University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program: Master of Science in Computer Science	Spring semester, 2018 Open access
Writer: Olav Salhus	_____ (Writer's signature)
Faculty supervisor: Reggie Davidrajuh	
Thesis title: Developing an online monitor for discrete simulations	
Credits (ECTS): 30 ECTS	
Key words: Discrete Event Dynamic Systems (DEDS) Petri net (P-N) GPenSIM MATLAB Graphical user interface (GUI)	Pages: 54 Attachment: source code + GPenSIM v10 Stavanger, 15 June 2018

Abstract

This paper covers the design and implementation of a graphical simulation monitoring program. The monitor works in real-time or simulation time. The monitor is compatible with the modelling and simulation tool GPenSIM [1] (**G**eneral-purpose **Pe**tri **N**et **Sim**ulator). The monitor shows the current state of a simulation in a graphical user interface (GUI) window while it is running. The monitor and GPenSIM are both running on the MATLAB platform [2]. The tool is easy to use while enabling user customization. It also includes the ability to show historical data in the form of logging and plotting.

1 Introduction

The monitor fetches data using a function call added to GPenSIM. MATLAB GUIDE (Graphical User Interface Development Environment) is used for GUI programming. GUIDE is included in the base version of MATLAB not depending on any additional toolboxes. MATLAB GUIDE does include a drag and drop environment, but it will not be used. Instead a programmatic workflow will be used throughout the program. Some knowledge from human-computer interaction (HCI) theory will be used when attempting to create a good user experience for the user.

2 Problem definition

A simulation in GPenSIM normally runs without any feedback until it is finished. Some simulations can take a long time to complete, especially if they run in real-time or have high complexity. The user running the simulation may want to monitor the state of the simulation while it is running. This is possible by modifying the simulation files to output to the command window or a plot and rerunning the simulation code. This approach can be impractical for several reasons:

- The person running the simulation might not have programming background needed to modify how logging is performed.
- The simulation must restart from the beginning whenever different events needs to get logged.
- Transitional definition files need to be modified to include logging.

This tool attempts to solves these problems. One of the biggest challenges is to not overwhelm the user with information. This is done by letting the user select what transitions and places to monitor or plot. This type of filtering can be done before and after the simulation has finished.

3 Previous work

A monitor for GPenSIM has already been created by Nigussie Girma Tulu in 2014 [3]. The main problem with the monitor was that it heavily modified GPenSIM source code even affecting how simulations are normally run. The new monitor tries to minimize modifications made to GPenSIM keeping a modular approach making it clearer what information is passed from GPenSIM to the monitor. The monitor mostly reads information from GPenSIM without modifying anything; however, it can slow down, pause and stop the simulation. While the new monitor does minimal changes to the source code it is important to note that it unfortunately relies on undocumented features. This is mainly related to logging states and events in the simulation. When a new GPenSIM version is released the monitor needs to adapt to any changes in its internal data structures.

4 Table of contents

5 Design

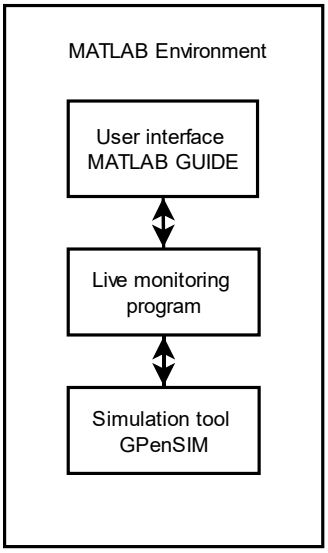
5.1 Human computer interaction and user experience

Human-computer interaction theory seeks to improve the usability of computer hardware and software. As the name suggests, it does this by analysing the interaction between a computer and its user.

A good user experience is critical when designing a graphical user interface (GUI). If the interface presents too much information at one time it will take a longer time to learn how to get started. While too little information at one time means the user must navigate back and forth to find relevant information. Having an idea of who will use the tool and what it will be used for is essential to create a good user experience.

The following are some example question that can be relevant to consider when designing the monitor GUI:

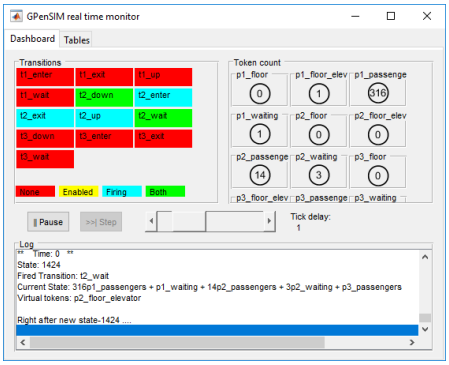
- The user might have other windows open at the same time. How should the user interface adapt to window resizes?
- The user might want to select what information he wants. Where can the user specify this?
- The user might gloss over the manual or not read it all, does he need to know any hotkeys or new terminology to get any use out of the program? Or will the user be able to get up to speed quickly yet be able to discover optional time saving features later?



5.2 Software architecture

Since GPenSIM uses the MATLAB environment the monitor is also created in MATLAB. The user interface uses GUIDE.

5.3 Dashboard



5.3.1 Transition

Each transition contains the name and a colour representing the state of the simulation, clicking the transition will open a live plot of the transition state over time/state.

5.3.2 Places

Every place on the dashboard is named with a number inside a circle to represent the number of tokens inside the place. When clicking a place, a live plot opens with token count as a function of time/state.

5.3.3 Simulation controls

Depending on the simulation type the simulation can be paused, resumed, stopped, single stepped and slowed down.

5.3.4 Log

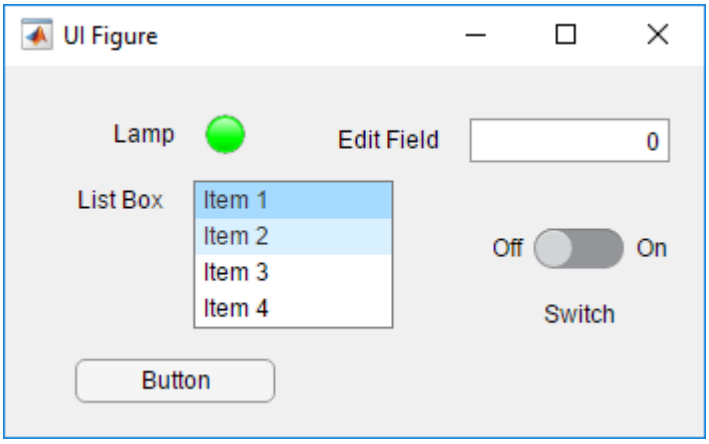
The logging is identical to how the "prnss" function from GPenSIM can output a history of state changes after a simulation. The function that does this has been adapted to log during the simulation to a section at the bottom of the GUI.

5.4 Configuration

The user can change what transitions and places are shown on the dashboard. This is done by selecting the configuration tab.

5.5 Design alternatives

5.5.1 MATLAB App designer



MATLAB App designer has a very similar API to MATLAB GUIDE even though App designer uses web components instead of the Java GUI components used by GUIDE. Unfortunately, App designer is only supported starting with MATLAB R2016a and is still under development [4]. This means that the application might break in newer versions of MATLAB and is incompatible with older versions. It is also missing some features from GUIDE, for example it only has absolute pixel positioning. The application could recalculate pixel sizes whenever the window resizes to work around this limitation. However, this will require more time and will add otherwise unnecessary complexity to the program.

6 Implementation

This chapter explains how the program was developed. It does so using explanations, screenshots and code snippets.

6.1 User interface

As mentioned, GUIDE is built into MATLAB allowing creation of user interfaces [4]. GUIDE uses a hierarchy of objects to define the UI. The root of the GUI is the main window which is created using the `figure` function. GUI elements (buttons, menus…) are children of either the root element or other GUI elements.

6.1.1 Window

The `figure` function is normally used for creating data plots in MATLAB. If the menubar is turned off (`Menubar = 'None'`) the window will be empty.

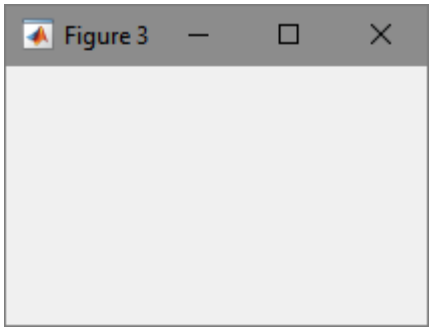
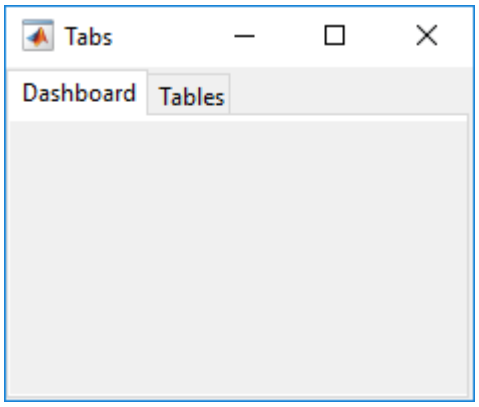


Fig 1: `figure('Menubar', 'none')`

6.1.2 Tabs



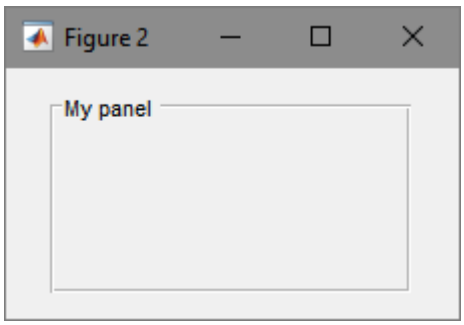
Tabs will be used in this application allowing the user to choose if they want to see a custom dashboard with only the selected tokens and transitions, or a table with all tokens and transitions. The "`uitabgroup`" function generates a tabbed UI element that has multiple tabs as children (`uitab`). Each tab "`uitab`" is a child of "`uitabgroup`" and can have its own UI elements as children, these are only shown when the tab is selected.

Code used to generate a tabbed UI:

```
tabs = uitabgroup(fig,'Position',[0 0 1 1]);
tab1 = uitab(tabs,'Title','Dashboard');
tab2 = uitab(tabs,'Title','Tables');
```

6.1.3 Panels

```
uipanel(root, 'Title', 'My panel',
'Position', [0.1 0.1 0.8 0.8]);
```



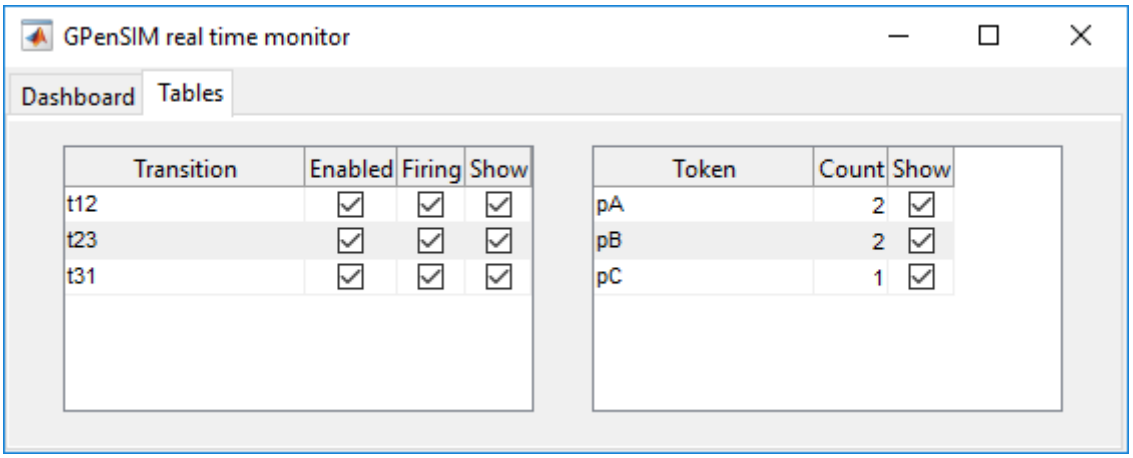
This application shows the state of transitions and places. A `uipanel` is used as a container for more GUI elements, it also provides a visible border and a title that can be set for the panel (see fig3). The position property specify the position of the bottom right corner (x,y = 0.1, 0.1) and the width and height (w, h = 0.8, 0.8). The position and size are relative to the parent where 0,0 is the bottom left corner and 1,1 is the top right corner of the parent element. The relative positioning is maintained when resizing the window.

When showing the state of transitions; a panel for each transition will be created. The transition panel's background colour will be updated to indicate the state of that transition.

Similarly, each "place" in the simulation has its own panel, the number of tokens is shown in the centre for each place.

6.1.4 Tables

The second tab of the interface contains two tables, one for the transitions and another for the tokens.



The token table is defined using the following code:

```
table = uitable(tab2);
table.Units = 'normalized';
table.ColumnFormat = {'char' 'numeric' 'logical'};
table.ColumnName = {'Token', 'Count', 'Show'};
table.ColumnEditable = [false false true];
table.ColumnWidth = {110 37 33};
table.RowName = {};
table.Data = {'t12', 2, true; 't23', 2, true; 'tCA', 1, true;} % example!
table.Position = [.525 .1 .425 .825]; % lower right
```

Note: The enabled, firing and count columns have been removed completely for performance reasons explained in chapter **Feil! Fant ikke referansekilden.**

Similar code is used for the transition panel. Only the “Show” column is editable. The user can toggle the checkmark for each transition in the “Show” column, the values will be updated in the “Data” property of the table. The main dashboard will only show tokens and transitions that have the “Show” column set.

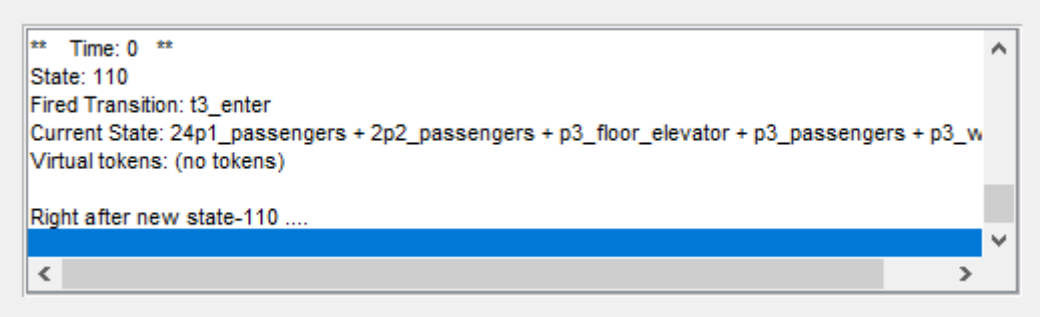
6.1.5 Listbox

A listbox is used for logging since it can be set up to automatically scroll to the bottom. A listbox is normally used for selecting a single element from a set. Selecting an element can be done programmatically which also conveniently reveals that element by scrolling the list. The normal textbox in MATLAB unfortunately does not support scrolling in the GUIDE API. [5]

Side note: It is possible to solve this problem by encapsulating a fully stretched out textbox inside a smaller panel. Then only revealing a small part of the textbox depending on the value from sliders (effectively replacing normal scrollbars). Implementing this reliably proved to be a challenge and was quickly abandoned once the `listbox` solution was discovered.

The listbox is created using “uicontrol” with ‘Style’ set to ‘listbox’. Each line in the log is added to a cell array of strings in the “uicontrol.String” property.

```
% Append to GUI window
self.StateLog.String{end} = [self.StateLog.String{end} string];
```




```
% Scroll to the bottom by selecting the last line
self.StateLog.Value = length(self.StateLog.String);
```

6.1.6 Interacting with GUI elements

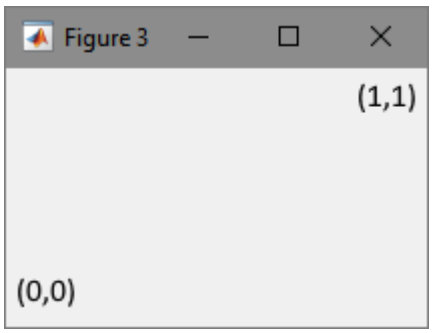
All GUI elements are referenced in the properties of the guiMonitor object (Window, Tabs, DashboardTab,TokenTable, TokenPanel,…). Whenever a property is changed in a GUI element it will take effect after the next redraw using the “drawnow” function. An example is changing the color of the second tokenpanel:

```
panel = self.TransPanels{2};
panel.BackgroundColor = 'green';
drawnow;
```

Some GUI elements also have different callbacks, this allows a referenced function to be triggered when a certain event occurs. The main window has an event called “CloseRequestFcn” that triggers when the user closes the window.

6.1.7 Element Positioning

Positioning of GUI elements can be done ‘normalized’ from 0 to 1 with respect to the parent size (see fig) or absolute based on various computer screen dimensions (pixels/mm/inches...). [5]



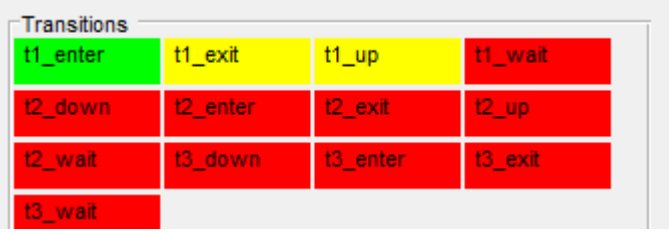
Normalized positioning is always relative to the parent element, ex: resizing the window will also cause all normalized elements to maintain relative positioning. This only means that elements are scaled, not reflowed.

6.1.8 Element reflow

To ensure good use of screen real estate the user interface is responsive to changes in window size. The user might dedicate a big screen to the GUI or just a quarter of the screen if he uses multiple programs at the same time. The program should in both cases present the given information as efficiently as possible.

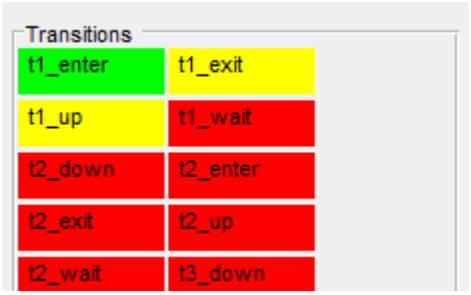
The dashboard elements reflows as the window changes size using the SizeChangedFcn event handler. Reflowing of elements must be explicitly programmed to change the position of the dashboard elements.

6.1.8.1 *resizePanel function*



The guiMonitor has a resizePanel function that takes a container element, a set of panels and the size each panel will occupy.

The function starts by getting the container dimensions in "character units", one character unit is the width and height of the character x using the standard font of the graphics root object.



The function loops through all container panels updating the position to make sure the elements are lined up. If the container position is about to collide with the edge it gets moved to the start of the next line.

The panels get shrunk slightly to add some space between the panels.

6.2 Interacting with GPenSIM

This chapter explains how the program accesses information from GPenSIM.

6.2.1 Additions to GPenSIM source code

The GUI elements in the dashboard should indicate the state of the simulation. This is done by changing the colours of the transition panels and drawing a number to indicate transition state and the number of tokens in a place. Extracting this information from GPenSIM at regular intervals is necessary. This happens at every simulation tick. To achieve this, a function call will be added to the bottom of the GPenSIM main loop.

The GPenSIM main loop (gpensim.m truncated with modifications highlighted):

```
while ~(SIM_COMPLETE),
    ...

    % stop if ((queue is empty) OR (max loop) OR (max log size)) is reached
    SIM_COMPLETE = simulations_complete(Loop_Nr, global_info.MAX_LOOP);

    % ADDON: Real time monitor (may be removed if unused)
    if isfield(global_info, 'MONITOR_RUNNING') && global_info.MONITOR_RUNNING
        global_info.MONITOR_GUI.showData(LOG, Enabled_Trans_SET, Firing_Trans_SET);
    end; % ADDON: Real time monitor

end %while ~(SIM_COMPLETE)
```

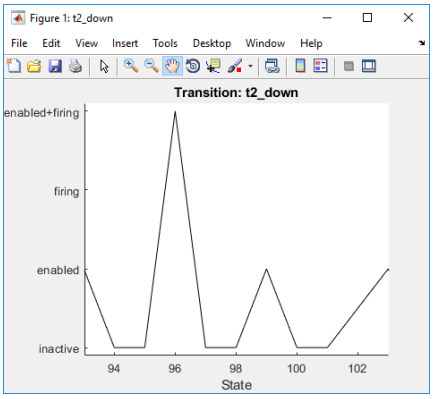
Existing simulations will only call the function if MONITOR_RUNNING is a field inside global_info and is set to true. This is convenient to prevent slowing down simulations that don't use monitoring and can also be used to temporarily stop monitoring of a monitored simulation.

Notice that the showData function is called on the object `global_info.MONITOR_GUI` when the `guiMonitor` object is created it will add a reference to itself in `global_info.MONITOR_GUI`. This is convenient since GPenSIM already uses `global_info` .

Note: The astute reader might expect that passing the entire LOG object might cause a problem if it gets copied. The good news is that MATLAB uses a copy-on-write model (aka. Lazy copy) [7]. This means that the LOG object only gets copied in memory if a modification is made to it, which conveniently is not the case.

6.2.2 Live plotting

Plotting is used to give the user the ability to easily observe state changes over time during- or after simulation. Clicking a transition or place will open a separate plotting window. Places will show number of tokens over time (or states). The transition plot will show which of the 4 possible states it is in at a given time (enabled/firing/both/none).



When the user clicks a transition or place panel it triggers its "`ButtonDownFcn`" callback. The callback function for transitions and places opens a plot in a separate figure window.

```
transPanel.ButtonDownFcn = {@self.initLiveTransPlot, i};
tokenPanel.ButtonDownFcn = {@self.initLiveTokenPlot, i};
```

Both figures contain an animatedline plot which is a built-in MATLAB plotting interface optimized for streaming data in real time. The addpoints function is used to add one or more points to the plot. The axis object is updated when adding points causing the plot to scroll along the X axis during simulation. The x-axis represents time, in an untimed simulation (non real-time and without firing times) the state number is used instead.

Initially all existing history stored in the LOG object is used to plot, transitions additionally refer to the `Enabled-/Firing_Trans_SET` matrices.

For places:

```
function [x, y] = plotHistoryOfTokenCount(self, place_id, ...
                  No_of_places)
if self.IsTimedPN
      column_end_times = No_of_places + 6;
      x = self.LOG(:, column_end_times);
      y = self.LOG(:, place_id);
else
       ... handle untimed petri net ...
end
```

For transitions:

```
function [x, y] = plotHistoryOfTransition(self, trans_id, ...
      No_of_places)

if self.IsTimedPN
      x = self.Firing_Trans_SET(:, 1);
      ETS_i = 1:size(self.Firing_Trans_SET, 1);
      y = 2 * self.Firing_Trans_SET(ETS_i, 1 + trans_id) + ...
              self.Enabled_Trans_SET(ETS_i, 1 + trans_id);
else
       ... handle untimed petri net ...
end
```

Once this is done `showData` will add new datapoints for every simulation tick and scroll the x axis accordingly. `showData` will go through all valid animatedline objects and add new points and move

the x axis. In the case of token counts it will change the y axis to be able to show the highest point of the graph.

6.2.3 showData function

The showData function itself takes the LOG, Enabled_Trans_SET and Firing_Trans_SET matrices. These are all used for logging. The global object PN is used to grab the token counts (PN.X), PN.Firing_Transitions and PN.Enabled_Transitions. As well as names of places and transitions (PN.global_places and PN.global_transitions).

6.2.3.1 Only show tokens and transitions that are selected

When showData gets called it will check if the user has changed the shown tokens and transitions. If it detects a difference between currently shown tokens and selected tokens, it will simply reset the panels representing each place.

```
% if the user changed the shown tokens
currentlyShownTokens = cell2mat(self.TokenTable.Data(:, 2))';
if ~isequal(self.ShownTokens, currentlyShownTokens)
    self.ShownTokens = currentlyShownTokens;
    self.resetTokenPanel();
end
```

The same code structure is used for transitions.

The resetTokenPanel will remove all existing token panels, it will then add back the selected tokens to be shown in the GUI. This is more of a "brute force/fix all" approach compared to reusing existing panels. This is acceptable since it will only happen when the user changes the currently shown tokens. The same goes for transition panels.

6.2.3.2 Updating transitions on the dashboard

Indicating the state for each transaction is done by changing the color of the visible panels. The visible transition panels are referenced in `guiMonitor.TransPanels`. 2 logical arrays `firing` and `enabled` represents the current state of all transitions.
It is important to note that since some transitions might be hidden in the dashboard the indices may not correspond to those in `guiMonitor.TransPanels`.

Example:

```
transitions names = ['t1', 't2', 't3']
Self.ShownTrans = [1 0 1]
Self.TransPanels = [<t1 panel object> <t3 panel object>]
```

Since the second transition is hidden (`self.ShownTrans[2] == 0`) the second element of self.TransPanels is going to be t3. The ordering of transitions is consistent even when transitions are removed or added to the dashboard (the transition index is always increasing inside `self.TransPanels`).

When looping and filtering the transitions, two indices are used. `j` represents the transition index and `i` represent the dashboard panel index. Depending on the state of each transition a unique color is added to the transition panels background color:

```
% update transitions shown in dashboard
index_trans_panels = 1;
for j = 1:length(self.ShownTrans)
    if self.ShownTrans(j)
```

```
        panel = self.TransPanels{index_trans_panels};

        newColor = self.getTransColor(enabled{j}, firing{j});

        % only perform expensive operation when needed
        if ~isequal(panel.BackgroundColor, newColor)
            panel.BackgroundColor = newColor;
        end

        index_trans_panels = index_trans_panels + 1;
    end
end
```

Note that `j` increases for each transition while `i` only increases for each shown transition. `i` is used to reference the actual shown GUI elements in `self.TransPanels`.

6.2.3.3 Updating tokens on the dashboard

The same loop structure also applies to tokens. The difference is that tokens are represented using a number inside a circle. The circle is added to get a familiar look compared to actual places in a Petri net diagram.

The circle and text are children of an axes element which again is a child of the corresponding token/place panel.

Relevant source code:

```
% update tokens shown in dashboard
i = 1;
for j = 1:length(self.ShownTokens)
    if self.ShownTokens(j)
        panel = self.TokenPanels{i};
        if panel.UserData.currentTokenCount ~= tokens{j}
            panel.UserData.currentTokenCount = tokens{j};
            panel.UserData.text.String = num2str(tokens{j});
        end
        i = i + 1;
    end
end
```

6.2.4 updateLog function

In the showData function the parameters `LOG`, `Enabled_Trans_SET` and `Firing_Trans_SET` are passed into the guiMonitor.updateLog function. This function logs the state of the simulation in a listbox. The source code is taken from GPenSIM's **"prnss"** function and adapted for real time logging to the GUI.

6.2.4.1 GPenSIM log structure

The log output corresponds to the information inside `LOG`, `Enabled_Trans_SET` and `Firing_Trans_SET` matrices.

The following example shows log output corresponding to highlighted rows in the GPenSIM datastructures. The transition and place names are accessible from `PN.global_transitions.name` and `PN.global_places.name` during simulation.

```
{PN.global_places(:).name}       => {'pA'}       {'pB'}       {'pC'}
{PN.global_transitions(:).name} => {'tAB'}      {'tBC'}      {'tCA'}
```

```
Output:
**    Time: 7.2    **
State: 12
Fired Transition: tAB
Current State: pB
Virtual tokens: pC

Right after new state-12 ....
At time: 7.2,  Enabled transitions are:    tAB    tBC
At time: 7.2,  Firing transitions are:     tAB    tBC


**    Time: 7.2    **
State: 13
Fired Transition: tCA
Current State: pA + pB
Virtual tokens: (no tokens)

Right after new state-13 ....
At time: 7.2,  Enabled transitions are:    tAB    tBC
At time: 7.2,  Firing transitions are:     tAB    tBC
```

LOG:

Row	Places			FT	State		ETS ID FTS ID	Time		Virtual tokens		
	pA	pB	pC		current	previous		Start	Stop	pA	pB	pC
54	0	0	0	0	0	0	0	7	7	1	0	1
55	0	0	0	0	0	0	0	7.2	7.2	1	0	1
56	**0**	**1**	**0**	**1**	**12**	**11**	**43**	**6.2**	**7.2**	**0**	**0**	**1**
57	**1**	**1**	**0**	**3**	**13**	**11**	**43**	**6.2**	**7.2**	**0**	**0**	**0**
58	0	0	0	0	0	0	0	7.2	7.2	1	1	0
59	0	0	0	0	0	0	0	7.4	7.4	1	1	0

Note: The enabled transition row number (ETS ID) has the same value as its corresponding firing transition row number (FTS ID). Therefore, there is no need for a seperate `FTS_index` in the source code. Example from GPenSIM:

```
function [] = print_statespace_enabled_and_firing_trans(ETS_index)
...
enabled_trans = PN.Enabled_Trans_SET(ETS_index, 2:end);
firing_trans  =  PN.Firing_Trans_SET(ETS_index, 2:end);
...
```

When the log outputs “right after new state-12…” it takes the ETS ID/FTS ID from the row with current state 12 (see LOG row 56) and increments it by 1. In this case 43 + 1 = 44. It then outputs the enabled and firing transitions from row 44 in `Enabled_Trans_SET` and `Firing_Trans_SET`.

The output is identical under “right after new state-13…” since state 12 and 13 both refer to the same ETS/FTS ID.

Enabled_Trans_SET

Row	Time	tAB	tBC	tCA
42	7	0	0	0
43	7.2	0	0	0
44	**7.2**	**1**	**1**	**0**
45	7.4	0	0	0

Firing_Trans_SET

Row	Time	tAB	tBC	tCA
42	7	1	0	1
43	7.2	1	0	1
44	**7.2**	**1**	**1**	**0**
45	7.4	1	1	0

6.2.4.2 Logging states

When updateLog gets called for the first time it will loop through all LOG rows from **self.NextLogIndex** to the end. The self.WaitForETSIndex will be explained later, it will be zero the first time updateLog is called.

```
if self.WaitForETSIndex
    ...
end
```

```
if ~self.WaitForETSIndex
    for row = self.NextLogIndex:no_of_rows
        current_row = LOG(row, :);
        fired_trans = current_row(Ps+1);
        if (fired_trans)
            ...Start logging...
        end
    end
    self.NextLogIndex = row + 1;
end
```

Note: self.NextLogIndex cannot be set to no_of_rows + 1 because a condition can break out of the for loop before the for loops is finished meaning that row is not equal to no_of_rows.

If a row in the LOG matrix has a fired transition it will be logged. Inside the if (fired_trans) block, relevant information is extracted from the LOG entry and then added to a character array called "`string`" containing the output in plain text.

```
        if (fired_trans)
            state = current_row(Ps+2);
            ETS_index = current_row(Ps+4);
            % start_time = current_row(Ps+5); % not used by "prnss"
            finishing_time = current_row(Ps+6);
            current_markings = current_row(1:Ps);
            virtual_tokens = current_row(Ps+7:end);

            lines = self.log_prnss_state(fired_trans, finishing_time, ...
```

```
                    current_markings,state, virtual_tokens);
            string = [string lines];
```

`self.log_prnss_state` presents the given parameters in a readable form and is almost identical to `prnss_state`. The only difference is that it returns a string instead of just displaying the output in the MATLAB command window. Example output:

```
**    Time: 7.2   **
State: 12
Fired Transition: tAB
Current State: pB
Virtual tokens: pC

Right after new state-12 ....
```

The next step is to show the enabled and firing transitions right after the current state. This is done by taking the `ETS_index` (=`FTS_index`) and incrementing it by 1. The function `log_prnss_ET_FT` outputs a string and is based on `print_statespace_firing_trans`. It takes the `Enabled_Trans_SET, Firing_Trans_SET and ETS_index`, to output a string containing the relevant transitions. Example:

```
At time: 7.2,  Enabled transitions are:    tAB   tBC
At time: 7.2,  Firing transitions are:     tAB   tBC
```

The problem is that the Firing- and Enabled transition sets may not have advanced to that point yet. A check is therefore added to make sure. If the transition sets have not advanced; the `ETS_index` is stored in `self.WaitForETSIndex` which causes the `self.updateLog` function to not log anything until the transition sets have advanced. Once the transition sets include the `self.WaitForETSIndex` the firing and enabled transitions are outputted. The `self.WaitForETSIndex` is then set back to zero indicating that `self.updateLog` should continue with normal operation. Source code where the relevant logic is highlighted:

```
function updateLog(self, LOG, Enabled_Trans_SET, Firing_Trans_SET)

...Variable definitions from function parameters and global PN...

if self.WaitForETSIndex
    if self.WaitForETSIndex <= len_ETS
        lines = self.log_prnss_ET_FT(...
                Enabled_Trans_SET, Firing_Trans_SET, self.WaitForETSIndex);
        string = [string lines ' ' newline ' ' newline];
        self.WaitForETSIndex = 0; % done waiting for an ETS_index
    end
end

if ~self.WaitForETSIndex
    for row = self.NextLogIndex:no_of_rows

        current_row = LOG(row, :);
        fired_trans = current_row(Ps+1);
        if (fired_trans)
            ...

            ETS_index = current_row(Ps+4);
```

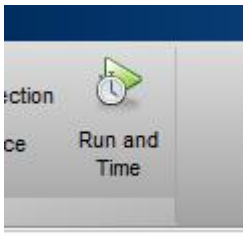
```
            % Check if the next ETS_index exists
            if len_ETS >= (ETS_index + 1)
                % show the enabled and firing transitions
                lines = self.log_prnss_ET_FT( ...
                    Enabled_Trans_SET, Firing_Trans_SET, ETS_index + 1);
                string = [string lines ' ' newline ' ' newline];

            % If not: wait for the Transition sets to get populated
            % for the next loop (see "if self.WaitForETSIndex ..." above)
            else
                self.WaitForETSIndex = ETS_index + 1;
                break;
            end
        end
    end
    self.NextLogIndex = row + 1;
end
```

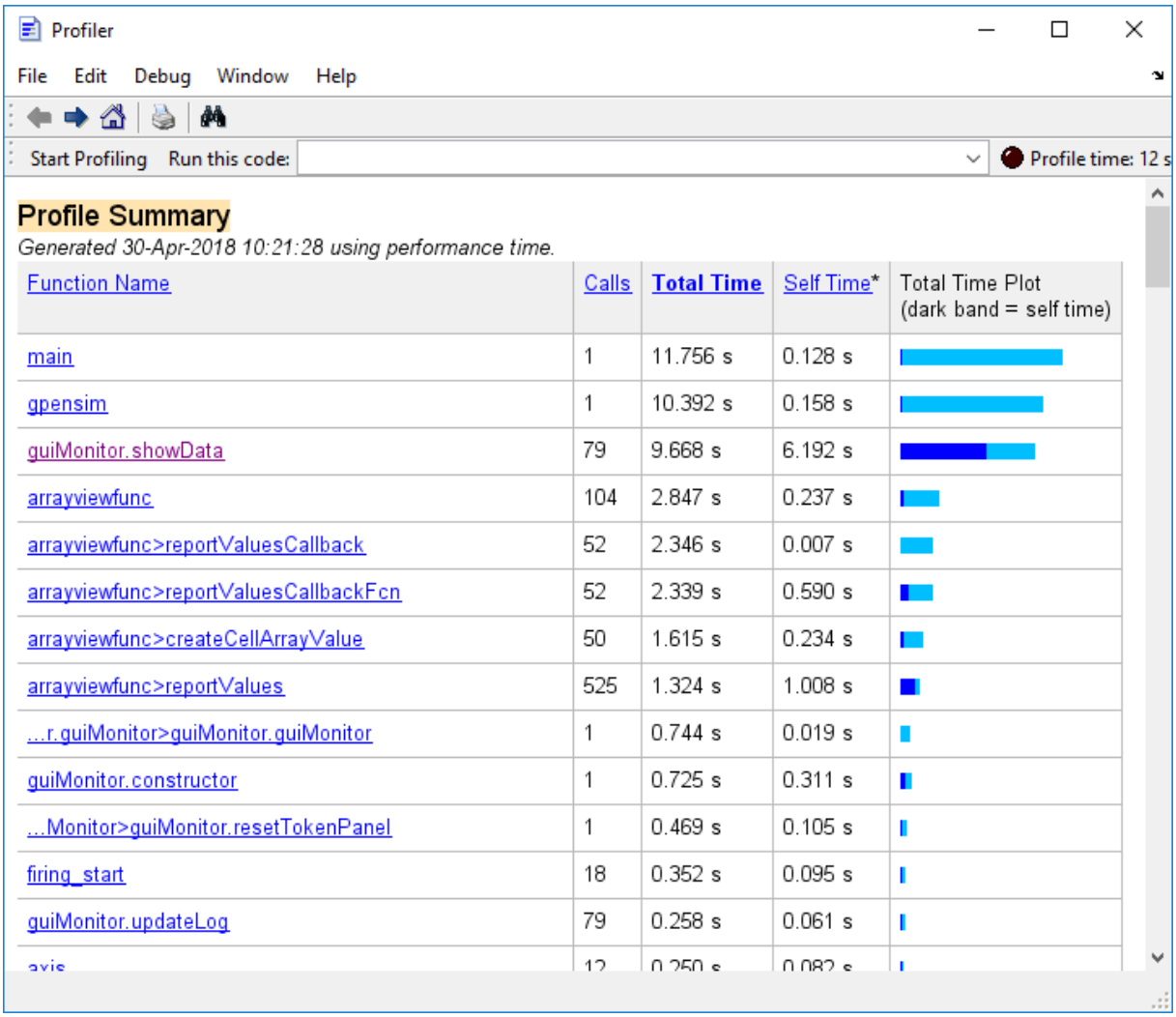
6.3 Optimizations

This tool is designed to save time by letting the user quickly and easily add live logging into any simulation. Having a responsive user interface without any unnecessary waiting is therefore critical for its usability.

6.3.1 MATLAB profiler (“Run and time”)

MATLAB has a built-in profiler accessible by pressing “Run and Time” instead of “Run”. It tells how much time is spent by each line of the source code. The following is an example output:



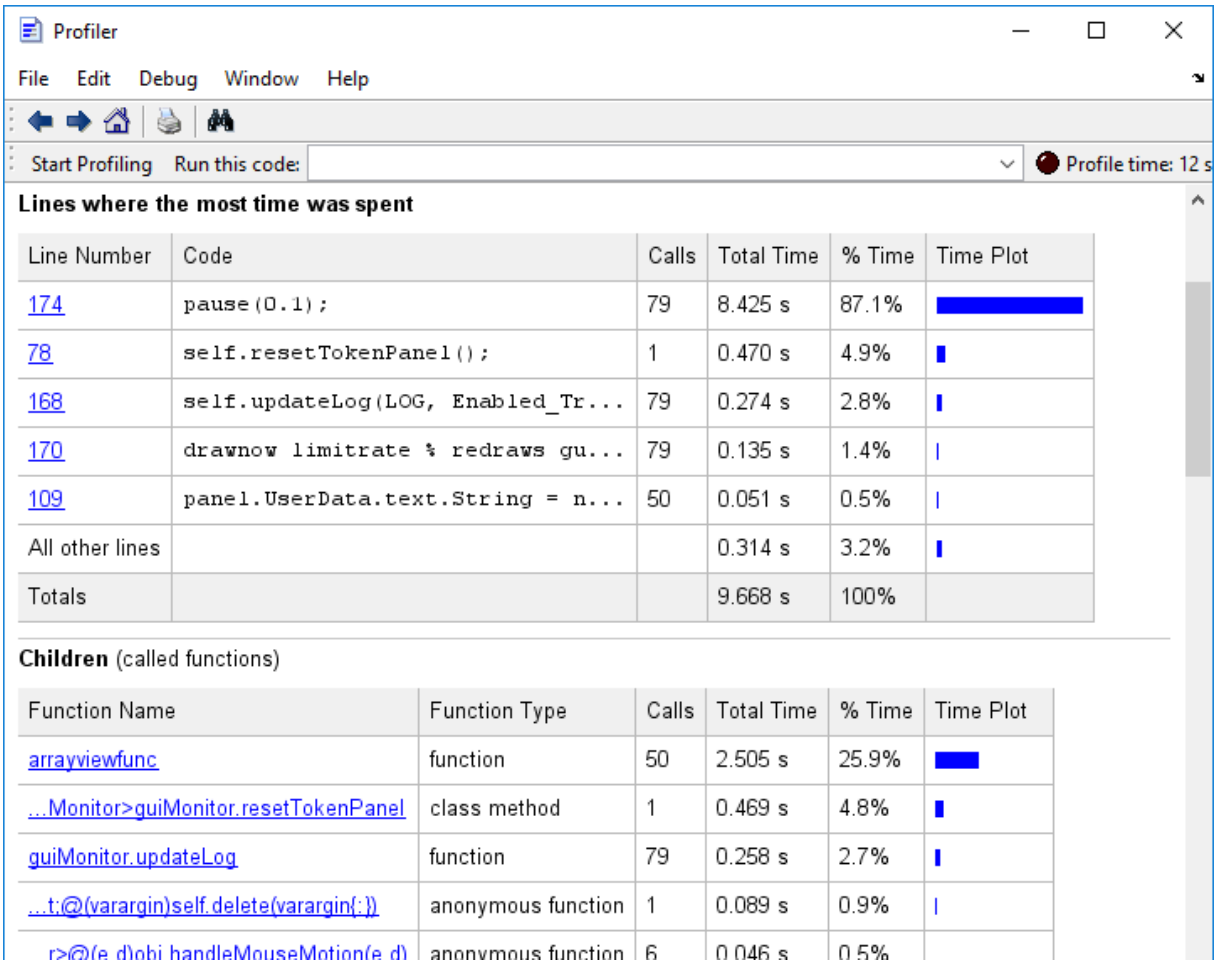


Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
main	1	11.756 s	0.128 s	
gpensim	1	10.392 s	0.158 s	
guiMonitor.showData	79	9.668 s	6.192 s	
arrayviewfunc	104	2.847 s	0.237 s	
arrayviewfunc>reportValuesCallback	52	2.346 s	0.007 s	
arrayviewfunc>reportValuesCallbackFcn	52	2.339 s	0.590 s	
arrayviewfunc>createCellArrayValue	50	1.615 s	0.234 s	
arrayviewfunc>reportValues	525	1.324 s	1.008 s	
...r.guiMonitor>guiMonitor.guiMonitor	1	0.744 s	0.019 s	
guiMonitor.constructor	1	0.725 s	0.311 s	
...Monitor>guiMonitor.resetTokenPanel	1	0.469 s	0.105 s	
firing_start	18	0.352 s	0.095 s	
guiMonitor.updateLog	79	0.258 s	0.061 s	
axis	12	0.250 s	0.082 s	

Total time is the time used by the function *including* its child functions.

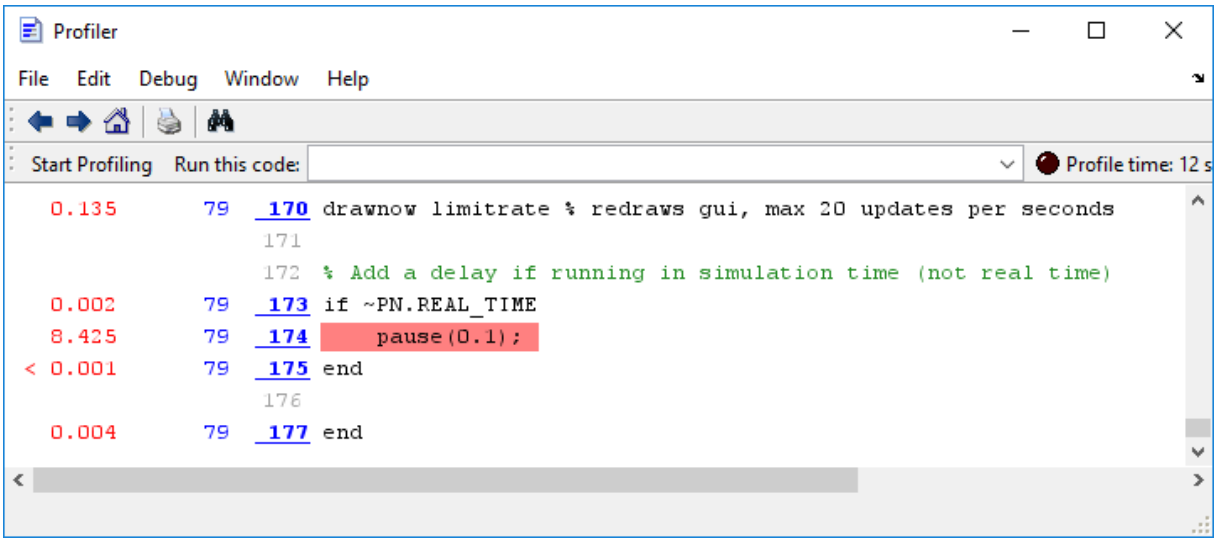
Self time is the time used by the function *excluding* its child functions.

Clicking the function name in the profiler will give more information about what lines use the most time. The following comes from guiMonitor.showData:



Most of the time is spent by the pause function. This is used to intentionally slow down the simulation to make it easier for the user to observe the simulation as it happens.

Scrolling down the page reveals the source code of the function including time spent, times called for each line. The most time-consuming lines are highlighted in red by the profiler:



6.3.2 Skipping unnecessary GUI property updates

The profiler reveals some problems when updating properties of GUI handles. Note how line number 91 and 94 are both called 531 times. The difference is that line 91 only use 4 ms line 94 uses 75 ms. This causes slowdowns of the GUI if the value gets assigned frequently.

Before: **75 ms**

```
  0.001      531     90        colormap = {[1 0 0] [1 1 0] [0 1 1] [0 1 0]};
  0.004      531     91        newColor = colormap{1 + 2 * firing{j} +  enabled{j}};
                     92
                     93        % only perform expensive operation when needed
  0.075      531     94        panel.BackgroundColor = newColor;
```

The good news is that this value often stays the same and only needs to get assigned if the value is different. An if statement is added to ensure that the value only gets assigned if needed.

After: 16 ms + 12 ms = **28 ms**

```
                     93        % only perform expensive operation when needed
  0.016      531     94        if ~isequal(panel.BackgroundColor, newColor)
  0.012      106     95            panel.BackgroundColor = newColor;
< 0.001      106     96        end
```

Reading gui property values also seems to consume some time. We can avoid reading the BackgroundColor at each loop if a copy of the newColor gets stored some time after the if statement.

After additional optimization attempt: 7 ms + 16 ms + 8 ms = **31 ms** (about the same)

```
                     93        % only perform expensive operation when needed
  0.007      531     94        if ~isequal(self.PreviousTransColor(j,:), newColor)
  0.016      106     95            panel.BackgroundColor = newColor;
< 0.001      106     96        end
                     97        % used in the next call to showData
  0.008      531     98        self.PreviousTransColor(j,:) = newColor;
```

The if statement is about twice as fast when reading from the variable instead of the GUI property directly. The extra matrix assignment needed to achieve this negates the advantage. And even if there was a small advantage the added complexity would decrease readability of the program which is also undesirable. Other GUI properties where the assignment often is rewritten with the same value will get an if condition added. For example the token values:

Before: 84 ms + 206 ms = **290 ms**

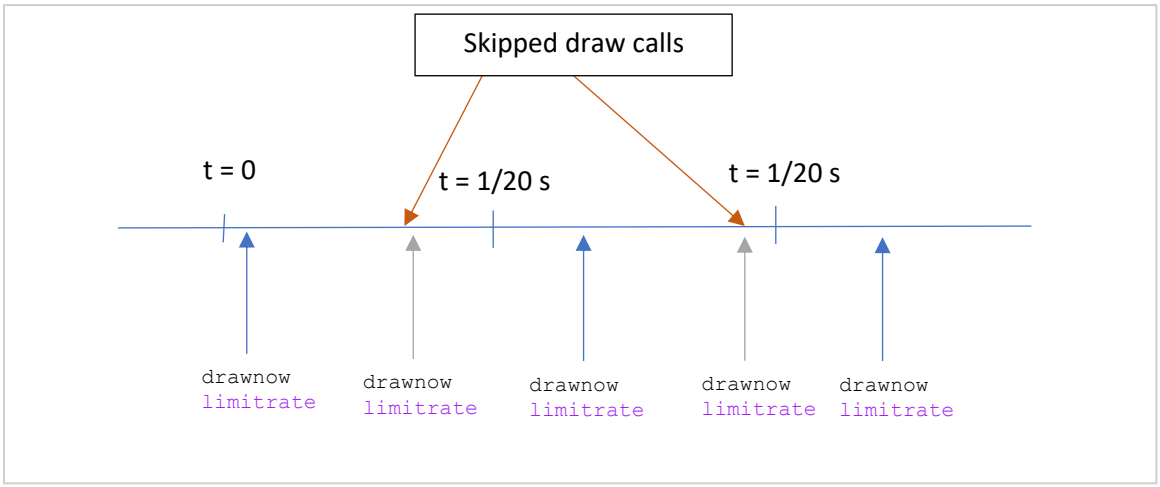
```
  0.084      531    109            panel.UserData.currentTokenCount = tokens{j};
  0.209      531    110            panel.UserData.text.String = num2str(tokens{j});
```

After: 28 ms + 19 ms + 77 ms = **124 ms**

```
  0.028      531    108        if panel.UserData.currentTokenCount ~= tokens{j}
  0.019      106    109            panel.UserData.currentTokenCount = tokens{j};
  0.077      106    110            panel.UserData.text.String = num2str(tokens{j});
< 0.001      106    111        end
```

6.3.3 Limit effective draw calls

After updating the state of the GUI the changes will only get reflected after calling drawnow (or during idle times if the program pauses). However just because it is possible to run the drawnow function as often as possible doesn't mean it is necessary to trigger a redraw every time. The limitrate option for drawnow tries to maintain a refreshrate of 20 gui updates per second. However it does not block until 1/20 s has passed but instead returns much quicker since it is not performing the actual work needed to redraw the GUI. This behaviour is good when the user wants to run the simulation at a high speed since it will not be intentionally slowed down to a 20 Hz update rate.



Time used by 1000 drawnow calls:

`drawnow limitrate` x 1000 = 1.0 s = 11% of total (8.7 s)

`drawnow` x 1000 = 8.3 s = 52% of total (15.9 s)

The program runs almost twice as fast using the limitrate option. The user is still able to slow down the program using a pause function that takes a value from a slider in the GUI.

6.3.4 Logging performance

When running simulations with around 500 or more states the simulation seem to slow down dramatically over time. As can be seen after running the profiler the `guiMonitor.addToLog` function is the culprit consuming half of the simulation time:

Profile Summary

Generated 08-May-2018 21:23:28 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
main	1	23.284 s	0.055 s	
gpensim	1	22.320 s	0.133 s	
guiMonitor.showData	500	20.530 s	6.061 s	
guiMonitor.updateLog	500	13.740 s	0.054 s	
...tor.guiMonitor>guiMonitor.addToLog	501	13.337 s	13.337 s	
firing_start	500	1.212 s	0.088 s	

time	Calls	line	
		311	function addToLog(self, string)
		312	% Append to GUI window
12.282	501	313	self.StateLog.String{end + 1} = string;
		314	
		315	% Scroll to the bottom by selecting the last line
0.001	501	316	if self.LogAutoScroll
1.047	501	317	self.StateLog.Value = length(self.StateLog.String);
< 0.001	501	318	end

Again it can be seen that adding to GUI elements take a significant time. Given that the problem gets worse over time might indicate that the time used to assign to this variable increases as the log grows. An acceptable solution would be to only keep a certain number of lines while logging. The full log can be written back once the simulation is paused or finished.

Performance after limiting to 50 lines at a time:

Function Name	Calls	**Total Time**	Self Time*	Total Time Plot (dark band = self time)
main	1	5.015 s	0.058 s	
gpensim	1	4.097 s	0.114 s	
guiMonitor.showData	500	2.466 s	1.088 s	
firing_start	500	1.107 s	0.082 s	
guiMonitor.updateLog	500	0.697 s	0.045 s	
prnss	1	0.553 s	0.056 s	

The guiMonitor.updateLog function went from **13.7** to **0.7** seconds. This is a major improvement when running long simulations. The complete log is stored in `guiMonitor.FullLogString` by the `guiMonitor.addToLog` function and is written to the log window when the simulation is paused or finished:

```
if SIM_COMPLETE
    ...
    self.StateLog.String = self.FullLogString;
    ...
end
```

6.4 Input parameters

Input parameters to the monitor can be specified in the simulation file, example:

```
g = guiMonitor(...
    'WantedDelay', 1, ...
    'ShownTokens', ...
    {'p1_floor_elevator', 'p2_floor_elevator', 'p3_floor_elevator'}, ...
    'ShownTrans', ...
    {'t1_exit', 't2_exit', 't3_exit'} ...
);
sim = gpensim(pni);
```

When running this example the monitor will add pauses to maintain a 1 second delay between simulation loops. The “ShownTokens” and “ShownTrans” parameters will cause the monitor to only show the selected places and transitions (assuming they exist). An error is thrown when a parameter has an incorrect name or value (see source code: guiMonitor.parseArguments).

6.4.1 WantedDelay

Default: 0 (The simulation + monitor will run as fast as possible)

Number of seconds (double) that controls the time added between simulation loops. If the simulation loop and guiMonitor uses less time than the WantedDelay a pause is added to maintain a consistent simulation rate. If the WantedDelay is too small compared to the time used in the simulation loop + guiMonitor then the tick rate will not be maintained (why "Wanted" is in the name).

6.4.2 ShownTokens and ShownTrans

Default: {} ("The empty set means that all tokens are added")

Sets of names (cell array) for places and transitions to be shown on the dashboard initially. If a name does not exist or is mispelled it is ignored without warning. If no names are specified all entities will be shown. The user can still show or hide tokens in the table view.

6.4.3 LogAutoScroll

Default: true

Logical value controlling the behaviour of the logs scrollbar. If disabled the logging will not move to the bottom by itself.

6.4.4 TokenPanelSize and TransPanelSize

Default:

```
TokenPanelSize = [15 4]
TransPanelSize = [15 2]
```

Two element vector specifiying width and height respectively of all places or transitions on the dashboard.

Character dimensions are used, explanation from MATLAB documentation [5]:

> *"These units are based on the default uicontrol font of the graphics root object:*
>
> *Character width = width of the letter x.*
>
> *Character height = distance between the baselines of two lines of text*
>
> *...."*

The width can be changed to accommodate for length of transition and token names names.

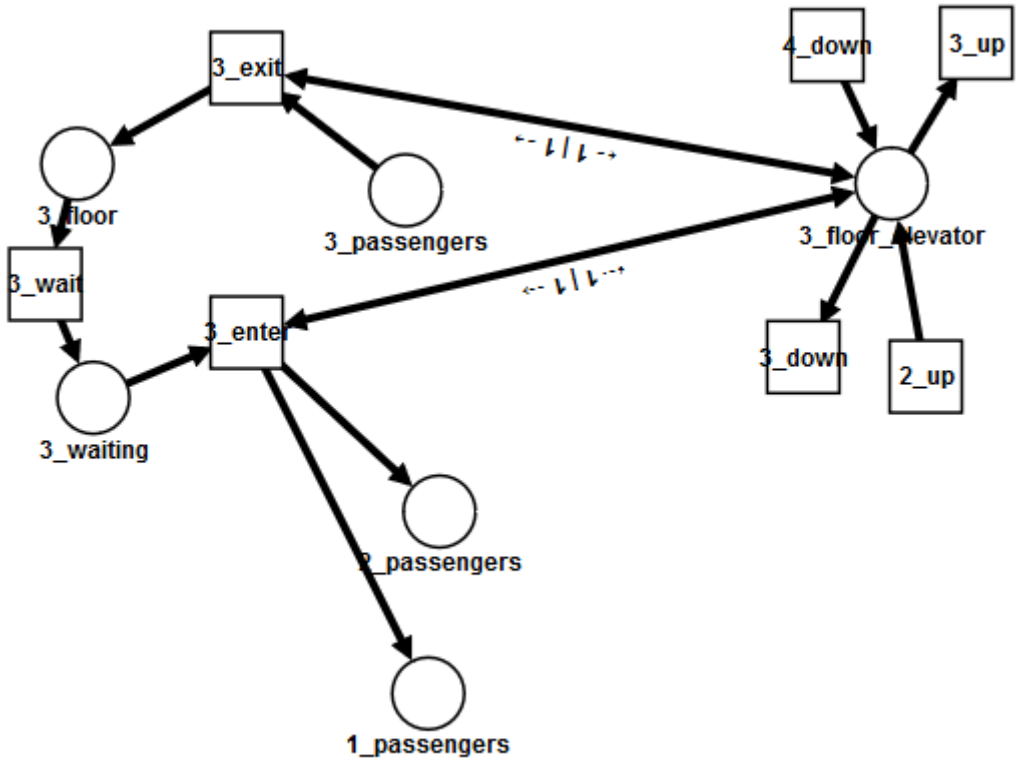
7 Example simulations

8 Example simulation: Elevator

To better demonstrate the features of the monitor an example simulation was created.

Note: This example is not finished. However it does something and allows adding extra places dynamically (extra floor) which makes it good for stress testing.

This example simulates an elevator. It will simply move in one direction only chaging direction when no passengers want to enter or exit a floor in that direction.



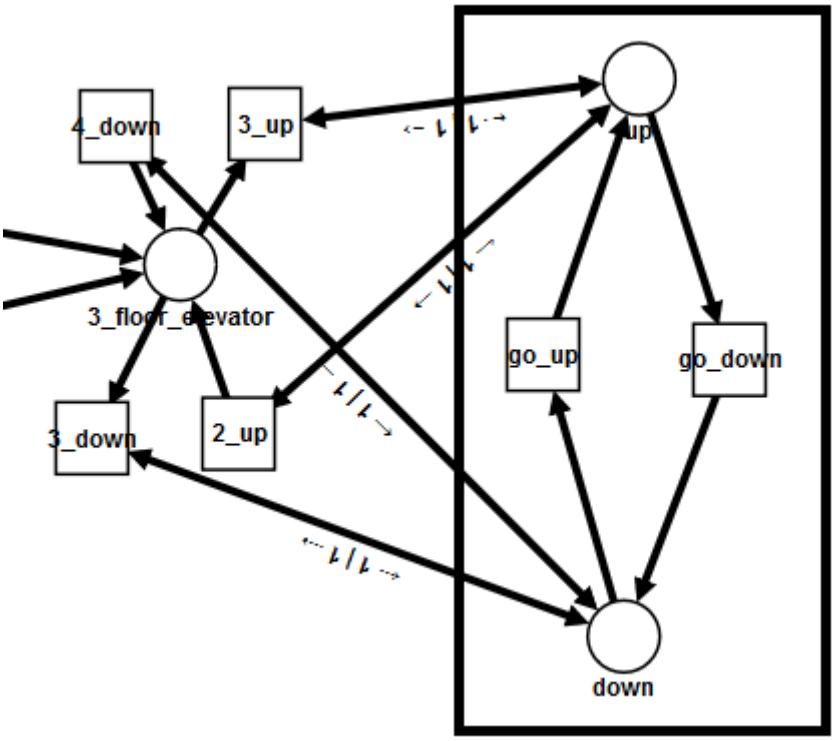
The petri net of the third floor has the same structure as other floors. The place “3_floor_elevator” indicates if the elevator is on the third floor. The transitions “3_up” and “3_down” moves the elevator up and down from the third floor. When “2_up” triggers the elevator has moved up to floor 3, and “4_down” triggers when moving from floor 4 down to 3.

There is a place for each floor indicating that passengers inside the elevator wish to move to that floor , “2_passengers” ... “n_passengers” where n is the total number of floors. Assuming that passengers cannot change their minds while in the elevator, exiting the elevator on floor 3 “3_exit” can only happen if passengers wish to move to the 3rd floor (“3_passengers” has tokens) and the elevator is on the 3rd floor (“3_floor_elevator” contains a token). When exiting the elevator, “3_floor_elevator” will be instantly put back due to a bidirectional arc (see figure). The passenger token will be moved to the 3rd floor outside the elevator indicated by the place “3_floor”. A passenger on the 3rd floor can start waiting for the elevator, the “3_wait” transition indicates that the passenger started waiting for the elevator by transfering it to place “3_waiting”. When a passenger waits we assume that the call elevator button has been pressed thus notifiying the elevator to stop at that floor. The transition “3_enter” lets waiting passengers enter the elevator. Entering only happens if the elevator is on the third floor and when there are passengers waiting (“3_waiting”). After

entering the elevator the passengers get transferred to “1_passengers”, “2_passengers”... depending on what floor they want to move to.

The elevator will move up “3_up” or down “3_down” depending on the current direction. If the current direction is up and there are no passengers wanting to go up (“4,5...n_passengers” is empty) and noone is waiting in the upper floors (“4,5...n_waiting” is empty) the elevator will change direction. Although this logic could be hard wired into the petri net, in this example transition pre-processors will be used to only allow travel in one direction. Hard wiring the petri net would have made it very cluttered with bi-directional arcs going to all “k_passengers” and “k_waiting” where k is higher than the current floor if going up “3_up” or lower if going down “3_down”. All this wiring would have to be added to each floor.

The alternative used in this example is creating a direction reversal module. There is one module needed per elevator and it has places “up” and “down”, only one of these can contain a token at a time. The transitions going up “1,2,...(n-1)_up” requires a token in the “up” place to fire, the same goes for the “down” place required for “2,3,...n_down” to fire. A bidirectional arc is added to all relevant transition to check for the requirement. The module has two transitions which can reverse the direction, one is going from down to up (“go_up”) and another in the opposite direction (“go_down”). These transitions have preprocessors checking what floor the elevator is on (“k_floor_elevator”). If noone wants to enter (“k_waiting”) or exit (“k_passengers”) the elevator in the current direction, the elevator will reverse its direction. The direction reversal module is connected as shown in the following sketch (only connections near floor 3 shown):



9 Discussion

9.1 User manual

9.1.1 Installation

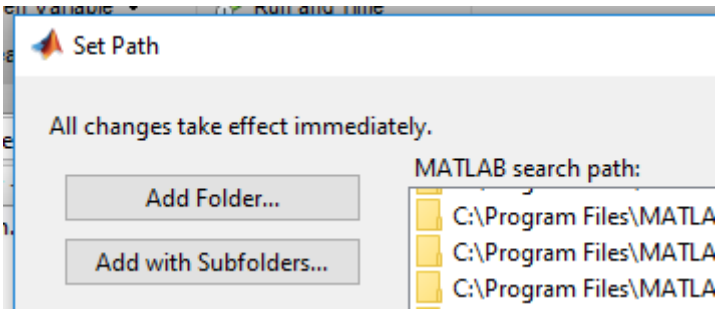
9.1.1.1 Install GPenSIM and the guiMonitor

Download GPenSIM (v9/10 supported) and the **guiMonitor**

Decompress each archive to a chosen location, ex.: C:\MATLAB\gpensim and C:\MATLAB\guimonitor

Add the folders and subfolders to MATLAB:

Goto: MATLAB -> Home tab -> **Set path -> Add with Subfolders...**



Select the GPenSIM folder and the guiMonitor folder.

If GPenSIM was succesfully installed:

```
>> gpensim
--------
GPenSIM version 10.0;   Lastupdate: October 2017

(C) Reggie.Davidrajuh@uis.no

http://www.davidrajuh.net/gpensim
--------
```

If the guiMonitor was successfully installed:

```
>> guiMonitor

ans =

  guiMonitor with properties:

                  Window: [1×1 Figure]
                    Tabs: [1×1 TabGroup]
...
```

A monitor window without any places or transitions will also appear.

9.1.1.2 Modify GPenSIM source code (add function call)

The GPenSIM main loop (gpensim.m truncated with modifications highlighted):

Edit the gpensim.m file by typing "`open gpensim`" in MATLAB, or navigate to GPenSIM\gpensim.m .

Add the highlighted text:

```
while ~(SIM_COMPLETE),
    ...

    % stop if ((queue is empty) OR (max loop) OR (max log size)) is reached
    SIM_COMPLETE = simulations_complete(Loop_Nr, global_info.MAX_LOOP);

    % ADDON: Real time monitor (may be removed if unused)
    if isfield(global_info, 'MONITOR_RUNNING') && global_info.MONITOR_RUNNING
        global_info.MONITOR_GUI.showData(LOG, Enabled_Trans_SET, Firing_Trans_SET);
    end; % ADDON: Real time monitor

end %while ~(SIM_COMPLETE)
```

Note: If this step was done incorrectly the monitor will appear but without any information about the simulation (as if just typing `guiMonitor` in the command line).

9.1.2 Getting started with the guiMonitor

Create an instance of the guiMonitor (see highlight) before the gpensim() function call.

Using the "MAX_LOOP" example from GPenSIM v10,

Source: GPenSIM_v10_Examples\example-15-OPTION-MAX_LOOP\maxloop_demo.m

```
% Example-15: MAX-LOOP demo

clear all; clc;
```

```
global global_info                % user data
global_info.MAX_LOOP = 15;

guiMonitor();

pns = pnstruct('maxloop_pdf'); % create petri net structure

dyn.m0 = {'p1',3, 'p2',4};
dyn.ft = {'t1',10};

pni = initialdynamics(pns, dyn);
sim = gpensim(pni); % perform simulation runs

plotp(sim, {'p1','p2','p3'}, 0, 2); % plot the results
```

If everything went according to plan the monitor window and a plot window will open. This simulation finishes in the blink of an eye so it can be a good idea to have a one second delay between each simulation loop. This is done by setting the 'WantedDelay' input parameters for the guiMonitor.

```
guiMonitor('WantedDelay', 1);
```

The simulation should take 15 seconds now (given 15 loops). It is also possible to control the simulation speed during simulation using a slider:

Note: The "step" button is only available once the simulation is paused. It will advance one simulation step each time it is pressed.

9.2 Originality

Even though the previous work has been useful as a point of reference, this tool has been redesigned from the ground up. The implementation of the monitor is original and only uses GPenSIM and the built-in GUI API from MATLAB. Source code from the previous work have not been reused.

9.3 Further work

This section includes some ideas to further improve the monitoring tool.

9.3.1 Simulation replay

The object returned from the gpensim function contains all information about the state of places and transitions. It might be useful to be able to store and replay this information in the monitor. This could be implemented using a new input parameter that took a GPenSIM simulation object and replayed this information.

9.3.2 Resizable user interface elements

A user might want to put more emphasis on certain elements of the user interface. Example: The user wants to monitor significantly more places than transitions. Currently the transition and place section allocate the same amount of space regardless of what is needed. If the section with transitions has available space it should be allocated to the places section if needed and the other way around.

9.3.3 Combined plots

The user might want to combine different plots with a shared x axis and often a shared y axis. A solution could be to include a selector inside every plot that lets the user add simulation elements into the same plot.

9.3.4 Log customization

The current log simply behaves identical to the “print state-space” (prnss) function from GPenSIM. A user may want to customize and filter what and how things are displayed in the log. A configuration button next to the log could open a dialog that presents what options the user have when it comes to modifying logging behaviour.

9.3.5 Log search

Scrolling through a huge log to find gets impractical as the size of the log increases. A search box should enable the user to find information based on a certain parameter. Perhaps a certain point in time, a certain number of state. The user might want to search through all cases where a given transition has fired. A selector can reveal possible parameters to search for and the search box should contain what the user is looking for.

9.3.6 Organizing transitions and places

Currently simulation elements (transitions and places) on the dashboard have a fixed order. Let the user easily organize what order transitions and places are shown in. If drag and drop was implemented for elements the user would have an easy way of logically arranging the interface elements. The user may additionally want to keep transitions and places together in groups.

9.3.7 Feature creep

While there are a lot of ideas to work with, it is important to ensure that less important features are not overwhelming to the first-time user. This can be achieved by implementing a “simple mode” with only the essentials or letting the user toggle features in the settings menu.

References

[1] R. Davidrajuh, “GPenSIM: A tool for modeling, simulation, and performance evaluation of discrete-event systems,” 2018. [Online]. Available: http://www.davidrajuh.net/gpensim/.

[2] “MATLAB Documentation,” [Online]. Available: http://www.mathworks.com/help/matlab/index.html. [Accessed 18 5 2018].

[3] N. G. Tulu, “Monitor for displaying the status of Real-Time simulation.,” 27 06 2014. [Online]. Available: http://hdl.handle.net/11250/221465. [Accessed 20 05 2018].

[4] The MathWorks, Inc, “MATLAB documentation: UI Figure Properties,” [Online]. Available: https://se.mathworks.com/help/matlab/ref/matlab.ui.figureappd-properties.html. [Accessed 27 5 2018].

[5] MathWorks, “Ways to build MATLAB GUI's,” 2017. [Online]. Available: https://se.mathworks.com/help/releases/R2017b/matlab/creating_guis/ways-to-build-matlab-guis.html#bu7g5rv-1.

[6] MathWorks, “MATLAB documentation: uicontrol properties,” 2017. [Online]. Available: https://se.mathworks.com/help/releases/R2017b/matlab/ref/matlab.ui.control.uicontrol-properties.html.

[7] L. Shure, “MATLAB: Memory Management for Functions and Variables,” 10 May 2006. [Online]. Available: https://blogs.mathworks.com/loren/2006/05/10/memory-management-for-functions-and-variables/. [Accessed 10 6 2018].

[8] MathWorks, “MATLAB,” 2018. [Online]. Available: https://www.mathworks.com/products/matlab.html.

10 Appendix

10.1 Source code: guiMonitor (stored inside @guiMonitor folder)

10.1.1 guiMonitor.m

```
classdef guiMonitor < handle
    %GUIMONITOR Graphical user interface and API for showing state of
    %gpensim in real time.
    %   TODO: Detailed explanation goes here
    properties (Access = public)
        SimulationPaused = false
    end

    properties (Access = private)
                Window % main window
        Firing_Trans_SET
        Enabled_Trans_SET
        TokenPanelSize = [15 4]
        MainTransPanel
        TransPanels
        TransPanelSize = [15 2]
        %       [R G B]    None   Enabled Firing   Both
        TransColorMap = {[1 0 0] [1 1 0] [0 1 1] [0 1 0]}
        ShowLegend = true
        GraphWidth = 10
        ShownTokens = {}
        ShownTrans = {}
        PreviousTransColor
        ShownTableNeedsUpdate
        IsTimedPN = true
        WantedDelay = 0

        LogAutoScroll = true

        TokenTable
        TransTable
        MainTokenPanel
        TokenPanels
        GraphedPlaces
        GraphedTrans
```

```
        WantedDelaySlider
        WantedDelayText
        LastTime = 0
        SecondsInDay = 3600 * 24
        Strings
        PlayPauseButton
        StepButton
        FullLogString = {}
        StateLog
        LogPanel
        FirstLoop
        Tabs
        DashboardTab
        TablesTab
        WaitForETSIndex
        LogFileId
        NextLogIndex

        LOG
    end

    methods
        function self = guiMonitor(varargin)
            self.parseArguments(varargin{:});
            self.constructor();
        end

        % Color lookup from TransColorMap
        function transColor = getTransColor(self, firing, enabled)
            transColor = self.TransColorMap{1 + 2 * firing + enabled};
        end

        string = getStrings(self)

        % Called after the axes moves in x direction (axes.XLim changes)
        function axisChangedUpdateDateticks(~, ~, eventData)
            axes_handle = eventData.AffectedObject;
            datetick(axes_handle, 'x', 'HH:MM:SS', 'keeplimits');
        end

        function playPauseButtonPressed(self, button, ~, ~)
            % hObject    handle to togglebutton1 (see GCBO)
            % eventdata  reserved - to be defined in a future version of
MATLAB
            % handles    structure with handles and user data (see GUIDATA)
            self.SimulationPaused = button.Value == button.Max;
            if self.SimulationPaused
                % allow single stepping the simulation while paused
                %self.StepButton.Enable = 'on';
                % clicking the button will resume the simulation
                button.String = self.Strings.RESUME_BUTTON;
            else
                % single stepping has no effect when the simulation
                % is running. Disable the button to avoid confusion (gray).
                self.StepButton.Enable = 'off';
                % resume the guiMonitor.showData function
                uiresume(self.Window);
                % clicking the button will pause the simulation
                button.String = self.Strings.PAUSE_BUTTON;
            end
```

```
end

function stepButtonPressed(self, ~, ~)
    uiresume(self.Window);
    % the user cannot click the button again until stepping is
    % complete
    self.StepButton.Enable = 'off';
end

resizePanels(~, container, panels, panelSize)

function resizePlacePanels(self, ~, ~)
    self.resizePanels(self.MainTokenPanel, self.TokenPanels, ...
        self.TokenPanelSize);
end

function resizeTransPanels(self, ~, ~)
    self.resizePanels(self.MainTransPanel, self.TransPanels, ...
        self.TransPanelSize);
end

function sliderChanged(self, ~, ~)
    self.WantedDelay = self.WantedDelaySlider.Value;
    self.WantedDelayText.String = ...
        ['Tick delay:' newline num2str(self.WantedDelay)];
end

parseArguments(varargin)

function tokenTableEdited(self, ~, ~)
    self.ShownTableNeedsUpdate = 1;
end

function transTableEdited(self, ~, ~)
    self.ShownTableNeedsUpdate = 1;
end

function [x, y] = plotHistoryOfTokenCount(self, place_id, ...
        No_of_places)

    if self.IsTimedPN
        column_end_times = No_of_places + 6;
        x = self.LOG(:, column_end_times);
        y = self.LOG(:, place_id);
    else
        column_current_state = No_of_places + 2;
        [State, LOG_i] = unique(self.LOG(:, column_current_state));
        % skip zero state (always at beginning of LOG)
        State = State(2:end);
        LOG_i = LOG_i(2:end);

        x = State;
        y = self.LOG(LOG_i, place_id);
    end
end

function [x, y] = plotHistoryOfTransition(self, ...
        trans_id, No_of_places)
```

```
            if self.IsTimedPN
                x = self.Firing_Trans_SET(:, 1);
                ETS_i = 1:size(self.Firing_Trans_SET, 1);
            else
                % TODO better/more elegant solution?
                TS_col = No_of_places + 4;
                State_col = No_of_places + 2;
                [ETS_i, LOG_i] = unique(self.LOG(:, TS_col));

                % remove the unwanted ETS_i == 0 element
                % gpensim always creates the first row with state == 0
                LOG_i = LOG_i(2:end);
                ETS_i = ETS_i(2:end);
                x = self.LOG(LOG_i, State_col); % x = state
            end

            y = 2 * self.Firing_Trans_SET(ETS_i, 1 + trans_id) + ...
                self.Enabled_Trans_SET(ETS_i, 1 + trans_id);


        end

        % open a new window for live plotting of transitions
        initLiveTransPlot(self, panel, ~, trans_id)

        % open a new window for live plotting of tokens in places
        initLiveTokenPlot(self, panel, ~, token_id)

        % Show the selected transitions
        % Called when a token is added or removed from the transition panel
        function resetTransitionPanel(self)

            transitionNames = (self.TransTable.Data(:, 1));
            delete(self.MainTransPanel.Children);
            self.TransPanels = {};

            for i = 1:length(transitionNames)
                if self.ShownTrans(i)

                    panel = uipanel(self.MainTransPanel);
                    panel.Title = transitionNames(i);
                    panel.BorderType = 'none';
                    panel.ButtonDownFcn = {@self.initLiveTransPlot, i};
                    self.TransPanels{end + 1} = panel;

                end
            end

            if self.ShowLegend
                % Show transition states and its corresponding colors as a
                % legend
                legendEntries = [ ...
                    struct('Color', self.getTransColor(0, 0), 'Name',
'None') ...
                    struct('Color', self.getTransColor(0, 1), 'Name', 'Ena-
bled') ...
                    struct('Color', self.getTransColor(1, 0), 'Name', 'Fir-
ing') ...
```

```
                    struct('Color', self.getTransColor(1, 1), 'Name',
'Both') ...
                ];

                % Dimensions in character units
                BottomPadding = 0.5; SidePadding = 1; Width = 10; Height =
1.1;

                for i = 1:length(legendEntries)
                    entry = legendEntries(i);
                    panel = uipanel(self.MainTransPanel, 'Title', en-
try.Name);
                    panel.BackgroundColor = entry.Color;
                    panel.BorderType = 'none';

                    panel.Units = 'characters';
                    x = (i - 1) * (Width + SidePadding);
                    panel.Position = [x BottomPadding Width Height];
                end
            end

            self.resizeTransPanels();
        end

        % Show the selected tokens
        % Called when a token is added or removed from the token panel
        function resetTokenPanel(self)

            tokenNames = (self.TokenTable.Data(:, 1));
            % remove existing panels
            delete(self.MainTokenPanel.Children);
            self.TokenPanels = {};

            % dimensions
            w = 1 / 3; h = 1 / 3;
            x = 0.000; y = 1 - h;

            for i = 1:length(tokenNames)
                if self.ShownTokens(i)
                    if x >= 1
                        x = 0; y = y - h;
                    end

                    % Draw a panel with a circle inside
                    panel = uipanel(self.MainTokenPanel);
                    panel.Title = tokenNames(i);
                    panel.Position = [x y w h];
                    axes(panel, 'Position', [0 0 1 1], 'Visible', 'off');

                    % draw a circle (using a utf8 text char(9711))
                    text(panel.Children, ...
                        0.5, 0.5, self.Strings.CIRCLE_ICON, 'HitTest',
'off',...
                        'HorizontalAlignment', 'center', ...
                        'VerticalAlignment', 'middle', ...
                        'FontSize', 20);

                    panel.UserData.text = text(panel.Children, ...
                        0.5, 0.5, '', 'HitTest', 'off',...
```

```
                        'HorizontalAlignment', 'center',...
                        'VerticalAlignment', 'middle');
                    panel.ButtonDownFcn = {@self.initLiveTokenPlot, i};
                    self.TokenPanels{end + 1} = panel;

                    % this number will be used to check if updating
                    % the token count is necessary without having to use
                    % relatively expensive str/number conversions
                    panel.UserData.currentTokenCount = -1;


                    x = x + w;
                end
            end
            self.resizePlacePanels();
            %self.MainTokenPanel.Children = self.TokenPanels;
            %self.MainTokenPanel.Position = [.05 .500 .425 .45];
        end

        % showData is called from the end of the gpensim mainloop to update
        % tokens+transition states.
        % This function should be called at each simulation step
        % This should be added to gpensim.m at the end of the main loop

% while ~(SIM_COMPLETE),
%     ...
%
%     % ADDON: Real time monitor (may be removed if unused)
%     if isfield(global_info, 'MONITOR_RUNNING') && ...
%       global_info.MONITOR_RUNNING
%         global_info.MONITOR_GUI.showData( ...
%           LOG, Enabled_Trans_SET, Firing_Trans_SET);
%     end; % ADDON: Real time monitor
%
% end

        showData(self, LOG, Enabled_Trans_SET, Firing_Trans_SET, ...
            SIM_COMPLETE)

        [string] = log_prnss_state(self, fired_trans, finishing_time, ...
                    current_markings,state, virtual_tokens)

        updateLog(self, LOG, Enabled_Trans_SET, Firing_Trans_SET)

        % add to log, no additional newlines added
        function addToLog(self, string)
            % Append to GUI window (side effect: sets listboxtop = 1)
            lines = split(string, newline)';
            self.StateLog.String = [...
                self.StateLog.String{max(end - 50, 1):end} lines];

            self.FullLogString = [self.FullLogString lines];

            % Scroll to the bottom by selecting the last line
            if self.LogAutoScroll
                % scroll and select bottom value
                self.StateLog.Value = length(self.StateLog.String);
            end
```

```
        end

        % Called when the user closes the window
        function delete(self, ~, ~)
            global global_info PN;

            % close transition graphs
            for i = 1:length(self.GraphedTrans)
                graph = self.GraphedTrans{i};
                if isvalid(graph)
                    delete(graph.Parent.Parent);
                end
            end

            % close token graphs
            for i = 1:length(self.GraphedPlaces)
                graph = self.GraphedPlaces{i};
                if isvalid(graph)
                    delete(graph.Parent.Parent);
                end
            end

            % close main window
            delete(self.Window);

            % stop monitoring unless there is another active instance of
            % the gui
            if global_info.MONITOR_GUI == self
                global_info.MONITOR_GUI = 0;
                global_info.MONITOR_RUNNING = 0;

                % TODO REMOVE (stops simulation after window closes)
                PN.STOP_TIME = 0;
            end

        end
    end
end
```

10.1.2 constructor.m

```
function self = constructor(self)

    self.Strings = self.getStrings();

    self.Window = figure('Menubar', 'None', 'NumberTitle','off',...
        'Name', self.Strings.WINDOW_TITLE,...
        'CloseRequestFcn', @self.delete);


    self.Tabs = uitabgroup( self.Window, 'Position', [0 0 1 1]);
    self.DashboardTab = uitab(self.Tabs,...
        'Title', self.Strings.DASHBOARD_TAB_TITLE);
    self.TablesTab = uitab(self.Tabs,...
        'Title', self.Strings.TABLE_TAB_TITLE);

    self.MainTransPanel = uipanel(self.DashboardTab, ...
        'Title', self.Strings.TRANS_PANEL_TITLE, ...
        'SizeChangedFcn', @self.resizeTransPanels, ...
```

```
    'Visible', 'off');
self.TransPanels = {};

self.MainTransPanel.Visible = 'on';

self.MainTokenPanel = uipanel(self.DashboardTab, ...
    'Title', self.Strings.PLACE_PANEL_TITLE, ...
    'SizeChangedFcn', @self.resizePlacePanels, ...
    'Visible', 'off');
self.TokenPanels = {};
self.MainTokenPanel.Visible = 'on';

self.LogPanel = uipanel(self.DashboardTab, ...
    'Title', self.Strings.LOG_PANEL_TITLE, 'Units', 'normalized');
self.StateLog = uicontrol(self.LogPanel, 'Style','listbox', ...
    'Max', 2, ... % enables multiline
    'Units', 'normalized', 'String',{''});


% the index of the first unprocessed log entry (start = 1)
self.NextLogIndex = 2; % 2 for testing

% when this > 0, wait for transition set to catch up
% Enabled_Trans_SET and Firing_Trans_SET
self.WaitForETSIndex = 0;

% Time when previous simulation tick is finished, used to calculate how
% much additional time to wait if the user wants a fixed simulation
% tick rate.
self.LastTime = clock();
% Contains a table of tokens
self.TokenTable = uitable(self.TablesTab);
self.TokenTable.Units = 'normalized';
self.TokenTable.ColumnFormat = {'char' 'logical'};
self.TokenTable.ColumnName = {...
    self.Strings.COLUMN_PLACE, ...
    self.Strings.COLUMN_PLACE_SHOWN ...
};
self.TokenTable.ColumnEditable = [false true];
self.TokenTable.ColumnWidth = {110 33};
self.TokenTable.CellEditCallback = @self.tokenTableEdited;
self.TokenTable.RowName = {};

% Contains a table of transitions, the user can select which
% transitions show up in the main panel
self.TransTable = uitable(self.TablesTab);
self.TransTable.Units = 'normalized';
self.TransTable.ColumnName = {'Transition', 'Show'};
self.TransTable.ColumnEditable = [false true];
self.TransTable.ColumnFormat = {'char' 'logical'};
self.TransTable.ColumnWidth = {120 33};
self.TransTable.CellEditCallback = @self.transTableEdited;
self.TransTable.RowName = {};


self.PlayPauseButton = uicontrol(self.DashboardTab, ...
    'Style', 'togglebutton', 'String', self.Strings.PAUSE_BUTTON,...
    'Units', 'normalized', ...
    'Callback', @self.playPauseButtonPressed);
```

```
    self.StepButton = uicontrol(self.DashboardTab, ...
        'Style', 'pushbutton', 'String', self.Strings.STEP_BUTTON, ...
        'Units', 'normalized', 'Enable', 'off', ...
        'Callback', @self.stepButtonPressed);

    self.WantedDelaySlider = uicontrol(self.DashboardTab, ...
        'Style', 'slider', 'Min', 0, 'Max', 5, ...
        'Value', self.WantedDelay, 'SliderStep', [.1 .5], ...
        'Units', 'normalized');

    addlistener(self.WantedDelaySlider, ...
        'Value', 'PostSet', @self.sliderChanged);

    self.WantedDelayText = uicontrol(self.DashboardTab, ...
        'Style', 'text', 'Units', 'normalized', 'Max', 2, ...
        'String', [self.Strings.TICK_DELAY num2str(self.WantedDelay)]);

    % Indicates that a user changed which transitions/tokens are showns
    % When this is set to 1 the showData function will reflect this change
    self.ShownTableNeedsUpdate = 1;

    % Indicates the first iteration of the loop. This is used in showData.
    self.FirstLoop = 1;

    % GUI component positioning
    % TAB: dashboard
    self.MainTransPanel.Position = [.02 .500 .47 .48]; % upper left
    self.MainTokenPanel.Position = [.51 .500 .47 .48]; % upper right

    self.PlayPauseButton.Position = [.05 .405 .10 .07];
    self.StepButton.Position = [.17 .405 .10 .07];
    self.WantedDelaySlider.Position = [.32 .405 .30 .07];
    self.WantedDelayText.Position = [.65 .405 .10 .07];
    self.StateLog.Position = [0 0 1 1];

    self.LogPanel.Position = [.02 .02 .96 .37];

    % TAB: tables
    self.TokenTable.Position = [.525 .1 .425 .825]; % lower right
    self.TransTable.Position = [.05 .1 .425 .825]; % lower left

    drawnow;

    global global_info;

    % indicates that the updateMonitor() function should run in the main
loop
    % this value can be set to zero to pause monitoring (window will not be
    % closed) and setting it to 1 again will continue monitoring
    global_info.MONITOR_RUNNING = 1;

    % start the GUI, information will be shown once updateMonitor
    % is called from gpensim
    global_info.MONITOR_GUI = self;
end
```

10.1.3 getStrings.m

```
function [string] = getStrings(~)

feature_locale = feature('locale');

RESUME_ICON = '>';
PAUSE_ICON = '||';
STEP_ICON = '>>|';
CIRCLE_ICON = char(9711);

% UTF-8 media control icons: https://en.wikipedia.org/wiki/Media_controls
if strcmpi(feature_locale.encoding, 'utf-8')
    RESUME_ICON = char(11208);
    PAUSE_ICON = char(9208);
    STEP_ICON = char(9197);
end

string = struct(...
    'WINDOW_TITLE', 'GPenSIM real time monitor',...
    'DASHBOARD_TAB_TITLE', 'Dashboard', ...
    'TABLE_TAB_TITLE', 'Configuration', ...
    'TRANS_PANEL_TITLE', 'Transitions', ...
    'COLUMN_TRANS', 'Transition',...
    'COLUMN_ENABLED', 'Enabled', ...
    'COLUMN_FIRING', 'Firing', ...
    'COLUMN_SHOW', 'Show', ...
    'PLACE_PANEL_TITLE', 'Token count', ...
    'COLUMN_PLACE', 'Place',...
    'COLUMN_TOKEN_COUNT', 'Count',...
    'COLUMN_PLACE_SHOWN', 'Show',...
    'LOG_PANEL_TITLE', 'Log', ...
    'CIRCLE_ICON', CIRCLE_ICON, ...
    'PAUSE_BUTTON', [PAUSE_ICON ' Pause'], ...%char(9208) => pause icon
    'STEP_BUTTON', [STEP_ICON ' Step'], ...%char(9197) => next track icon
    'TICK_DELAY', 'Tick delay: ', ...
    'RESUME_BUTTON', [RESUME_ICON ' Resume']... % char(11208) => play icon
);

end
```

10.1.4 initLiveTokenPlot.m

```
function initLiveTokenPlot(self, panel, ~, place_id)
% initLiveTokenPlot: opens a new window for live plotting of token count in
% the selected place
global PN;

% show figure instead if it already exists
for animline_handle_cell = self.GraphedPlaces
    animline_handle = animline_handle_cell{1}; % TODO fix

    if isvalid(animline_handle) && ...
        isequal(place_id, animline_handle.UserData.place_id)

        figure_handle = animline_handle.Parent.Parent;
        figure(figure_handle);
        return;
    end
end
```

```
% same structure as initLiveTransPlot(...)
figure('Name', cell2mat(panel.Title));
title(['Place: ' cell2mat(panel.Title)], ...
    'Interpreter', 'none'); % prevent underscore to subscript conversion

% An animatedline plot is optimized for real time updates
% the axis and plot is modified at regular intervals
% changes are shown when drawnow is called. The place_id
% is stored in the UserData property for use in showData.
h = animatedline('UserData', struct('place_id', place_id));
ax = h.Parent;
ylabel('Tokens');


if self.IsTimedPN
    xlabel('Time');
    xlim = [PN.current_time - self.GraphWidth, PN.current_time];
else
    xlabel('State');
    xlim = [PN.State - self.GraphWidth, PN.State];
end

if PN.REAL_TIME
    xlim = xlim / self.SecondsInDay;
    % Whenever the axis changes datetick must be updated on the axes
    addlistener(ax, 'XLim', 'PostSet', @self.axisChangedUpdateDateticks);
end
% If end of simulation: PN.LOG exists

% plot all token times from PN.LOG
[x, y] = self.plotHistoryOfTokenCount( ...
    place_id, PN.No_of_places);
ax.YLim = [-0.1, max([y; 10.1])];
if PN.REAL_TIME
    x = x / self.SecondsInDay;
end
addpoints(h, x, y);
ax.XLim = xlim;
self.GraphedPlaces{end + 1} = h;

end
```

10.1.5 initLiveTransPlot.m

```
function initLiveTransPlot(self, panel, ~, trans_id)
% initLiveTransPlot: opens a new window for live plotting of transitions
global PN;

% show figure instead if it already exists
for animline_handle_cell = self.GraphedTrans
    animline_handle = animline_handle_cell{1}; % TODO fix

    if isvalid(animline_handle) && ...
        isequal(trans_id, animline_handle.UserData.trans_id)

        figure_handle = animline_handle.Parent.Parent;
        figure(figure_handle);
        return;
```

```
    end
end

% Draw the plot in a new window
figure('Name', cell2mat(panel.Title));

title(['Transition: ' cell2mat(panel.Title)], ...
    'Interpreter', 'none'); % prevent underscore to subscript conversion

% An animatedline plot is optimized for real time updates
% the axis and plot is modified at regular intervals
% changes are shown when drawnow is called. The transition id
% is stored in the UserData property for use in showData.
h = animatedline('UserData', struct('trans_id', trans_id));

%h.Marker = 'o';
% Setup axis and ticks
ax = h.Parent;
ax.YLim = [-0.1, 3.1];
% panning in the y direction is unnecessary since y is bounded
pan xon;
if self.IsTimedPN
    xlim = [PN.current_time - self.GraphWidth, ...
        PN.current_time];
else
    xlim = [PN.State - self.GraphWidth, PN.State];
end
if PN.REAL_TIME
    xlim = xlim / self.SecondsInDay;
    % Whenever the axis changes datetick needs to be updated on the axes
    addlistener(ax, 'XLim', 'PostSet', @self.axisChangedUpdateDateticks);
end
yticks(ax, 0:3);
yticklabels(ax, {'inactive', 'enabled', 'firing',...
    'enabled+firing'});



if self.IsTimedPN
    xlabel('Time');
else
    xlabel('State');
end

% if the simulation is finished PN.Firing_Trans_Set is made
% available in that case PN.*_Trans_Set will be available
[x,y] = self.plotHistoryOfTransition(...
    trans_id, PN.No_of_places);

if PN.REAL_TIME
    x = x / self.SecondsInDay;
end
addpoints(h, x, y);

ax.XLim = xlim;
% The showdata function will update all animatedline handles
% inside this array with its respective transition.
self.GraphedTrans{end + 1} = h;
end
```

10.1.6 log_prnss_ET_FT.m

```
function [string] = log_prnss_ET_FT(~, Enabled_Trans_SET, Firing_Trans_SET,
ETS_index)

% function call inlined and adapted:
% print_statespace_enabled_and_firing_trans(ETS_index+1)

% The event time is the same for Firing_Trans_SET
event_time = Enabled_Trans_SET(ETS_index, 1);

enabled_trans = Enabled_Trans_SET(ETS_index, 2:end);
firing_trans  = Firing_Trans_SET(ETS_index, 2:end);

%inlined:
%prnss_enabled_trans(enabled_trans, event_time); %
global PN;
if (PN.HH_MM_SS),
    string = ['At time: ', string_HH_MM_SS(event_time),...
        ',  Enabled transitions are: '];
else
    string = ['At time: ', num2str(event_time), ...
        ',  Enabled transitions are: '];
end;

% row - 2: Enabled Transitions
et_index = find(enabled_trans);

for j = 1:length(et_index)
    etn = et_index(j);
    enabled_event_name = PN.global_transitions(etn).name;
    string = [string, '    ',enabled_event_name];
end;

string = [string newline];
%END of prnss_enabled_trans

%inlined:
%prnss_firing_trans(firing_trans, event_time);    %

if (PN.HH_MM_SS),
    string = [string 'At time: ', string_HH_MM_SS(event_time), ...
        ',  Firing transitions are:  '];
else
    string = [string 'At time: ', num2str(event_time), ...
        ',  Firing transitions are:  '];
end;

ft_index = find(firing_trans);

for k = 1:length(ft_index),
    ftn = ft_index(k);
    firing_event_name = PN.global_transitions(ftn).name;
    string = [string, '    ', firing_event_name];
end;
string = ([string newline]);

end
```

10.1.7 log_prnss_state.m

```
function [string] = log_prnss_state(~, fired_trans, finishing_time, ...
                    current_markings,state, virtual_tokens)
global PN;
if (PN.HH_MM_SS)
    string = ['**    Time: ',...
        string_HH_MM_SS(finishing_time), '   **' newline];
else
    string = ['**    Time: ',...
        num2str(finishing_time), '   **' newline];
end

string = ([string 'State: ', num2str(state) newline]);
string = ([string 'Fired Transition: ', ...
    PN.global_transitions(fired_trans).name newline]);
string = ([string 'Current State: ', ...
    markings_string(current_markings) newline]);
string = ([string 'Virtual tokens: ',  ...
    markings_string(virtual_tokens) newline]);
% END of inlined prnsss_state

string = [string ' ' newline];
string = [string 'Right after new state-',...
    int2str(state), ' ....', newline];

end
```

10.1.8 parseArguments.m

```
function parseArguments(self, varargin)
%PARSEARGUMENTS Parse input arguments and apply them to the guiMonitor
%   Detailed explanation goes here

% TODO consider using inputparser
% https://se.mathworks.com/help/matlab/ref/inputparser.addparameter.html

if mod(length(varargin), 2)
    error('guiMonitor: Odd number of arguments');
end

for i = 1:2:length(varargin)
    property = varargin{i};
    value = varargin{i + 1};

    switch property

        case {'WantedDelay', 'GraphWidth'}
            if ~isreal(value)
                error(['guiMonitor: ''' property ''' must be a number']);
            end
            self.(property) = value;

        case {'ShownTokens', 'ShownTrans'}
            if ~iscell(value)
```

```
                error(['guiMonitor: ''' property ''' must be a cell ar-
ray']);
            end
            self.(property) = value;
            % self.ShownTokens/Trans will be replaced with a logical vector
            % where ones indicate that the place/transition with that index
            % should be shown: {'p1', 'p3', 'p6'} => [1 0 1 0 0 1]
            % This will take place in the first call to guiMonitor.showData

        case {'LogAutoScroll', 'ShowLegend'}
            if ~islogical(value)
                error(['guiMonitor: ''' property ''' must be a logical']);
            end
            self.(property) = value;

        case {'TokenPanelSize', 'TransPanelSize'}
            if ~isreal(value) || ~isequal(size(value), [1 2])
                error(['guiMonitor: ''' property ''' must be a logical']);
            end
            self.(property) = value;

        otherwise
            error(['guiMonitor: property:' property ' does not exist'])
    end
end
```

10.1.9 resizePanels

```
function resizePanels(~, container, panels, panelSize)

% get container panel dimensions in character units by
% temporarily changing the units. This will maintain the
% position while showing in the correct units.
container.Units = 'characters';
container_width = container.InnerPosition(3);
container_height = container.InnerPosition(4);
container.Units = 'normalized';

% dimensions in character dimensions, 19 was found to
% contain 10 letters of 'M'. GPenSIM truncates all names to 10
% characters before simulation.
w = panelSize(1); h = panelSize(2);
x = 0.00; y = container_height - h;

xmargin = 0.2; ymargin = 0.1;
% adding this to the position vector will shrink the panel
% depending on the margin sizes
transform = [xmargin, ymargin, -2 * xmargin, -2 * ymargin];

for i = 1:length(panels)

    if (x + w) >= container_width
        x = 0; y = y - h;
    end
    panel = panels{i};
    panel.Units = 'characters';
    panel.Position = [x y w h] + transform;
```

```
    panel.Units = 'normalized';

    x = x + w;

end

end
```

10.1.10 showData

```
function showData(self, LOG, Enabled_Trans_SET, Firing_Trans_SET, SIM_COM-
PLETE)

% extract relevant information from gpensim global object (PN)
global PN;
tokens = num2cell(PN.X);
firing = num2cell(logical(Firing_Trans_SET(end,2:end)));
enabled = num2cell(logical(Enabled_Trans_SET(end, 2:end)));

% used in order to refer to the log outside showData copy-on-write prevents
% this from being a performance issue even as the LOG grows over time
% https://blogs.mathworks.com/loren/2006/05/10/memory-management-for-func-
tions-and-variables/#11
self.LOG = LOG;
self.Firing_Trans_SET = Firing_Trans_SET;
self.Enabled_Trans_SET = Enabled_Trans_SET;
% initialize the transitions and tokens table on the first function call
if self.FirstLoop
    self.FirstLoop = false; % only run once

    if PN.REAL_TIME
        % disable pause, step and tickdelay controls when in real time
        self.PlayPauseButton.Enable = 'off';
        self.StepButton.Enable = 'off';
        self.WantedDelaySlider.Enable = 'off';
        self.WantedDelayText.Visible = 'off';
    end

    % Check if the simulation is timed, used to decide x axis when
    % plotting: x axis represent state when time is unavailable
    self.IsTimedPN = any(PN.Set_of_Firing_Times) || PN.REAL_TIME;

    % Fetch entity names
    token_names = {PN.global_places(:).name};
    trans_names = {PN.global_transitions(:).name};

    % Show specified tokens (indicated by ones in a logical array)
    self.ShownTokens = ismember(token_names, self.ShownTokens);
    self.ShownTrans = ismember(trans_names, self.ShownTrans);

    % If no simulation entities are whitelisted then assume the user
    % wants to see all entities.
    if ~any(self.ShownTokens)
        self.ShownTokens = true(1, PN.No_of_places);
    end

    if ~any(self.ShownTrans)
        self.ShownTrans = true(1, PN.No_of_transitions);
    end
```

```
    % Initial table data
    self.TokenTable.Data = [token_names' num2cell(self.ShownTokens')];
    self.TransTable.Data = [trans_names' num2cell(self.ShownTrans')];

    % Printing the first log entry is handled as a special case
    % TODO refactor into function (ex. firstLog() )
    string = ['Simulation of "', PN.name, '":' newline];
    string = [string ' ======= State Diagram ======= ' newline];
    HH_MM_SS = PN.HH_MM_SS; % printing style for time

    len_ETS = size(Enabled_Trans_SET, 1);
    Ps = PN.No_of_places;
    start_time = LOG(1, Ps+5);
    if (HH_MM_SS)
        string = [string '**    Time: ', string_HH_MM_SS(start_time), '
**' newline];
    else
        string = [string '**    Time: ', num2str(start_time), '   **' new-
line];
    end

    initial_markings = LOG(1, 1:Ps); % print initial state
    string = [string 'State:0 (Initial State): ', markings_string(ini-
tial_markings) newline];

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Enabled and firing transitions at start
    ETS_index = 1;
    if (len_ETS)
        string = [string 'At start ....' newline];
        lines = self.log_prnss_ET_FT(...
            Enabled_Trans_SET, Firing_Trans_SET, ETS_index);
        string = [string lines];
    end

    string = [string ' ' newline];
    % end of change -1
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    self.addToLog(string);

end


% if the user changed the shown transitions/tokens
if self.ShownTableNeedsUpdate
    % reset everything (does not occur often)
    self.ShownTrans = cell2mat(self.TransTable.Data(:, 2))';
    self.resetTransitionPanel();
    self.ShownTokens = cell2mat(self.TokenTable.Data(:, 2))';
    self.resetTokenPanel();
    self.ShownTableNeedsUpdate = 0;
end

% update transitions shown in dashboard
index_trans_panels = 1;
for j = 1:length(self.ShownTrans)
    if self.ShownTrans(j)
```

```
        panel = self.TransPanels{index_trans_panels};

        newColor = self.getTransColor(enabled{j}, firing{j});

        % only perform expensive operation when needed
        if ~isequal(panel.BackgroundColor, newColor)
            panel.BackgroundColor = newColor;
        end

        index_trans_panels = index_trans_panels + 1;
    end
end

% update tokens shown in dashboard
index_place_panels = 1;
for j = 1:length(self.ShownTokens)
    if self.ShownTokens(j)
        panel = self.TokenPanels{index_place_panels};
        if panel.UserData.currentTokenCount ~= tokens{j}
            panel.UserData.currentTokenCount = tokens{j};
            panel.UserData.text.String = num2str(tokens{j});
        end
        index_place_panels = index_place_panels + 1;
    end
end


% Transitions with live plot (updating animatedline)
secondsInDay = 3600 * 60;
for i = 1:length(self.GraphedTrans)
    if isvalid(self.GraphedTrans{i})
        graph = self.GraphedTrans{i};
        trans_id = graph.UserData.trans_id;
        ax = graph.Parent;

        if self.IsTimedPN
            x = PN.current_time;
        else
            x = PN.State;
        end

        y = 2 * firing{trans_id} + enabled{trans_id};
        if self.IsTimedPN
            xlim = [PN.current_time - self.GraphWidth, PN.current_time];
        else
            xlim = [PN.State - self.GraphWidth, PN.State];
        end

        if PN.REAL_TIME
            % datetick requires converting from seconds to days
            xlim = xlim / secondsInDay;
            x = x / secondsInDay;
        end
        ax.XLim = xlim;

        addpoints(graph, x(1:length(y)), y );

    end
end
```

```
% graphed tokens (updating animatedline)
for i = 1:length(self.GraphedPlaces)
    if isvalid(self.GraphedPlaces{i})
        graph = self.GraphedPlaces{i};
        place_id = graph.UserData.place_id;
        ax = graph.Parent;

        [~, token_counts] = getpoints(graph);

        % first run
        if isempty(token_counts)
            [x, y] = self.plotHistoryOfTokenCount(...
                place_id, PN.No_of_places, LOG);
        else
            if self.IsTimedPN
                x = PN.current_time;
            else
                x = PN.State;
            end
            y = tokens{place_id};
        end
        xlim = [x(end) - self.GraphWidth, x(end)];
        ax.YLim = [-0.1, max([token_counts + 0.1, 10.1])];

        if PN.REAL_TIME
            x = x / self.SecondsInDay;
            xlim = xlim / self.SecondsInDay;
        end
        ax.XLim = xlim;
        addpoints(graph, x, y);

    end
end

self.updateLog(LOG, Enabled_Trans_SET, Firing_Trans_SET);

drawnow limitrate; % redraws gui, max 20 updates per seconds

if SIM_COMPLETE
    % Write full log to listbox
    self.StateLog.String = self.FullLogString;
    % Move to the most recent entry (bottom of log)
    self.StateLog.Value = length(self.StateLog.String);

    % Grey out simulation control buttons
    self.PlayPauseButton.Enable = 'off';
    drawnow;

    % prevent any plots accidentaly appearing in the GUI
    graphics_root = groot();
    graphics_root.CurrentFigure = [];
end


if ~PN.REAL_TIME && self.SimulationPaused
    % When paused the user will be able to scroll through the full log:
    self.StateLog.String = self.FullLogString;
    % Move to the most recent entry (bottom of log)
```

```
    self.StateLog.Value = length(self.StateLog.String);
    % The user can step through one simulation loop/tick at a time
    self.StepButton.Enable = 'on';
    drawnow;
    uiwait(self.Window);
else

    % Add a delay if running in simulation time (not real time)
    % the time used by the simulation and GUI are subtracted
    currentTime = 60 * 60 * 24 * datenum(clock()); % seconds since year 0
    % seconds spent simulating and updating GUI since last loop
    overheadTime = (currentTime - self.LastTime);
    % if the overhead is too long (> self.WantedTime) pause will complete
    % instantly. pause(-1) == pause(0)...
    pause(self.WantedDelay - overheadTime);

    self.LastTime = currentTime; % used next loop
end

end
```

10.1.11 updateLog.m

```
function updateLog(self, LOG, Enabled_Trans_SET, Firing_Trans_SET)
% prnss function from gpensim adapted
% to enable real time logging


global PN;
% Note: PN.LOG, PN.Enabled_Trans_SET and PN.Firing_Trans_SET are not added
% to the PN datastructure until the end of the simulation. Since updateLog
% is meant for use during simulation these parameters must be provided
% in the input parameters.
Ps = PN.No_of_places;
string = '';

% the Firing_Trans_SET has the same as Enabled_Trans_SET, a "len_FTS"
% variable is therefore not needed since len_ETS == len_FTS.
len_ETS = size(Enabled_Trans_SET, 1);

no_of_rows = size(LOG, 1);

% wait for the Transition sets to catch up in case a previously non
% existing index was referenced
if self.WaitForETSIndex
    if self.WaitForETSIndex <= len_ETS
        lines = self.log_prnss_ET_FT(...
                Enabled_Trans_SET, Firing_Trans_SET, self.WaitForETSIndex);
        string = [string lines ' ' newline ' ' newline];
        self.WaitForETSIndex = 0; % done waiting for an ETS_index
    end
end

if ~self.WaitForETSIndex
    for row = self.NextLogIndex:no_of_rows

        current_row = LOG(row, :);
        fired_trans = current_row(Ps+1);
        if (fired_trans)
```

```
            state = current_row(Ps+2);
            ETS_index = current_row(Ps+4);
            % start_time = current_row(Ps+5); % not used by "prnss"
            finishing_time = current_row(Ps+6);
            current_markings = current_row(1:Ps);
            virtual_tokens = current_row(Ps+7:end);

            lines = self.log_prnss_state(fired_trans, finishing_time, ...
                    current_markings,state, virtual_tokens);
            string = [string lines];

            % Check if the next ETS_index exists
            if len_ETS >= (ETS_index + 1)
                % show the enabled and firing transitions
                lines = self.log_prnss_ET_FT( ...
                    Enabled_Trans_SET, Firing_Trans_SET, ETS_index + 1);
                string = [string lines ' ' newline ' ' newline];

            % If not: wait for the Transition sets to get populated
            % for the next loop (see "if self.WaitForETSIndex ..." above)
            else
                self.WaitForETSIndex = ETS_index + 1;
                break;
            end
        end
    end
    self.NextLogIndex = row + 1;
end

if ~isempty(string)
    self.addToLog(string);
end

end
```

10.2 Source code: Unfinished elevator example

10.2.1 main.m

```
clear all; clc;

global global_info;
%global_info.STOP_AT = 30;%current_clock(3) + [0 0 10];
global_info.DELTA_TIME = 1;
global_info.MAX_LOOP = 1000;
global_info.NO_OF_FLOORS = 3;
global_info.ELEVATOR_DIRECTION = 'up';
pns = pnstruct('pdf');

%dyn.ft = {};
% elevator is on the first floor with 3 passengers inside wanting to go to
% floor 3
dyn.m0 = {'p1_floor_elevator', 1, 'p3_passengers', 4};

pni = initialdynamics(pns, dyn);
g = guiMonitor('WantedDelay', 0.1);
sim = gpensim(pni);
global PN;
```

10.2.2 pdf.m

```
function [ pns ] = pdf()
%TEST_PDF Summary of this function goes here
%   Detailed explanation goes here
global global_info;
top_floor = global_info.NO_OF_FLOORS;

pns.PN_name = 'Elevator';

pns.set_of_Ps = {};
pns.set_of_Ts = {};
pns.set_of_As = {};

% add all floors to indicate elevator position
for floor_number = 1:top_floor
    % Add places, transitions and arc connections for nth floor
    n = num2str(floor_number);

    % Places for each floor

    % people who have called for and are waiting for elevator on nth floor
    n_waiting = ['p' n '_waiting'];
    % people on nth floor
    n_floor = ['p' n '_floor'];
    % passengers in elevator who wants to go to the nth floor
    n_passengers = ['p' n '_passengers'];
    % indicates if the elevator is on the nth floor
    n_floor_elevator = ['p' n '_floor_elevator'];

    pns.set_of_Ps = [pns.set_of_Ps n_waiting n_floor n_passengers...
        n_floor_elevator];

    % Transitions connected to the nth floor
    % passengers who are waiting for a called elevator
    n_wait = ['t' n '_wait'];
    % passengers who are exiting the elevator at this floor
    n_exit = ['t' n '_exit'];
    % passengers who entered the elevator
    n_enter = ['t' n '_enter'];

    pns.set_of_Ts = [pns.set_of_Ts n_exit n_enter n_wait];

    arcs = {};
    % if not on the top floor: add transitions+arcs over the elevator
    if ~(floor_number == top_floor)
        n_up = ['t' n '_up'];
        from_above = ['t' num2str(floor_number + 1) '_down'];
        pns.set_of_Ts = [pns.set_of_Ts n_up];
        arcs = [arcs...
            n_floor_elevator, n_up, 1, ...
            from_above, n_floor_elevator, 1, ...
        ];
    end

    % if not on the bottom floor: add transitions+arcs under the elevator
    if ~(floor_number == 1)
        from_below = ['t' num2str(floor_number - 1) '_up'];
        n_down = ['t' n '_down'];
        pns.set_of_Ts = [pns.set_of_Ts n_down];
```

```
        arcs = [arcs...
            n_floor_elevator, n_down, 1, ...
            from_below, n_floor_elevator, 1, ...
        ];
    end



    % Arcs for each floor
    arcs = [arcs ...
        n_floor_elevator, n_exit, 1, ...
        n_exit, n_floor_elevator, 1, ...
        n_floor_elevator, n_enter, 1, ...
        n_enter, n_floor_elevator, 1, ...
        ...
        n_passengers, n_exit, 1, ...
        n_exit, n_floor, 1, ...
        n_floor, n_wait, 1, ...
        n_wait, n_waiting, 1, ...
        n_waiting, n_enter, 1, ...
    ];

    % when a passenger enters (n_enter) he can go to any other
    % floor. Passengers who are in an elevator and wants to go to floor m
    % are indicated as a token in place pm_passengers.
    for m = 1:top_floor
        if m ~= floor_number
            m_passengers = ['p' num2str(m) '_passengers' ];
            arcs = [arcs, n_enter, m_passengers, 1];
        end
    end

    pns.set_of_As = [pns.set_of_As arcs];
end


end
```

10.2.3 passengersWantDirection.m

```
function [wantDirection] = passengersWantDirection(current_floor, up)
%PASSENGERSWANTDIRECTION Check if any passengers needs the elevator to go
%in a given direcion
%   Detailed explanation goes here
global global_info;

% Check floors depending on direction

% check floors above current floor if going up
if up
    floorsToCheck = (current_floor + 1):global_info.NO_OF_FLOORS;

% check floors below current floor if going down
else
    floorsToCheck = 1:(current_floor - 1);
end

for k = floorsToCheck
```

```
    k_passengers = ['p' num2str(k) '_passengers']; % waiting to exit eleva-
tor
    k_waiting = ['p' num2str(k) '_waiting']; % waiting to enter elevator

    % if passengers want to exit or are waiting to
    % enter the elevator on the k-th floor return true
    if ntokens(k_passengers) || ntokens(k_waiting)
        wantDirection = true;
        return;
    end
end

wantDirection = false;

end
```

10.2.4 COMMON_PRE.m

```
function [fire,transition] = COMMON_PRE(transition)
%COMMON_PRE Summary of this function goes here
%   Detailed explanation goes here
global global_info;
top_floor = global_info.NO_OF_FLOORS;
fire = 1;

name = split(transition.name(2:end), '_');

current_floor = str2double(name(1));
type = join(name{2:end});


down = isequal(type, 'down');
previous_direction = global_info.ELEVATOR_DIRECTION;

if isequal(type, 'up')
    needToGoUp = passengersWantDirection(current_floor, true);
    % check if we need to go up
    if needToGoUp
        % if the elevator was already going up then we continue
        if isequal(previous_direction, 'up')
            fire = 1;
            return;
            % if elevator is on the way down
        else
            % only reverse if we dont need to go down
            needToGoDown = passengersWantDirection(current_floor, false);
            if ~needToGoDown
                global_info.ELEVATOR_DIRECTION = 'down';
                fire = 1;
            else
                fire = 0;
            end
            return;
        end

    % Don't need to go up
    else
        fire = 0;
        needToGoDown = passengersWantDirection(current_floor, false);
```

```
        if needToGoDown
            global_info.ELEVATOR_DIRECTION = 'down';
        else
            % dont need to go up or down, stay at current floor
            global_info.ELEVATOR_DIRECTION = 'stay';
        end
    end
end

if isequal(type, 'down')
    needToGoDown = passengersWantDirection(current_floor, false);
    % check if we need to go down
    if needToGoDown
        % if the elevator was already going down then we continue
        if isequal(previous_direction, 'down')
            fire = 1;
            return;

        % if elevator is on the way up
        else
            % only reverse if we dont need to go up
            needToGoUp = passengersWantDirection(current_floor, true);
            if ~needToGoUp
                global_info.ELEVATOR_DIRECTION = 'up';
                fire = 1;
            else
                fire = 0;
            end
            return;
        end

    % Don't need to go down
    else
        fire = 0;
        needToGoUp = passengersWantDirection(current_floor, true);
        if needToGoUp
            global_info.ELEVATOR_DIRECTION = 'up';
        else
            % dont need to go up or down, stay at current floor
            global_info.ELEVATOR_DIRECTION = 'stay';
        end
    end
end
```