primarily for those not familiar with ECG. If you fall into this category, keep reading. A lot of the rest of this document may not make sense to you unless you have read this section. For a more complete description of ECG (as well as the appropriate references), see "Embodied Construction Grammar in Simulation-Based Language Understanding" by Bergen and Chang.

SCHEMAS

A "schema" is a way of describing conceptual categories such as containers, mammals, or commercial transactions. Schemas are defined through their relationships to other schemas (see "Relationships Between Entities of the Same Type" and "Bindings") and through their "roles." Roles are simply the constituents that are required to define a category. For example, you could not have a commercial transaction without a buyer, a seller, some sort of good or service to be exchanged, and some sort of payment. Hence, a commercial transaction schema would need roles such as "buyer," "seller," "payment," and "goods." As a more abstract example, a "Container" schema would need roles such as "interior" and "exterior."

Roles can also have types. Role types are similar to data types in a programming language; they specify what type of thing a role can be filled with. For this reason, they are also often called role restrictions – they restrict the type of thing that can be assigned to a particular role. To learn more about how role types are used, see "Bindings."

METAPHORS

The concept of metaphor in ECG is the same as the intuitive concept of metaphor. However, this concept can be applied to a great deal of everyday language not typically considered metaphorical. For example, ECG treats the statement "prices rose" as a metaphor. If you don't see how this could be the case, ask yourself this: When the prices in question increased, did they, in any literal sense, move? The verb "rose" belongs to the domain of movement through physical space. In this example, spatial movement is the "source domain" of the metaphor – the conceptual category in which the literal meaning of the statement belongs. “Prices,” on the other hand, belong in a domain we might call commerce. In this example, commerce is the “target domain” of the metaphor – the conceptual category in which the subject of the metaphorical statement belongs.

You might have noticed that I referred to the source and target domains of a metaphor as “conceptual categories,” the same phrase that I used to describe schemas. This was no accident – in ECG, the source and target domains of a metaphor are, in fact, just schemas. A metaphor is a mapping between the roles of these two schemas. In the “prices rose” example, we might say that the “height” role of the “spatial movement” schema (the source domain) maps to the “expense” role of the “commerce” schema (the target domain). A single mapping like this is generally referred to as a “pair.” There would, of course, be several other pairs involved in this metaphor, all of which should in theory be enumerated when creating its formal representation.

RELATIONSHIPS BETWEEN ENTITIES OF THE SAME TYPE

We would be hard pressed to define any reasonable number of schemas without being able to express relationships between them. The same is true of metaphors. There are, in fact, two ways that entities of the same type can be related in ECG – entities can be “subcases” of other entities of the same type, or they can “evoke” other entities of the same type.

A “subcase” of an entity is a special or more specific case of that entity. For example, a “vessel” is a specific type of “container;” accordingly, the “vessel” schema would be a “subcase” of the “container” schema. Entities can also be subcases of multiple other entities of the same type. For example, a schema for “building” would certainly be a subcase of the “container” schema (or a subcase of one of the “container” schema’s subcases, and so on), but buildings can also be bought and sold, so the “building” schema would also need to be a subcase of the “good” schema.

When we say that an entity “evokes” another entity, we are stating that the evoked entity is somehow required in order to define the entity that evokes it. What this means is best illustrated by an example. Suppose we intended to define a schema for the concept “into.” It would be difficult to imagine such a concept without knowing about the “container” schema or a common spatial movement schema called “source-path-goal.” However, “into” is not a special case of either of these schemas, but rather a new concept built with the use of these schemas (see “Bindings”).

Additionally, evoked entities are given “local names,” names used to reference them in the context of the entity being defined. When we assign a local name to an evoked entity we can be said to be “instantiating” it. Instantiation is a concept from the realm of computer science that essentially means making a local copy of an entity that we can make changes to without affecting its properties in the general case. It is necessary to give each evoked entity a local name since it is possible to evoke two separate instances of the same type of entity. For example, we might want to define a schema for the concept of “meeting.” It would be difficult to describe the dynamics of “meeting” without one “instance” of the “source-path-goal” schema for each involved party.

BINDINGS

A binding is an association between two entities. Binding two entities can be thought of as setting them equal to each other. Returning to the “into” example, simply evoking the “container” and “source-path-goal” schemas doesn’t define how they relate to each other in order to help form the concept “into.” This is why we need bindings. Bindings are what tell us that the “exterior” role of the “container” schema refers to the same thing as (that is, is equal to) the “source” role of the “source-path-goal” schema, and that the “interior” role of the “container” schema refers to the same thing as the “goal” role of the “source-path-goal” schema.

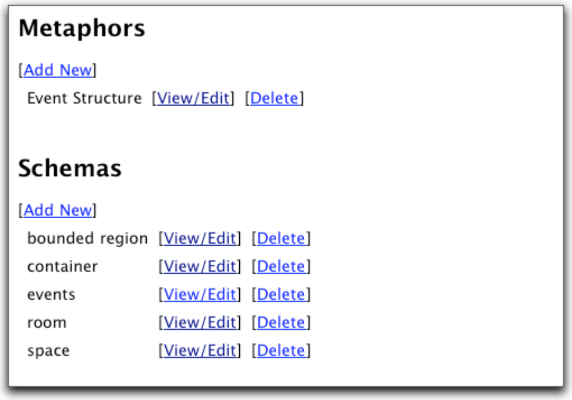
Now it should be clearer why role types/restrictions are useful. An “interior” in the “container” schema and a “goal” in the “source-path-goal” schema both refer to locations in physical space, so it makes sense that we would be able to bind them together. It wouldn't make sense, however, to create a binding between the "buyer" role in the "commercial transaction" schema and the "interior" role in the "container" schema, for example, since a "buyer" is not a location in physical space (he may reside in one, but is not actually one himself).

CONSTRAINTS

Bindings are a subset of a more general class called "constraints," which can also include any other restrictions required to define a specific entity. One type of constraint that is not a binding is an assignment of a value to a role. One example that might require such an assignment is a "triangle" schema. A triangle is a type of polygon, so we would want "triangle" to inherit from a general "polygon" schema, which would have a role for "number of sides." In order to distinguish subcases of this schema, such as "triangle," we would need to constrain the number of sides to the correct number for the specific subcase – in this case, three.

USER INTERFACE TUTORIAL

A sample version of the MetaDB user interface is currently available at http://mrm.mine.nu/metanet/list.php. You can actually take this tutorial at the aforementioned address if you like; otherwise, just follow along with the illustrations.


Metaphors

[Add New]

Event Structure	[View/Edit]	[Delete]

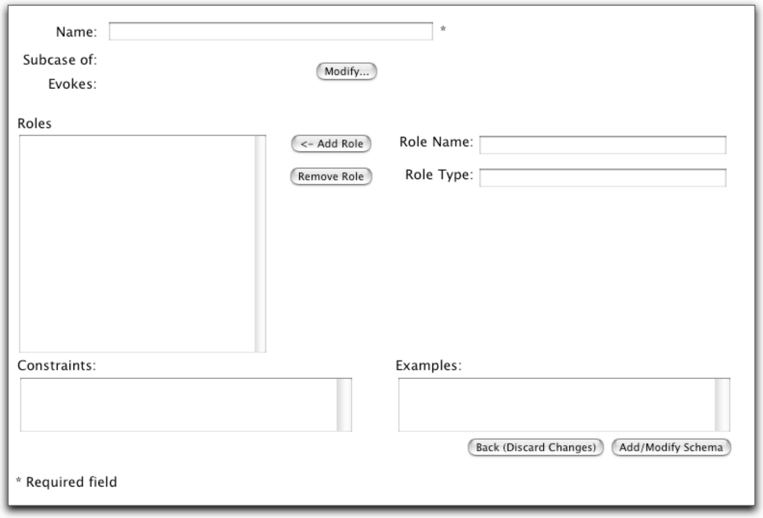
Schemas

[Add New]

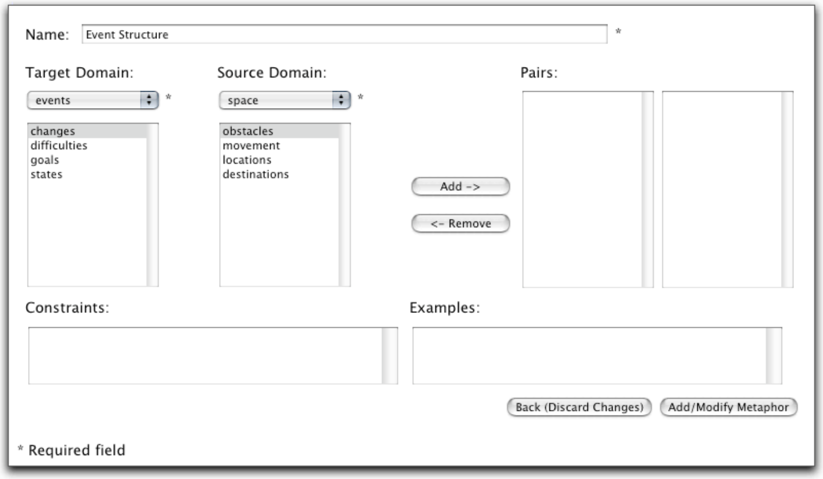
bounded region	[View/Edit]	[Delete]
container	[View/Edit]	[Delete]
events	[View/Edit]	[Delete]
room	[View/Edit]	[Delete]
space	[View/Edit]	[Delete]

The first page you will see when you load MetaDB's web interface is the main list. The main list displays all schemas and metaphors in the database. You can click on any of the "View/Edit" links to modify an existing entry, or click on one of the "Add New" links to add a new schema or metaphor to the database. You can also remove entries by clicking on the appropriate "Delete" link. For now, try clicking on the "Add New" link in the "Schemas" section. This will take you to the "Edit Schema" page.

The "Edit Schema" page allows you to edit existing schemas, or, in our case, create new ones. To create a new schema, first enter a name in the field near the top of the page. The system will now allow you to add this schema to the database if you like, but a schema wouldn't be very useful without any roles, so let's enter some. You can add roles one at a time by typing the name of the role into the field labeled "Role Name", optionally providing a type in the "Role Type" field, and clicking the "<- Add Role" button. After you click the button, the role will appear in the list to the left. If you make a mistake or simply change your mind, you can remove a role by selecting it in the list and clicking the "Remove Role" button. When you are done, click the "Add/Modify Schema" button to save your new schema in the database. You will be taken back to the main list page where you should now see your new schema listed in the "Schemas" section

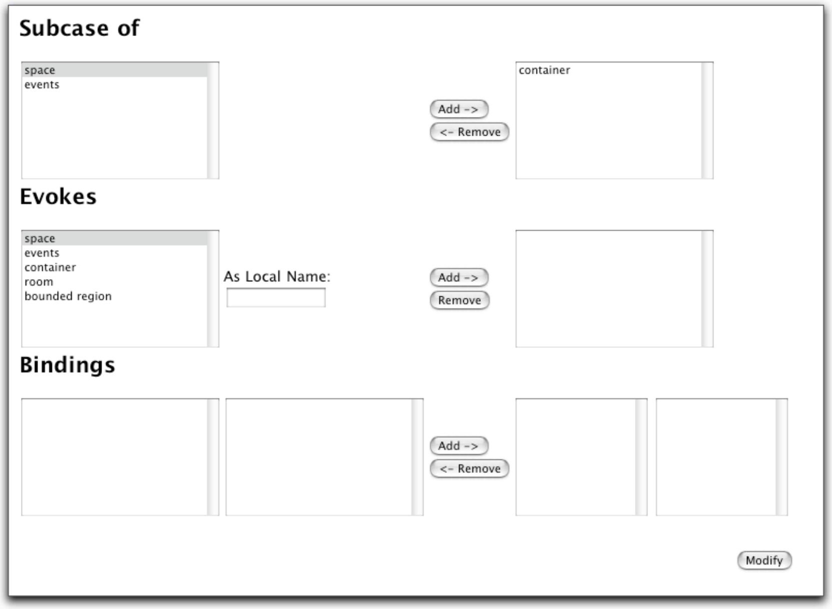


(in some browsers you may have to refresh the page first).



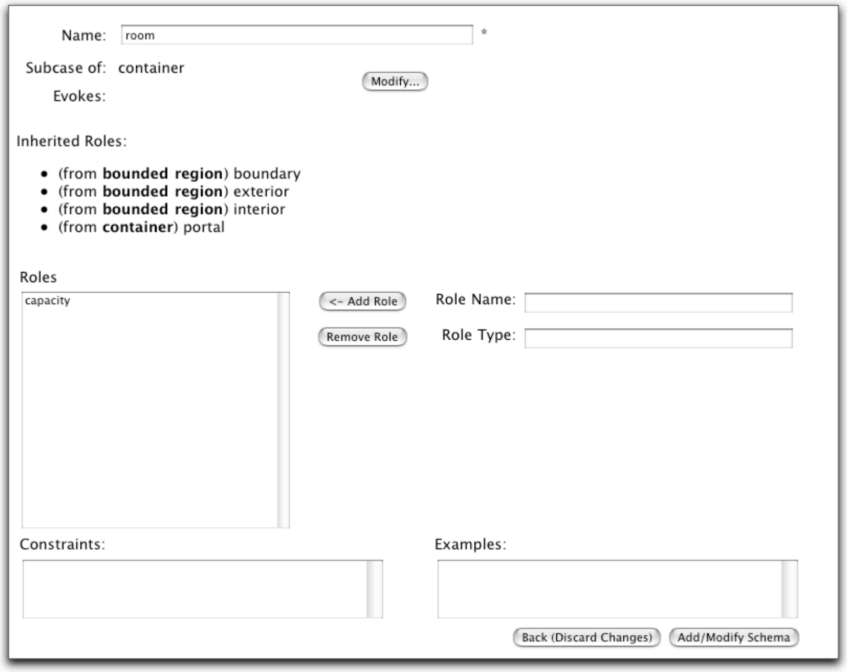
Once there are at least two schemas in the database, we can move on to metaphors. Click on the "Add New" link in the "Metaphors" section. This is the "Edit Metaphor" page. Just like on the "Edit Schema" page, the first step is to name the metaphor. Once you've done this, pick the source and target domains from the popup menus labeled "Source Domain:" and "Target Domain:", respectively. These popup menus contain all of the same schemas listed on the main list. When you select the source and target domains, you'll see the roles for the schema you selected appear in the list below. Now, you can create pairs for the metaphor by selecting one role in each list and clicking the "Add ->" button. You will notice that creating a pair with two roles removes them from the lists on the left. This does not mean that the role has been removed from the schema; this behavior is only to make sure that roles are not paired more than once. If you make a mistake or change your mind, you can remove a pair by clicking on one of its constituents in the list of pairs on the right (the matching role will be selected for you automatically) and clicking the "<- Remove" button. Once you have completed your pairings, click the "Add/Modify Metaphor" button to save your new metaphor and return to the main list.

Let's go back to the "Edit Schema" page for a moment; click on the "View/Edit" link next to one of the schemas listed in the "Schemas" section. You may have already noticed the "Subcase of:" and "Evokes:" labels just below the "Name" field. If you created this schema yourself earlier, there will probably be nothing listed there. Click the "Modify…" button to the right. You are now looking at the "Edit Relations" page. This is where you can specify which schemas this schema is a subcase of, as well as what schemas it evokes (evokes relations are not yet implemented) and any bindings required (also not yet implemented). Doing any of these things is fairly straightforward; the interface is quite similar to other portions of the MetaDB user interface that you have already used. To mark this schema as a subcase of some other schema, simply select that schema from the list in the "Subcase of" section and click the "Add ->" button to the right. To create an evokes relation, select the schema to be evoked in the list on the left of the "Evokes" section, type a local name into the field labeled "As Local Name:", and click the "Add ->" button directly to the right. The interface for creating bindings is the same as the one that you used for creating pairs in a metaphor. In the future, this same page will be used to create relations for metaphors as well. For right now, make this schema a subcase of some other schema (as described



above) and click the "Modify" button.

You are now back at the "Edit Schema" page, but a few things have changed. First of all, as you might expect, the schema you selected on the previous page is now displayed in the "Subcase of:" list near the top of the page (to the left of the "Modify…" button). Secondly, there is a new section of the page labeled "Inherited Roles." Each bulleted item in this section consists of the schema that the role is inherited from (in parentheses) followed by the name of the inherited role. Your schema will have these roles in addition to any roles that you define. The difference is that inherited roles cannot be modified directly. (To modify an inherited role, either change what schemas this schema is a subcase of or change the roles in their original schemas.) For example, the illustration below shows a schema for "room," which is a subcase of the "container" schema. The "portal" role comes from the "container" schema as you might expect, but the other roles are from the "bounded region" schema. This may seem confusing since the "room" schema is not a subcase of the "bounded region" schema. However, the "container" schema is a subcase of the "bounded region" schema, and therefore inherits its roles. Since those roles are (inherited) roles of the "container" schema, they also become (inherited) roles of the "room" schema, which is a subcase of the "container" schema. The "Inherited Roles" section tells you from which schema the role originated. This way, if a role is named incorrectly, for example, you know which schema you need to modify in order to fix it. You may see similar behavior now that you've made your schema a subcase of some other schema.


Name: room *
Subcase of: container
Evokes:

Inherited Roles:

- (from **bounded region**) boundary
- (from **bounded region**) exterior
- (from **bounded region**) interior
- (from **container**) portal

Roles
capacity

You may have noticed the "Constraints" field on both this and the "Edit Metaphor" page. This field allows you to enter arbitrary constraints by simply describing them in textual form. The preferred format for textual constraints is that used in the ECG formalism (see Feldman). As bindings are not yet implemented, a good temporary solution is to enter them into the "Constraints" field. To save your changes, click the "Add/Modify Schema" button.

IMPLEMENTATION

MetaDB has two separate components: a relational database and a web-based user interface (UI). The data model is designed to be flexible enough to represent the complex relationships that can occur in an ECG-based system. As a result, the database system is unable to enforce many important integrity constraints on the data (such as ensuring that the entities referenced as the target and source domains of a metaphor are schemas). Instead, the UI takes responsibility for enforcing these integrity constraints by simply not providing a way for the user to input incorrect data in such cases.

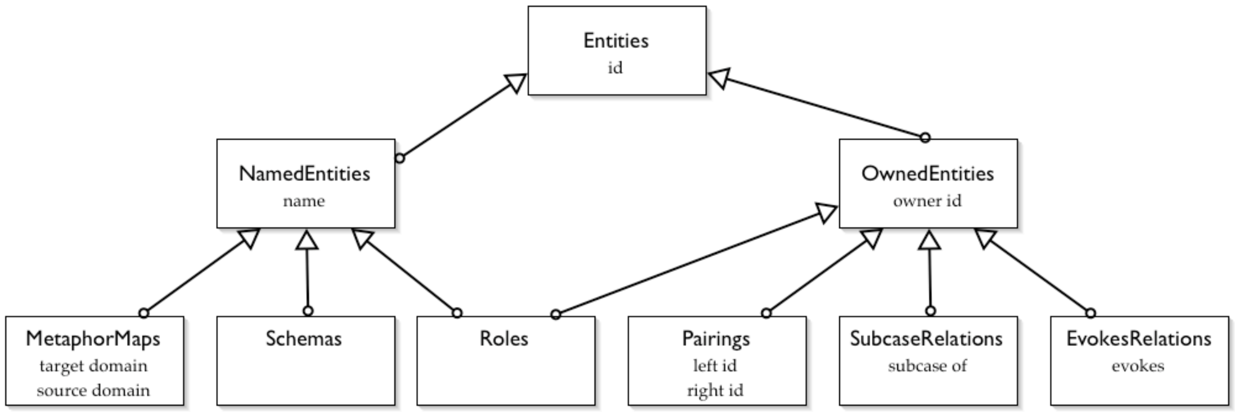
User Interface

The MetaDB UI is designed to require knowledge of the linguistic concepts involved (as described above), but not of the ECG formalism itself. Since it is beneficial to the MetaNet project to make data entry as easy as possible, using the MetaDB UI should require only the linguistic knowledge necessary to understand the data itself. As a result, the UI is designed to be as intuitive and straightforward as possible.

The UI is implemented using PHP and JavaScript. PHP scripts generate the HTML pages using data from the database, and the JavaScript allows the interface to be responsive to the user, providing feedback as the user works.

When creating or modifying metaphors or schemas, all necessary data is stored temporarily in HTML-form fields before the form is submitted for entry into the database. Form field values are initialized with either values from the database or values sent to the file using the POST method. Form field values can be changed dynamically using JavaScript.

Data Model



The data model for MetaDB is designed to be easily extensible. Every entity in the database has a unique integer id, all of which appear in the "Entities" table. This allows flexibility in referencing other elements in the database – for example, a subcase relation can be owned by a schema, a metaphor, or any new type of entity that might be added to the database in the future without any change to the "SubcaseRelations" table itself.

MetaDB uses the concept of table inheritance to accomplish this flexibility. Tables that inherit from other tables automatically inherit all of their column names, and the parent table automatically includes all entries in any sub-tables (but not vice-versa). In MetaDB, there are two subcategories of entities: named entities and owned entities. This dichotomy is meant to distinguish entities that are meaningful on their own from entities that are only meaningful with regards to other entities. A table may be a member of both subcategories if its entities can be referenced on their own but still in some sense "belong" to other entities. The "Roles" table is such a table. The diagram above shows the relationships between the various tables currently present in the system, where the arrows point from the sub-table to its parent. As bindings are not yet implemented, there is as of yet no table present to represent them.

Requirements

SERVER-SIDE

MetaDB requires PostgreSQL version 7.3 or higher. The UI requires a web server that supports PHP 4 with PostgreSQL extensions.

CLIENT-SIDE

The MetaDB UI should run on any modern web browser that supports JavaScript and Cascading Style Sheets (CSS). It has been thoroughly tested with the Gecko engine (Netscape 6 and newer, Mozilla, Phoenix), the KHTML engine (Konqueror, Safari with some minor bugs), and the latest versions of Microsoft Internet Explorer for both Macintosh and Windows.

Files

DB.PHP

This file contains wrapper functions for all of the database specific calls. This allows the system to be re-implemented using a different data storage method if necessary. However, all of the functions are still expected to return an array of arrays where the elements in the sub-arrays can be referenced by the column names from the existing data model.

LIST.PHP

This is essentially the root page for the system. It displays available metaphors and schemas and directs the user to the correct page to create a new metaphor or schema or modify an existing one. It also handles deletion.

METAPHOR.PHP

This page is the interface for creating new metaphors and modifying existing ones. Creating and modifying a metaphor's relationship to other metaphors is handled in relations.php. Any changes to the database are performed in submit.php.

PAIRINGS.JS

This file contains JavaScript functions for implementing a set of controls consisting of four HTML "SELECT" elements where items from the first two lists can be paired together and the pairs displayed as corresponding elements in the second two lists.

RELATIONS.PHP

This page is the interface for modifying the relationships between entities of the same type. In the current version, only schemas support this relationship, and bindings are not yet supported at all. In future versions, this page will handle editing relationships between metaphors as well.

SCHEMA.PHP

This page is the interface for creating new schemas and modifying existing ones. Creating and modifying a schema's relationship to other schemas is handled in relations.php. Any changes to the database are performed in submit.php.

SUBMIT.PHP

This page has no interface, but instead handles all modifications to the database other than deletion, which is handled by list.php.

UTILS.PHP

This file contains some useful utility functions for use in other PHP code.

DB-SCHEMA.SQL

This file contains the actual SQL code that defines the current data model. The proper column names that the system expects from the functions in db.php can be found here.

KNOWN ISSUES AND FUTURE PLANS

The system should make use of the "Extras" table. This would allow saving the contents of the examples field (which is not currently saved), and keeping track of updates to the database.

"Evokes" relationships are not yet saved to the database.

Bindings are not yet implemented, though the user interface for creating them is. However there should also be field for "setting."

The "Edit Relations" page is not yet hooked up to the "Edit Metaphor" page, so it is as of yet not possible to create relationships between metaphors.

There is a bug in the "Edit Metaphor" page that causes roles that are inherited from a schema's parents' parents (and parents' parents' parents, and so on) not to appear when the schema is selected as the source or target domain for the metaphor.

Role types/restrictions should be better specified. There are three types of role restrictions – references to schemas, references to elements in an external ontology, and references to "basic types." A "basic type" is simply an ordered, possibly infinite set from whose elements the role in question can take its values. A few examples of basic types are the integers, level of animacy, and gender. In order to make proper use of basic types, we would need to develop a way to store them in the database and an addition to the user interface for creating and modifying them.

Metaphors should also be able to have roles. Without giving a specific example, it seems that there are situations where a metaphor requires additional roles not present in either the source or the target domain.

The ultimate goal of the MetaNet project is not only to provide a tool for linguists but also to support an implementation of a language understanding system. The crucial step in doing this is to create a library for programmers to access data in the MetaDB database in a manner that enforces the same constraints that the user interface enforces and does not require knowledge of the data model. The library should provide a similar set of operations to those provided by the user interface. Hopefully, MetaDB will eventually be used as a primary storage and retrieval system for linguistic information in a larger language understanding system with supporting input provided through the user interface.

REFERENCES

Bergen, Benjamin and Nancy Chang. "Embodied Construction Grammar in Simulation-Based Language Understanding." 2002. To appear in Jan-Ola Östman and Mirjam Fried (eds.), *Construction Grammar(s): Cognitive and Cross-Language Dimensions*. Johns Benjamins.

Feldman, Jerome A. "A Proposed Formalism for ECG Schemas, Constructions, Mental Spaces, and Maps." 2002. Unpublished.

Lakoff, George and Mark Johnson. *Metaphors We Live By*. Chicago: University of Chicago Press, 1980.