Overview of TinkerCell Code

Introduction

The complete TinkerCell application is specifically for synthetic biology. However, if the plug-ins are removed, then TinkerCell is just a library for creating network drawing applications. The library that provides all the GUI functions is called the TinkerCell "Core" library. Developers who want to write a GUI application for drawing networks can use the Core library to make the task simpler. The Core library provides many basic features such as copy/paste, undo/redo, multithreading, Python/Octave/Ruby interpreters, alignment, and text boxes. The Core library can optionally support ontologies for more intricate network diagrams.

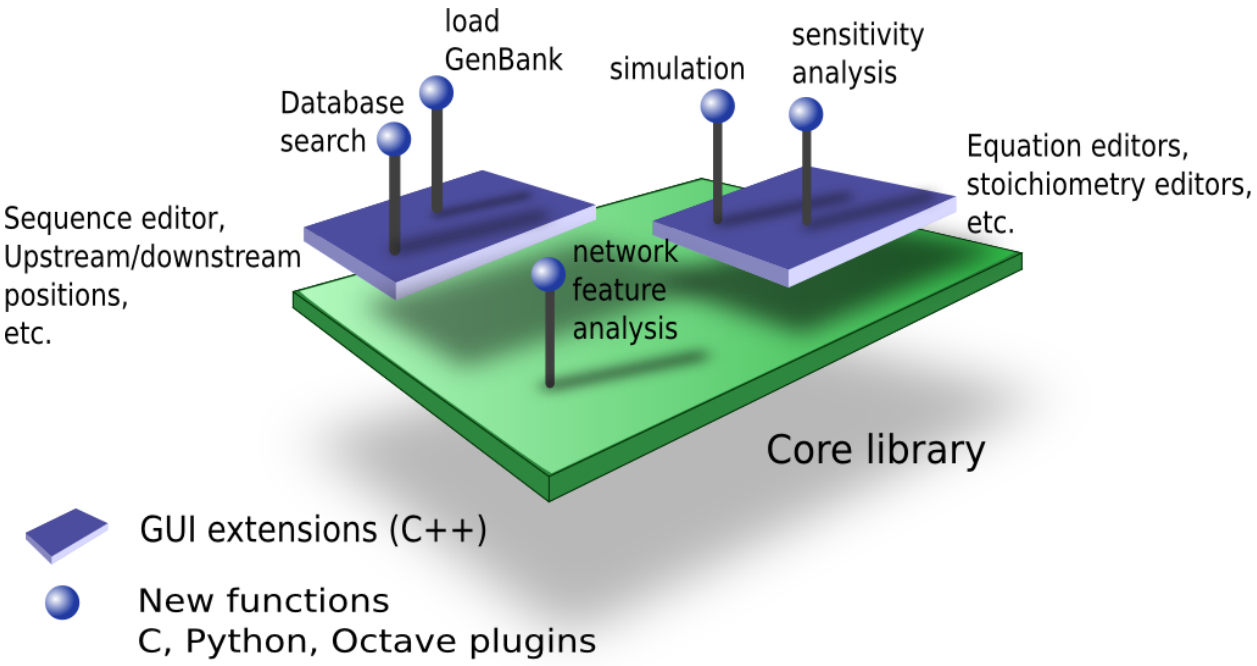
Developers interested in synthetic biology (or systems biology) will be more interested in writing plugins using scripting languages. There is another document that provides more details on the Python and Octave interface: https://docs.google.com/View?docid=dcx9fkfb_234gf8st9hc

If you are interested in using the Core library to develop new applications or new GUI features in TinkerCell, continue reading this document and also take a look at examples of extensions: https://docs.google.com/View?docid=dcx9fkfb_431gjjp72cv

TinkerCell code can be summarized as a layered architecture. The bottom-most layer is the Core library, which provide all the functions and classes for drawing networks and interacting with the user. Resting on the Core library is the plug-ins (or Tools) layer, which adds specialized features depending on the goal of the application. The Core library provides a C application programmer interface (API) allowing C programs to interact with TinkerCell. The C++ plug-ins can extend this Core library's C API by providing additional functions and supporting header files. A developer can work on any one of the layers.

The C++ plug-ins, or Tools, can provide GUI features, such as special widgets, and new C API functions. These widgets and API functions become

the foundation for C programs that perform specific tasks. The diagram below illustrates an example of the layered architecture.



API

The C++ API documentation can be found at: http://tinkercell.sourceforge.net/Core/index.html
Examples of using the C++ API to create extensions can be found here: https://docs.google.com/View?docid=dcx9fkfb_431gjjp72cv

The C/Python/Octave API documentation can be found at: http://tinkercell.sourceforge.net/CAPI/index.html

Examples of Python and Octave code can be found here:
https://docs.google.com/View?docid=dcx9fkfb_234gf8st9hc

Goals

The TinkerCell application is a collection of plug-ins that utilize the TinkerCell Core library. The TinkerCell Core is a general purpose library for constructing networks. The TinkerCell Core by itself is not a usable application. Plug-ins that use the API provided by the Core library add all the GUI features to make a complete application.

It is important to understand that the full TinkerCell application, which includes the plug-ins, has a different goal from the Core library. The set of goals from each is listed below.

Core library

TinkerCell Core is a collection of functions for constructing a network making graphical application. The purpose of TinkerCell Core can be divided into the two areas described below.

1. **A host to C/Python/Octave libraries** : Various algorithms for analysing networks are written in languages such as C. Using them thus requires programming experience. Applications such as the statistical software R or the scripting language Python have made programming simpler by providing a scripting interface to various fast C algorithms. The goal of TinkerCell Core is similar, except using a visual interface in addition to the scripting interface. TinkerCell Core will allow algorithm writers to use it as an interface to their algorithms, thus allowing numerous users without C++ programming skills to still use those functions. Note that a C interface can be extended to other languages such as Python, Ruby, Perl, R, or Octave. Support for Python and Octave is included in the Core library.

2. **Flexible and extensible** : In order to serve as a *host* to various algorithms, it is important for the underlying structure of TinkerCell Core to be highly flexible. Different functions would require different information to be available in order to perform the analyses. Networks constructed using TinkerCell Core should be able to provide all the information required by those functions.

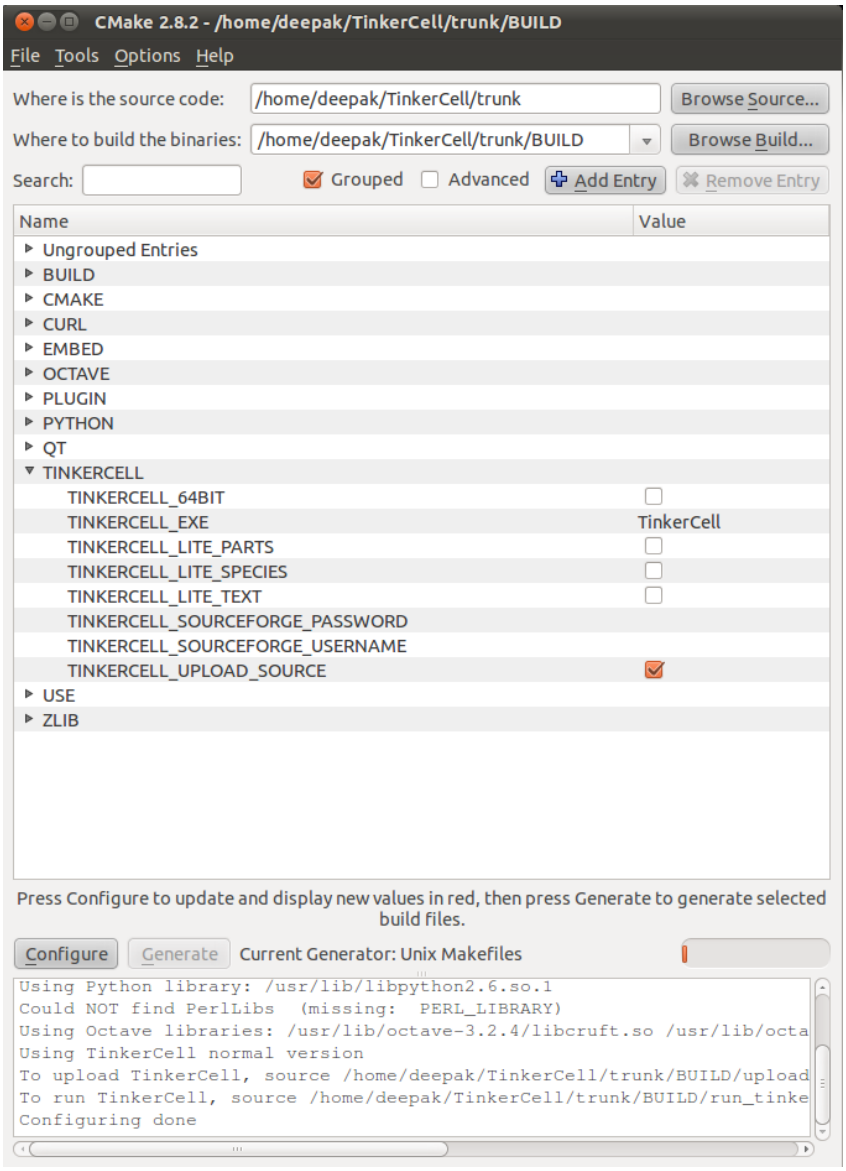
Synthetic biology

The TinkerCell application is a collection of plug-ins and the Core library. The TinkerCell application is intended for constructing cellular networks for the purpose of engineering. A few specific goals are needed to meet this end.

1. **Support for externally defined set of parts and connections** : The user should be able to select biological parts, or nodes of the network, from a catalog of parts and connect them using connections from a similar catalog. These catalogs will represent the various types of biological parts and connections between those parts. For example, parts would include proteins, genes, promoters, and RNA, to name a few. Connections would include metabolic reactions, transcription regulation, and so on. These catalogs should be editable, allowing it to be connected to some external source.
2. **Modules** : Engineering requires modularity, i.e building one network from existing ones. The TinkerCell application should support an interface that would allow a user to convert a network into a "module" as well as connect two or more modules to construct a new network without having the make changes to the original networks.
3. **Genetic networks** : The TinkerCell application should support construction of genetic networks using generic parts, such as promoters, operators, ribosomal binding sites, and protein coding regions. The network should store the DNA sequence for each part as well as parameters for each part, allowing a user to replace one part with another with another in order to change the behavior of the network.

4. **Organized information** : The TinkerCell application should store parameters in the context of the node or connection where it is relevant. This feature will be required for the TinkerCell application to crosstalk with external databases that house information about biological parts and reactions. If a reaction, e.g. a transcriptional regulation, is to be loaded from a database, then the parameters specific to that reaction must be identified and replaced. These parameters may be used in other parts of the model.
5. **Simulation** : A user should be able to simulate networks constructed in the TinkerCell application. It would be ideal if different simulation techniques were available. At the least, simulation using ODEs and stochastic simulation should be available.

Building the TinkerCell project



TinkerCell relies on CMake to compile all the files. CMake is a tool that allows easy compilation of TinkerCell code in a platform-independent manner. CMake can be downloaded at www.cmake.org. It comes with a graphical user interface (CMake GUI). To compile TinkerCell, run CMake GUI and specify the TinkerCell/trunk folder as the source folder and TinkerCell/trunk/BUILD as the binary folder. Press "configure" and then "generate" to generate the platfor
m-specific makefile or project for the entire TinkerCell source code. Once CMake has generated the makefile or project, you will need to compile the code using the specific compiler. For example, if CMake generated a g++ makefile, you can just type "make" in the TinkerCell/trunk/BUILD folder to make the entire project. If CMake generated a project file, you can open it from the TinkerCell/trunk/BUILD folder.

When running the CMake GUI, check the "grouped view" and "advanced view" check boxes. Sometimes CMake is not able to find folders, such as the

Qt folder. In such cases, you will need to manually enter those paths in the GUI window. Usually, CMake just needs a hint about where to look; for instance, if you provide the location of the qmake executable, CMake will automatically find all the Qt libraries using qmake as a starting point.

TinkerCell includes scripts for automatically generating installers and/or uploading the installer to Sourceforge (if you are a TinkerCell developer). Instructions for running these scripts are provided when you configure the code using CMake. For example, in Windows, the CMake windows shows the following messages:

Using Octave libraries: C:/Octave/lib/octave-3.2.3/libcruft.dll.a C:/Octave/lib/octave-3.2.3/liboctave.dll.a C:/Octave/lib/octave-3.2.3/liboctinterp.dll.a

Using TinkerCell normal version

Run C:/Users/Deepak/Projects/Tinkercell/trunk/BUILD/win32/makeWin32Installer.bat to build the program, the installer, and upload it to the sourceforge site

If you want to enable automatic upload to Sourceforge, you must create two New Entries of type STRING, TINKERCELL_SOURCEFORGE_USERNAME and TINKERCELL_SOURCEFORGE_PASSWORD, and populate then with your Sourceforge username and password

In the above list of messages, the first message is just indicating the Octave libraries that are being used (just to confirm). The second message indicates the version of TinkerCell that will be build, and the third message tells you that there is a CreateInstaller.bat file that has been created. This file will compile TinkerCell and generate the installer (if you have ISTool) installed. If you provide the Sourceforge username and password, the same file will also upload the installation files to Sourceforge, as indicated by the fourth message.

Similarly, for Mac and Linux, script files will be generated that will automatically package the application and optionally upload the files to

Sourceforge. Look for messages in the CMake window. You may read the README.txt file that comes with the TinkerCell source code for detailed setup instructions.

To generate Windows installer, you will need to install Inno Setup and ISTool because that is what is used to create the Windows installer. The .iss file for Inno in located in the win32 folder.
If you want to change the installation procedures, take a look at the following files: linux/upload.sh, mac/CreateApp.sh, win32/uploadTinkerCell.winscp.in, win32/TINKERCELLSETUP.iss, win32/CreateInstaller.bat.in

Recommended compilers: MinGW for Windows, GCC/G++ for Unix (i.e. Mac and Linux)

Third-Party packages included with TinkerCell source distribution

All of the following third-party *source code* are included with TinkerCell source distribution. All of these package will be automatically compiled. These packages are included in order to avoid annoying dependencies.

1. LAPACK (linear algebra)
2. Sundials CVode numerical integrator
3. MuParser (math parser)
4. libSBML for parsing systems biology models
5. COPASI for simulations and optimizations
6. libStructural for matrix analysis of reaction networks
7. libAntimony for parsing model scripts
8. Nelder-Mead optimization
9. QWT for plotting 2D graphs
10. QWT3D for plotting 3D graphs
11. Systems Biology Workbench Core
12. Raptor RDF parser
13. Expat XML parser

14. Cluster 3.0

Enabling Python, Octave, or Ruby

If you want to enable the embedded Python and Octave, check the EMBED_PYTHON and EMBED_OCTAVE options in the CMake GUI window. If you are lucky, CMake will automatically find the correct libraries and include folder for each. On Linux, this is usually easy. On MS Windows, MinGW compilers would need to be pointed to the .a files and Visual Studio compilers would need the .lib files.

Important note for Windows: the Octave include folder is NOT Octave/include; it is Octave/include/octave-3.2.3 and similarly, the library files are inside Octave/lib/octave-3.2.3. Since Octave only comes with MinGW libraries, you will need to use MinGW compiler if you want to embed Octave (unless you want to compile Octave using Visual C++).

For Octave in Mac and MS Windows, an additional step may be required: open the config.h find in the Octave include folder and comment the #define HAVE_HDF5 and HAVE_REGEX lines (unless you have these packages installed, which is not included with MinGW). On a Mac, I have not resolve embedded Octave yet. Of course, Linux does not require any additional complication.

The SimpleDesigner example program contains the code necessary for embedding Octave (or Python) -- it is about 3 lines of code.

Overview of the Core library

The TinkerCell Core library is a set of C++ classes that utilize Nokia's Qt Toolkit. The classes provide functions for drawing networks as well as storing information associated with each node and connection in the network.

Being built using Qt Toolkit, the Core library makes extensive use of Qt's Signal/Slot framework. When signals are emitted, e.g. mousePressed(...), the signals are received by one or more slots. Slots are functions that respond to the signals. In the Core library, the MainWindow class acts like a "signal hub". Numerous Tools classes (aka "plug-ins") implement the slots for processing the MainWindow's signals. The Core library does not do anything by itself, except display the main window. Tools, or plug-ins, perform all the work. The set of plug-ins in the "BasicTools" folder perform numerous tasks such as inserting, highlighting selected items, renaming an item when the text is changed, etc. Other folders such as "ModelingTools" consist of plug-ins that are used to generate dynamic models of biological system. These plug-ins are *not* part of TinkerCellCore, but they are very important for the TinkerCell application.

C Programming Interface and Multi-threading

Although a low level language, C has many advantages, one being that many useful algorithms are available in C. For this reason, TinkerCell Core library has built-in support for running functions inside C libraries. There are three main features that the Core library provides:

1. Run C library functions as separate threads
2. Allow C programs running on separate threads to interact with the TinkerCell using the C API
3. The C API can be *extended* by plug-ins. Plug-ins can add new functions that can be called from C programs.
4. Python and Octave can be embedded within TinkerCell using SWIG (www.swig.org)

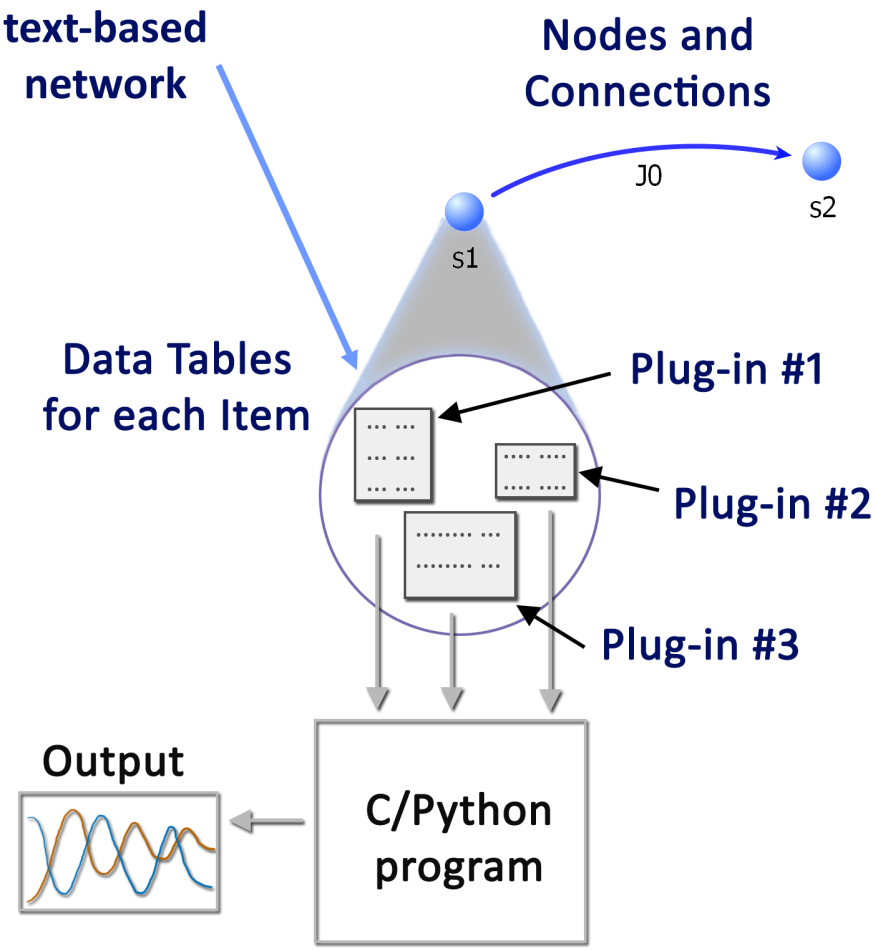
The framework is setup so that C functions are converted to Qt Signals, which are then processed by the plug-ins. This allows the C programs to run on a separate thread, because Qt's Signal/Slot framework can work across different threads.

The thread class that is used to run functions inside C libraries is called CThread. This class provides several features, such as a dialog with progress bar that can be used to display the progress of the running C program.

There is a standards "protocol" that is used for extending the API. This protocol and the CThread class are discussed at the end of the Core Library section.

Object-oriented network design

The Core library uses an object-oriented framework to describe properties of a network. A network is defined using nodes and edges, or connection. Each connection can connect multiple nodes. Each node and connection contains two important fields: *family* and *data*. Family, defined in class ItemFamily, describes the family of a node or connection. Data is a collection of 2D tables of doubles or strings. Each table has a title, e.g. "parameters". These titles are used to identify the information stored within these tables. Information about the network is thus stored within the nodes and connections of the network. For example, parameters describing a particular node's behavior will be stored within that node; this framework allows one to replace nodes and connections along with the information associated with those nodes and connections. It is also possible to store *global* information by using the *modelItem* member in the SymbolsTable class, which is discussed later.

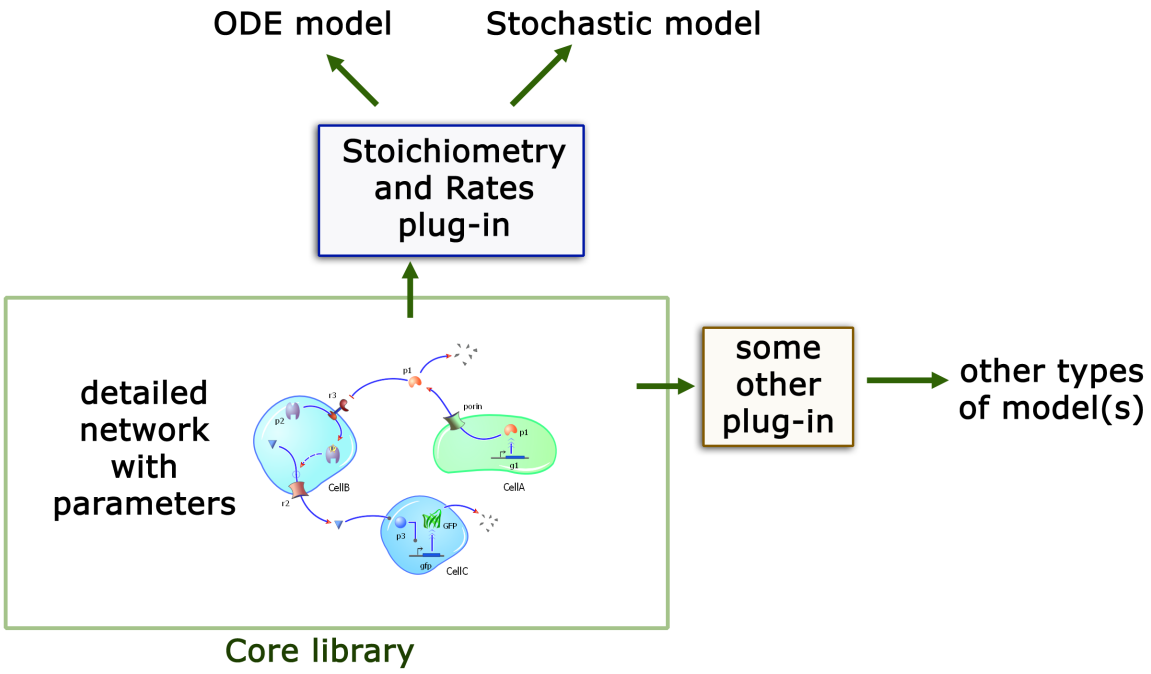


The data tables defined inside each node and connection is generally decided by the plug-ins. Different plug-ins may add different information to nodes and connections based on the family of that node or connection. The plug-ins can then provide C functions for editing those data tables. For examples, an application using the TinkerCell Core library may require each node to contain a list of parameters. For this purpose, a plug-in would be used to add a data table called "parameters" to each node as they are inserted into the network. The plug-in would ideally provide slots connected to C function

pointers such as getParameter or setParameter which can be used by a C program to alter parameters in the network.

Flexible modeling framework

A TinkerCell network without the support of plug-ins is not a complete computational model. It is a generic network diagram with information about the network. Different plug-ins determine how to generate the computational model (e.g. differential equations, stochastic model, etc.). So, new modeling methods can be implemented side by side with existing methods. The objective was to not have a bias toward one method. The following diagram illustrates the idea.



Settings

The class called GlobalSettings contains several parameters for enabling and disabling features in TinkerCell. See the documentation for all of these settings.

Core library classes

The **MainWindow** class provides the top-level window. It is also a "hub" for numerous signals. Any programmer writing a plug-in must be familiar with all of these signals in order to utilize the Core library well. The MainWindow holds multiple **NetworkHandle** class instances. The NetworkHandle class is basically what defines a "network". The NetworkHandle stores a collection of **ItemHandle** instances. The ItemHandle class represents individual nodes (**NodeHandle**) or connections (**ConnectionHandle**). It is important to understand that each network can be displayed in multiple windows and each node or connection can be displayed using multiple graphical items on the screen. The NetworkWindow class is a single window that represents either the entire network or just part of a network. A NetworkHandle contains one or more NetworkWindow instances. Each NetworkWindow hold either a GraphicsView or a TextEditor, but never both. Therefore, a "network" (i.e. NetworkHandle) can displayed to the user using one or more graphical diagrams (GraphicsView) or text (TextEditor).

To understand the design of the Core library, it is imperative to understand ItemHandle. To build well-behaved plug-ins, it is imperative to understand how the Core library uses Undo Commands and Signals. It is also important to review the functions available in the MainWindow, GraphicsScene, and NetworkHandle classes.

DataTable

This is a template class that stores a 2 dimensional table, including the row and column headers. The contents of the table can belong to any type. Typically, TinkerCell only uses double and QString types because those are the two allowed data types in the ItemHandle class. The DataTable class is composed of three vectors: the data, the column headers, and the row headers. The class provides functions for obtaining the data values using header names or index values, removing or adding rows and columns, swapping rows and columns, and resizing the table.

Note: the headers are stored in hash tables, which allows fast access to data using header names and indices.

When an ItemHandle contains a data table, then all of the row names of the data table are treated like local properties of the item, but the columns are not. This becomes very important when renaming components in a model. If x is renamed to y, then any data table column with name "x" will be renamed to "y", but a row with name "x" will remain unchanged. In order to rename the row named "x", the rename function must be given the full row name, i.e. "p.x" (where "p" is the item that contains the table with row "x"). This is just a convention that seemed reasonable because columns are often associations and may refer to other objects in the network.

Undo Commands

Numerous classes are defined in the UndoCommands.h file that inherit from QUndoCommand. These classes contain an undo() and a redo() method. These functions undo and redo a single action without any other side effects. All changes made to a network are generally done using one of these QUndoCommand classes. Examples of undo command classes include MoveCommand, InsertGraphicsCommand and RemoveGraphicsCommand, InsertTextCommand and RemoveTextCommand, ChangeDataCommand, and RenameCommand. There are several more, one for each "atomic" operation. CompositeCommand can be used to construct a more complex command from atomic commands. For example, the "paste" operation is a composite command made from InsertCommand, MoveCommand, and

RenameCommand (for renaming newly inserted items). Other plug-ins also use the composite command.

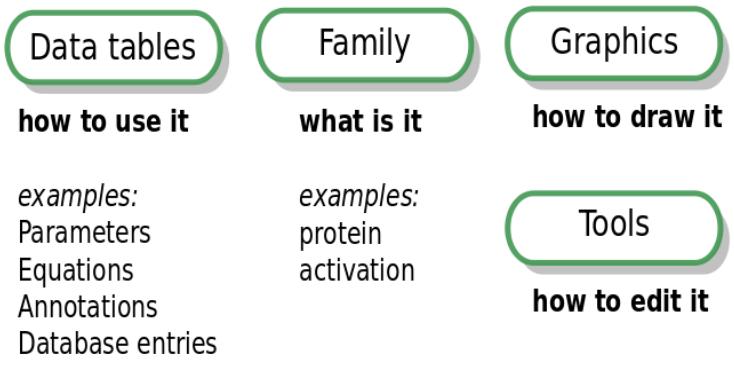
The common procedure for using an undo command is as follows:

```
            QList<QGraphicsItem*> graphicsItems;
            QUndoCommand * cmd = new InsertGraphicsCommand("some
informative message here",graphicsItems,handles);

            if (mainWindow && mainWindow->historyStack())
                mainWindow->historyStack()->push(cmd);
```

Alternatively, the NetworkWindow class and GraphicsScene class provide functions that automatically do the same operations:

```
            QList<QGraphicsItem*> graphicsItems;
            GraphicsScene * scene = currentScene();
            scene->insert("informative message here",
graphicsItems);
```

ItemHandle class



This class is arguable the most integral aspect in the TinkerCell Core library. The ItemHandle can be thought of as a "package" with four important components: the graphics items for drawing a node or connection, the data table associated with that node or connection, the tools associated with the node of connection, and the family that the node or

connection is identified with. The ItemHandle is the complete package that is required to obtain all the information about any item in the network. Since TinkerCell networks can be constructed using text of graphics interface, the ItemHandle is not required to have graphical items. For networks constructed using the text editor, the data inside each ItemHandle is what is most important.

NodeHandle and ConnectionHandle inherit from ItemHandle. For text based models, it is possible to store connections between nodes and connections using ConnectionHandle::addNode() method, which takes a NodeHandle and an integer describing the "role" of that node in the connection. The interpretation of the "role" is open to the plug-in using it. Note that this "role" has no direct connection with the concept of role in ConnectionFamily.

Here is a code example, where two graphics items are placed inside a handle, and a new table is added to the handle:

```
            NodeHandle * nodeHandle = new NodeHandle;

            //make a node item from an XML file
            NodeGraphicsItem * node = new NodeGraphicsItem;
            NodeGraphicsReader reader;
            reader.readXML(node,"mynode.xml");

            //make a text graphics item
            TextGraphicsItem * text = new
TextGraphicsItem("hello world");

            //add graphics items to the handle
            nodeHandle->graphicsItems << node << text;

            nodeHandle->textData("magic word") = "please";
            nodeHandle->numericalData("magic
numbers","pi","value") = 3.141593;
```

```
            nodeHandle->numericalData("magic
numbers","e","value") = 2.718282;

            //get the entire table
            DataTable<qreal> magicNumbers = nodeHandle-
>numericalDataTable("magic numbers");
            //set the entire table
            nodeHandle->data->numericalData["magic numbers"] =
magicNumbers;
```

ItemHandle contains several functions for conveniently retrieving information or the list of child items. Please see the ItemHandle documentation .

Family

The ItemFamily class is used to describe a family that a node or connection belongs in. Nodes and connections are not required to belong in a family. Each family can have multiple parent families. The two main child classes are NodeFamily and ConnectionFamily. NodeFamily stores the default graphics item(s) that is used to draw an item of that family, and ConnectionFamily stores the default arrow head that is used when drawing connections of a given family. The family information is useful for tools in order to distinguish items and insert data tables according to the family of the item.

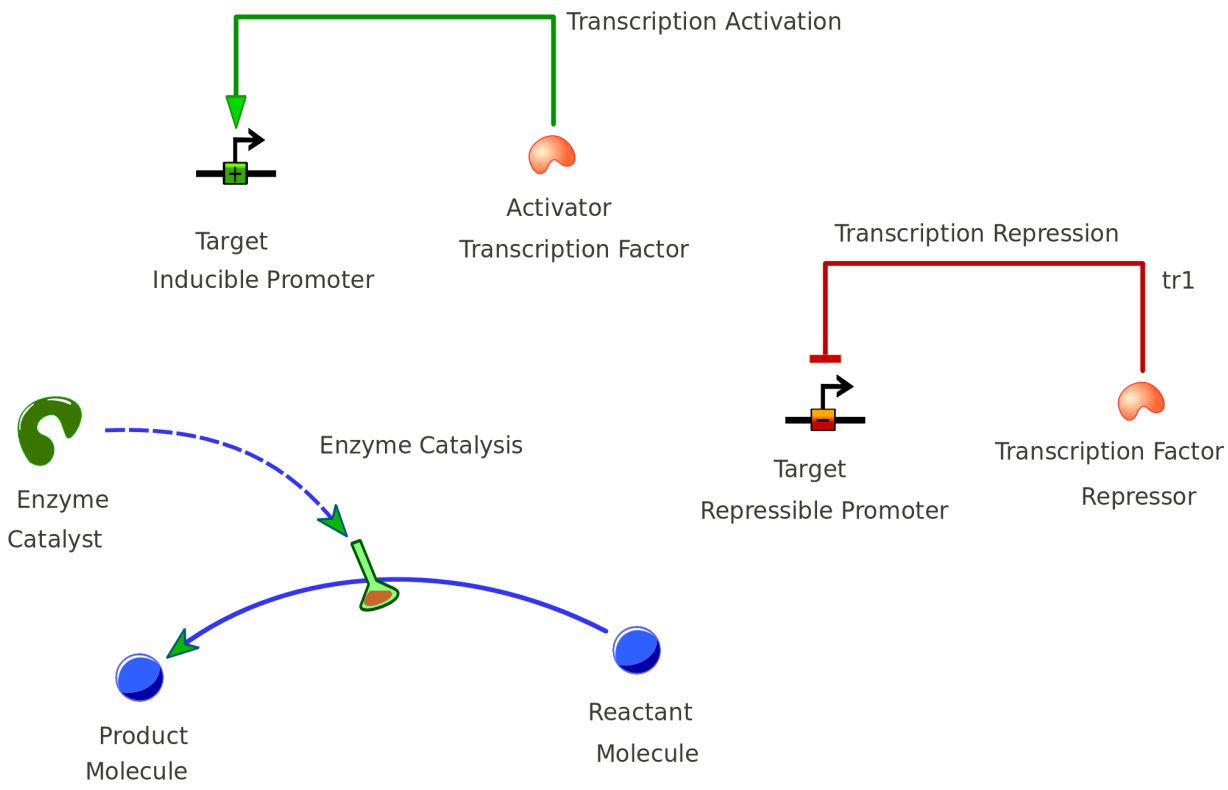
```
            NodeFamily * f1 = new NodeFamily("family A");
            NodeFamily * f2 = new NodeFamily("family B");
            f2->setParent(f1);  //family B is a sub-family of
family A

            NodeHandle * node = new NodeHandle("x",f2);

            if (node->isA("family A")) // will return true
```

```
            //... do something ...
```

Note that the **isA** function does *not* perform string matches. Instead, all families contain an ID (integer), which is created when the family is given its name. Within the ItemFamily class, the IDs are used to identify family types, not strings.

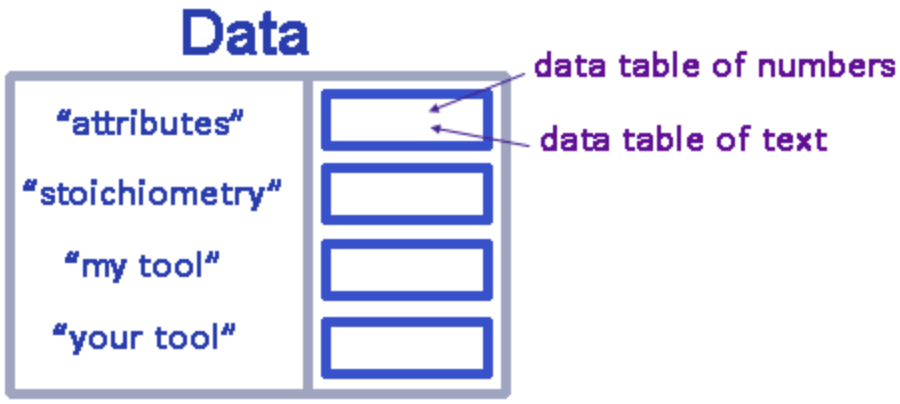
NodeFamily and ConnectionFamily

All items in a TinkerCell network are either nodes, connections, or undefined types. Nodes are identified by the fact that they belong to the NodeFamily (i.e. *NodeFamily::cast(handle->family()) != 0*). Similarly connections belong with ConnectionFamily. ConnectionFamily is also defined using two lists: nodeFamilies and nodeRoles. These lists describe what types of nodes are connected by this family of connections and what role each node takes in this family of connections. A few examples are shown in the diagram below, where the connection name is shown next to the connection and the node role and families are shown next to the nodes.



Methods such as **isValid** and **findValidChildFamilies** can be used to check whether a given set of nodes are compatible with a given connection family. ConnectionFamily contains some advanced features, such as checking how many family types are shared between two connection families, which is useful to determine whether one connection type can be directly attached to another (I hope that made sense).

Data



The "Data" inside an ItemHandle is an instance of class ItemData. This class is just composed of two hash tables, numericalData and textData. Each hash table maps a string to a DataTable. These hash tables store all the information needed to describe a node or connection. For example, numericalData["parameters"] might contain all the parameters belonging to this item. The data tables inside each item are added by tools, which often use the family information to decide what data tables to insert in a given item. For example, connections might contain textData["rates"] to describe the flux equations whereas nodes of a particular family might contains some other information, such as textData["DNA sequence"]. It is important to note that each entry is a 2D table of strings or numbers; of course, they can be a 1x1 table as well.

Tools

Each ItemHandle instance contains a list of pointers to tools, or classes that inherit from class Tool. These tools are associated with this item. When items are selected by a user, the list of contextMenuActions from each of these tools is placed in context menu and the list of graphics items are displayed to the side.

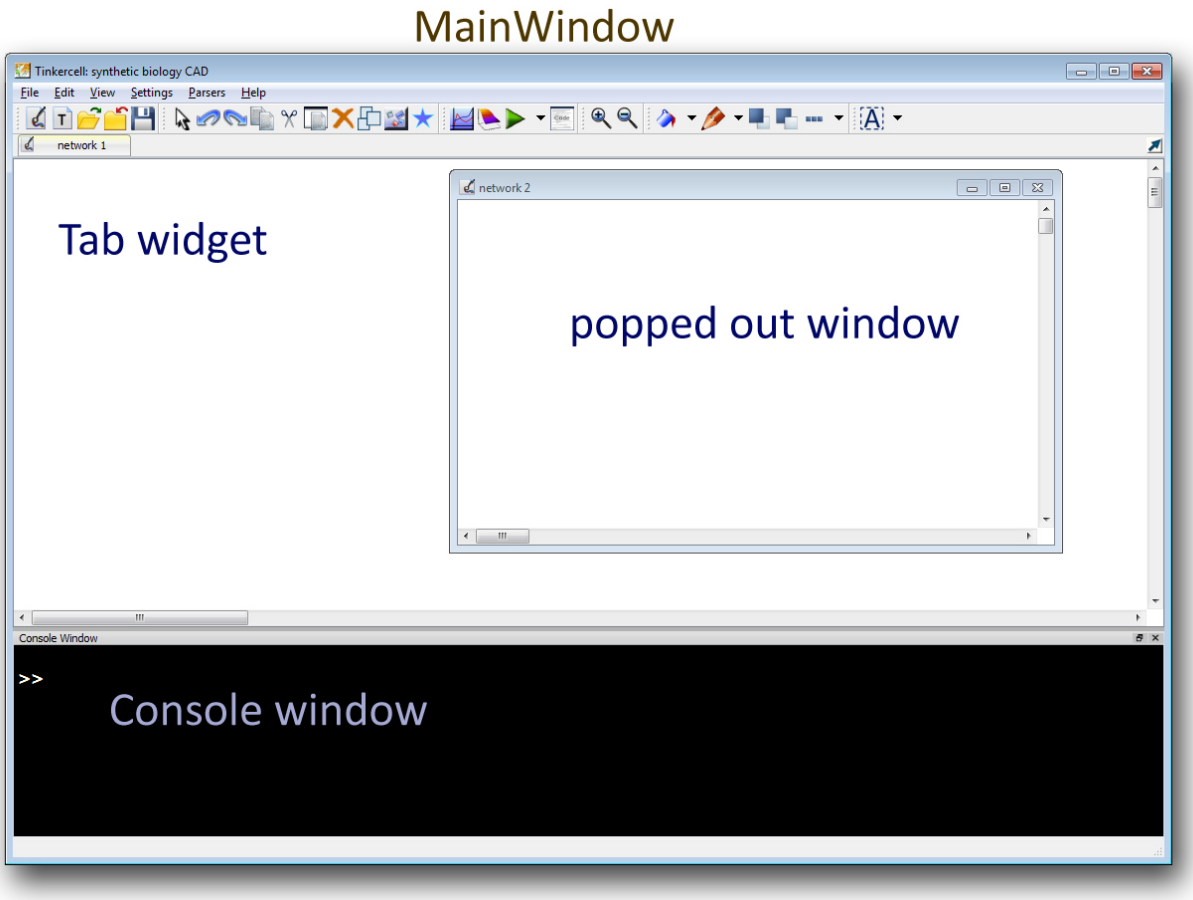
Graphics and Text items

Since TinkerCell Core supports graphical or text interface for constructing networks, each handle can be represented as a QGraphicsItem (Qt's graphics item) or TextItem (a TinkerCell class). Both of these classes are discussed later. There may be tools that simply look at the handle, family,

and data information. For such tools, it is irrelevant whether the network was constructed using a text interface or graphical interface.

MainWindow class

The MainWindow is always the top-most widget that is created in the main() function. The central widget inside the MainWindow is a Tab Widget with windows that can be popped out. Each widget inside the tab widget is a NetworkWindow. Each NetworkWindow can contain a TextEditor or a GraphicsScene. The MainWindow constructor has two arguments for specifying whether the documents should only contain TextEditors or only GraphicsScene or both. Each GraphicsScene is displayed using a GraphicsView.



The MainWindow class inherits from Qt's QMainWindow. The MainWindow has two main functions:

1. Provide the main window for the docking windows, menus, text editors, and drawing canvas
2. Serve as a Signal *hub* that routes the signals from each scene or text editor to the plug-ins listening to those signals. Thus, the plug-ins do not need to connect to every single scene and text editor; they only need to connect to the MainWindow's signals. These connections are made in a plug-in's setMainWindow() method.

The MainWindow also provides several Slots that are connected to C function pointers via the C_API_Slots class. These functions include find, rename, move, remove, and other functions for changing the data tables within an item in the network.

Nearly all the members in the MainWindow class are public. This includes the three toolbars: 1. toolBarBasic, which stores buttons for basic functions such as new, open, and save; 2. toolBarEdits, which stores buttons such as copy and pase; 3. toolBarForTools, which is intended for other tools. Tools may also add new toolbars using the addToolBar method in QMainWindow. The context menu (mouse right button) for TextEditor and GraphicsScene are also defined in MainWindow. The menus named contextItemsMenu and contextScreenMenu are used by GraphicsScene when items are selected and when no item is selected, resp.. The menus named contextSelectionMenu and contextEditorMenu are used by TextEditor when text is highlighted and when no text is highlighted, resp. Menu items such as file menu, edit menu, settings menu, and view menu are also public, allowing tools to add new actions to them.

When items are inserted or removed from a GraphicsScene or TextEditor, each class emits a signal indicating that graphics item(s) have been removed and text item(s) have been removed, resp. These signals are connected to signals in the MainWindow with the same names. In addition, MainWindow

also emits two signals called itemsInserted and itemsRemoved that only contain the ItemHandles instead of the graphics items or text items. Signals that contain only ItemHandles are useful for tools that do not need to know whether the network was constructed using text or graphical interface.

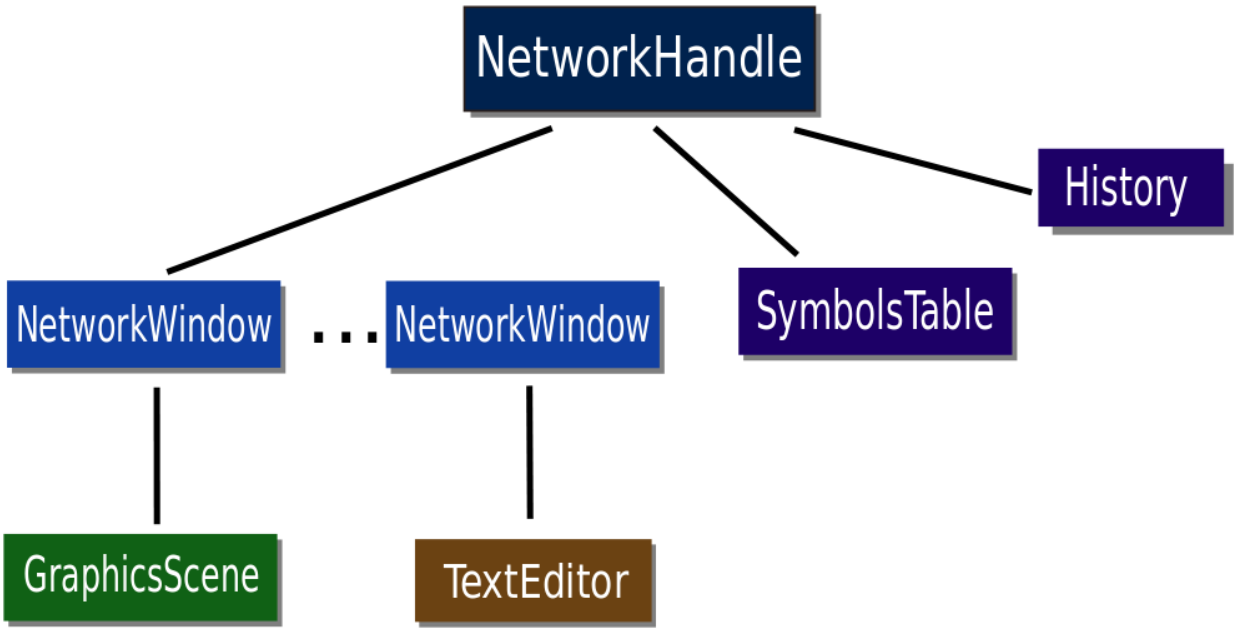
Some important signals

- itemsAboutToBeInserted and itemsAboutToBeRemoved: these signal are emitted just before items are inserted or removed from a scene, respectively. It can be used to automatically add or remove items from the list. The signal contains a list of QUndoCommands; new commands can be added to this list to perform additional actions along with the insertion event.
- itemsInserted and itemsRemoved: these signals are emitted after items are inserted or removed from a scene, respectively. It can used to modify the items that have been inserted based on the placement of the items or other conditions. It is also used to add tools to the handle::tools list of the new items.
- dataChanged: this signal is emitted whenever any handle's data entry is changed. It is also emitted when items are inserted or removed. This signal can be used to check when a model need to be updated. Note that undo events are not captured by this signal, which is only captured by historyChanged signal.
- historyChanged: this signal is emitted whenever any recorded change occurs. This signal can be used to check when a model need to be updated.
- networkOpened, networkClosed, and networkChanged: these signals are emitted whenever a new network is opened, a network has been closed, or a the user has clicked on a different network window (respectively). These signals are usually used to reset contents of widgets that display information about a network.
- networkOpening and networkClosing: these signals are sent before opening or closing networks (respectively). They can be used to check if the network has been saved.

- mousePressed, mouseReleased, mouseDragged, mouseDoubleClicked, sceneRightClicked: These signals are emitted due to mouse events. These signals are emitted even if the useDefaultBehavior switch is off in GraphicsScene.
- keyPressed, keyReleased: These signals are emitted due to keyboard events. These signals are emitted even if the useDefaultBehavior switch is off in GraphicsScene.

NetworkHandle class

The NetworkHandle is used to store all the information inside a network. The three main components of a NetworkHandle are: historyStack, symbolsTable, and networkWindows. The history stack is used to store the QUndoCommands that provide the undo/redo capabilities. The symbolsTable stores all the nodes and connections in the network. The list networkWindows stores all the windows that are used to display the network to the user. The NetworkHandle provides convenience functions such as changeData(...) or rename(...). These functions create a QUndoCommand, add it to the history stack. Each NetworkHandle can be represented using one or more windows. All of these windows are connected to the same symbols table and the same history stack. NetworkHandle also contains functions such as find() for finding any string in the network and parseMath for validating a mathematical expression (uses muparser).



NetworkWindow class

The NetworkWindow is a window (QMainWindow) inside the MainWindow's tab widget. This window can contain either a TextEditor or a GraphicsScene, but not both. Each NetworkWindow can contain its own toolbar or dock widgets. Each NetworkWindow has functions for replacing its current scene or text editor (warning: this operation cannot be undone). Each NetworkWindow can contain an ItemHandle pointer. This handle can be used for multiple purposes. It is designed for particular scenarios in which each individual window is associated with a handle. By default, this pointer is zero.

SymbolsTable class

The SymbolsTable class is used to store all the string found in a network model. These strings include the node and connection names and the row names and column names of all the data contained within each node and connection. The purpose of the symbols table is to easily look-up a symbol

and find the network objects associated with that symbol. The symbols table keeps a hash table of names and pointers to the node or connection with that name.

The SymbolsTable is also used to get all the ItemHandles in a network, except for "hidden" ItemHandles. ItemHandles represent objects in a network, whether the model is represented as text or graphics.

Full names are always unique, e.g. Cell1.p1. Just the first name, e.g p1, need not be unique. The symbols table keeps a one-to-one hash table that maps full names to object pointers and a one-to-many that maps the first names to object pointers. The uniqueData hash table stores prefixed strings, e.g. p1.param1, as well as non-prefixed strings, e.g. param1. For each string, the hash table stores all the objects that contain that string and the name of the data table which contains that string.

Each NetworkHandle contains one SymbolsTable instance. This instance is updated during any change (history update) to the network. NetworkHandle's makeUnique method uses the symbols table to quickly assign unique names.

GraphicsScene class

The GraphicsScene class is used to construct a network visually. It is one of the largest classes in TinkerCell. The GraphicsScene inherits from Qt's QGraphicsScene. The primary duty of the GraphicsScene class is to receive mouse and keyboard events and emit necessary signals such as itemsSelected, itemsMoved, or mouseOverItem.

The GraphicsScene also handles selection of objects on the scene and moving objects on the scene. The selected objects are placed in the selected() list, and the moving objects are placed in the moving() list. These lists can be modified by plug-ins in order to modify which objects are selected or moved. Moving items are always grouped together when moving; this makes the movement smoother. For example, if a node has

other nodes attached to it, a plug-in can ensure that all the nodes move together by adding each node to the moving() list when any one of them is selected. The GraphicsScene's selection and moving operations can be disabled by setting useDefaultBehavior = false**.**

In addition to emitting signals and handling selection and moving, the GraphicsScene houses numerous functions for conveniently making changes to a network. The functions include insert, remove, move, rename, and changeData. Each of these functions do three things: make a QUndoCommand object, push the undo command to the history stack, and emit the necessary signal(s) such as itemsInserted or itemsRemoved.

The GraphicsScene is always contained inside a NetworkWindow. Therefore it uses the parent NetworkWindow's history stack and symbols table. Many functions such as changeData, rename, or allHandles simple call the parent NetworkWindow's function.

Configuring

Various visual features, such as the color of the selection rectangle in a scene and default grid size can be set using global variables: GraphicsScene::SelectionRectangleBrush, GraphicsScene::SelectionRectanglePen, GraphicsScene::BackgroundBrush, GraphicsScene::ForegroundBrush, GraphicsScene::GRID, GraphicsScene::GridPen. GraphicsScene::MIN_DRAG_DISTANCE can be used to set the minimum distance that is considered a valid drag, i.e. moving the mouse less than this distance will be considered an accidental movement of the mouse and ignored.

GraphicsView class

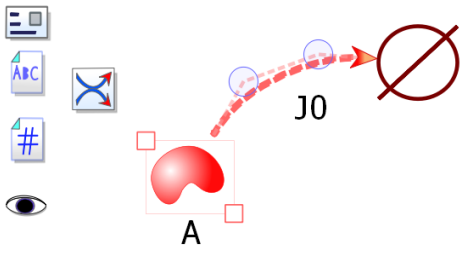
The GraphicsView is a class for viewing a GraphicsScene. It inherits from QGraphicsView, and provides a few extra features such as drag-and-drop and zooming.

Graphics item classes

Qt's QGraphicsItem class is used to draw all the items in the GraphicsScene. The two main graphics item classes are NodeGraphicsItem and ConnectionGraphicsItem. Supporting graphics items are TextGraphicsItem and ControlPoint.

The qgraphicsitem_cast<class> function can used to cast a generic QGraphicsItem to one of these four classes. In addition, NodeGraphicsItem::cast and ConnectionGraphicsItem::cast can also be used to get the top-most node or connection item from a generic QGraphicsItem instance. Each NodeGraphicsItem and ConnectionGraphicsItem also contains a string named ClassType, which is used to statically cast sub-classes of Node or Connection. For example, ArrowHeadItem is a NodeGraphicsItem with classType = "Arrow Head Item". example usage: if (node->className == ArrowHeadItem::CLASSNAME) static_cast<ArrowHeadItem*>(node)

ControlPoint



The ControlPoint class is used to identify key locations of a NodeGraphicsItem or ConnectionGraphicsItem that can be used to change the appearance of that item. For example, NodeGraphicsItem uses control points around its bounding box, allowing a user to drag the control points in order to resize the item. ConnectionGraphicsItem uses control points to define the line or beziers used to draw the connection. See

image to the right: the small squares and circles are control points. Control points are generally not child items of the item that they belong with. The two main sub-classes of ControlPoint are NodeGraphicsItem::ControlPoint and ConnectionGraphicsItem::ControlPoint.

NodeGraphicsItem

This class is used to draw nodes on the GraphicsScene. NodeGraphicsItem inherits from QGraphicsItemGroup, which is used to group several graphics items together. Each NodeGraphicsItem is a set of points and a set of shapes that are defined using those points. The points belong to the ControlPoint class and the shapes belong to the Shape class. The entire NodeGraphicsItem can be saved as an XML file using NodeGraphicsItemWriter (and NodeGraphicsItemReader for reading the XML). The XML file uses the SBML render extension format, which is similar to SVG.

The NodeGraphicsItem has convenient functions such as connections(). The set of connections connected to a given node is retrieved by looking at the control points that are child items of that node. Each connection must have a control point that is the child item of the node that is it connected to.

Shape

This class is a polygon constructed using lines, beziers, or arcs. The Shape class inherits from QGraphicsPolygonItem. The polygon must be closed. The refresh() method is used whenever the shape's control points are changed. This updates the shape's polygon.

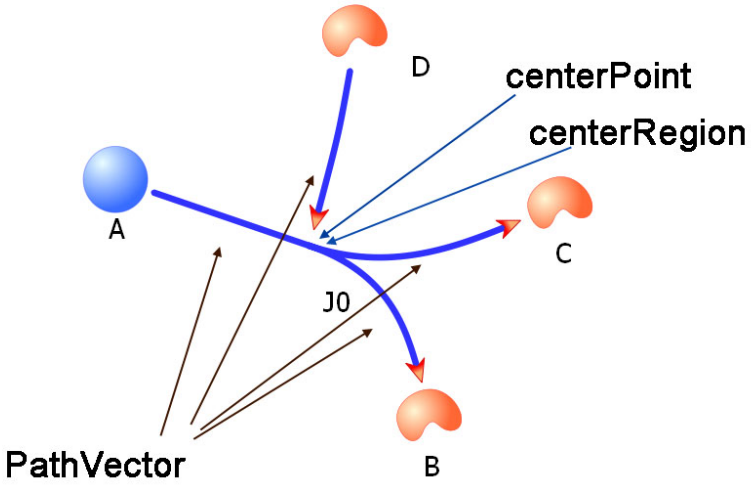
The groupID variable is used to identify the items that belong in the same scene. This is important for models that span multiple scenes.

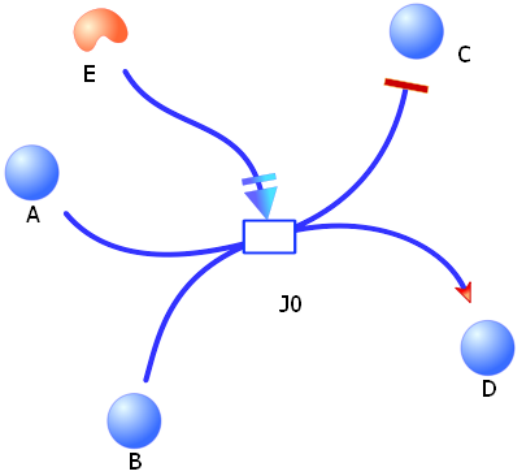
ConnectionGraphicsItem

This class is used to draw connections between nodes. ConnectionGraphicsItem is composed of a list of CurveSegment instances.

Each CurveSegment is a collection of control points that define a single path, usually with the same central control point. Each curve segment also has two arrow head items -- one at either ends (they can be null). If there is a node at the end of any of the paths, then the control points at the end will be child items (see QGraphicsItem) of that node; so, looking at the parent items of each of the control points at the ends is the correct way to find all the nodes that are connected by a connection.

The ConnectionGraphicsItem also contains an optional centerRegionItem, which is a node that sits at the center of the connection. This node is used when one connection item needs to connect to another connection item. Since connections can only be connected to nodes, the center region item is used when connecting a connection to another.





The control points that constitute a connection are generally parent-free, except for the end control points. As mentioned earlier, if a control point is at the end of a connection and is connected to a node, then the control point will be set as the child of the node item. This allows the control point to move along with the node. The ConnectionGraphicsItem class retrieves all the nodes that it is connected to by looking at the parent items of each of its end control points. ConnectionGraphicsItem provides convenient functions such as nodes(), nodesWithArrows(), nodesWithoutArrows(), where "WithArrows" means that there is an arrow head at the arc leading to the node. It is important to

understand that these functions do not imply that the curve segments represent a reaction or some other specific process. They indicate the visual representation, which is then translated to more specific meanings by the plug-ins.

refresh() is used whenever the connection is changed. This function updates the arcs and the shape() of the connection using the control point positions.

The groupID variable is used to identify the items that belong in the same scene. This is important for models that span multiple scenes.

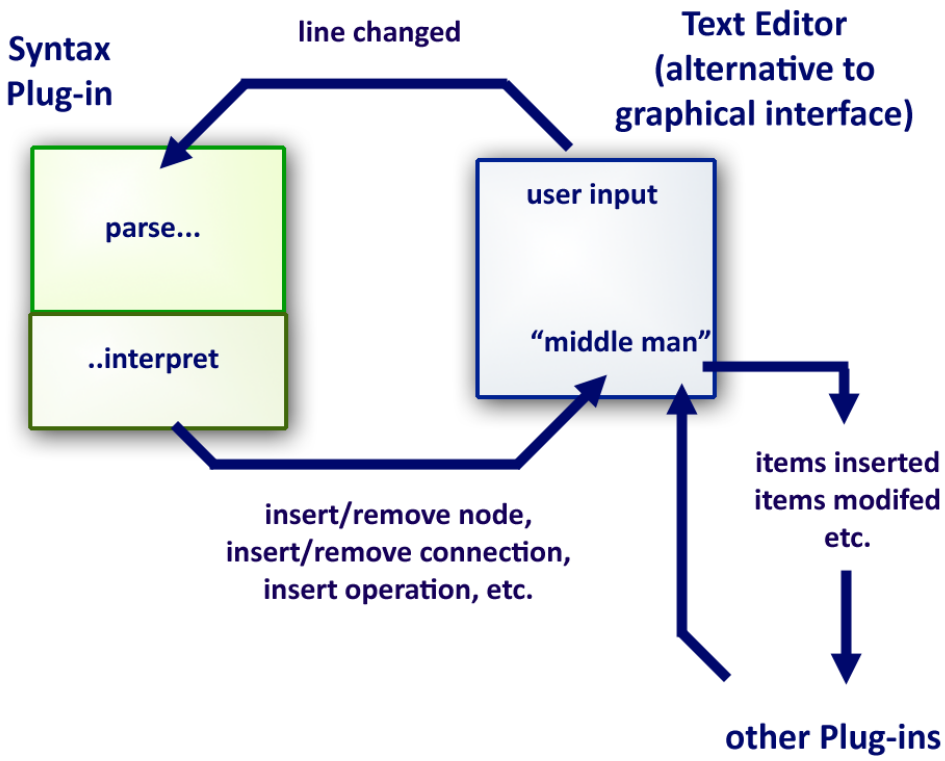
The ConnectionGraphicsReader and Writer can be used to read and write a connection item to an XML file.

The default arrow head can be set using ConnectionGraphicsItem::DefaultArrowHeadFile. Similarly, the default middle item (the box at the center) can be set using ConnectionGraphicsItem::DefaultMiddleItemFile. For example:

```
ConnectionGraphicsItem::DefaultArrowHeadFile = appDir +
QString("/ArrowItems/Reaction.xml");
ConnectionGraphicsItem::DefaultMiddleItemFile = appDir +
QString("/OtherItems/simplecircle.xml");
```

TextEditor class

The TextEditor class is used to construct a network using a text-based language. The syntax is not defined by TextEditor and must be provided by a supporting plug-in. The supporting plug-in is expected to make use of the lineChanged(...) and textChanged(...) signals emitted by TextEditor to identify changes by a user and call the insertItem(...), removeItem(...), or setItem(...) methods in order to modify the network.



Tool

Tool is the parent class for all TinkerCell "plug-ins". The most important method in the Tool class is setMainWindow(), which is used by a new tool to connect with the MainWindow's signals and slots.

Each Tool can choose to create instances of Tool::GraphicsItem and place them on the scene. When these graphics items are selected by the user, TinkerCell Core will call the select(int) method of the Tool that is associated with the graphics item.

Ontology

The Ontology class is an optional class that can be used to read RDF files. The Ontology class will create and maintain a set of node and connection family instances. These instances can be accessed by different plugins in TinkerCell. The Ontology class is used by the LoadSaveTool for mapping from family names to family instances (if families are used in the application)

Console Window

The ConsoleWindow class provides a generic framework for Tools to receive command-line input as well as display messages or execute commands. Each tool can access the ConsoleWindow using console() or mainWindow->console(). For example:

```
 if (console())
    console()->message("hello world");    //print a message on
the console window
 if (console())
    console()->error("incorrect response");  //print an error
message on the console window
 if (console())
    console()->eval("print 1+2");  //evaluate this expression
(only runs if a plugin such as python plugin is available)
 DataTable<double> data;
 //fill in data
 if (console())
    console()->printTable(data); //print a table (tab-delimited)
```

Tools can also interact with the user by connecting to the ConsoleWindow's **commandExecuted signal**. This signal is emitted whenever the user pressed <return> after entering a text at the command prompt. The Tools can process the string and carry out necessary operations.

```
ConsoleWindow * console = console();
if (console)
{
    connect(editor, SIGNAL( commandExecuted(const QString&) ),
            this, SLOT( commandExecuted(const QString&) ));
}
```

Tools may also disable and re-enable the ConsoleWindow while they are processing the command by using:

```
 console()->freeze();    //lock the console window
 console()->unfreeze();  //unlock the console window
```

Alternatively, Tools may also connect with the freeze() and unfreeze() slots:

```
CommandTextEdit * editor = console()->editor();
if (editor)
{
    connect(this, SIGNAL(freeze()), editor,SLOT(freeze()));
    connect(this, SIGNAL(unfreeze()), editor,SLOT(unfreeze()));
    connect(this, SIGNAL(setFreeze(bool)),
editor,SLOT(setFreeze(bool)));
    connect(editor, SIGNAL( commandExecuted(const QString&) ),
            this, SLOT( commandExecuted(const QString&) ));
}
```

Plotting tools

The core library comes with classes for drawing 2D line graphs, histograms, and 3D surface plots. The graphs can also be viewed as tab-delimited text file within the plotting window. The PlotTool class is a multi-document interface (MDI) that can hold multiple PlotWidget windows and toolbars.

Each PlotWidget is responsible for drawing a particular type of graph, e.g. surface plot, histograms, text output.

The plotting tools can be enabled by:

```
MainWindow mainWin;
mainWin.addTool(new PlotTool);
mainWin.addTool(new GnuplotTool); //gnuplot
```

Using the plotting tool from another tool:

```
NumericalDataTable data1, data2;
//fill in data1, data2
if (mainWindow->tool("plot"))
{
     QWidget * widget = mainWindow->tool("plot");
     PlotTool * plotTool =
static_cast<PlotTool*>(widget);
     plotTool->plot(data, "title 1");

     //second plot with a different x-axis and plot
type
     plotTool->hold();
     plotTool->plot(data2, "title 2", 1,
PlotTool::ScatterPlot);
}
```

The Plot2DWidget has lots of features. One of the interesting features is the ability to append new plots on top of existing plots. The PlotTool performs the append whenever the append action is checked in the options menu.

The PlotTool can also cluster multiple plots and display each cluster as a separate sub-plot.

Basic graphics toolbox

The core library comes with a simple class that provides buttons for coloring, zooming, aligning, and a few other such features. This toolbox can be enabled by:

```
MainWindow mainWin;
mainWin.addTool(new BasicGraphicsToolbar);
```

CThread

This class is used to run C plug-ins as separate threads. The CThread class loads a shared (dynamic) library and calls specific functions. By default, the MainWindow class tells CThread to look for "tc_main" function. But C plugins or C++ plugins can load specific shared libraries and ask CThread to load specific functions.

InterpreterThread

This class inherits from CThread. It is used to run interpreters such as Python and Octave interpreter.

PythonInterpreterThread

This class inherits from InterpreterThread. It is used to embed Python interpreter. This class uses the C program python/runpy.c.in
Python can be enabled in the Console using the following code:

```
ConsoleWindow * console = mainWindow.console();
OctaveInterpreterThread * octaveInterpreter = new
PythonInterpreterThread("_tinkercell",&mainWindow);
```

```
octaveInterpreter->initialize();
console->setInterpreter(octaveInterpreter);
```

In the above code, _tinkercell is the prefix for the TinkerCell C API wrapped for Python.

OctaveInterpreterThread

This class inherits from CThreads. It is used to embed Octave interpreter. This class uses the C++ program octave/runOctave.cpp (for embedding Octave) and assumes that SWIG has been used to generate tinkercell.oct library (which extends Octave).
Octave can be enabled in the Console using the following code:

```
ConsoleWindow * console = mainWindow.console();
OctaveInterpreterThread * octaveInterpreter = new
OctaveInterpreterThread("tinkercell.oct","libtcoct",&mainWindow);

octaveInterpreter->initialize();
console->setInterpreter(octaveInterpreter);
```

In the above code, tinkercell.oct is the TinkerCell C API wrapped for Octave, and libtcoct is the prefix for the library that is used to embed Octave (code in API/runoctave.cpp).

Loading and Saving

The Core library provides the LoadSaveTool that enables saving and loading. The MainWindow::OPEN_FILE_EXTENSIONS and MainWindow::SAVE_FILE_EXTENSIONS store the list of acceptable file extensions. If they are empty, then LoadSaveTool sets the file extension to "TIC". To enable LoadSaveTool, just do mainWindow->addTool(new LoadSaveTool).

If the program makes use of families, then LoadSaveTool needs to know how to map family names to family instances. For this purpose, it uses two static maps:

```
QMap<QString,NodeFamily*> nodeFamilies;
QMap<QString,ConnectionFamily*> connectionFamilies;
```

These two QMaps need to be populated, otherwise all loaded items will have NULL families (which is not a problem if the program does not use the family information).

Plug-in Framework

While TinkerCell supports C or C++ plug-ins, C++ plug-ins have far greater control. C plug-ins are intended for providing new algorithms, not interfaces. Octave and Python plug-ins work practically the same way as C plug-ins (they use the same functions).

The C++ plug-ins make use of the Signals provided by the Core library classes. Plug-ins can add new user interface and/or new information to the network. All C++ plug-ins are classes that inherit from the Tool class. There is a standard procedure for adding new plug-ins to the TinkerCell core:

1. Write a class that inherits from the **Tool** class (inside the namespace Tinkercell).

```
#include "Tool.h"
class TINKERCELLEXPORT MyTool : public Tinkercell::Tool
{
    public:  void setMainWindow(Tinkercell::MainWindow *);
};
```

2. In most cases, the setMainWindow function provided in Tool will need to be overridden. In this method, you would connect the signals from MainWindow to the slots on your class. See the main window documentation

to see the set of signals available. You may also add your tool as a dock widget using addDockingWindow.

```
void MyTool::setMainWindow(Tinkercell::MainWindow * main)
{
     //connect signals and slots
}
```

3. The next step depends on how the plug-in will be loaded. The plug-in can either be loaded directly in the main program, or it can be loaded from the dynamic library.

3(a). Loading the plug-in directly in the main program would usually look like this:

```
int main()
{
    QApplication app(argc, argv);
    MainWindow mainWindow;
    MyTool * tool = new MyTool;
    mainWindow.addTool(tool);
    mainWindow.show();
    return app.exec();
}
```

3(b). Loading the plug-in as a dynamic library requires the plug-in classes to be built into a dynamic library with dependency on the libTinkerCellCore library. In order to load the plug-in library, use the following method in MainWindow:

```
mainWindow->loadDynamicLibrary("myplugin");  //do NOT use .dll, .so, .dylib suffix
```

When a library is dropped on the TinkerCell window, the above function is automatically called on the file (if it is a library file).

In the above method call, the "myplugin" string is the name of the dynamic library file **without the suffix** (.dll, .so, .dylib) , where mainWindow in an instance of MainWindow. This method will look for a particular function called **loadTCTool** in the plug-in's dynamic library. So, the plug-in **must define** a function that would usually look like this:

```
	extern "C" TINKERCELLEXPORT void
loadTCTool(Tinkercell::MainWindow * main)
	{
		if (!main) return;
		Tinkercell::MyTool * myTool = new
Tinkercell::MyTool;
		main->addTool(MyTool);
	}
```

The function shown above is called by MainWindow when the library is loaded. Generally, loading the library means adding the tool class into MainWindow using the addTool method, as shown above. However, one may also do other things when a library is loaded inside the loadTCTool function. The tool will be deleted by MainWindow, so no memory management is needed (and do NOT statically allocate Tools, since MainWindow will try to delete them).

When a Tool is being loaded, other tools can prevent it from loading. MainWindow emits the toolAboutToBeLoaded(Tool,bool) signal, the second argument of which is a bool that can be set to false to prevent the tool from being loaded.

A typical plug-in would use the itemsInserted signal and insert new data into new handles.

Simple Examples

Please refer to the TinkerCell Extensions document for examples: https://docs.google.com/View?docid=dcx9fkfb_431gjjp72cv

The C Interface

TinkerCell Core library has built-in support for running functions inside C or C++ libraries on separate threads. Since TinkerCell Core is not a complete program but a library for constructing an application, it is expected that a complete application would want an API that is specific for that application. In order to achieve this, the C interface is structured so that the plug-ins can add new functions to the API. There is a standard protocol that must be used by plug-ins for *extending* the C API.

Protocol for extending the C API

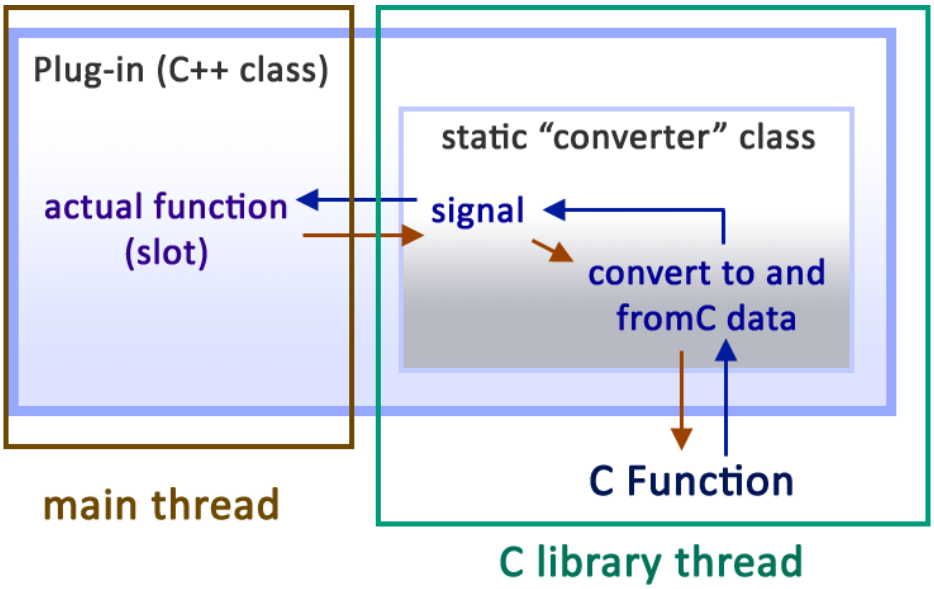
The C API is composed of a set of header files. Each header file generally belongs with one plug-in (or sometimes a group of plug-ins). All of these header files are then included in the TC_api.h header file. TC_api.h is the header file that will be used by C programs for interacting with TinkerCell.

The header files define a set of function pointers and one function for initializing those pointers. For example:

```
//function pointers
 void (*tc_plot)(Matrix data,const char* title) = 0;
 Matrix (*tc_getPlotData)() = 0;

 //initialize function pointer
 void tc_PlotTool_api(
              void (*arg1)(Matrix,const char*)
              Matrix (*arg2)())    )
 {
         tc_plot = arg1;
         tc_getPlotData = arg2;
```

```
}
```

It is expected that the above functions, i.e. plot and getPlotData, are two features that are provided by some plug-in. The second function, or the initializing function, is called by the plug-in that provides the features. After initializing, the function pointers will point to static methods inside the plug-in class. These static methods call an identical method inside a static class. The methods inside this static class emit signals which are received by the plug-in classes themselves. In summary, the function pointers in the C header files are converted to Qt signals. Conversion to the signal/slot framework is important in order to run the C programs on separate threads.



The code below shows the plug-in class that provides the tc_plot and tc_getPlotData functions defined in the header files. Note the arguments in each function. **tc_matrix** is a simple C struct and its C++ counterpart is **DatatTable<qreal>**. Similarly, QString is the C++ counterpart of the C char*.

ConvertValue(...) is used to convert between C data structures and C++ data structures. In the code below, the PlotTool_FToS is a class that is used to convert C functions to Qt signals, which is why the signals in this class have the C++ data structures and the methods have the corresponding C

data structures. "setupFunctionPointers" is a signal that is emitted by MainWindow. It allows plug-ins to initialize function pointers inside a newly loaded library. A summary is provided at the end.

```
namespace Tinkercell
{
    //an intermediate class
    class PlotTool_FToS : public QObject
    {
            Q_OBJECT
        public:
            void plot(tc_matrix a0,int a1,const char*);
            Matrix getPlotData();
       signals:
            void plot(QSemaphore*, DataTable<qreal>&,const
QString&);
            void getPlotData(QSemaphore*, DataTable<qreal>*);
    };

    class PlotTool : public Tool
    {
           Q_OBJECT
           static PlotTool_FToS converter;
       public:
           PlotTool();
           bool setMainWindow(MainWindow *);
       private:
           static void _plot(tc_matrix arg1, const char* arg2);
           static Matrix _getPlotData();
       private slots:
           void setupFunctionPointers( QLibrary * library );
           void plot(QSemaphore*, DataTable<qreal>& arg1,const
QString& arg2);
```

```
            void getPlotData(QSemaphore*, DataTable<qreal>*
returnValue);
    };

    PlotTool_FToS PlotTool::converter;  //STATIC

    //THE C FUNCTIONS
    void PlotTool::_plot(tc_matrix arg1,const char* arg2)
    {
        converter.plot(arg1,arg2);   //just pass it on to
converter
    }

    Matrix PlotTool::_getPlotData()
    {
        return converter.getPlotData();  //just pass it to
converter and return
    }

    bool PlotTool::setMainWindow(MainWindow * TinkercellWindow)
    {
          Tool::setMainWindow(TinkercellWindow);
          if (mainWindow)
          {
                connect(mainWindow,
                        SIGNAL(setupFunctionPointers( QLibrary *
)),
                        this,
                        SLOT(setupFunctionPointers( QLibrary *
)));
          }
          connect(&converter,
```

```
                SIGNAL(plot(QSemaphore *,
DataTable<qreal>&,const QString&)),
                this,
                SLOT(plot(QSemaphore *, DataTable<qreal>&,const
QString&)));
          connect(&converter,
                SIGNAL(getPlotData(QSemaphore
*,DataTable<qreal>*)),
                this,
                SLOT(getPlotData(QSemaphore
*,DataTable<qreal>*)));

          return true;
    }

    void PlotTool::plot(QSemaphore * s, DataTable<qreal>&
matrix,const QString& title)
    {
          //so something here
          if (s)
              s->release();  //VERY IMPORTANT
    }

    void PlotTool::getPlotData(QSemaphore* s, DataTable<qreal>*
matrix)
    {
        DataTable<qreal> data;
        //fill in data
        (*matrix) = data;  //RETURN VALUE
        if (s)
             s->release();  //VERY IMPORTANT
    }
```

```
    typedef void (*tc_PlotTool_api)( void
(*plot)(tc_matrix,const char*) , Matrix (*plotData)() );
    void PlotTool::setupFunctionPointers( QLibrary * library)
    {
        tc_PlotTool_api init = (tc_PlotTool_api)library-
>resolve("tc_PlotTool_api");
        if (init)
        {
            init( &(_plot), &(_getPlotData) );   //INITIALIZE C
POINTERS
        }
    }

    void PlotTool_FToS::plot(tc_matrix a0,const char* title)
    {
        DataTable<qreal>* dat = ConvertValue(a0);  //convert
Matrix to DataTable
        QSemaphore * s = new QSemaphore(1);
        s->acquire();
        emit plot(s,*dat,ConvertValue(title));
        s->acquire();  //wait until done...
        s->release();  //finished.
        delete s;
        delete dat;  //free memory
    }

    Matrix PlotTool_FToS::getPlotData()
    {
        QSemaphore * s = new QSemaphore(1);
        DataTable<qreal> * p = new DataTable<qreal>;
        s->acquire();
        emit plotData(s,p);  //NOTICE: p is the RETURN VALUE
        s->acquire();
```

```
        s->release();
        delete s;
        tc_matrix m = emptyMatrix();
        if (p)
        {
            m = ConvertValue(*p);  //convert back to C data
structure
            delete p;
        }
        return m;
    }

} //end of namespace Tinkercell
```

In summary, this is the check-list for extending the C API. Each step (not in order) is required for correctly providing new C functions.

On the C side:

1. make a header file with list of function pointers (initialize to 0)
2. make an initialization function for the pointers
3. include this header file in TC_api.h

On the C++ side:

1. make a Tinkercell::Tool class
2. make a QObject class
3. In the QObject class:
    1. define a set of methods with the same definition as the function pointers in the C header file
    2. define a set of signals with the definition f(QSemaphore*, return_type, arg1, arg2...), where return_type is a reference or pointer to the return type for the C functions. Arg1 and arg2 are the arguments, whose types are the C++ counterparts of

the C argument types, i.e. DataTable instead of Matrix and QString instead of char*.

4. In the Tool class:
    1. define a set of **static** methods with the same definition as the function pointers in the C header file
    2. define a set of slots with the same definition as the signals in the QObject class
    3. define setupFunctionPointers method. In this method:
        1. find the initialization function in the C library using library->resolve
        2. call the initialization function with the static methods as the arguments
    4. connect the signals in the QObject class to the slots in the Tool class
    5. connect the setupFunctionPointers signal in MainWindow to the setupFunctionPointers in the Tool class

The static methods in the Tool class will just call the methods in the the QObject class and return the value returned by them.

The methods on the QObject class will have the following structure:

```
return_type function( type1 a, type2 b )
{
        QSemaphore * s = new QSemaphore(1);
        s->acquire();
        Type1 A = ConvertValue(a);
        Type1 B = ConvertValue(b);
        ReturnType C;
        emit functionSignal(s,C,a,b);
        s->acquire();
        s->release();
        return ConvertValue(C);
}
```

TinkerCell: Computer-Aided Design Tool for Synthetic Biology

The TinkerCell application is a design tool for synthetic biology. The TinkerCell application is a collection of plug-ins that build on the TinkerCell Core API. The most important plug-ins are the NodesTree and ConnectionsTree classes. These classes read two XML files that list the node and connection families and their attributes. For example, the XML file may define the family called "Protein" and list "Molecular Weight" as an attribute. The file also specifies the units used to quantify the item in a model. For the family "Protein", this field would include "Concentration(uM)". The NodeInsertion and ConnectionInsertion tools listen to signals from the NodesTree and ConnectionsTree tools and allow the user to insert items from these lists onto the graphics scene, using the GraphicsScene's insert() method. The insert() method emits itemsInserted() signal, which is received my numerous other tools. As a response to items inserted, the other tools add information such as parameters and rate expressions.

For example the BasicInformationTool looks at the attributes listed in the NodesTree and ConnectionsTree and adds those attributes into the objects using

```
handle->numericalDataTable("Parameters") = ...
handle->textDataTable("Text Attributes") = ...

or

handle->numericalData("Parameters",i,j) = ...
handle->textData("Text Attributes",i,j) = ...
```

Similarly, the StoichiometryTool checks if inserted items are connections using ConnectionHandle::asConnection and then inserts the "Rates" table and "Stoichiometry" table into the handles, again using

```
handle->numericalDataTable("Reactant Stoichiometries") = ...
handle->textDataTable("Rate equations") = ...
```

Tools involved in kinetic information

A handful of tools, or plug-ins, are involved in generating most of the kinetics information needed for a kinetic model, i.e. a stoichiometry matrix and reaction rate equations. The **Stoichiometry and Rates** tool generates rate equations for each reaction and a corresponding stoichiometry table. When generating the rate equations, this tool uses **Parameters** tool to check for undefined parameters. The **Numerical Attributes** tool stores parameters associated with each node or connection in a model. The **Functions and Assignments** tool stores function definitions associated with each node or connection. This tool also accesses the Numerical Attributes to check for undefined parameters.

The **Model File Generator** tool uses the above tools to generate the C and Python code for the rates function and stoichiometry matrix that define the system. This tool is used by the simulators to generate the rates function and stoichiometry table.

Plugin categories

The most important plug-in is the Nodes and Connections Catalog, which provides the catalog of parts and connections for constructing models. More importantly, it reads two XML files and loads all the ItemFamily instances that can exist in a model, which is used by most of the other plug-ins to determine the type of a model item. Next comes the plug-ins in Basic Tools folder, which are responsible for highlighting selected items, inserting nodes and connections from the Catalog, and moving child items along with their

parents. Insert Connections plug-in is a bit tricky, because it tries to determine the type of connection a user *meant* from the types of the components the user has selected. The Modeling Tools are a set of plug-ins responsible for generating stoichiometry, reaction rate equations, parameters, forcing functions, and events. The Module Tools provide the interface for connecting modules.

Biological Modularity in TinkerCell

A module is a "process", and what is a process? A process is described in its role players and the *types* (i.e. families) or each role player. For example, a reaction of *family* "enzyme catalysis" will have *catalyst, substrate,* and *product* as the role player. The families for the role players will be *protein, molecule,* and *molecule* (respectively). We can extend this simple definition to more complex modules. Lets take an incoherent feed-forward genetic network module. It would have *input* and *output* as the role players, and the types of the role players would be *transcription factor* and *protein*. So what is the difference between a Connection and a Module? Answer: nothing! A connection represent a process, and a module is essentially a process that has been abstractly represented.

Collaboration via TinkerCell

tinkercellextra.sf.net is a repository for storing models and script-based plugins (i.e. Python and Octave). When the TinkerCell application starts, all plug-ins available in this repository are automatically downloaded to the Documents/TinkerCell folder. This automatic download requires Subversion to be installed. Contributors who wish to add plug-ins to TinkerCell just need to submit updates to this central repository; all TinkerCell users will automatically get the updates. The repository is organized into four main folders: Modules, python, octave, plugins. The plugins folder contains

additional folders for Windows, Mac, and Ubuntu C/C++ plugins. The updates.txt is used to store recent updates and TinkerCell current version.

Contributors should ask for access to this repository so that they can submit new code and/or models.