An Enhanced GUI for the Tinker tool

by Pierre Le Bras

with contributions from Gudmund Grov & Yuhui Lin

September 2014

Heriot Watt University
Edinburgh Campus,
Riccarton, Currie
EH14 4AS
Abstract

Tinker is a tool that implements the proof-strategy graph (PSGraph) formalism. Here, tactics are represented as nodes and are linked with each other with edges. Each tactic will consume goals on input edges and generate new sub-goals which are sent to the output edges. The tool is generic w.r.t. theorem provers, and currently supports Isabelle and ProofPower. Users will use the normal interface provided by the theorem proving system, with the additional support for creating and evaluating proof strategy using Tinker’s GUI. Tinker is built on top of Quantomatic.

The first version of the GUI was very limited and highly integrated with the Quantomatic GUI. This document presents a new version of the GUI with several new features; including support for hierarchies and mode guided evaluation. The new version is decoupled further from Quantomatic. It is the outcome of the employment of the first author on EPSRC platform grant EP/J001058/1 over the summer of 2014.
# Contents

1 Introduction 3

2 Definitions 4

3 High level overview 5

4 Components 7

4.1 Views 7

4.1.1 Main frame 7

4.1.2 Menu bar 7

4.1.3 Graph edit panel 8

4.1.4 Sub-graph preview panel 12

4.1.5 Element edit panel 12

4.1.6 Library panel 12

4.2 Controllers 13

4.2.1 Main 13

4.2.2 Menu 13

4.2.3 Graph edits 13

4.2.4 Sub-graph preview 14

4.2.5 Element edition 15

4.2.6 Library 15

4.3 Services 15

4.3.1 Main service 16

4.3.2 Document service 17

4.3.3 Communication service 18

4.3.4 Dialog service 18

4.3.5 Events 19

4.4 Models 19

4.4.1 PSGraph model 19

4.4.2 Hierarchy model 22

4.5 Utils 22

4.5.1 Popup menu 22

4.5.2 Toolbar 22

4.5.3 Arguments parser 23

4.5.4 Mouse state 23

5 Quantomatic integration 24

6 Conclusion & future work 25

A Screenshots 26
1. Introduction

The first implementation of the GUI was developed on top of a top of the Quantomatic GUI, which includes features that are not relevant for PSGraphs. We made the new GUI separately from the Quantomatic GUI — albeit still using key features of it as a library. E.g. by calling some the classes to create the visual aspect of graphs and allow the interaction with their components. As in Quantomatic, we used the Scala language to create the GUI, and more especially the Swing package. As Scala uses the Java Virtual Machine to run its applications, some Java classes have easily been integrated.

The work has been accentuated on graph editor, by allowing new functionalities (merging nodes, dragging edges, editing subgraphs, ...) and disabling previous ones (adding boundaries manually, draing edges between boundaries, ...). Also efforts were put in the creation of a new representation of PSGraph for communication with the core of the system\(^1\).

One of the main addition in this version of the Tinker GUI was to handle graph hierarchies in teh GUI. A nested node in the graph would symbolize a nested graph and then reveal the presence of a sub-graph, or a choice between sub-graphs. There is two motivations behind this system:

- "Factorizing" a sequence of tactics into one (when there is one graph).
- Implementing a key functionality of PSGraphs : creating alternatives path in the proof evaluation. Basically the user can decide to give multiple ways of evaluating a graph at one point in the proof, and also set up an order or not in those ways by switching it to an "OR" or an "ORELSE" nested tactic.

The short term of the development process did not allow us to pursue work on the evaluation part. The idea would have been to build a robust socket connection between the GUI and the core of Tinker. Ideas were mentioned, and the development of this communication is future work.

---

\(^1\) All the magic, such as evaluation and theorem prover integration, happens in the core, which is written in Poly/ML. However, this report only focuses on the GUI so we will not give further details of the core.
2. Definitions

Through the document, we will use terms that are specific to the PSGraph theory or even just the GUI itself, here is a list of them with a brief explanation of what they really mean:

- Nested tactic: an evaluation tactic that contains one or more graphs to evaluate before being completed. There are two ways to evaluate a nested tactic; evaluate all the sub-graphs ("OR"), or only evaluate one sub-graph if the previous graph failed ("ORELSE").

- Atomic tactic: a tactic that only represent an atomic evaluation of a goal. Note that the model of atomic tactics contains a "tactic" field which does not correspond to their name, but to a complete description of their purpose.

- Graph: a set of nodes, representing tactics, goals or breakpoints linked by edges. There are two representations of it: one graphical and one in the Json format.

- Nested graph: a graph wrapped inside a nested tactic.

- Current graph: the graph the user is currently working on or editing, the one viewed on the central part of the window.

- Sub-graph: a directed nested graph of the current graph.
3. High level overview

The GUI is made of four types of components, spread in three packages:

- The View package, containing the actual views of the GUI.
- The Controller package, containing the controllers and the services.
- The Model package, containing the model of a psgraph.

As you can see the GUI implements the MVC pattern.

The following diagram (figure 3.1) describes the structure made by the different files.

![Figure 3.1: High level architecture of new GUI](image)

The views’ files are actually the different visual components of the window. Each of them has an associated controller in the controller package. A Service ensure the communication between the controllers or other services and the model. There are four other services:

- A Document Service, used to write and read files on the user’s device as well as keeping track of modifications on the current graph.
- A Communication Service, managing the sockets of communication with the core of the application.
- A Dialog Service, used to trigger dialogs on the view, such as error messages, confirmations or customs inputs for edits.
• A Quantomatic API, which uses classes from the Quantomatic project to create and interact with visual graphs, and generate a Json representation of it.

The controller package also contains a file with the signatures of all events used in the application.

There are two models in the model package. The main one is the PSGraph model, containing all the informations needed to build a Json object later used by the Tinker core. The second one represent the hierarchy between the graphs, using a tree architecture. We could have reused the PSGraph model for that, but an exclusive model for this purpose allows time savings.

There is also a Util package that provides utilities such as:

• A popup menu, which is not native in the scala swing package.

• A Toolbar, not native as well.

• A arguments parser, a custom parser for the tactics inputs, i.e. :

  \[
  \text{tactic(arg1: subarg1 subarg2, arg2)}
  \]

• A mouse state, adapted from the quantomatic project, used for knowing what the user is doing when drawing a graph, for example draging an item or drawing one.
4. Components

In this chapter you will find a list and description, for each package, of all the classes and objects implemented for the application.

4.1 Views

The complete view is actually split into eleven files, one implementing the top container. This last file calls sub-containers, which eventually call sub-sub-containers and so on. A sub-container can be located in the same file, depending on how deep we are in the hierarchy.

4.1.1 Main frame

The main frame is implemented in the *MainGUI.scala* file, which contains one object of the same name. This object inherits from the *SimpleSwingApplication* abstract class of the swing package which initialized our application. In this class, one method, called *top*, sets up and return the top-level frame of the application.

In order to initialize our top-level frame, we give it many arguments such as:

- A title (updated according to the document status, section 4.2.1).
- A menu bar (section 4.1.2).
- A content.

In our case the content in a sequence of *SplitPane*. A *SplitPane* is a division (horizontally or vertically) of a container, where we can specify the two sub-components, which can be a *SplitPane* as well, like in our case. At this point, we specified four sub-containers:

- A library panel (section 4.1.6)
- A graph edition panel (section 4.1.3)
- A sub-graph edition panel (section 4.1.4)
- An element edition panel (section 4.1.5)

The following picture (figure 4.1) represents the layout of our main frame:

4.1.2 Menu bar

The current menu is a standard application menu, it is implemented in the *TinkerMenu.scala* file. It offers a list of actions to manipulate files like:

- Creating a new graph.

---

1 See: *SimpleSwingApplication* in swing API.
2 See: *SplitPane* in swing API.
4.1.3 Graph edit panel

The graph edit panel, specified in the *GraphEditPanel.scala*, is where the user builds the current graph he/she is working on. The way it works is that the application gets a panel from quantomatic (section 5). This panel displays the current graph as well as listens to the user interactions (mostly through the mouse) and acts accordingly by launching events. Different actions will be triggered...
according to the mouse state (section 4.5.4). They are mostly launched by the controller (section 4.2.3).

A left click will only react when the user is on the selection mode (it will eventually select the element where user clicked, or start a selection box if no element has been hit) or the edge addition mode (it will start adding an edge). There are two ways to start the addition of an edge: either an element is present at the start and we simply draw an edge from it, or there is nothing and we create a boundary node and draw the edge from it. Note that the user cannot draw an edge from an existing boundary node, boundary nodes can only have one incoming or outgoing edge.

Dragging the mouse will react if the user is dragging a node or nodes (previously selected with a left click or the selection box), drawing an edge (previously started with a left click) or drawing a selection box (previously started with a left click). Note that drawing an edge can happen when the user is creating one or simply moving an existing one.

Releasing the mouse will react if the user is dragging a node or nodes (it will move it or them to the final position of the mouse), drawing an edge (it will either link the edge with an existing node on the final position of the mouse or create a boundary if necessary), adding a node (it will create one according to the selected type on the final position of the mouse) or drawing a selection box.

A right click will trigger a popup menu that has different options if the user hit a specific type of node, an edge or nothing. When nothing is hit, the menu only offers to add a specific node. Otherwise, it offers the same options described in section 4.1.5.

There are five types of nodes that a user can draw:

- An identity node, referring to the identity tactic.
- An atomic node, referring to an atomic tactic.
- A nested node, referring to a nested tactic.
- A boundary node, only created when the user draw an edge from or to no specific node on a graph. Note that in practice, a break node split the edge into two edges and is placed as an intermediate between them. It is impossible then to add incoming or outcoming edges on the breakpoint.
- A break node, can only be created when selecting an existing edge and asking to place a break on it. An edge can have one and only one breakpoint on itself.

There also exists goal node, but they can only be created when evaluating.

If many nodes are selected (by a selection box or a Shift+left click) the user has the possibility to merge them into a single nested node. Unmerging is still a functionality to be added.

On the top of the actual graph edition space, we put up few buttons and informations displayed in sub-containers, using the BorderPanel class of swing. The figure 4.2 shows the layout of the whole graph edit panel:

### Hierarchy tree

Graph hierarchy are supported. In order to clearly navigate the hierarchies, several additional features have been added. One of them is a hierarchy tree. On the top-left corner this icon opens a new frame (i.e. a new window). This frame is implemented in the HierarchyTree.scala file. It contains a single panel that uses the hierarchy tree controller (section 4.2.3) to draw a tree representation of the nested tactics in our model. Each tactic is displayed as a rectangle that has one and only one parent and zero or more children. The top level graph or root is labelled "main" (it is labelled so

---

3See : BorderPanel in swing API.
Figure 4.2: Graph edition layout

throughout the application). The current graph is displayed with a larger blue border and name. All the tactics are clickable, clicking one will set it as the current tactic, the current graph being the first one.

Hierarchy breadcrumbs

Another way to display hierarchy between graphs/tactics is through breadcrumbs. At anytime, the user can see the path in the hierarchy to arrive at the graph he/she is currently working on, starting from the top level one, "main". The file GraphBreadcrumbs.scala displays a list of labels, separated with the symbol ▶, from "main" to the current tactic via all the tactics in between. Every label, a part from the last one i.e. the current graph, is clickable, clicking one will set it as the current graph. The list of name to display as labels is obtained by the controller (section 4.2.3).

Hierarchy navigation

This panel is meant to give the user the possibility to navigate through same level graphs. Basically a nested tactic can contain one or more subgraphs, so when the user is editing a tactic, he/she has to be able to edit all the different graphs this tactic offers. Note that the root tactic, "main", can have one and only one graph related, therefore the navigation panel is hidden when the user edit this graph. This panel is implemented in the GraphNavigation.scala file. It offers four actions:

- Switch to the previous graph, stops at the first one.
- Switch to the next graph, stops at the last one.
- Add a new subgraph, disabled if the last graph is empty.
- Delete the current graph; there are multiple cases for this actions: by default we delete the graph and switch to the previous one, if we are at the first graph, we delete it a switch to the next one (new first one) and there were only one graph in the tactic we empty it (delete and then add new).
These actions are launched with clickable and intuitive icons (two arrows, a green plus and a red cross) and are mostly computed in the controller (section 4.2.3). The panel also include a index on total counter.

**Edit controls tool bar**

This tool bar (in file `EditControlsPanel.scala`) is a set of toggle buttons. It gives the user the ability to change the edition mode via mouse states (see 4.5.4). There are five modes:

- Selection, ▼
- Drawing identity nodes, △
- Drawing atomic nodes, □
- Drawing nested nodes, □
- Drawing edges, ▼

The change of state is made through the controller (section 4.2.3).

The same file also set up a second tool bar which has only one button with this icon ▴. It opens a new frame containing a single text area where the user can edit the goal types of the PSGraph. This frame does not have a proper controller, and interacts directly with the main service.

**Evaluation controls tool bar**

This tool bar is similar to the previous one. It is defined in the file `EvalControlsPanels.scala` and has a set of button that give the user options when evaluating a proof:

- Undo, ‹, to cancel the previous evaluation step.
- Step into, △, to view evaluation steps in a nested tactic.
- Step over, □, to evaluate a nested tactic without viewing the details.
- Complete, △, to complete the evaluation of the current tactic.
- Finish, △, to complete the whole evaluation.
- Stop, ▴, to stop evaluating.
- Debug, ▴, to give debug options and details to the user.

At this moment only the buttons exist, they are not linked to any kind of action, and no controller has been created for this tool bar.
4.1.4 Sub-graph preview panel

This panel is located on the top right corner of the window and is implemented in the `SubGraphEditorPanel.scala` file. Its controller is described at section 4.2.4. It has one purpose: when the user selects a nested tactic, it displays its graphs giving the user a preview of the tactic.

To obtain the preview, we use Quantomatic once again, but without allowing any edition in the previewed graph. We also use a BorderPanel class to layout this part of the view, with one main part for the preview and a top bar similar to the hierarchy navigation (see 4.1.3) with one additional icon: a pen to picture the action of editing this sub-graph, it will switch the currently edited graph with the one the user was previewing.

When no nested tactic is selected, this panel is empty.

4.1.5 Element edit panel

This panel is located below the sub-graph preview panel. It is implemented in the `ElementEditorPanel.scala` file and its controller is described in section 4.2.5. It has contents only when one or more elements of the graph are selected (node or edge). It allows the user to edit specific values of the selected element(s) as well as giving access to actions on those element. It also prints information about the selected elements such as there id on the graph, or the type of node (if it is a node).

The same actions are available via a popup menu (right click on graph elements):

- Single identity node: delete node.
- Single atomic node: edit name and arguments, edit tactic, delete node.
- Single nested node: edit name and arguments, switch between "OR" or "ORELSE" tactic, add a sub-graph, delete node.
- Single break node: remove breakpoint.
- Single edge: edit goal types, add/remove breakpoint (depends if there is already one), change source and/or target (only existing node that are not boundaries), delete edge.
- Multiple nodes: merge nodes into a nested tactic, delete nodes.

Since those actions are available via the popup menu, this panel may be deleted and the space in the window reused for another purpose in further developments, like printing results from evaluation.

4.1.6 Library panel

On the left side of the window, a panel has been set up to allow the use of a library of graphs. This panel is divided horizontally by a SplitPane and both parts are implemented in one file (`TinkerLibraryTree`) and have a common controller (section 4.2.6).

The upper panel simply displays a tree structure of a library directory. When the user double click one file (the tree only recognize .psgraph extensions) the graph it contains is previewed in the lower panel. The library preview work the same way the sub-graph preview does. On the top of the preview, a button allow the user to add this graph (and all its potential sub-graph) in the one he/she is currently working on.

When added in the PSGraph structure, the library graph imports its model as well, and renames its tactics names (atomic and nested) into "libraryFileName-previousName". On the graph edition panel, the library graph elements are by default selected.
4.2 Controllers

Foreach view described earlier there is a controller associated. A view as access to its controller via the main service as we instanciate a controller in this object in order to ease the inter-controller communication. Controllers were design to provide their view specific computations, but at this moment some only forward action to services. Although we may want to refactor this state a delete unnecessary controllers, we can also think of keeping all controllers and provide a place for specific computations for future development. The communication from a controller to its view is made exclusively via events, since only views knows their controller. A future refactor may change this state, making controllers "aware" of their view.

4.2.1 Main

File: MainGUIController.scala. Corresponds to the main view (section 4.1.1).

Provides the current document title to its view, computed in the document service, and listen to the same service for any change to eventually update the title in the view.

4.2.2 Menu

File: MenuController.scala. Corresponds to the menu view (section 4.1.2).

Provides all the functions related to the actions describe in the view section. It also listens to the document service for changes in the document status to enable or disable some action like "save" or "undo".

4.2.3 Graph edits

File: GraphEditController.scala. Corresponds to the graph edition view (section 4.1.3).

This controller is the one keeping track of the mouse state (section 4.5.4) for the graph edits. It implements two functions with the same name to change it, one with only a string as parameter (corresponds to state) and another one which allows a second parameter that can be either a string (refers to a node id) or a Point \(^4\) (refers to a position on the graph). These functions, called changeMouseState, are mostly meant to be send as parameters of other functions allowing a change of the mouse from another class or object.

It is also responsible for getting the graph view (a Panel object) from Quantomatic, and provides a function returning it, getGraph.

The last thing this controller does is to listen to the Quantomatic API and react to the mouse events it publishes. Those reactions actually calls functions from the Quantomatic API giving them parameters computed from the mouse state. It is in this reactions that the popup is launched, popup implemented as a variable of this class.

Hierarchy tree

File: HierarchyTreeController.scala. Corresponds to the hierarchy tree view (section 4.1.3).

This controller does all the computation needed to draw the hierarchy tree of our tactics. It works like this:

- The function drawTree, called by the view, takes a Graphics2D \(^5\) object as parameter, and calls the next function with the following parameters: the Graphics2D object, the root tactic (that it got from the main service), a default x coordinate, a default y coordinate, and the

---

\(^4\)See: Point in java API.

\(^5\)See: Graphics2D in java API.
depth number 1 (the depth symbolize the depth in the tree, the root is 1, its children 2, their children 3, ...).

• The second function, `drawElement` takes five parameters: a `Graphics2D` object, a `TreeElement` (section 4.4.2), a x coordinate, a y coordinate and a depth number. The idea is to recursively draw the tree with this function. We do it by iterating through the children of a tactic, foreach of them we compute its coordinates, draw a line from the parent to the child and call `drawElement` on the child. `drawElement` actually return the width taken by an element and its children, so we can draw the next child far enough to avoid overlaps of nodes. Once we have drawn all the children of a tactic, we can draw its actual representation, i.e. a rectangular node by calling the next function.

• Finally we call `drawNode`, a function that takes a `Graphics2D` object, a string and coordinates as parameters and returns the width of the node. It draws a rectangle representing the tactic’s node according its name (in order to fit the string inside the rectangle). An slightly different function, `drawActiveNode`, draws the rectangle with a blue border and string, it represent the tactic the user is currently viewing.

Throughout the process we keep track of the tactics’ positions in a map. Therefore a function `hit` can be called when the user click on the view, if it hits a node we can call all necessary functions to modify the current graph. Also a function `redraw` launch a event to notify the view to repaint the tree.

Hierarchy breadcrumbs

File: `GraphBreadcrumbsController.scala`. Corresponds to the hierarchy breadcrumbs view (section 4.1.3).

This controller provides three functions, the first `addCrum` takes a string parameter (name of the tactic to add in the breadcrumbs) and publish an event for its view in order to make it add this string in its list of breadcrumbs. The second, `changeGraph`, is called when the user clicks on a parent breadcrumb, and make the main service change the current graph. And the last one, `rebuildParent`, sends to the view a list of string via an event to rebuild the breadcrumbs sequence from scratch.

Hierarchy navigation

File: `GraphNavigationController.scala`. Corresponds to the hierarchy navigation view (section 4.1.3).

It provides the functions for navigating through the graphs of the same tactic as well as adding or deleting them. There is also a function called `viewedGraphChanged` made to update the navigation buttons (disable the add button if needed) and informations (index of graph on total) according to the current graph.

Edit controls

File: `EditControlsController.scala`. Corresponds to the graph edit toolbar (section 4.1.3).

This controller listens to the toggle buttons of the toolbar and reacts accordingly to which one is selected, changing the mouse state.

4.2.4 Sub-graph preview

File: `SubGraphEditController.scala`. Corresponds to the main view (section 4.1.4).

This controller retrieves the JSON representation of a subgraph from the model (through the main service) and make Quantomatic generate a view (panel) of it to inject in the sub-graph panel. It
also provides functions for navigation between sub-graphs (similar to the hierarchy navigation) and a function edit to set the sub-graph as the current graph. Finally it makes the view display an empty panel if no nested tactic/node is selected on the current graph.

4.2.5 Element edition

File: ElementEditController.scala. Corresponds to the main view (section 4.1.5).

This controller main function is to give listeners to the edition fields of the view. There is one for the name on the nodes (which the name of a tactic and its argument), it listens to any key input and if the values are different from the previous ones (avoiding extra process for non actual updates), the argument parser (section 4.5.3) separate them and construct the list of arguments for the model. Note that the argument parser returns a proper string format for the arguments and update the field with it. This is due to other parts of the application using string equality to update fields in the Json representation, and the compared strings are constructed from the list of argument into a specific format. So to prevent errors we make avoid "bad" formated inputs.

Another input listener exists for the tactic field of an atomic node. It listens to the key inputs and update if any changes are made. The controller also has a function to retrieve this information from the model (through the service), since it is not an occurring information on the graph.

The last input listeners focus on the goal types of an edge and it source and target. Source and target listenes tak e the nodes' ids as values to update, they are only actually updated if the nodes actually exist and are valids (not breakpoint or boundaries).

Otherwise many functions are available: getter and setter for the "OR"/"ORELSE" value of a nested tactic, deletion of a node or an edge, addition/removal of breakpoints, addition of subgraphs and merging nodes. The merge function triggers a dialog (via the dialog service, section 4.3.4) to ask the user for confirmation. The actual merging creates a nested node on the graph and a nested tactic in the model, then it create a sub-graph in it containing the merged nodes, with eventual boundaries for inputs and outputs, finally it deletes the merged nodes on the current graph and connects the nested node with eventual input and output edges.

This controller is also where the selection informations (one node, one edge, multiple nodes ...) are listened from the Quantomatic API. Depending on what the selection contains, different events are send to the view.

4.2.6 Library

File: TinkerLibraryController.scala. Corresponds to the main view (section 4.1.6).

It contains two functions. One that, as the sub-graph preview, gets a Json representation of the graph from the file of the library, and then makes the Quantomatic API generates a graphical panel of it, in order to allow the view access it.

The other one is the function that takes the Json model from the file, updates the name to avoid eventual unwanted information duplication, integrate it to our PSGraph model and uses the root graph of the file to add it in the current graph.

4.3 Services

Services are, in this application, controllers meant to interact with the model or external resources of the application. In order to maintain their unicity and ease their access from anywhere in the application, they are developed to be singletons (a simple import in the headers of a file allow their use, no instantiation is needed).
4.3.1 Main service

In the file Service.scala you will find the main service. It instantiates all the controllers, that way communication with or between controllers is eased. A view can then simply access a controller object as they have a public access modifier.

It also implements the models of the application. And as this service is the only one in the controller package to "know" the models, a major part of its code consists in modifying / accessing model values and impact this changes in the views, via the controllers.

Also we instantiate the communication service, as it needs a initialisation to start listening for connections.

Getters on the PSGraph model allow to have access to the following values/objects:

- The whole PSGraph's Json representation.
- A specific subgraph's Json representation.
- The size of a tactic.
- The "OR"/"ORELSE" value of a nested tactic.
- The index (in a nested tactic) of the current graph.
- The size of the current nested tactic.
- The name of the current tactic.
- The tactic value of an atomic tactic.
- The goal types of the PSGraph.

Getters on the hierarchy model allow access to three objects: the root element, the active element and the list of a specified element's parents.

The following list describes the functions of this service that manipulates the models, sometimes the view (like, updating the hierarchy tree) and also the document service (mainly to register unsaved changes):

- **createNode**, given a name and two booleans (is it a nested tactic and if so is it an "OR" tactic), it creates the tactic in the model, update the hierarchy model and views if needed and return the actual name. The actual name is a modified name where we appended a number in order to avoid duplication of tactics' name.

- **updateTacticName**, it changes the name of a tactic, update hierarchy if needed and return an actual name (same reason as above)

- **deleteTactic**, deletes a tactic. At this moment this function is not used, i.e. can not be launched by the user's actions. The reason is that we need to specify a case where this has to be done.

- **parseAndUpdateArguments**, for a specified tactic, it takes a string input, parse it into a list of arguments (being a list of strings) using the argumentParser (section 4.5.3), and update it in the model. It returns a string, representing well formatted arguments.

- **updateArguments**, same as above only it takes a list of lists of strings as input (or a list of arguments), so no needs to parse it.

- **changeTacticParent**, updates the parent of an element in the hierarchy model.
• *addSubgraph*, creates a subgraph of a tactic in the model, and update the views for the edition of this subgraph.

• *deleteSubgraph*, delete a subgraph of tactic in the model.

• *editSubgraph*, given a tactic name and an index, it sets a specific subgraph as current in the model, gets its representation, loads it into the Quantomatic API (so it can return its graphical representation in the view) and updates the views for the edition of this subgraph. If the subgraph can not be found in the model, we call *addSubgraph*.

• *setIsOr*, sets the "OR"/"ORELSE" value of a nested tactic.

• *saveGraphSpecificTactic*, adds a graph Json representation in a tactic.

• *setAtomicTacticValue*, sets the tactic value of an atomic tactic.

• *setGoalTypes*, sets the goal types.

• *refreshGraph*, gets the current graph's Json representation from the model, and loads it in the Quantomatic API, updates the hierarchy tree as well.

• *changeGraphEditMouseState*, changes the mouse state in the graph edition controller.

A listener is also implemented. On a change in the graphical view of a graph, an event is launched from the Quantomatic API with a Json parameter (the graph's Json). The service then updates the model.

Finally there are few functions to interact with the document service. Although we could have used the document service directly, we go through the main service as informations from the model are most of the time needed:

• *saveToJsonToFile*, gets the PSGraph's Json (with last updates) and save it to the current file, if there no current file, we call the *saveAs* function.

• *saveJsonAs*, gets the PSGraph's Json (with last updates) and calls the *saveAs* function.

• *closeDoc*, gets the PSGraph's Json (with last updates) and check if it needs to be saved before returning a boolean, validating the closure of the application

• *newDoc*, gets the PSGraph's Json (with last updates) and check if it needs to be saved before creating a PSGraph in the model, updating the hierarchies and calling the *refreshGraph* function.

### 4.3.2 Document service

The document service (DocumentService.scala file) we implemented is largely inspired by the Quantomatic project's one. Its main purpose is to interact with the file management system.

It has two variables, one keeping track of the file the user is working on with a default value set to "None", as it is an optional variable. The other variable is a boolean telling whether or not there are unsaved changes on the PSGraph, the default value is false, it is set to true at every modification on the PSGraph, and back to false when the user saves its work.

Some method in this object are meant to simply set and access some value related to the document status. *setUnsavedChanges* sets up the unsaved changes boolean to the value specified as parameter, it also trigger an event stating a change in the document status. *title* return the title of the document, i.e. the file name, if there is no file specified, it is "untitled", it also appends a "*" at the end of it if there are unsaved changes. *previousDir* as two behaviour, one returns the
last directory the user worked on with our application, or, by default, the user's home. The other
one sets up this last directory, using the file specified in parameter.

This service also launches two prompts. One checking for the existence of a file, for example
when the user saves a PSGraph and specified a file name. If the file already exists, the prompt asks
the user for a confirmation. The second prompt checks for unsaved changes (when the user wants
to close the application or open a new file) and asks the user if he/she wants to save the change,
discard the changes or cancel the action. It returns the value "true" when everything has been done.

There are two function regarding the save of a file. A simple one called save that writes a
specified Json in the current file or a specified one. And one called saveAs that first launch a file
chooser, where the user can choose in which file to save the specified Json, then calls the save
function with the chosen file and the Json as parameters.

Finally there are two functions related to the opening of file. The first one, showOpenDialog,
opens a file chooser where the user choose the file to open. And there is load, which actually reads
a specified file and parses its content into Json. Both of these functions return an optional Json
(the value "None" meaning there has been a problem when loading a Json), load usually returns it
to showOpenDialog, which returns it to the calling object.

In future development an undo stack management should be added to the document service,
allowing a better use of it. At the moment we use the undo stack from the Quantomatic part of
the application, so it only works fro the current graph.

4.3.3 Communication service

This service, CommunicationService.scala file, is meant to work with socket connections with core
of Tinker. The way it works is the following :

- First it initializes a server socket on port 1790, and waits for connection from a client on
  that port. To avoid blocking the application while waiting, we use a Future type, allowing
  asynchronous computation.

- When a connection is made, it starts to listen.

- Listening means waiting for messages in the input streams. When a first line comes, we get
  the whole message.

- To get the complete message, we read line after line until the input stream is empty.

- When we have the message, we parse it, it is suppose to a valid Json. We try to find a
  command and eventual parameters, launch actions according to that, and maybe send an
  answer.

- Finally we loop back to the listen state, or to the first state, if the command was to close the
  connection.

Future development might change this protocol, since the communication protocol is not finalised
on the core side.

4.3.4 Dialog service

The dialog service, implemented in the TinkerDialog.scala file, provides the application common
dialogs that can be used in many situation. It is actually one dialog changing its layout depending
on how we use it. There are three modes :

- A confirmation dialog, displaying a message and a list of buttons corresponding to actions
  specified as parameters.
4.3.5 Events

In the file `TinkerEvents.scala`, you will find a list of events’ signatures, used in the entire application. This list may change in the future after refactoring has been made.

4.4 Models

Two models are used in our application: one for the actual PSGraph, and one for the hierarchy between tactics. This last model was developed later in the development process to help with the hierarchy view, as well as with the breadcrumbs. We could technically have reused the PSGraph model, but this new model is a lot simpler.

4.4.1 PSGraph model

This model is spread in five files: `PSGraph.scala`, `GraphTactic.scala`, `AtomicTactic.scala`, `Argument.scala` and `HasArguments.scala`. It describes the following organisation (figure 4.3):

![PSGraph model organisation](image)

In practice, a PSGraph is a singleton, only one instance of it is made. It contains many variables like:

- A boolean telling if the current graph is the main one.
- A main graph.
- A current nested tactic. If the `isMain` value is false, then we will pick the current graph in this tactic.
- A current index, telling which of the graph in the nested tactic is the current one (if isMain is false).
- A goal types string.
- A list of atomic tactics.
- A list of nested tactics, or graph tactics.

A Graph object is a Json generated by Quantomatic, it is with this Json that we construct a graphical view of a graph. An atomic tactic has three variables, a name (string), a tactic name (string) and a list of arguments. A nested tactic (in file GraphTactic.scala) has four variables, a name (string), a list of arguments, list of graphs and a boolean saying if it is an "OR" or an "ORELSE" tactic. As atomic and nested tactics have the same behaviour regarding arguments, an interface, HasArguments, manages everything regarding the list of arguments of the tactics. An argument is a list of string.

Various functions to manipulate the model are implemented in PSGraph.scala. This file also ensures an important feature of the model: tactics' unicity. Indeed when editing a tactic we do not want to mix tactics so we have to make sure that they are all unique at every time: when created and when edited. This unicity is now only supported through the name of the tactic. The function generateNewName ensure the unique names, by looping through all the tactic names already used and appending an index at the given name if necessary. However, it is clear that this system leads to a major issue. What if the user want to use the same tactic, it is of course legitimate for the atomic tactics, and can only be done under certain circumstances (like not putting a nested tactic inside itself). Further development should allow that and leave options to the users: if he/she creates two tactics with the same name but a different number of arguments, if he/she creates two tactics with the same signature, does he/she actually want two distinct but look-a-like tactics, or did he/she forget about the first one and realises he/she want the same thing. When editing, should the application edit all nodes with the same tactic name or create a clone. When deleting the last node representing a tactic should we erase the tactic from model, or keep track of it?

Finally, the main use of this model is to generate a Json object, that will be used by the application core, and save on the file system. It has the same structure as below:

```
{
    "ps_graph":{
        "current":"nested", // if name = main then main graph
        "current_index": 0,
        "graph":{
            "dir_edges":{ // list of the edges
                "e1":{ // edge id
                    "data":{ // edge data
                        "gtyp_name":"any", // the goal types
                        "type":"GT" // type of edge ?
                    },
                    "src":"v0", // source vertex
                    "tgt":"v1" // target vertex
                },
                "e0":{ // ...
                },
            },
            "wire_vertices":{ // list of the boundaries
                "b1":{ // boundary id
```
"annotation": { // boundary data
  "boundary": true, // is it a boundary
  "coord": [1.1, -3.55] // boundary coordinates
},
"b0": { // ...
},
"node_vertices": { // list of the interesting nodes
  "v1": { // node id
    "data": { // node data
      "rt_name": "rtech", // node name
      "appf": { // (Atomic/Identity/Nested)
        "type": "Atomic" // type name
      }
    },
    "type": "RT_ATM" // type id (RT_ATM/RT_ID/RT_NST)
  },
  "annotation": {
    "coord": [-1.14, 1.51] // node coordinates
  },
  "v0": { // ...
  }
},
"graph_tactics": [
  {
    "name": "thing", // name of the tactic
    "args": [ // list of the arguments
      [ // one argument : list of string
        "string",
        "string2"
      ], [ // ...
      ]
    ],
    "isOr": true, // applying OR or ORELSE
    "graphs": [ // list of the possible graphs
      { // same signature as main graph
        // (generated by quantomatic)
      },
      [ // ...
      ]
    ]
  },
  [ // ...
  ]
],


```
"atomic_tactics": [
    {
        "name": "simp", // the name
        "args": [ // same as graph tactics arguments
        ],
        "tactic": "simplify" // the tactic
    },
    {
        // ...
    },
    // ...
],
"goal_types": "..."
}
```

4.4.2 Hierarchy model

The hierarchy model was made to simplify hierarchy related views (breadcrumbs and hierarchy tree mainly).

It basically implements a tree architecture of all the nested tactics, the root being the main graph. Every node in this tree as three variables:

- A name (string), typically the name of the tactic.
- Children (list of nodes).
- A parent (string), referring to the parent tactic.

We keep track in this model of the current tactic as the active node.

Beside basic manipulation functions, this model, implemented in the file `HierarchyModel.scala`, also provides function to rebuild the hierarchy with a PSGraph model.

4.5 Utils

This package is meant to provide classes, not natives in `scala.swing` but useful in our application.

4.5.1 Popup menu

This code, in file `PopupMenu.scala`, was found in a StackOverflow discussion. As every contribution on this website, it is under the Creative Commons Licence.

The idea is to simply reuse the popup menu of `java.swing` and introduce its use into our `scala` environment.

A basic use of the popup is to implement a `PopupMenu` object and add a list of menu items in its `contents` variable.

4.5.2 Toolbar

The toolbar is implemented in the file `ToolBar.scala`. It comes from the Quantomatic project, but was externalised so we could use it in a standalone way.

It gives an implementation of the tool bar above the graph view.

---

6See : Discussion.
4.5.3 Arguments parser

In the file `ArgumentParser.scala` a list of functions has been implemented to help in the parsing of node values in a graph (on a graphical point of view). Those values are describing the name of the tactic and its arguments. These functions then separate the name from the arguments and take the string of arguments to separate them from each other and put them into a list. Each argument is then parse a last time to become a list of strings.

The following diagram (fig 4.4) illustrates the process:

![Diagram](image)

Figure 4.4: Arguments parsing.

Of course the reverse functions are also implemented in the same file.

4.5.4 Mouse state

It is implemented in the file `MouseState.scala`. As for the tool bar, it originally comes from the Quantomatic project, but is used separately to ease the standalone use of our application.

It is a datatype specifying the state in which the mouse is for edition: selecting, drawing, dragging ...
5. Quantomatic integration

Using Quantomatic in our application is possible through an API file called `QuantoLibAPI.scala`. It allows a library like use of Quantomatic, and although at the moment its code is integrally under the `lib` folder, the idea would be to have it as an installed package in the user’s system.

Therefore the API was made so it can access objects related to the edition of graphs:

- A view, a class keeping managing the visual aspect of a graph.
- A document, a class keeping track of the modifications made on the graph, as well as managing its writing and reading in the file management system (like our document service).
- A graph, the model of a graph.
- A theory, a class helping building the model, depending on theory used for this graph (in our case, is it always a $psgraph$ theory).

All of them being attributes of a "graphPanel", an actual visual element. This element is the one send to our application graph edition view, where the user really edits the graph.

The majority of the functions implemented in this file are meant to be called by the controllers or the services of our application in order to modify the visual aspect of a graph, but also its model. At every changes, it refresh its view for the user and publishes an event so the main service can update our model with the Json representation of the current graph.

It also listens to mouse and keyboard events (user editing the graph) to either modify things itself or publish events so this actions can be handle by our controllers if further computation is needed.

Although this was made so the Quantomatic files do not have to be changed, we had to modify them at one point: when we wanted to introduce new shapes for the nodes. To change the colors we simply had to modify the theory file `strategy_graph_modified.qtheory` under the folder `resources/quanto/data`. But then we introduced new shapes:

- A triangle for the identity nodes.
- A double rectangle for the nested nodes.
- A octagon for the breakpoints.

We then modified the files `Theory.scala` and `VertexDisplayData.scala` respectively to:

- Read new entries in the theory file ("triangle", "octagon", "multiRect") to notify the vertex display data file.
- Draw the new shapes, compute the hit value (how far from the origin of the shape, do we consider the user hit the node, or if he/she draws an edge from or to this node, what is the coordinate of the point where the edge hits the node) and modify the font size of the string in the nodes (for breakpoints).
6. Conclusion & future work

This report has detailed the implementation of the new GUI for Tinker. Screenshots and installation guidelines can be found in the appendices.

The main limitation of the tool is support for evaluation and communication with the core. Next, we will develop a more robust communication protocol to achieve this. As discussed in section 4.3.3, the protocol on the GUI side as been partially done, the only missing thing is a clear command interpreter and the functions that will be needed for the evaluation part (connecting the buttons to commands in the controller, changing the display of the graph ...). However, we already have some ideas how we would like to do this:

- An asynchronous method waiting for connection on the GUI side, avoiding blockade during execution.
- A synchronised variable keeping the connection on the core side (to handle parallel execution in e.g. Isabelle).
- States on the core side, like: Proving, Drawing, Failed, Idle, Connected ...
- Json formatted messages, with fields for commands, parameters, responses ...

The main use case for evaluation will be as following (if the user started from the GUI of Tinker):

- The user draws a proof strategy graph on the GUI and requests an evaluation.
- The core gets the graph’s Json representation and evaluates it.
- It returns another graph with goals drawn on it stating the success or failure of a strategy. Either step by step (tactics by tactics) on the user request or just the final result.
- If any breakpoints has been added to the graph, and the user is on debug mode, the evaluation will pause there.

We do also expect some other changes to the GUI based on user experience and feedback.
A. Screenshots

Here is screenshots of the new GUI.

![New GUI](image)

Figure A.1: Layout of the application when just opened, i.e. an empty project.
Figure A.2: Layout of the application when editing a graph.

Figure A.3: Idem, with comments.
Figure A.4: Dialog when editing a node.

Figure A.5: Hierarchy tree window.
B. Installation guidelines

Dependencies

In order to run the GUI on your computer you will need the following packages installed on your computer:

- SBT (Scala Build Tool) : http://www.scala-sbt.org/download.html.
- Java 6 or above : https://java.com/en/download/manual.jsp

Download

You can then download a .ZIP file of Tinker from Github, https://github.com/ggrov/tinker, and extract it in a folder of your choice. Otherwise, you may instead run the following commands in your terminal (assuming you moved to the folder of your choice using the cd command):

```
$ git clone https://github.com/ggrov/tinker
```

or

```
$ git clone git://github.com/ggrov/tinker.git
```

Run

To run the GUI, execute the following commands:

```
$ cd path_to_your_folder/tinker/tinkerGUI
$ sbt run
```