CharGer: A Graphical Conceptual Graph Editor

Harry S. Delugach
Computer Science Department
University of Alabama in Huntsville
Huntsville, AL 35899
delugach@cs.uah.edu

Overview of Features

CharGer is a conceptual graph editor intended to support research projects and education. Its current version is primarily an editor to create visual display of graphs. It is deliberately and explicitly a research tool meant for conceptual graph researchers to explore implementation issues in conceptual graph interfaces. Its main features are summarized below. This paper discusses some of the features of CharGer that are specific to conceptual graphs or are considered by the author to be of sufficient interest to the community. Typical or ordinary operations (e.g., cut/paste, etc.) are not discussed.

Features currently supported

These features are currently supported (as of version 2.4b, dated 2000-09-05):

- Save and retrieve graphs from files, in a proprietary (i.e., non-standard) text format. These graphs have the extension .cg and are portable across all platforms.
- Save and retrieve graphs from files in CGIF standard interchange format. These graphs have the extension .CGF and are portable across all platforms.
- Display a graph's CGIF version or a graph's natural language paraphrase, or copy it to the clipboard as text, or save it to a text file.
- Any number of graph windows may be opened for editing.
- Concepts, relations and actors are all supported for editing.
- Type and relation hierarchies may be edited and saved the same way as a graph, and may be intermixed on the same sheet of assertion.
- Graphs may be labeled as to their intent (e.g., query, definition, etc.)
- Contexts are supported, including arbitrary nesting.
- Auto-save safeguards to prevent losing a graph. Other modifications to prevent losing data.
- Copy/paste of graphs using an internal clipboard
- Unlimited undo/redo for editing changes (limited only by available system memory)
- Portability to all major platforms (i.e., as portable as Java based on JDK 1.2)
- Activation of some built-in actors, including several primitive ones for arithmetic and elementary operations, with an optional “animation” to show their operation.
- Database access through actors, although restricted to tab-separated text file “databases” at present.

- Capability to automatically create a skeleton graph from a database suitable for providing database semantics.
- Hotkeys to move and explicitly resize graph objects and contexts
- Keystrokes to switch editing tools.
- A natural language paraphrase feature, to paraphrase a graph in English (other languages on the way)
- Ability to set user preferences and save them between sessions.
- Some conceptual graph operations (e.g., join, match) (see Known Bugs and Restrictions)
- Adjustable parameters for the matching scheme applied to joins and matching

Features not currently supported

The following useful features are not currently supported; plans are to implement them in the future.

- Validation facilities (except for enforcing CG formation rules)
- Ability for user-defined actors.
- Type and relation hierarchies in CGIF form.

Why the name "CharGer"?

The University of Alabama in Huntsville has several sports teams nicknamed the "Chargers". A catchy name at that. Since the "CG" initials appear in it, it seemed a natural choice. And I like the image of forging ahead, attacking research problems and leading the way. So there it is.

Documentation

Documentation of ***CharGer*** consists of a set of HTML files (generated from javadoc) documenting the classes, some tool tips and informative messages during execution, and a manual.

Some General Features

GRAPH STRUCTURE

A **`graph`** in the ***CharGer*** environment is any collection of concepts, relations, actors and type labels, linked by their appropriate arrows, co-referent links, input/output arrows or super/sub-type arrows. A collection of un-connected elements can still be called a graph.

We have introduced a new graphical element not previously defined in conceptual graph display environments, namely, a **`typelabel`** as a text string above a horizontal line, as **`ANIMAL`**. This gives us an easy facility to draw type hierarchies and have them stored either separately, or included as part of another graph.

"PURPOSE" OF A GRAPH

Several authors have made reference to a graph's *intent*, that is, whether the graph is intended as a query, an empirical observation, a rule, a schema, a definition, etc. We consider this to be an important aspect of the content of a graph and therefore include facilities for indicating such intentions. As yet, there is no further use for these labels; they can be safely de-activated in the current version. In a practical knowledge base, however, we expect they would be of some use.

“PHILOSOPHY” OF CHARGER’S ARCHITECTURE

The “philosophy” behind the editor is that it supports an open world. That is to say, the user may enter anything he/she wants as long as it is well-formed by the general rules of conceptual graphs. If the user enters some elements that are already defined, then the system should use that definition to enforce any constraints that result from it. If the user enters an element that is *not* already defined in the system, it can still be included, but no constraints will be enforced on it and little processing should be expected. Of course, the user is free to then provide definitions, facts, etc. that deal with the new elements.

Selections

CharGer’s idea of a selection tries to accommodate conceptual graph semantics in the following ways:

- Arrows, co-referent links, etc. are in the selection if they are clicked on, shift-clicked on, or included in a dragged selection rectangle.
- If a concept, relation, actor or type label is cut or cleared, its arrows, co-referent links, etc. are cleared with it, even if they lie outside the selection; it makes no sense to have an arrow without a graph node at both its ends.
- If a selection is pasted, only those arrows, co-referent links, etc. that are entirely within the selection are pasted; other arrows, co-referent links, etc. are not pasted, even if they were selected; this is because there is no practical way to connect the links’ other ends.

Installation

Using CharGer requires availability of a Java package, at the JDK 1.2 level (Java 2) or higher. Do the following:

- Obtain the `CharGer24b.zip` file.
- Un-zip the `CharGer24b.zip` file. This step should create the `CharGer24b` folder and all the rest.
- Send an email to delugach@cs.uah.edu to let me know you were here.

The `CharGer24b` directory is referred to as your "top level" directory. Within it, there should be the files:

- **CharGer.jar** - Java archive (jar) containing the classes needed to execute CharGer; these are platform independent at the JDK 1. 2 level. This file is ready to run as long as Java, or at least a Java Virtual Machine (VM) is installed on your computer. Also includes the Notio package, courtesy of Finnegan Southey.
- **Lib/Preferences** - A file containing a number of parameters and options to CharGer. Self-documenting?
- **Lib/GIF/*.gif** - All the pictures and icons CharGer uses.

- **Graphs** - Where CharGer expects to find graphs by default. Can be opened and saved from/to any directory.
- **Databases** - Where CharGer expects to find its tab-separated text databases for the actor. See below for more information.

To run CharGer, you must have Java available. On a PC, this is called `java.exe`. You should be using Java's JDK 1.2 version or higher. It is possible that earlier versions will work in some cases, but since the JDK is freely available from www.javasoft.com it is probably a good idea to get the newest version first and install it before running CharGer. Follow its instructions for installing Java, paying particular attention to the setting of your local PATH variable to include Java's binaries (e.g., **java.exe** and **jar.exe**). To check whether Java is in your path (on a PC), do "set" at the command prompt and see if JDK paths are in the PATH variable.

Charger's main class is called `CGMain`.The Notio classes, included with this release, are also required.

From the MS-DOS prompt on Windows or a shell prompt under LINUX, make the "top level" directory your current directory, and use the following:

```
cd ...\charger24b
java -cp CharGer.jar CGMain
```

For the convenience of Windows users, the file **cg.bat** will invoke this from the command line as "cg". To run the system on a Macintosh, you must run a Java Virtual Machine and identify CGMain as the main class to execute. See your Macintosh Java documentation for more information.

Operations

MAKE CONTEXT

Make the current selection into a context. Use the Selection Tool to establish a selection before using the **`MakeContext`** Tool. If the selection would cause overlapping contexts, the user is prevented with a warning.

UNMAKE CONTEXT

Removes the outermost context border in the selection, thus attaching its contents to the graph in which it is nested (or the entire graph if the context was not nested). If there are concepts, etc. selected that are not in the outermost context, they are ignored. If there is more than one equally nested outermost context, one of them is chosen arbitrarily to be un-made.

Among the more interesting operations in CharGer are the following:

MAKE GENERIC

Remove the referents of the selected concepts and contexts. In other words, make individual concepts into generic ones. If a concept or graph has no referent, then it is unchanged. If a context has a graph descriptor, that descriptor is unchanged. Relations, actors, and types are unaffected.

Match to Open Graphs

Matches the current graph to any other graphs in an open window. If a match is found, then the matched graph is selected and placed in a **new** editing window. If no match is found, then a message is displayed.

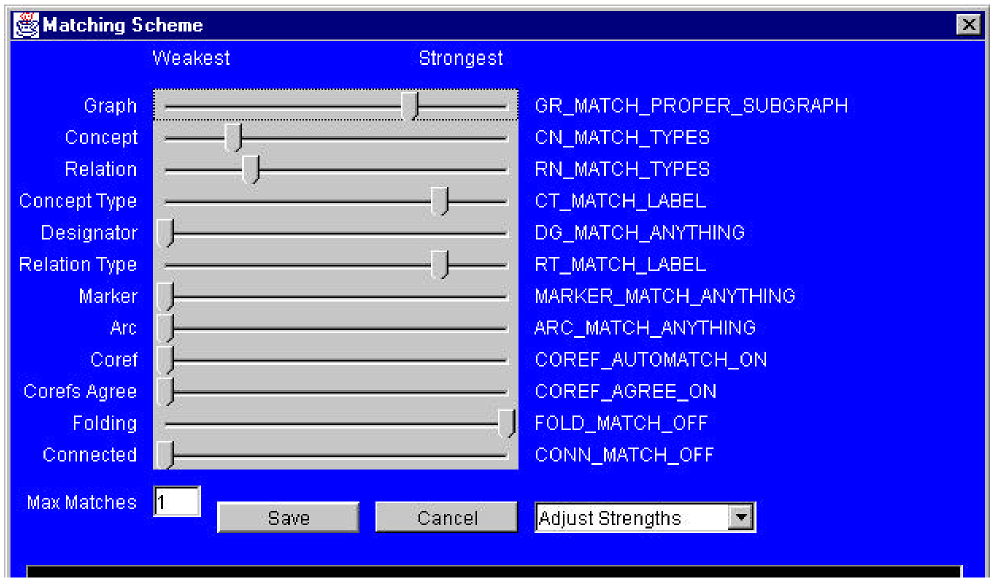
Note: The graph matching is not well tested and only crudely implements Notio's graph matching capabilities. It is for illustration purposes only.

Join selected nodes in open graphs

Performs a join operation using two or more graphs, starting with the current graph. Each graph should have its own set of one or more selected items. If there is a valid way to join the graphs, according to the current matching scheme, a joined graph is created and opened in its own new window.

Modify Matching Scheme

Allows the user to adjust the parameters of matching and joining. The command brings up a window like Figure 1.



Matching Scheme

Weakest Strongest

Graph	GR_MATCH_PROPER_SUBGRAPH
Concept	CN_MATCH_TYPES
Relation	RN_MATCH_TYPES
Concept Type	CT_MATCH_LABEL
Designator	DG_MATCH_ANYTHING
Relation Type	RT_MATCH_LABEL
Marker	MARKER_MATCH_ANYTHING
Arc	ARC_MATCH_ANYTHING
Coref	COREF_AUTOMATCH_ON
Corefs Agree	COREF_AGREE_ON
Folding	FOLD_MATCH_OFF
Connected	CONN_MATCH_OFF

Max Matches 1

Figure 1. Matching Scheme Parameters.

To find out what the slider positions mean, move the slider and see the meaning of the slider's parameter. (These parameters are obtained from Notio's MatchingScheme class, which defines the parameters for matching in Notio.)

The principle behind matching is that the matching process can be less constrained (weaker) so that more graph elements may potentially match, or be more constrained (stronger) so that fewer graph elements may potentially match.

An interesting feature for a tool would be its ability to determine the "best" (i.e., strongest) match between selections. In many applications dealing with typicality or ambiguity, it would be useful to have multiple matches possible, ordered by the strength by which they match.

Tools

Some of the CharGer's tools are novel, since there is no formal display form for type and relation hierarchies. We have adopted the following tool conventions to support these features.

Type Label Tool

The type label tool allows you to insert types (usually in a hierarchy) onto the graph drawing area. The type label tool can also be chosen by pressing the "T" key on the keyboard.

Relation Type Label Tool

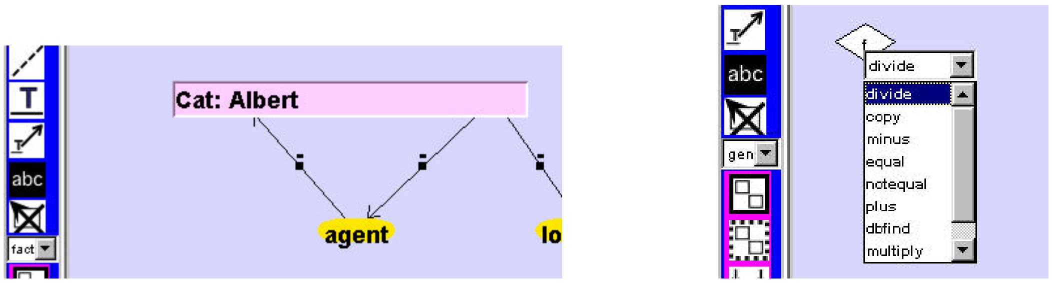
The relation type label tool allows you to insert relation types (usually in a hierarchy) onto the graph drawing area. The relation type label tool can also be chosen by pressing the "L" key on the keyboard.

Generalization/Specialization Line Tool

This tool draws a generalization/specialization relationship between two concept types or between two relations. The generalization/specialization tool can also be chosen by pressing the "," (comma) key (an un-shifted "<") on the keyboard.

Edit Text Tool

abc With this tool chosen, a click on a concept, relation, actor, context border, or type label will open a Text Edit Box for the selected element's label. For a concept or context, a textedit box will appear, as in Figure 2(a). For an actor or relation, a pop-up menu will show the list of pre-defined actor labels from which to choose, as in Figure 2(b). The user may pick an actor/relation name from the menu or create a new (undefined) actor by picking the name "Other…" and then editing it again, inserting the desired name. To edit the label on an arrow, click on the dot (■) and edit its label.



(a) (b)

Figure 2. Editing Node Names.

Text labels for concepts, relations and contexts can be changed through use of a Text Edit Box. Open a Text Edit Box by doing either of the following:

- With the Selection Tool, double-click on the label you want to change.
- With the Edit Text Tool selected, click once on the label you want to change.

Use conventional text editing keys to make your changes. To save the changes, position the cursor in the Text Edit Box and press return or enter.

Commands

Make Context

Make the current selection into a context. Use the Selection Tool to establish a selection before using the **`MakeContext`** Tool. If the selection would cause overlapping contexts, the user is prevented with a warning.

UnMake Context

Removes the outermost context border in the selection, thus attaching its contents to the graph in which it is nested (or the entire graph if the context was not nested). If there are concepts, etc. selected that are not in the outermost context, they are ignored. If there is more than one equally nested outermost context, one of them is chosen arbitrarily to be un-made.

Preferences

There is a preferences panel available that allows the user to adjust some settings for their own use. Preferences are arranged in three groups, Appearance, Compatibility, and Actor Settings. The Appearences preferences will not be dealt with here. Some (but not all) of the preferences are discussed below.

Compatibility

Activate graph purpose labels

Keep track of whether a given graph is generic, a fact, etc. The labels and their meanings are:

`def`	Definition
`rule`	Rule
`generic`	Generic
`type`	Type Hierarchy (for `convenience)`
`fact`	Empirical fact
`query`	Query
`schema`	Schema
`“” (empty string)`	None of the above

Enforce standard relation arguments

Enforce the standard relation rule such that a relation has at most one output argument. Never enforced for actors.

Actor Settings

Allow null actor arguments

If checked, considers a null argument legal to an actor. Usually means that the actor will ignore the argument.

Allow actor links across contexts

The ANSI standard does not allow relation links across context boundaries (whether an actor or relation). Strictly speaking, an actor should link to a concept in its own context, with a coreferent link from that concept into the other context. Checking this box allows an actor link to connect to any concept, regardless of its context.

Animate actor firing

Whether to activate actor firings in increments, visually marking those actors and concepts that are involved in each firing.

Actor Activation

CharGer supports actors, generally using the techniques described in Sowa's original book. There are a few built-in actors, with pre-defined semantics. Most of these are simple arithmetic operations. For example, there is a plus actor to implement addition. Consider the following graph. It would be good practice for you to draw the graph in Figure 3 before proceeding:

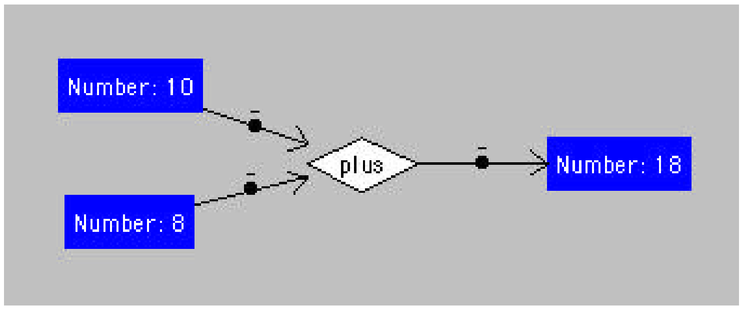


Figure 3. Actor example.

Note that the output number's referent is the sum of the two input number's referents. To understand how an actor works, draw the graph above, and then change the input **`Number: 10`** to read **`Number: 5`**. Note how the output number changes, as shown in Figure 4:

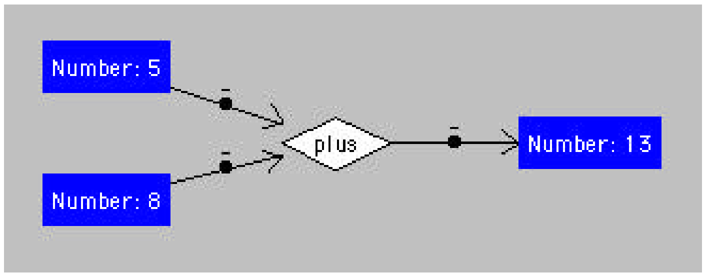


Figure 4. Changed actor graph.

Now change the output concept **`Number: 13`** to some other number. Note how it changes back to 13. The reason is that the plus actor denotes a functional dependency where the output concept is functionally dependent upon the input concepts; thus changing it causes the graph to re-evaluate itself and restore the original constraint.

The following executable actors are built-in to ***CharGer***. T means any type; null means bottom (⊥). In general, actors' inputs are not commutative (i.e., their order matters, as denoted by the numbers on their input arcs.) Future versions will also allow actors to have a varying number of input concepts, when the meaning would be clear (e.g., **`plus`** could have two or more numbers to be added).

	Input Concepts		**Output Concepts**		
Actor Name	**Quantity**	**Type(s)**	**Quantity**	**Type(s)**	**Semantics**
`plus`	Two	**`Number`**	One	**`Number`**	Output number is sum of the two inputs. Commutative.
`minus`	Two	**`Number`**	One	**`Number`**	Output number is concept 1 minus concept 2.
`multiply`	Two	**`Number`**	One	**`Number`**	Output number is product of the two inputs. Commutative.
`divide`	Two	**`Number`**	One	**`Number`**	Output number is concept 1 divided by concept 2.
`dbfind`	Two	**`Database`** **`T`**	One	**`Number`**	Output concept is the value associated with **`T`**'s referent in the file denoted by the **`Database`** concept.
`equal`	Two	**`T`**	One	**`T or null`**	Output type is **`T`** if inputs are strictly equal; otherwise **`null`**
`notequal`	Two	**`T`**	One	**`T or null`**	Output type is **`T`** if inputs are strictly not equal; otherwise **`null`**
`copy`	One	**`T`**	One	**`T`**	Input (referent only) is copied to output concept.
`greaterthan`	Two	**`T`**	One	**`T or null`**	Output type is **`T`** if input 1 greater than input 2; otherwise **`null`**
`greaterequal`	Two	**`T`**	One	**`T or null`**	Output type is **`T`** if input 1 greater than or equal to input 2; otherwise **`null`**
`lessthan`	Two	**`T`**	One	**`T or null`**	Output type is **`T`** if input 1 less than input 2; otherwise **`null`**
`lessequal`	Two	**`T`**	One	**`T or null`**	Output type is **`T`** if input 1 less than or equal to input 2; otherwise **`null`**

There is as yet no actor-definition mechanism; that is a future enhancement. (Suggestions for appropriate mechanisms are welcome.)

Database Linking

One of ***CharGer***'s features is its ability to use an external "database" that actor <dbfind> can use. This ability is built into ***CharGer***'s actor definitions. For ***CharGer***'s purposes (as of the current version) a database file is a tab-separated tabular text file. For example, suppose the file DBElement.txt contains the following tab-separated values:

```
Number  Element     Symbol
1       Hydrogen    H
2       Helium      He
3       Lithium     Li
```

CharGer's **`<dbfind>`** actor is designed to illustrate ***CharGer***'s interface to a database. The following graph shows how such a database would be used in a conceptual graph with actors:

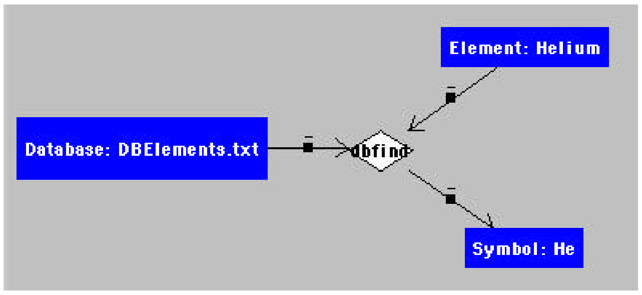


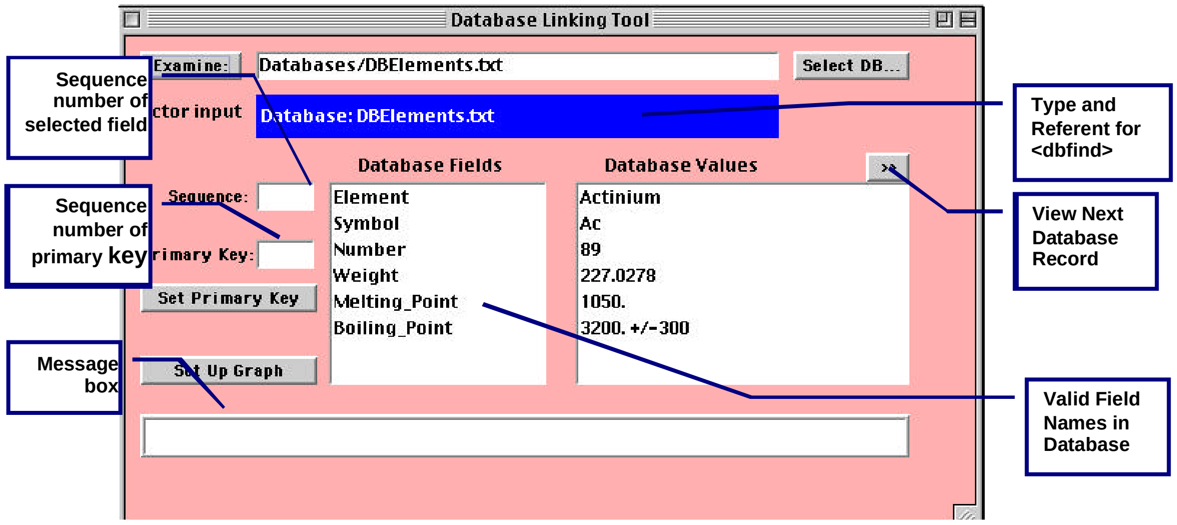
Figure 5. Illustration of the <dbfind> actor.

To see how the actor works, use the “abc” tool to change “Helium” to “Lithium”. Note how the Symbol and Number concepts now have new referents! Another example is to change the Element referent to “Earth” (which is not an element) and note how Symbol becomes null (which is equivalent to ⊥ in conceptual graph terms).

There are a few important guidelines for how the **`dbfind`** actor works. First, it requires an input concept of type **`Database`** whose referent is a real file name. Second, another input must have a type which exactly matches some field type (i.e., a header name from the file). Third, there must be a single output concept, whose type also matches some field type. (It’s permitted for the input and output types to be the same; it’s a good way to see whether the input concept’s referent value is actually found in the database.) Use the **Database Linking Tool** to see what type names are valid in a given database.

A database with no tab characters will be treated as a one-column table whose entire first line is considered the only valid field name.

To make it easier for users to create usable graphs with database actors, the **`Database Linking Tool`** window has been provided. It is accessed through the **`Tool`** menu in the Hub. When activated, the window looks something like Figure 6:


Database Linking Tool
Examine:
Select DB...
ctor input
Database Fields
Database Values
>>
Sequence:
rimary Key:
Set Primary Key
Set Up Graph
Sequence number of selected field
Sequence number of primary key
Message box
Type and Referent for <dbfind>
View Next Database Record
Valid Field Names in Database

Databases/DBElements.txt

Database: DBElements.txt

Database Fields	Database Values
Element	Actinium
Symbol	Ac
Number	89
Weight	227.0278
Melting_Point	1050.
Boiling_Point	3200. +/-300

Figure 6. Database Linking Tool Window.

This window helps the user determine what are valid inputs and outputs to the database lookup actor(s). The window can also be used to set up a template graph for defining a database's semantics (with or without a primary key).

`Examine:` will open the "database" file whose name is provided. The first line of the file must be a header consisting of a series of two or more tab-separated strings, each of which is the field name corresponding to the subsequent lines in the file. At present, ***CharGer*** only works with text-only, tab-separated lines.

`Select DB` will open a file dialog giving the user a chance to pick a database file. The database file must reside in the chosen Databases folder; ***CharGer***'s default, or a folder selected by the user. This is a known limitation to be rectified in the future.

`Input Concept` indicates the valid input concept representing the database, to be used as an input to a <dbfind> actor.

`Database Fields` contains the exact names of the columns in the database file. These names are to be used as type-labels for the input concepts to the lookup actor. Referent values for such actors can be any value for the type. For example, [Number: 5] would be a valid input concept for the fields given in the example screen. Selecting one of the database fields puts its sequence number into the **`Sequence`** box.

`Set Primary Key` makes the selected field name into the database file's primary key. When one of the field names is selected, show index will give its sequence number in the list. This is the same sequence number that will be used for the primary key sequence number.

`Set Up Graph` creates a new graph, in an editing window, with the database concept as input to a set of <dbfind> actors, a primary key set up as the other input (if a primary key has been selected), so that the user can begin a graph already connected to a database. This is a handy tool for describing database semantics in a conceptual graph, and checking the semantics of the graph using actual values in a database.

Known Restrictions

Some of CharGer's restrictions are shown below.

RESTRICTIONS

- New primitive (built-in) actors must be coded in Java, according to the procedures outlined in comments to the **`GraphUpdater`** class.
- Actors do not yet operate with contexts as input or output, although such input and output contexts can be drawn. Inputs and outputs for actors must be simple concepts. The concepts themselves may be nested in contexts.
- Actors can only be activated by editing one of their input or output concept referents, or the actor name itself. There is no explicit activation.
- Lambda expressions are not yet supported.
- Arithmetic with real numbers lacks precision; ***CharGer*** is not yet meant as a reliable calculator for real numbers.
- When moving a context, all it contents go with it logically; if the move causes additional graph elements to appear enclosed visually, those additional elements are not logically part of the context! This issue will be addressed in a future release.

- Type labels are NOT exported in the CGIF form except insofar as they are passed as type labels of concepts, contexts and relations.
- A concept can safely be a member of only one coreference set. It is not yet clear to this author (and others) how to interpret the semantics of a concept that's a member of more than one coreference set.
- When joining or matching forms a new graph, elements of the new graph may overlap, since each set of elements is derived from a separate graph. The graph is stored internally in its correct form (as would be displayed by the CGIF format).
- Matching is not completely implemented. That is, several combinations and/or matching parameters will either not work at all or produce unpredictable results. The matching in ***CharGer*** is provided by Notio, which is currently undergoing additional development.

Accomplishments

CharGer has been a labor of love for its author, as well as a useful tool in discovering many remaining issues both in the CG theory and in interactive interfaces to support CGs. Incorporating the Notio API package was instrumental in discovering minor flaws in its architecture, as well as enhancing CharGer's capabilities through fundamental code re-use. It is my belief that coordination among CG tool makers is crucial to advancing the use and popularity of CGs. I hope that CharGer's development has given some ideas to other tool makers that will help them in enhancing their tools.