FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: DATMAS Computer Science	Spring semester, 2018 Open
Author: Ronny Wathne	…….. (signature of author)
Programme coordinator: Supervisor(s): Reggie Davidrajuh	
Title of master's thesis: GUI for petri net coverability tree	
Credits: 30	
Keywords: Petri Net, Coverability, MATLAB, GUI,	Number of pages: 62 + supplemental material/other: one zip file containing 10 script files Stavanger, 13/06/2018 date/year

Title page for Master's Thesis Faculty of Science and Technology

GUI For Petri Net Coverability Tree

By Ronny Wathne

Abstract

Petri nets are a long time established modeling concept for modelling and simulation of discrete-event systems. This thesis concerns the development of tools to perform analysis of Petri nets that contains places that are unbounded. The tools are coded in MATLAB, and produces graphical representations of reachability/coverability trees, as well as providing an interactable interface to make changes to how the generation is done.

Table of contents

1 Introduction

This thesis is made to accompany the MATLAB file petNetCoverTreeGUI.m, and its dependencies. It will detail the problems this program was made to solve, the features and design of the program, and the design of the algorithm. It is a tool made for analyzing unbounded petri nets using reachability/coverability trees.

1.1 Motivation

The motivation for the creation of *petNetCoverTreeGUI.m* are as follows:

- Make an algorithm to construct coverability/reachability trees, based on petri nets that can have an infinite number of states.
- Display these trees in a graphical User interface that allows us to manipulate the displayed tree in some ways that lets the user shift our focus to certain areas.
- Make GUI tools to manipulate the creation of the tree by including, or excluding certain transitions, or choosing different algorithms.

Existing MATLAB programs that already do some of these things, have limited scalability, and no GUI controls. This program was made to provide more features, better control, and scalability, compared to alternatives in the MATLAB programming language.

The program will make use of Human Computer Interaction(HCI) theory. Designing a Program with elements of HCI theory involves making an interface That allows for a back and forth loop of feedback and interaction between the user and the provided interfaces.

This program also adds an extension to the petri net algorithm that allows us to add one or more additional values to the transitions in the petri net. The additional values, and HCI elements will be further elaborated on in the Design section.

2 Literature Review

2.1 reachability

The reachability tree of a petri net is a graphical representation of all the possible place markings(states) that the petri net can reach, and the transitions that are used to reach these markings. By analyzing this tree, we can determine certain properties of the petri net we made the tree from.

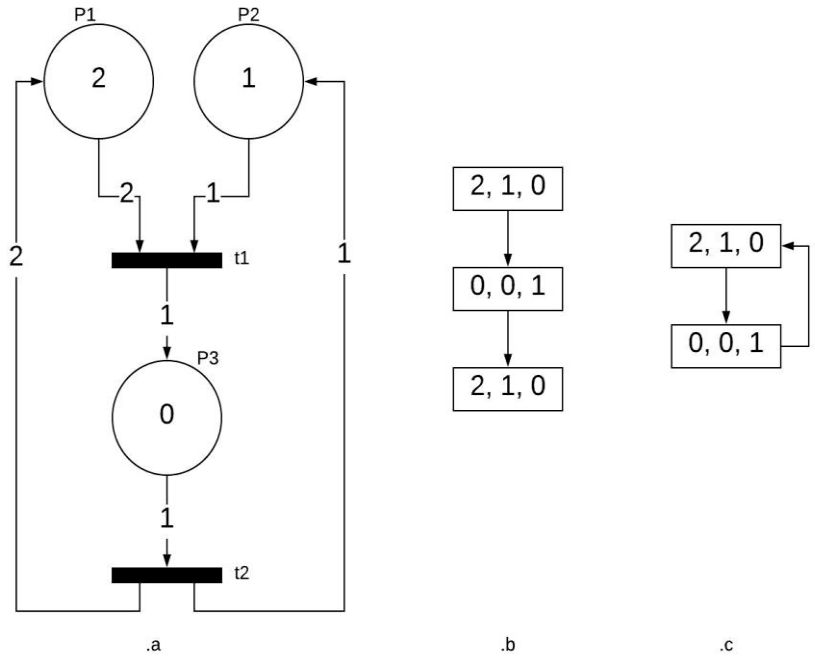

Figure 1: .a shows the Petri net .b shows the reachability/coverability tree with duplicate state
.c shows tree with duplicate state redirected.

The difference between reachability trees and coverability trees are that coverability trees have some additional rules in its algorithm when making new states. Using the algorithm for constructing a reachability tree based on a petri net, will in some cases provide an identical result to what we would get if we were to use the coverability algorithm. Figure 1 shows an example of a petri net that would produce the same tree from using either algorithm.

2.2 coverability

The goal of the coverability algorithm, is to allow us to make a tree of a petri net where some of the places are unbounded, meaning that the value of one or more of the places int the markings can increase into infinity. We can not make a reachability tree of such a petri net, but we can make a coverability tree.

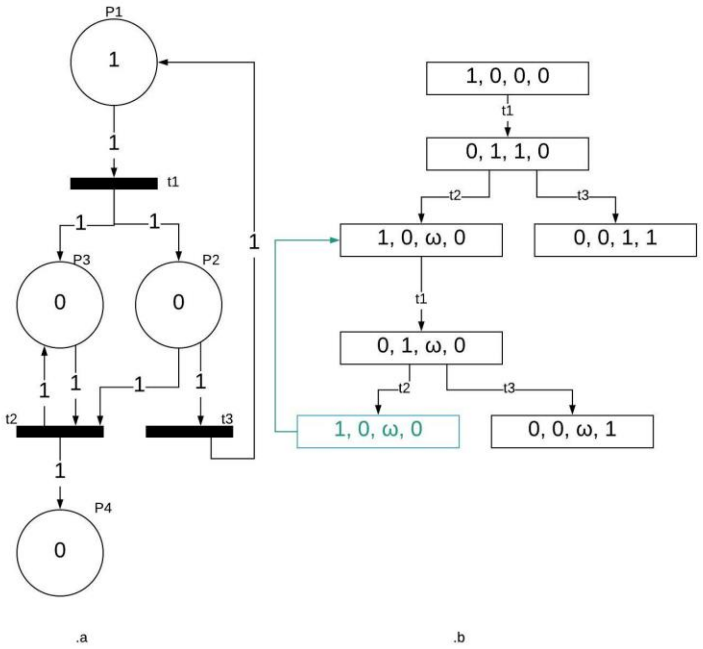

Figure 2: .a petri net .b: coverability tree

The coverability algorithm is made to detect when places are unbounded. A place that is unbounded can in theory eventually contain an infinite number of tokens. To avoid infinite running times, there are a few options available to us.

The coverability algorithm deals with unbound places by assigning them the Omega property, to signify that this place will grow infinitely. Any additions or removals of tokens in this place after it is given this property, will be ignored.

This has the downside of losing some information about the Petri Net that we want to analyze in some cases. This report includes four algorithms based on the reachability and coverability algorithms. Each of these algorithms can provide us with different degrees of detail of Petri Nets.

2.3 HCI

Human-Computer Interaction(HCI) theory are concepts of designing an interface that provides functionality to the end user. It is the idea that interfaces provided by the program, contain some interactable elements that allows the user of the program to make an evaluation of the information the interface provides. The user should be able to affect some change based on this evaluation, by using the provided interface, and then re-evaluate the results.

The goal is to keep an active feedback loop of interaction between the user and the program, where the flow of interaction is not broken by the need to re-adjust parameters by manually programming them.

If we for example wanted to see how the removal of a transition affected a Petri Net, we would need to manually write changes to the code and re-run the program, unless the program provided us with tools that make these changes through interface options.

Likewise, if we are presented with a figure Where the text is too small to read, we should have access to interface elements that allows us to zoom and pan, until we reach a view that provides us with the information that we may have wanted.

3 Design

3.1 Overview

The step by step running of the program goes as follows:

1. Provide proper inputs to *petNetCoverTreeGUI*
2. A GUI window will open, allowing the user to make some choices about how we want the program to handle the inputs.
3. Click the “make states” button, that will generate a list of states, provide feedback about the method used, and how many states were made.
4. Click the “draw” button. This will open a MATLAB figure window, that contains the generated tree.

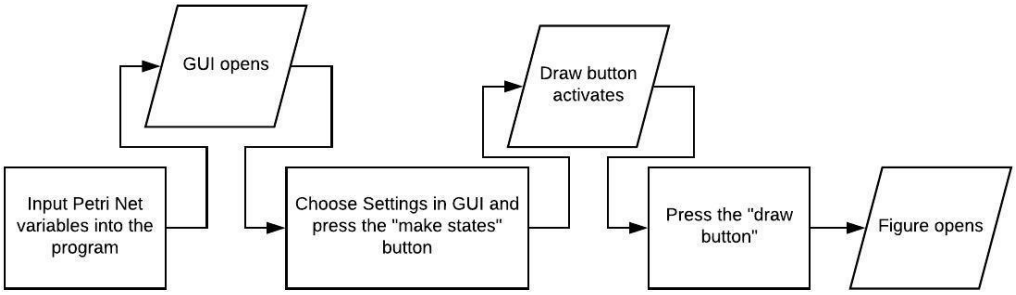

Figure 3: program flowchart from MATLAB command line, to the presentation of a tree

3.2 GUI design

The main GUI is made using MATLABs “uifigure” “container”. This GUI provides us with two basic choices to make.

The first choice is the type of algorithm that will be used to generate the states of the figure. Some of the choices will require some additional parameter in a corresponding value box to the right of the choices.

The other choice is a toggleable option to ignore certain transitions in the state generation. This option works for all algorithms. The field that becomes active when this option is toggled needs the index numbers of the transitions that should be ignored, otherwise the generation will fail.

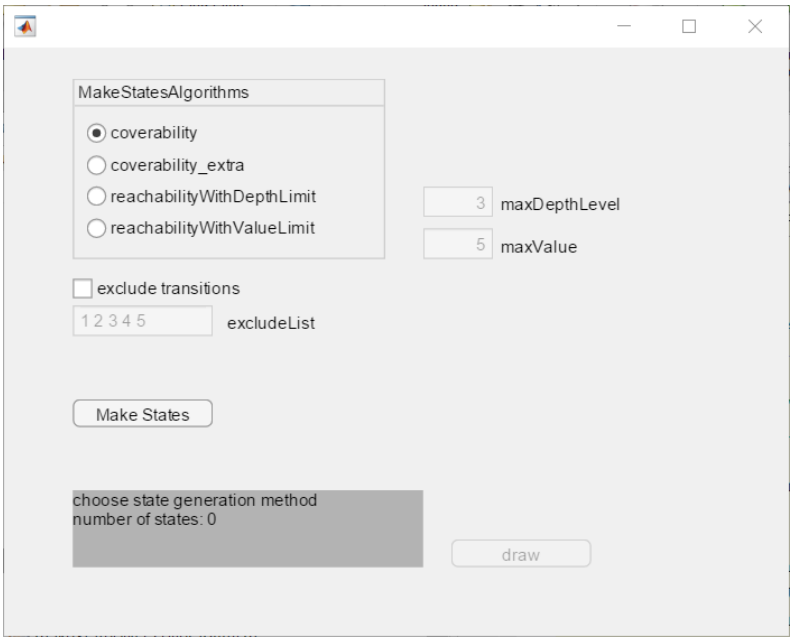

Figure 4: GUI made with MATLAB uiFigure

When the states have been generated, the GUI provides feedback for how many states were made in the currently selected algorithm.

The “draw” button will be disabled until there exists a list of states with more than 0 states in it. When it is pressed, a new MATLAB “figure” “container” window will be opened, where the coverability/reachability tree will be drawn. Every time the “draw button” button is pressed, a new figure will be made based on the currently active configuration in the GUI. This can be used to make side by side comparisons of trees generated by different methods and parameters, purely by using the GUI.

The MATLAB figure comes with some default basic functionality that allows us to zoom and pan, letting us navigate some of the more expansive trees. However, zooming in a MATLAB does resize text objects. This needed to be coded manually in the tree drawing part of the code.

States in the figure are colored according to three properties. Green denotes a unique marking, yellow denotes a marking that already exists, and red denotes a deadlock marking

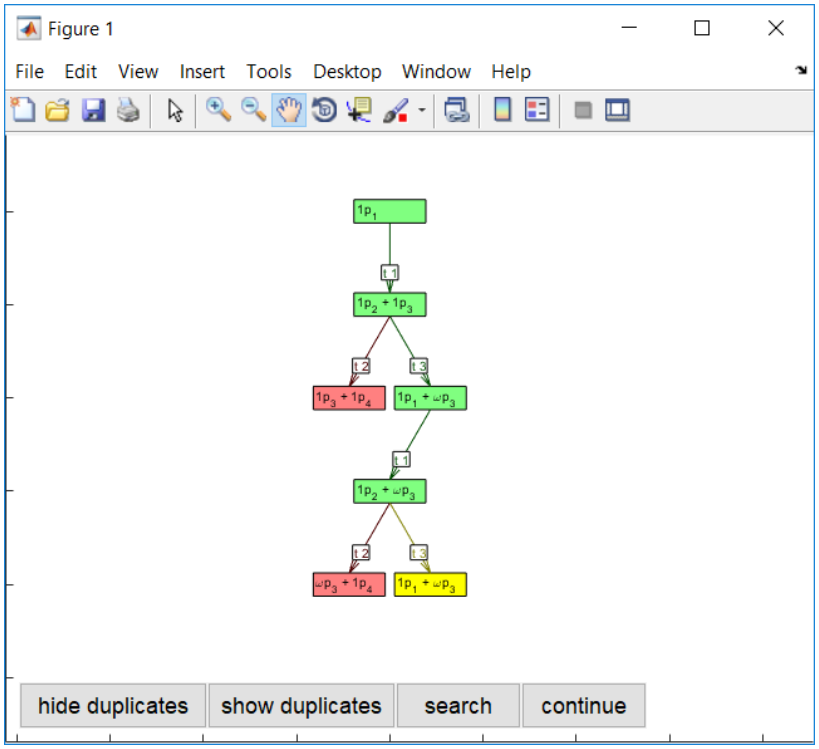

Figure 5: Output tree based on the Petri Net in figure 2

The top bar in figure 5 contains the functions that comes with all MATLAB figures. The zoom, pan, and select modes, can be toggled here. The window itself can be resized and moved as needed, like most GUI windows. Closing a figure window will not stop the program from running, and it is possible to have multiple figures open at a time by either using the draw button on the GUI, or the continue button on the figure.

The bottom row of buttons is made specifically for coverability/reachability tree analysis.

The “hide duplicates” button will remove all states that have a duplicate marking, making all states in the figure unique. All transitions pointing to duplicate markings are also redirected.

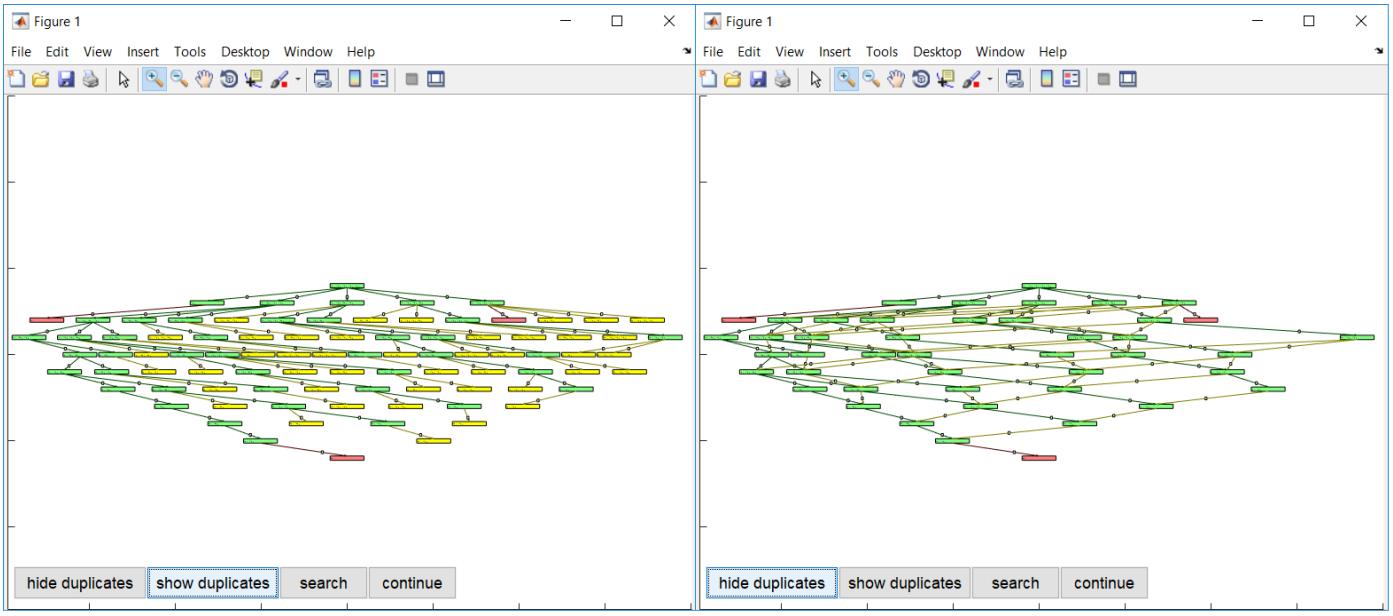

Figure 6: show/hide duplicates buttons

The search button prompts the user for a marking. The marking must be the same length as the markings in the tree. In cases where the marking is represented by the omega symbol, the program uses the letter 'W' as input. All states that match the search query will be highlighted in the figure window.

All states in the figure can be highlighted by clicking them (any MATLAB functionality such as zoom, or pan, must be de-selected first).

Highlighting a state and pressing the search button will enter the marking of the highlighted state into the search dialogue window as a default search.

Pressing the continue button, will make a new figure with a tree that has its home marking based on a state selection in the previous tree, and the algorithm that was used to construct it.

3.3 Algorithm design

petNetCoverTreeGUI.m provides four algorithms that are all designed to handle unbounded petri nets.

Coverability

This is the basic coverability algorithm.

For every transition into a new state, it keeps track of the markings of all previous states before it. The values of these markings are compared to the values of the marking of the new state. If the algorithm finds a previous state where all the new marking values are either greater or equal, it will then compare each value in both the old and new markings. If a value in the new marking is strictly greater, this value will be replaced with the omega symbol in the new marking. This is to denote that the value of that place can grow into infinity.

Coverability_extra

One of the problems with the coverability algorithm, is that the abstraction can hide the behavior of the petri net. The program provides a version of the coverability algorithm that can in some cases show more information than the basic algorithm. This algorithm only has an effect if there are transitions that remove more than one tokens from any place. If not, it will have the same result as the basic coverability.

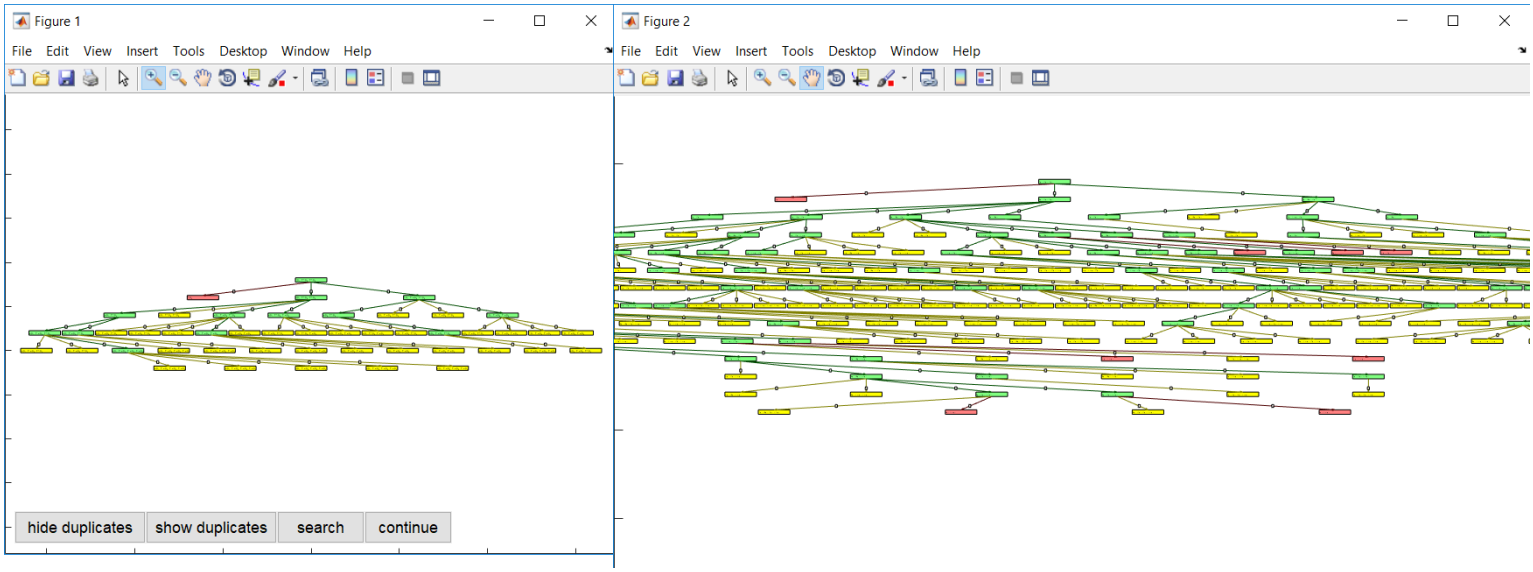

Figure 7: Left figure shows standard coverability. Right figure shows Coverability_extra of the same Petri Net

reachabilityWithDepthLimit

In the cases where we want the complete list of states that the reachability algorithm provides, while working with petri nets with infinite states, we define a terminating property that limits the size of the generated tree. The algorithm “reachabilityWithDepthLimit”, as the name implies, generate states until

it reaches a maximum “depth” in the tree. This algorithm requires one numeric input parameter that denotes the desired maximum depth.

reachabilityWithValueLimit

Another state generation algorithm that the program provides, is the “reachabilityWithValueLimit” algorithm. It takes one numerical input parameter and continues to generate states as long as none of the values in the states, exceed this parameter value.

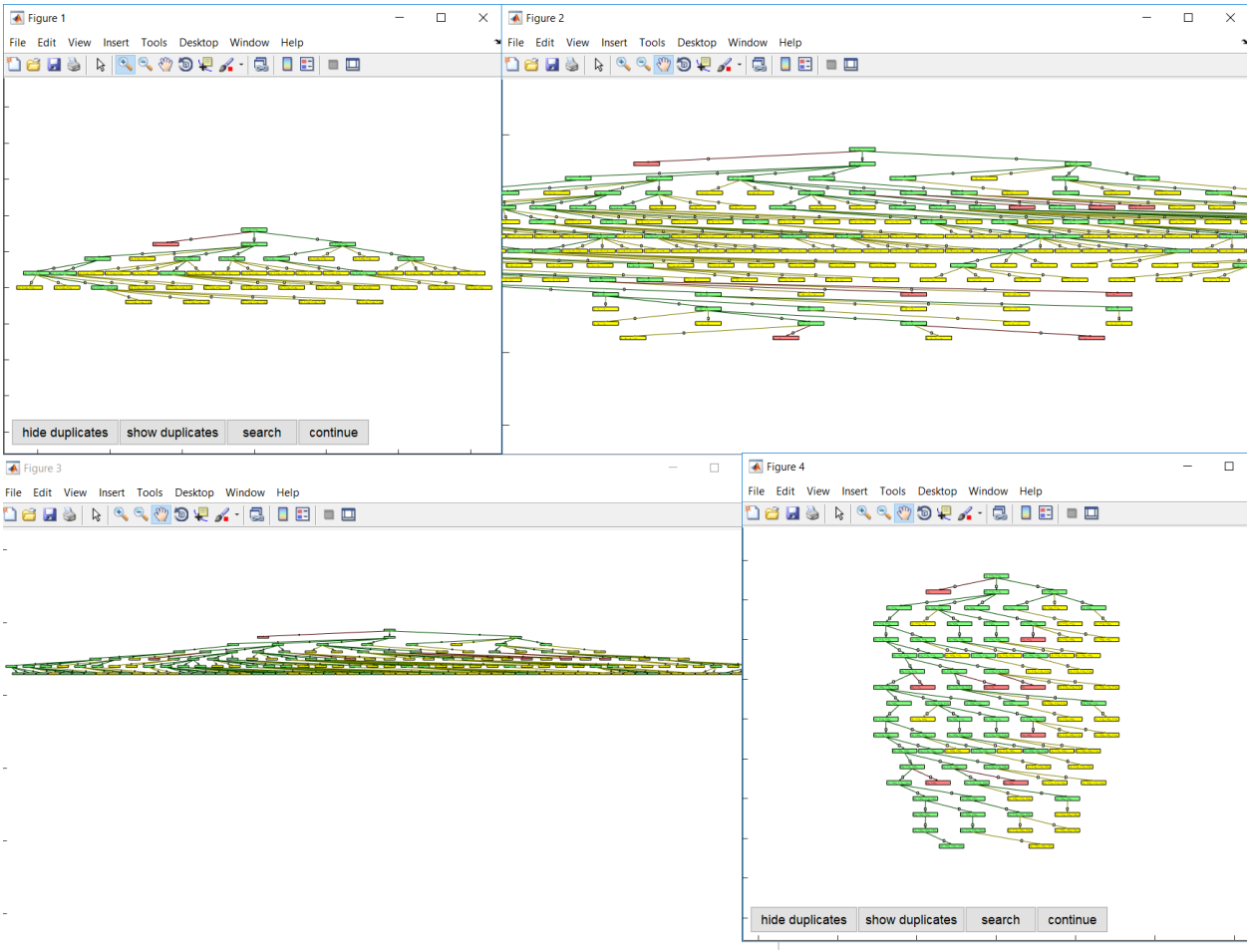

Figure 8: four different results from four different algorithms analyzing the same petri net. The bottom left shows a tree with a depth limit. The bottom right shows a tree with a value limit.

Coverability vs reachability

The coverability algorithms gives us a representation of all states of the petri net, at the expense of details about the flow of the petri net.

The reachability algorithms that use terminating properties, provides us with all the states in the petri net, within some given range of values.

Choosing different algorithms, making states, and pressing draw, produces figures of the input petri net that can be compared side by side.

When using the Coverability algorithms, it may be useful to make a note of the transitions that gives a marking the omega property, and then use a figure with one of the coverability algorithms to obtain greater detail of that area of the tree.

Extra transition parameters

The program includes a petri net extension, where we can add one or more costs to the firing of a transition. This can be used to for example model a system where a transition has a cost of some sort. There may be a cost in time and/or the cost of one or more other resources. Every state in the tree will have an array of values next to it that denotes all the extra values that are affected by the transitions used to reach that state.

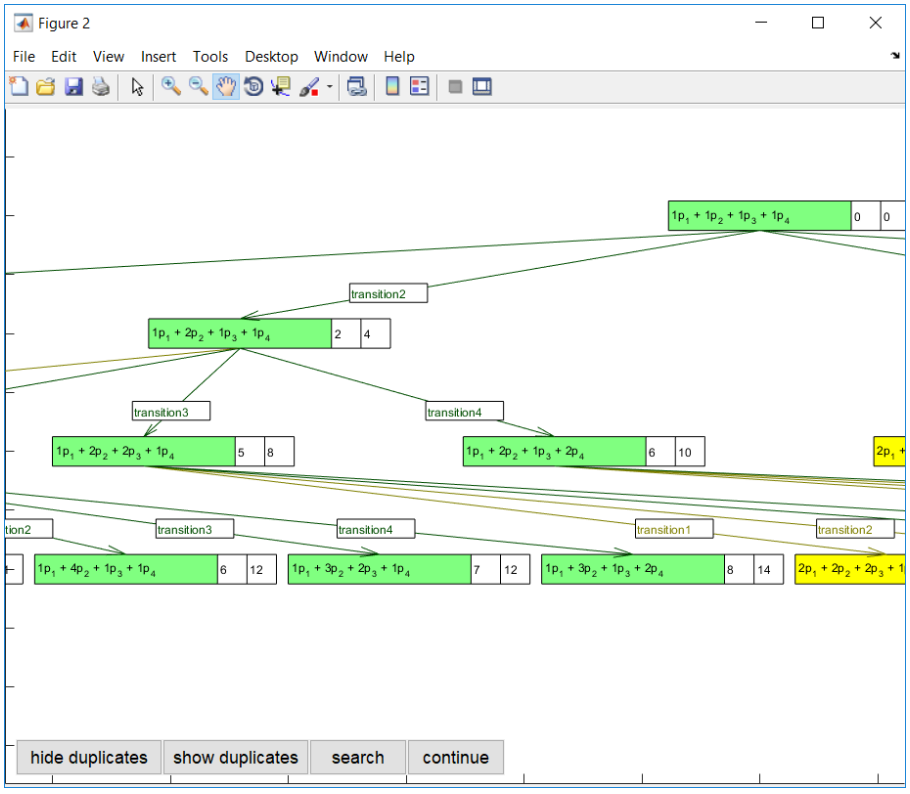

Figure 9: states with two extra values in a zoomed in tree

This is an optional input, that will only be displayed in the reachability algorithm trees. This is because, if one of the values in the state marking is ever given the omega value, all the extra values are essentially infinite.

4 The algorithm

4.1 The input

The program is started as a method on the MATLAB command line, or in a script file. It takes six input variables.

The first input is an array of numbers to represent the starting, or “Home” marking. Each number represents the value in each place in the petri-net.

Two other inputs are two matrices where the number of columns corresponds with the length of the Home marking, and where the number of rows corresponds with the number of transitions in the petri-net. The number of columns in the transition matrices must match the length of the home marking.

The fourth input is an array of names for each transition. The number of rows in the transition matrices must match the length of this array.

The two last inputs can be empty lists. These are used to provide extra values to each state based on the transitions that was used to reach it, but they are not needed to generate the trees themselves. The fifth input is an array of initial extra values for the home marking. The sixth input is a matrix that must have as many rows as the transition matrices, and as many columns as the length of the array of initial extra values.

Sample input values from MATLAB command line:

```
homeMarking=[1 0 0 0];

subMat=[   -1   0   0   0;
            0  -1  -1   0;
            0  -1   0   0];

addMat=[    0   1   1   0;
            0   0   1   1;
            1   0   0   0];

extraInitialValues=[]
extraTransitionValues=[]
transistionNames={'t 1' ; 't 2' ; 't 3'}
petNetCoverTreeGUI(homeMarking,subMat,addMat,extraTransitionValues,…
extraInitialValues,transistionNames)
```

This is the input for the Petri Net shown in figure two(page four).

If we for example wished to give the transitions a cost, possibly a cost of time, and a cost of resources, the definitions for *extraTransitionValues*, and *transistionNames,* would look more like this:

```
extraInitialValues=[0 0]
extraTransitionValues=[ 1 3;
                        2 5;
                        3 4;]
```

When the algorithm is run, it will make a MATLAB struct based on the home marking. It will use the home marking to calculate all transitions that are active from this marking. It will then continue to make structs of the new markings until no more transitions are possible. It will then return the produced array of structs.

4.2 The GUI control choices

These inputs are passed to the *petNetCoverTreeGUI* method. We will then be presented with a GUI interface, where we can make several choices.

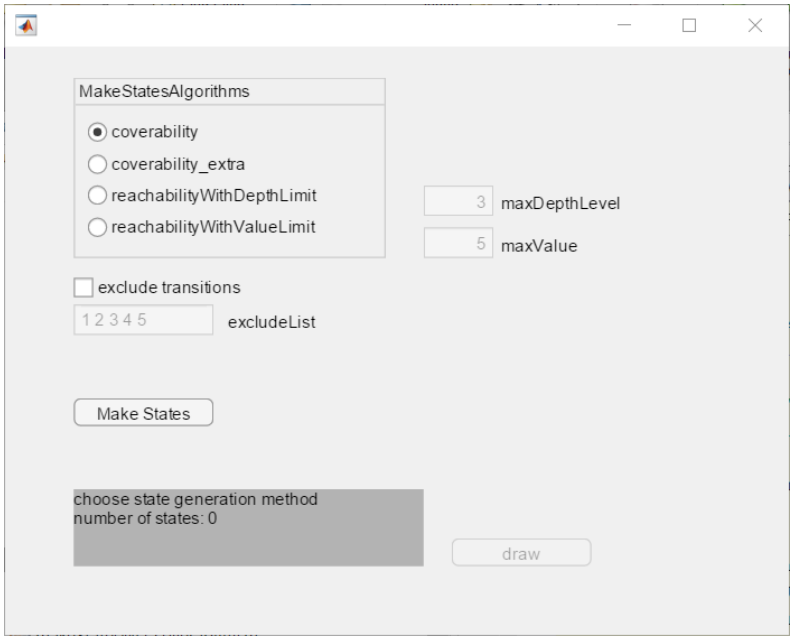

Figure 10: GUI control choices

First, there is a radio button selection to indicate what algorithm we want to use.

Second, we have a checkbox option of turning off transitions before we generate the states. This can be used with all the algorithms.

Changing the radio button selection will update a variable with the name of the currently selected button. When the “make states” button is pressed, the button handler will use a switch statement based on this variable.

The methods used, and their inputs will be dependent on the algorithm choice, but the result the that is returned will in all cases be a list of states, and the longest string of characters in those states markings.

The “draw” button will remain deactivated until here exists a list of states with a length greater than zero.

It is possible to make a figure of one set of settings, then draw a new set of states based on other settings, then do side by side comparisons of these two figures.

4.3 algorithm code

When we make states, we make a list of MATLAB structs that contain the following information

- The parent marking
- The last transition made to reach this state(index number, not name)
- The unique ID of this state
- The unique ID of the parent state
- The Depth in the tree of this state
- The marking of this state
- The type of this state(normal ,duplicate or deadlock)
- The extra values of this state
- The history of all transitions made to reach this state

Additionally, the states made by the coverability algorithms will contain a list of all previous markings.

This section details the four algorithms that the program lets the user choose between.

basic coverability

The following pseudocode shows how the program uses this algorithm to create the elements for a coverability tree

```
makeStatesCover(home, addTMatrix, subTMatrix)
      result= empty array
      queue= empty array
      parentStruct = Make struct based on home marking
      add parentStruct to the result array
      AT = findActiveTransitions(home , subTMatrix)
      For all elements i in AT
            newMarking = calTransWithCoverCheck
            (parentStruct.previousMarkings, home, subTMatrix(i) ,
            addTMatrix(i))
            childStruct = Make struct based on tmpMarking
            add childStruct to the queue array
      while queue is not empty
            parentStruct = Make struct based on queue(1).marking
            remove queue(1)
            add parentStruct to the result array
            AT = findActiveTransitions(parentStruct.marking ,
            subTMatrix)
            For all elements i in AT
                  newMarking = calTransWithCoverCheck
                  (parentStruct.previousMarkings, parentStruct.marking,
                  subTMatrix(i), addTMatrix(i))
                  childStruct = Make struct based on tmpMarking
                  add childStruct to the queue array
```

```
Return result
```

This will make a list of MATLAB structs that contains all the states that the input petri net can inhabit according to a coverability tree.

makeStatesCover makes use of two methods: *calTransWithCoverCheck* and *findActiveTransitions*.

The *calTransWithCoverCheck* method takes four inputs:

- A list of all the markings that reached between the home marking the current point
- The marking of the immediate parent state
- Two arrays of values obtained from the addition and subtraction matrices, based on the active transition AT(i)

For the basic coverability algorithm, the *calTransWithCoverCheck* method produces a marking based on the coverability algorithm.

```
calTransWithCoverCheck (allPrevMark , parentMark , addVal, subVal)
      newMarking = parentMark + addVal + subVal
      for all elements i in allPrevMark
            if all elements in i >= all elements newMarking
                  for j=0; j<length(i);j++
                        if newMarking(j) > i(j)
                              newMarking(j) = inf
Return newMarking
```

Not shown in the pseudocode here, are conversions between int values and char values. The method returns an array of strings where the inf value is represented by the value ‘W’, to represent an omega symbol.

The *findActiveTransitions* method returns a list of index numbers where the resulting marking have no values with a lower value than zero. It needs the marking to evaluate, and the matrix containing all values that are subtracted in each transition.

```
findActiveTransitions(marking , subTMatrix)
      AT =[];
      for elements i in subTMatrix
            tmp=( marking + subTransMatrix(i,1:end))
            if all elements in tmp >=0
                  add i to AT
Return AT
```

If all values in a marking are zero or non-negative, it means that the transition being evaluated is active, and the index is added to the “AT” variable.

Coverability_extra

This algorithm was made to remove some of the abstraction in the basic coverability algorithm by adding an additional condition for a value being given the omega property. At the beginning, the algorithm will examine the matrix with subtraction values, and produce a list of the greatest absolute value that are removed from each place.

For example: if a place has two transitions that remove markings from it, and these two transitions remove different numbers of values from that place, then the highest of these values will be used in the list.

The additional condition for giving a place the omega property, is that the new value, also needs to be greater or equal to the value in this list that corresponds to the given place.

```
calTransWithCoverCheck_extra (allPrevMark , parentMark , addVal,
subVal, maxsub)
      newMarking = parentMark + addVal + subVal
      for all elements i in allPrevMark
            if all elements in i >= all elements newMarking
                  for j=0; j<length(i);j++
                        if ( newMarking(j) > i(j))&(newMarking(j) >=
                  maxsub(j))

                              newMarking(j) = inf
Return newMarking
```

Because the code of coverability and coverability_extra is nearly identical, the program uses the same code for both algorithms, and differentiates by use of a switch statement.

Reachability_with_depth_limit

The reachability algorithms function largely the same with two main differences.

The method for calculating new markings is simpler, each state does not need to maintain a list of all its reachable markings.

In the queue loop, when we remove states from the queue, we need to decide if we should continue adding states.

The following is pseudocode for *makeStatesWithDepthLim.m*

```
makeStatesDepthLim(home, addTMatrix, subTMatrix, limit)
      result= empty array
      queue= empty array
      parentStruct = Make struct based on home marking
      add parentStruct to the result array
      AT = findActiveTransitions(home , subTMatrix)
      For all elements i in AT
            newMarking = calTrans
            (home, subTMatrix(i) , addTMatrix(i))
```

```
            childStruct = Make struct based on tmpMarking
            add childStruct to the queue array
      while queue is not empty
            parentStruct = Make struct based on queue(1).marking
            remove queue(1)
            add parentStruct to the result array
            if parentStruct.Depth +1 > limit
                  continue
            AT = findActiveTransitions(parentStruct.marking ,
            subTMatrix)
            For all elements i in AT
                  newMarking = calTrans (parentStruct.marking,
                  subTMatrix(i), addTMatrix(i))
                  childStruct = Make struct based on tmpMarking
                  add childStruct to the queue array
Return result
```

All states made has a “Depth” property, allowing the algorithm to skip the step of making child states if their depth will reach a greater value that the input variable “limit”.

Reachability_with_value_limit

makeStatesValLim.m works in a similar manner, but with one difference. In the depth limit algorithm, we will either make all or none of the child states. In the value limit algorithm, we need to do the check to either accept or reject the new state after we have calculated its marking.

4.4 draw figure code

The Method that draws the tree in a MATLAB figure, *drawGraphReachLimContinue.m,* is called by the “draw” button in the first GUI, or by the “continue” button present in all figures made by the method.

It takes 9 variables:

- A list of MATLAB structs
- A matrix of transition subtraction values
- A matrix of transition addition values
- A number representing a limit set by the algorithm used
- The name for what type of algorithm to use, as a string
- A numeric list of all transitions to ignore, if any
- A list of extra transition values
- A number representing the maximum space used by a state
- A list of strings to represent the transition names

When the “continue” button in the figure is pressed, it will make a new list of structs based on the settings the previous figure was made with. This requires that the setting be passed to the drawing method.

We find out how much horizontal each state needs in the figure based on the state with the largest number of nonzero marking places. An example of this is shown in figure 5(page 7), where a tree is made based on a petri net with four places, but we only need space to display two places.

The maximum depth of the tree, and how many states there are on each level are obtained from the input states.

Every depth level of the tree is then given a space between each transition based on how much larger the maximum is. This is to prevent long distances between states.

We calculate a total length on a depth level based on how many states there are on that level, the space for that level, and the total length of a state.

We then iterate over all the states and give each struct a set of coordinates based on the depth of the state.

The Figure is horizontally centered on the zero coordinate. Vertically it starts at zero and continues into the negative.

When the figure is drawn, the states are given mouse button click handlers to allow us to select them. The MATLAB rectangles and text are added to their respective structs. This allows us to modify them based on the ID number of the state it represents.

5 Testing

Using MATLAB figures to draw large trees eventually reach a performance limitation. Testing showed that that navigation such as zooming and panning actions become very slow and unresponsive when the number of objects in the figure becomes high enough.

On the system the program was developed on, responsiveness was becoming unreasonably slow when navigating a tree consisting of about 800 states. Moving the window is not affected, but resizing is.

Every new figure is opened as a new window in the upper right corner of the screen. They need to be moved around to make side by side comparisons.

Resizing the figure window, changes the size of the state rectangles, while not changing the size of the text. This can cause marking text to reach out of its state rectangle if the window is too small. The negative effects of this seems minimal, since you would need to minimize the window to the point that the rest of the figure becomes very hard to see for this to happen.

It is apparent that the MATLAB figure features was not made to be used as a graphical canvas in this manner. We may need to look for other

6 Discussion and Future work

Drawing limits

As mentioned in the testing section, we hit performance issues for very large trees. This can be alleviated to some extent by making drawing algorithms that does not draw the duplicate states at all. As it is now, even when they are hidden, states are still part of the figure, and need to be updated during navigation, affecting performance.

If the parameters for coverability algorithms are too large, the program may lock up. There is no time-out functionality.

Figures can sometimes have a lot of long transition arrows, if there is a big variance in how many states there are at each depth level. This can make it difficult to follow a specific transition path.

Algorithm limits

The algorithm does not detect if the new marking is the same as the old marking. There needs to be added a better way to display loops in the tree.

Make a new state type to denote a loop. Add new code for how to draw transitions to and from this state.

Future work

In reachability trees that stop producing states when it reaches some threshold, it may be useful to mark the states where some, or all, of its possible transitions ae outside the threshold. This could be done by defining it as a new state type.

There should be some tools for expanding functionality to the extra transition variables that are given to each state. Possibly a way to choose the path through the tree based on the highest or lowest cost of a selected variable.

States with duplicate markings should keep track of the ID of all the states that they are duplicates of. This would be useful to compare the extra values of the other states.

Make a function for the figure that allows us to highlight the path from the currently selected state, to the home marking.

Have every “normal” state keep a list of all paths it took to reach that state, by adding the paths from states with duplicate markings. This would make highlighting a path easier

In the settings GUI, add controls that allow the user to change the home marking, and maybe a way to change the parameters of the transitions. HCI theory, involves allowing the user to make changes to program parameters by using the GUI, rather than redefine input parameters in the command line.

There could be an added option of not providing any command line input parameters for the Petri Net at all, and allow parameters, like the number of places, and the number of transitions, to be set in the GUI. There should also be a list of all current transition in the GUI that is updated with each added or removed parameter.

It may be better to make the graphical elements of the transitions, part of the state struct list. This will allow us to edit transition arrows, by knowing the ID of the state they point to.

Instead of just hiding duplicate states, it may be better to make a new figure based on a new struct list that only contain normal and deadlock states. This could make figures easier to analyze

Output something for about every hundred states made when the “make states” button is pressed, to provide loading feedback.

When the States have been made, output more information about it to the GUI, such as how many deadlock states, or duplicate states. Maybe, how many times each transition is used.

Make a function that somehow marks all states that contain a marking where one or more values have the omega property.

There are some coding inefficiencies in the program that can be improved.

There is no error handling for out of bound, or wrong inputs.

Problems

The text in MATLAB figures does not resize when the inbuilt zoom functions are used. The the program code that generates the figure has a method that resizes the text manually every time those functions are used.

Attempting to use the figure MATLAB options to save a figure as an image makes MATLAB automatically resize the text of the markings and transitions to near unreadability. To get exactly what is shown on screen, the user needs to make use of a screenshot tool.

The MATLAB “quiver” arrows used, automatically resizes the arrow head based on the length of the arrow. To have the same arrow head size, on all arrows, they all need to be resized again after creation.

Redirecting transitions with the “hide transitions” button can cause text box overlap and hide information.

Due to how the MATLAB figure handles graphical elements and text, we need to use the “equals” property of the figure, otherwise MATLAB will automatically resize the rectangles, so the text no longer fit.

7. References

Cassandras, C. G., & Lafortune, S. (2009). Introduction to discrete event systems. Berlin: Springer Science & Business Media.

Davidrajuh, R. (2013). Extended reachability graph of petri net for cost estimation. In 8th EUROSIM Congress on Modelling and Simulation, Cardiff, Wales, September 10-12, 2013, pp. 378-382.

Davidrajuh, R. (2017) Modeling descrete event systems with GPenSIM An Introduction, Springer Briefs in applied sciences and technology

I. Scott MacKenzie (2013). Human-computer interaction : an empirical research perspective , Waltham Mass. , Morgan Kaufmann

GPenSIM general purpose Petri Net simulator http://www.davidrajuh.net/gpensim/

Jensen, K. (1996). Coloured petri nets : Basic concepts, analysis methods and practical use : Vol. 1 (2nd ed., Vol. Vol. 1). Berlin: Springer.

Vilar, P. (2010). Designing the User Interface: Strategies for Effective Human-Computer Interaction (5th edition. Journal of the American Society for Information Science and Technology, 61(5), 1073-1074.

Aniqa Sikandar Butt (2015) Graphical State – Space Diagram Two Problems Simulation (bachelor theisis) Retrieved from University of Stavanger

Diaz, M. (2009). Petri nets : Fundamental models, verification and applications (Vol. V.81, ISTE). London :: ISTE ; John Wiley and Sons.

Murata, T. (1989). Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 77(4), 541-580.

Shouguang Wang, Mengchu Zhou, Zhiwu Li, & Chengying Wang. (2013). A New Modified Reachability Tree Approach and Its Applications to Unbounded Petri Nets. Systems, Man, and Cybernetics: Systems, IEEE Transactions on, 43(4), 932-940.

A. Appendix

A1. Installation Guide

To use this program, you need to have a recent version of MATLAB installed.

This program consists a few MATLAB script files, probably gathered in a zip archive. To use it, unzip the files to whatever location you prefer for organizing your MATLAB script files. Either select this place as the current folder in MATLAB, or set a path to the folder with these files.

A2. User Manual

The program is run by the *petNetCoverTreeGUI* method from the MATLAB command line.

This method requires six inputs in a specific order:

1.A one dimensional matrix to represent the home marking.
2.A matrix containing negative or zero values, to represent tokens removed by all transitions.
3.A matrix containing positive or zero values, to represent tokens added by all transitions.
4. A one dimensional matrix to represent the extra values.(can be empty)
5. A matrix containing values to be added to the extra parameters in each transition. (can be empty)
6. A cell array containing all the names for the transitions.

The length of input 1 must be the same as the length of the rows in input 2 and 3.

The number of names in input 6 must be the same as the number of rows in input 2 and 3.

Inputs 4 and 5 can be empty. If there are elements in input 4, there must be a matrix in input 5 where the length of the rows are the same as the length of input 4, and the number of rows are the same as input 2 and 3.

Sample input values from MATLAB command line:

```
homeMarking=[1 0 0 0];

subMat=[   -1   0   0   0;
            0   -1  -1   0;
            0   -1   0  0];

addMat=[    0   1   1   0;
            0   0   1   1;
            1   0   0   0];

extraInitialValues=[0 0]
extraTransitionValues=[ 1 3;
                        2 5;
                        3 4;]
transistionNames={'t 1' ; 't 2' ; 't 3'}
petNetCoverTreeGUI(homeMarking,subMat,addMat,extraTransitionValues,…
extraInitialValues,transistionNames)
```

By running *petNetCoverTreeGUI* with the required inputs, you will be presented with this GUI.

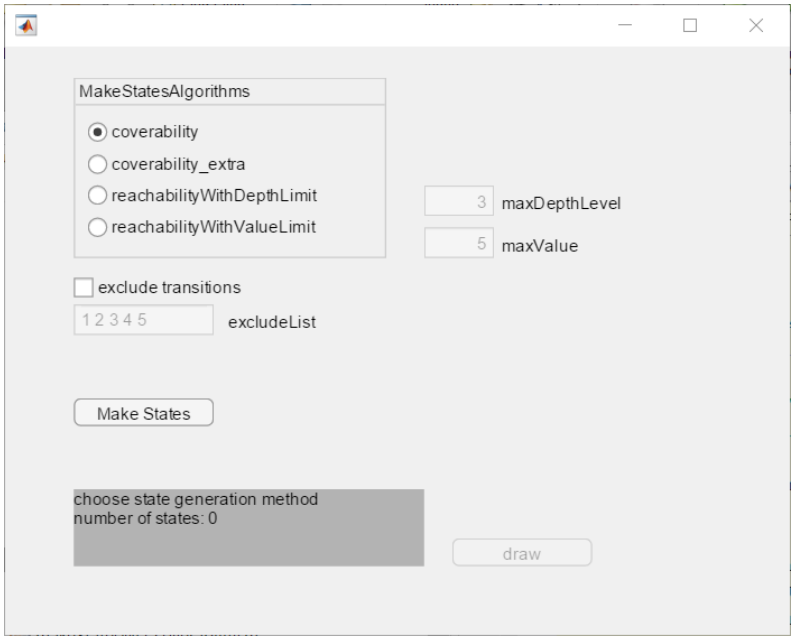

Figure 11: GUI control choices

You can choose four algorithms for generating states based on the input Petri Net, two of which take an extra parameter value that can be edited when the relevant algorithm is selected. The already present value is a default value.

If the “exclude transitions” box is checked, you will be able to edit the contents of the “excludeList” box. This box denotes the index numbers of the transitions to be excluded. The values in this box need to be numeric, with a space separating each number.

The results from pressing the “make states” button depends on all the above settings. There will be some information about the generated states in the gray box below. If a set of states is successfully generated. The draw button will be activated. If the “make states” button is pressed again, the current set of states will be overwritten.

Pressing the “draw” button will open a figure in a new window. Pressing it multiple times will continue opening new windows, displaying the tree of the current result from the “make states” button.

Choosing new settings and pressing “make states” will have no effect on figures that are already open.

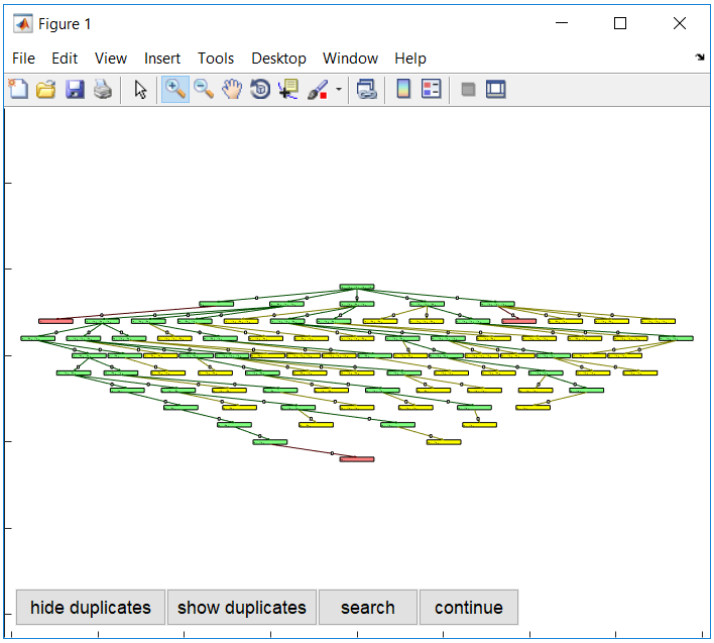

Figure 12: Figure window opened from pressing draw

Zoom and pan functions are found in the top toolbar.

States

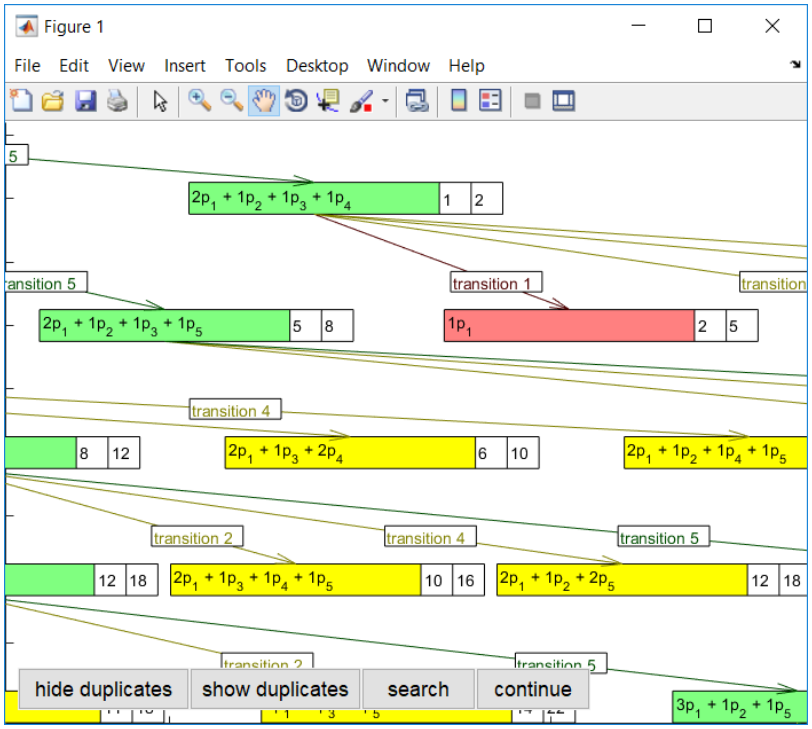

Figure 13: state marking

The example I figure 13 has five places. Only places that have tokens in them are displayed in the states. The size of the states is set by the largest marking size in the tree. Figure 5(page 7) shows a tree from a Petri Net that has four places, but the tree only needs space for two.

Hide/show duplicates

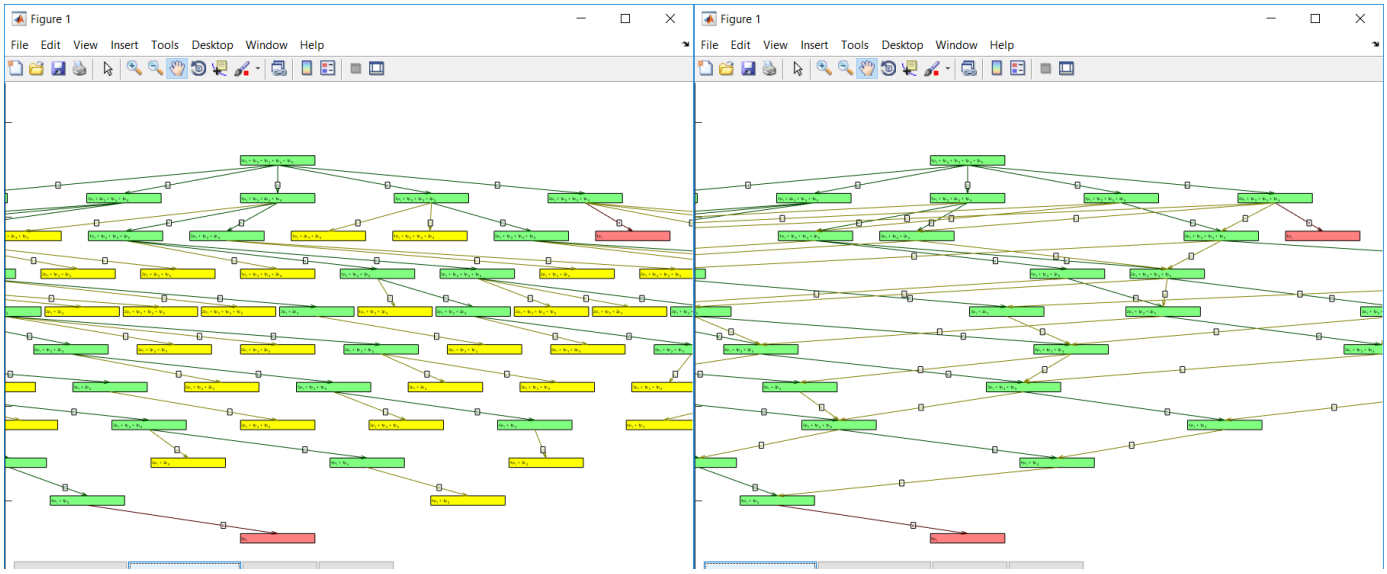

Figure 14: Show/hide example

Duplicate markings are colored yellow and can be hidden. Red markings denote Petri Net deadlocks. The buttons are in the bottom row. The default is to show duplicates

Search

Figure 15: search example. Used to find the normal state from a duplicate state.

All states in the tree can be selected, by clicking on them with the mouse pointer. If a function is active in the top toolbar, it needs to be de-selected first.

Clicking the search button will automatically fill the search dialogue window with a search query based on the currently selected state. It is possible to do a different search by filling in values for each state. Here the omega property is represented by the letter W.

Continue

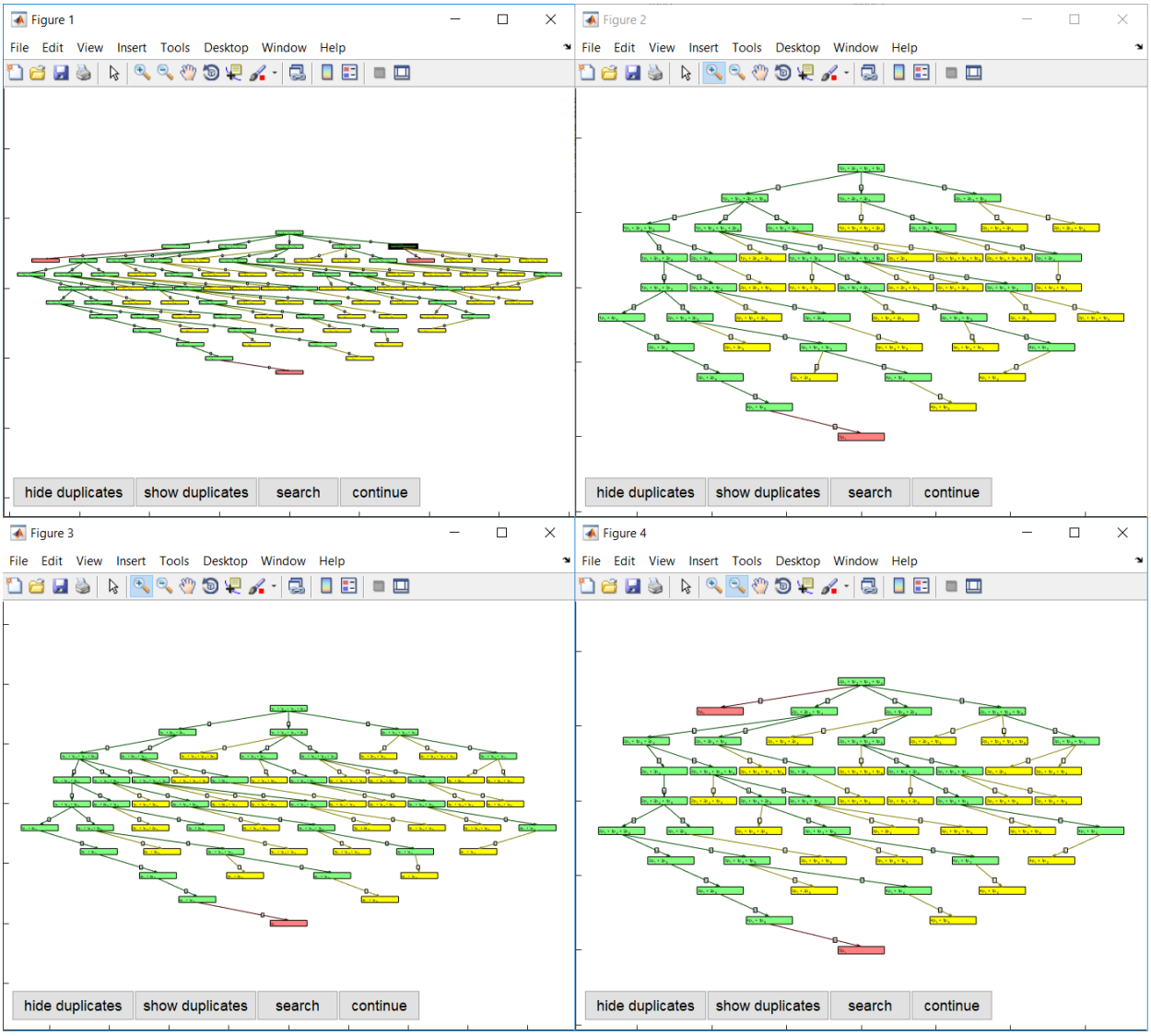

Figure 16: continue example. figure 2 3 and 4 are made from selecting states in figure 1 and using the continue function

The continue button makes a new figure based on the currently selected state and opens it in a new window. The new tree will be generated using the same algorithm and settings that the original tree was generated with. If the tree was generated with an algorithm that made use of a maximum value, then this value will be incremented, based on the values of the selected state.

A3: Complete Code

```
function petNetCoverTreeGUI(homeMarking,subtractTransitions,addTransitions,...
    extraTransitionValues ,extraInitialValues,transistionNames)
% takes six inputs, Opens a GUI for making graphical coverability/reachability trees
in matlab figures
% input 1 is a list of numbers representing the number of tokens in each
% marking, in the first state. the index number of the value will be
% considered the place name.
% input 2 is a matrix where each row represents the tokens removed from
% each place by a trasition. Contains only negative values or 0
% input 3 is a matrix where each row represents the tokens added to
% each place by a trasition. Contains only positive values or 0
% input 4 is a matrix, containing the extra costs for each transition.
% input 5 is a list of numbers with the extra values of the first state.
% input 6 is an array of strigs that need to have as many elements as there
% are rows in input 2 and 3. It is the names for each transition.
% makes use of drawGraphReachLimContinue.m, makeStatesCover.m, makeStatesDepthLim.m
and makeStatesPlaceValLim.m

extran=extraTransitionValues; %debugging
initval=extraInitialValues; %debugging

fig = uifigure;
fig.Position =[20   350   560   420];

generationChoice='coverability'; %default algorithm choice. value changes from the
radio buttons
States=[]; % states are generated by the make states button

% button group for radio buttons
bg = uibuttongroup(fig,...
    'Title','MakeStatesAlgorithms',...
    'Position', [50 270 223 129],...
    'SelectionChangedFcn',@(bg,event) bselection(bg,event));

% Create four radio buttons in the button group. The names are passed to
% the generationChoice to be used in later switch statements
r1 = uiradiobutton(bg,'Text','coverability','Position',[10 83 170 15]);
r2 = uiradiobutton(bg,'Text','coverability_extra','Position',[10 60 170 15]);
r3 = uiradiobutton(bg,'Text','reachabilityWithDepthLimit','Position',[10 38 170 15]);
r4 = uiradiobutton(bg,'Text','reachabilityWithValueLimit','Position',[10 15 170 15]);

%'Position' = [ x, y , lenght , height] x and y are distances from bottom right corner
% lenght, height goes up from base point.  x and lenght are horizontal
% distances, y and  height are vertical distances based off bottom right corner

chr = 'choose state generation method';
chr = [chr newline 'number of states: 0'];
statesInformation = uilabel(fig,'Position',[50 50 250 55],...
```

```
    'Text',chr,...
    'BackgroundColor', [0.7 0.7 0.7]);

% fields for limit parameters
DepthLimit = uieditfield(fig,'numeric',...
    'Position',[300 300 50 22] , 'Enable','off');
DepthLimitlbl = uilabel(fig,'Position',[355 295 150 22]...
    ,'Text','maxDepthLevel');
DepthLimit.Value = 2;
ValueLimit = uieditfield(fig,'numeric',...
    'Position',[300 270 50 22], 'Enable','off');
ValueLimitlbl = uilabel(fig,'Position',[355 265 100 22]...
    ,'Text','maxValue');
ValueLimit.Value = 2;

% field for excluding transitions
ExcludeList = uieditfield(fig,...
    'Position',[50 215 100 22] ,  'Enable','off');
ExcludeListbl = uilabel(fig,'Position',[160 210 100 22]...
    ,'Text','excludeList');
ExcludeList.Value = '1 2 3 4 5';
ExcludeListCbx = uicheckbox(fig , 'Position', [50 235 150 22] ,'Text','exclude
transitions'...
    ,'ValueChangedFcn',@(btn,event) excludeToggle(btn,event) );

makeStatesBtn = uibutton(fig,'push',...
    'Position',[50 150 100 20],...
    'Text', 'Make States',...
    'ButtonPushedFcn', @(btn,event) makeStatesButton(btn,event) );

drawBtn = uibutton(fig,'push',...
    'Position',[320 50 100 20],...
    'Text', 'draw',...
    'Enable','off',...
    'ButtonPushedFcn', @(btn,event) drawStatesButton(btn,event) );

excleList=[]
maxSpaceUsed=10;
typeChoice=''

%------------------------ functions --------------------
    function makeStatesButton(btn,event)
        % method for making states based on the chosen options
        excleList=[];

        switch generationChoice
            case 'coverability'

                disp('coverability')
                if ExcludeListCbx.Value == 1
                    disp('checkbox')
```

```
                    excleList=str2num(ExcludeList.Value);
                end
                coverType='normal'
                States=[];
                [makeRes1,makeRes2]
=makeStatesCover(homeMarking,subtractTransitions,addTransitions,excleList,coverType);
                States=makeRes1;
                maxSpaceUsed=makeRes2;

            case 'coverability_extra'
                disp('coverability_extra')
                if ExcludeListCbx.Value == 1
                    disp('checkbox')
                    excleList=str2num(ExcludeList.Value);
                end

                coverType='extra'
                [makeRes1,makeRes2]
=makeStatesCover(homeMarking,subtractTransitions,...
                    addTransitions,excleList,coverType);
                States=makeRes1;
                maxSpaceUsed=makeRes2;

            case 'reachabilityWithDepthLimit'
                disp('reachabilityWithDepthLimit')
                States=[];

                if ExcludeListCbx.Value == 1
                    disp('checkbox')
                    excleList=str2num(ExcludeList.Value);
                end

                depLim=DepthLimit.Value;
                initialHistory=[];

                [makeRes1,makeRes2]=makeStatesDepthLim(homeMarking ,
subtractTransitions, addTransitions, depLim,...
                    excleList,extran,initval,initialHistory);
                States=makeRes1;
                maxSpaceUsed=makeRes2;

            case 'reachabilityWithValueLimit'
                disp('reachabilityWithValueLimit')
                States=[]

                if ExcludeListCbx.Value == 1
                    disp('checkbox')
                    excleList=str2num(ExcludeList.Value);
                end
                disp(ValueLimit.Value)
```

```
                valLim=ValueLimit.Value;
                initialHistory=[];
                [makeRes1,makeRes2] =makeStatesPlaceValLim(homeMarking ,
subtractTransitions, addTransitions,...

valLim,excleList,extraTransitionValues,extraInitialValues,initialHistory);
                States=makeRes1;
                maxSpaceUsed=makeRes2;
            otherwise
                disp('error')
        end

        part = num2str(length(States));
        part = strcat('length:', part);
        part = [generationChoice newline part];
        if length(States) > 800
            part = [part newline 'warning: navigating figure can be slow'];
        end
        statesInformation.Text=part;

        if length(States)>0
            set(drawBtn,'Enable','on');
        end

    end

    function bselection(bg,event )
        % method for radio buttons, sets generationChoice value to be used elsewhere
        % activates relevant field, deactivate others.
        generationChoice=event.NewValue.Text;
        switch event.NewValue.Text
            case 'reachabilityWithDepthLimit'
                set(DepthLimit ,'Enable','on');
            case 'reachabilityWithValueLimit'
                set(ValueLimit,'Enable','on');
        end
        switch event.OldValue.Text
            case 'reachabilityWithDepthLimit'
                set(DepthLimit ,'Enable','off');
            case 'reachabilityWithValueLimit'
                set(ValueLimit,'Enable','off');
        end
    end

    function excludeToggle(bg,event )
        % method attached to toggle box. activated and deactivates field
        if  event.Value == 1
            set(ExcludeList,'Enable','on');
        elseif event.Value == 0
            set(ExcludeList,'Enable','off');
        end
```

```
    end

    function drawStatesButton(btn,event)
        limitNumber=1;
        % only used in 'reachabilityWithDepthLimit' and 'reachabilityWithValueLimit'

        switch generationChoice % the only difference is between
'reachabilityWithDepthLimit' and 'reachabilityWithValueLimit'
            case 'coverability'
                drawGraphReachLimContinue(States ,subtractTransitions, addTransitions
,...
                    limitNumber,generationChoice,excleList,[]
,maxSpaceUsed,transistionNames )
            case  'coverability_extra'
                drawGraphReachLimContinue(States ,subtractTransitions, addTransitions
,...
                    limitNumber,generationChoice,excleList,[]
,maxSpaceUsed,transistionNames )
            case 'reachabilityWithDepthLimit'
                limitNumber=DepthLimit.Value;
                drawGraphReachLimContinue(States ,subtractTransitions, addTransitions
,...

limitNumber,generationChoice,excleList,extraTransitionValues,maxSpaceUsed,transistionN
ames )
            case 'reachabilityWithValueLimit'
                limitNumber=ValueLimit.Value;
                drawGraphReachLimContinue(States ,subtractTransitions, addTransitions
,...

limitNumber,generationChoice,excleList,extraTransitionValues,maxSpaceUsed,transistionN
ames )
        end

    end

end
```

```
function [Result1 , Result2] = makeStatesCover(homeMarking , subtractTransitions,...
    addTransitions,excludeList,coverType)
% takes five inputs, produces two outputs. Result1 should be the struct list. Result2
should be a numerical value to represent how much space the largest state needs
% input 1 is a list of numbers representing the number of tokens in each
% marking, in the first state. the index number of the value will be
% considered the place name.
% input 2 is a matrix where each row represents the tokens removed from
% each place by a trasition. Contains only negative values or 0
% input 3 is a matrix where each row represents the tokens added to
% each place by a trasition. Contains only positive values or 0
% input 4 is a list of index numbers for transitions to ignore when making
% new states. used in continueFromSelectedMarking
% input 5 is a string for a switch statement to select the type of coverability
algorithm to use (coverability or coverability_extra)
% makes use of calTransWithCoverCheck.m, DecideType.m,
% findActiveTransitions.m and calculateTransition.m
tmp2=size(subtractTransitions);
transitionMaxSub=[];
for i=1:tmp2(2)
    transitionMaxSub=[transitionMaxSub ( max(  abs(subtractTransitions(:, i))  ) +1
)];
end

homeMarkingStringArray=[];
for i = 1:length(homeMarking)
    if homeMarking(i) < inf
        homeMarkingStringArray=[homeMarkingStringArray string(homeMarking(i))];
    else
        homeMarkingStringArray=[homeMarkingStringArray string('W')];
    end

end

States = []; % empty struct array fill with sis elements during run
maxSpaceUsed=0;
%first state
home.parent_marking = NaN;
home.transToThis=NaN;
home.parent_ID = NaN;
home.ID = 1; % should be the same as array index of the struct
home.depth = 1; % use for graphic coordinates
home.marking = homeMarkingStringArray;
home.type = 'N';
home.listOfPrevMarks = [];
home.extraValues=[];
States = [States home]; % add to array
id=1;
activeTransitions = findActiveTransitions(homeMarkingStringArray,
subtractTransitions,excludeList);
```

```
maxSpaceUsed = maxStateSpaceUsed(homeMarkingStringArray, maxSpaceUsed);
queueStructs = [];

for i = 1:length(activeTransitions)
    id = id+1;
    % make child and put into quene
    newSisElement.parent_marking = homeMarkingStringArray;
    newSisElement.parent_ID = 1;
    newSisElement.ID = id;
    newSisElement.transToThis = activeTransitions(i);
    newSisElement.depth = home.depth + 1 ;

    % calculate new marking

    newSisElement.listOfPrevMarks =[homeMarking];
    newSisElement.marking= calTransWithCoverCheck(homeMarking,
newSisElement.parent_marking , ...
        subtractTransitions(activeTransitions(i),1:end) ,
addTransitions(activeTransitions(i),1:end),...
        coverType,transitionMaxSub);
    newSisElement.type = DecideType(newSisElement.marking ,States,queueStructs);


    newSisElement.extraValues=[];

    queueStructs=[queueStructs newSisElement];

end

while length(queueStructs) >0

    %fromQ is the SIS element popped from the queue toQ are new element
    %being made before put into the queue
    fromQ= queueStructs(1); % take element from index 1
    queueStructs(1)= []; %  remove element from index 1
    maxSpaceUsed = maxStateSpaceUsed(fromQ.marking, maxSpaceUsed);
    activeTransitions = findActiveTransitions(fromQ.marking,
subtractTransitions,excludeList);

    if length(activeTransitions) ==0
        fromQ.type = 'E' ;% type E is a deadlock condition
    elseif fromQ.type =='D' % instead of putting duplicates in the queue place in the
main list instead
        States = [States fromQ];
        continue
    end

    States = [States fromQ];

    for i = 1:length(activeTransitions)
        match = false;
```

```
        id = id+1;
        toQ.parent_marking= fromQ.marking;
        toQ.parent_ID = fromQ.ID ;
        toQ.ID = id;
        toQ.transToThis = activeTransitions(i);
        toQ.depth = States(fromQ.ID).depth + 1 ;
        toQ.extraValues=[];

        intMarking=[];
        sizeinput =size(fromQ.marking);
        for j=1:sizeinput(2)
            if char(fromQ.marking(j)) == 'W'
                intMarking = [intMarking inf];
            else
                intMarking = [intMarking str2num(char(fromQ.marking(j)))];
            end
        end

        tmp2=[fromQ.listOfPrevMarks ; intMarking];

        toQ.listOfPrevMarks = tmp2;

        toQ.marking= calTransWithCoverCheck(toQ.listOfPrevMarks, toQ.parent_marking ,
...
            subtractTransitions(activeTransitions(i),1:end) , ...
            addTransitions(activeTransitions(i),1:end),coverType,transitionMaxSub);

        % check to see if duplicate exists, and add marks
        for j = 1:length(States)
            tmp = toQ.marking == States(j).marking;
            if all(tmp(:) > 0)
                match = true;
                tmp=[States(j).listOfPrevMarks ; toQ.listOfPrevMarks];
                tmp2=unique(tmp, 'rows');

                States(j).listOfPrevMarks=tmp2;
            end
        end
        for j = 1:length(queueStructs)
            tmp = toQ.marking == queueStructs(j).marking;
            if all(tmp(:) > 0)
                match = true;
                tmp=[queueStructs(j).listOfPrevMarks ; toQ.listOfPrevMarks];
                tmp2=unique(tmp, 'rows');
                %queueStructs(j).listOfPrevMarks=[queueStructs(j).listOfPrevMarks ;
toQ.listOfPrevMarks];
                queueStructs(j).listOfPrevMarks=tmp2;
            end
        end

        if match
```

```
            toQ.type ='D' ;% duplicate
        else
            toQ.type ='N' ;% normal
        end
        queueStructs=[queueStructs toQ];
    end

end
Result1=States;
Result2=maxSpaceUsed;
End
```

```
function [Result1 , Result2] = makeStatesDepthLim(homeMarking ,
subtractTransitions,...
    addTransitions, depthLimit,...
    excludeList,extraTransitionValues,extraInitialValues,initialHistory)
% takes eight inputs, produces two outputs. Result1 should be the struct list. Result2
should be a numerical value to represent how much space the largest state needs
% input 1 is a list of numbers representing the number of tokens in each
% marking, in the first state. the index number of the value will be
% considered the place name.
% input 2 is a matrix where each row represents the tokens removed from
% each place by a trasition. Contains only negative values or 0
% input 3 is a matrix where each row represents the tokens added to
% each place by a trasition. Contains only positive values or 0
% input 4 is a number. It is the maximum value used by this algorithm
% input 5 is a list of index numbers for transitions to ignore when making
% new states. used in continueFromSelectedMarking
% input 6 is a matrix, containing the extra costs for each transition.
% input 7  is a list of numbers with the extra values of the first state.
% input 8 is the transition history of the first state.
% this will have elements if this tree is the continuation of an already made tree.
% makes use of calculateTransition.m, DecideType.m,
% findActiveTransitions.m and calculateTransition.m

homeMarkingStringArray=[];
for i = 1:length(homeMarking)
    homeMarkingStringArray=[homeMarkingStringArray string(homeMarking(i))];
end

States = []; % empty struct array fill with sis elements during run

maxSpaceUsed=0;

%first state
home.parent_marking = NaN;
home.transToThis=NaN;
```

```
home.parent_ID = NaN;
home.ID = 1; % should be the same as array index of the struct
home.depth = 1; % use for graphic coordinates
home.marking = homeMarkingStringArray;
home.type = 'N';
if length(extraInitialValues)>0
    home.extraValues=extraInitialValues;
end
home.history=initialHistory;

maxSpaceUsed = maxStateSpaceUsed(homeMarkingStringArray, maxSpaceUsed);

States = [States home]; % add to array
id=1;
activeTransitions = findActiveTransitions(homeMarkingStringArray,
subtractTransitions,excludeList);

queueStructs = []; % use for BFS search

for i = 1:length(activeTransitions)
    id = id+1;
    % make child and put into quene
    newSisElement.parent_marking = homeMarkingStringArray;
    newSisElement.parent_ID = 1;
    newSisElement.ID = id;
    newSisElement.transToThis = activeTransitions(i);
    newSisElement.depth = States(newSisElement.parent_ID).depth + 1 ;

    % calculate new marking
    newSisElement.marking= calculateTransition(newSisElement.parent_marking ,
subtractTransitions(activeTransitions(i),1:end) ,
addTransitions(activeTransitions(i),1:end));
    newSisElement.type = DecideType(newSisElement.marking ,States,queueStructs);

    if length(extraInitialValues)>0

newSisElement.extraValues=home.extraValues+extraTransitionValues(activeTransitions(i)
, :);
    end
    newSisElement.history=[initialHistory activeTransitions(i)];

    queueStructs=[queueStructs newSisElement];

end

while length(queueStructs) >0
    %fromQ is the SIS element popped from the queue toQ are new element
    %being made before put into the queue
    fromQ= queueStructs(1); % take element from index 1
    queueStructs(1)= []; %  remove element from index 1
    maxSpaceUsed = maxStateSpaceUsed(fromQ.marking, maxSpaceUsed);
```

```
    activeTransitions = findActiveTransitions(fromQ.marking,
subtractTransitions,excludeList);
    if length(activeTransitions) ==0
        fromQ.type = 'E' ;% type E is a deadlock condition
    elseif fromQ.type =='D' % instead of putting duplicates in the queue place in the
main list instead
        States = [States fromQ];
        continue
    end

    if (fromQ.depth + 1) > depthLimit
        States = [States fromQ];
        continue
    end

    States = [States fromQ];

    for i = 1:length(activeTransitions) % make states to put into queue
        match = false;
        id = id+1;
        toQ.parent_marking= fromQ.marking;
        toQ.parent_ID = fromQ.ID ;
        toQ.ID = id;
        toQ.transToThis = activeTransitions(i);

        toQ.depth = States(fromQ.ID).depth + 1 ;

        toQ.marking= calculateTransition(toQ.parent_marking ,
subtractTransitions(activeTransitions(i),1:end) ,
addTransitions(activeTransitions(i),1:end));
        if length(extraInitialValues)>0

toQ.extraValues=fromQ.extraValues+extraTransitionValues(activeTransitions(i) , :);
        end
        toQ.history=[fromQ.history activeTransitions(i)]; % a history of all
transitions up to this point

        % check to see if duplicate exists
        for j = 1:length(States)
            tmp = toQ.marking == States(j).marking;
            if all(tmp(:) > 0)
                match = true;
                break
            end
        end
        for j = 1:length(queueStructs)
            tmp = toQ.marking == queueStructs(j).marking;
            if all(tmp(:) > 0)
                match = true;
                break
```

```
            end
        end

        if match
            toQ.type ='D' ;% duplicate
        else
            toQ.type ='N' ;% normal
        end
        queueStructs=[queueStructs toQ];
    end

end

Result1=States;
Result2=maxSpaceUsed;
end
```

```
function [Result1 , Result2] = makeStatesPlaceValLim(homeMarking ,
subtractTransitions, addTransitions,...
    valueLimit,excludeList,extraTransitionValues,extraInitialValues,initialHistory)
% takes eight inputs, produces two outputs. Result1 should be the struct list. Result2
should be a numerical value to represent how much space the largest state needs
% input 1 is a list of numbers representing the number of tokens in each
% marking, in the first state. the index number of the value will be
% considered the place name.
% input 2 is a matrix where each row represents the tokens removed from
% each place by a trasition. Contains only negative values or 0
% input 3 is a matrix where each row represents the tokens added to
% each place by a trasition. Contains only positive values or 0
% input 4 is a number. It is the maximum value used by this algorithm
% input 5 is a list of index numbers for transitions to ignore when making
% new states. used in continueFromSelectedMarking
% input 6 is a matrix, containing the extra costs for each transition.
% input 7  is a list of numbers with the extra values of the first state.
% input 8 is the transition history of the first state.
% this will have elements if this tree is the continuation of an already made tree.
% makes use of calculateTransition.m, DecideType.m,
% findActiveTransitions.m and calculateTransition.m

homeMarkingStringArray=[];
for i = 1:length(homeMarking)
    homeMarkingStringArray=[homeMarkingStringArray string(homeMarking(i))];
end

States = []; % empty struct array fill with sis elements during run
maxSpaceUsed=0;

%first state
home.parent_marking = NaN;
```

```
home.transToThis=NaN;
home.parent_ID = NaN;
home.ID = 1; % should be the same as array index of the struct
home.depth = 1; % use for graphic coordinates
home.marking = homeMarkingStringArray;
home.type = 'N';

maxSpaceUsed = maxStateSpaceUsed(homeMarkingStringArray, maxSpaceUsed);
if length(extraInitialValues)>0
    home.extraValues=extraInitialValues;
end
home.history=initialHistory;

States = [States home]; % add to array
id=1;
activeTransitions = findActiveTransitions(homeMarkingStringArray,
subtractTransitions,excludeList);

queueStructs = [];

for i = 1:length(activeTransitions)
    id = id+1;
    % make child and put into quene
    newSisElement.parent_marking = homeMarkingStringArray;
    newSisElement.parent_ID = 1;
    newSisElement.ID = id;
    newSisElement.transToThis = activeTransitions(i);
    newSisElement.depth = States(newSisElement.parent_ID).depth + 1 ;

    % calculate new marking
    newSisElement.marking= calculateTransition(newSisElement.parent_marking ,
subtractTransitions(activeTransitions(i),1:end) ,
addTransitions(activeTransitions(i),1:end));
    newSisElement.type = DecideType(newSisElement.marking ,States,queueStructs) ;
    if length(extraInitialValues)>0

newSisElement.extraValues=home.extraValues+extraTransitionValues(activeTransitions(i)
, :);
    end
    newSisElement.history=[initialHistory activeTransitions(i)];

    queueStructs=[queueStructs newSisElement];

end

while length(queueStructs) >0
    %fromQ is the SIS element popped from the queue toQ are new element
    %being made before put into the queue
    fromQ= queueStructs(1); % take element from index 1
    queueStructs(1)= []; %  remove element from index 1
    maxSpaceUsed = maxStateSpaceUsed(fromQ.marking, maxSpaceUsed);
```

```
    activeTransitions = findActiveTransitions(fromQ.marking,
subtractTransitions,excludeList);
    if length(activeTransitions) ==0
        fromQ.type = 'E' ;% type E is a deadlock condition
    elseif fromQ.type =='D' % instead of putting duplicates in the queue place in the
main list instead
        States = [States fromQ];
        continue
    end

    if fromQ.type =='D' % instead of putting duplicates in the queue place in the main
list instead
        States = [States fromQ];
        continue
    end

    States = [States fromQ];

    for i = 1:length(activeTransitions) % make states to put into queue
        interrupt=false;
        match = false;
        id = id+1;
        toQ.parent_marking= fromQ.marking;
        toQ.parent_ID = fromQ.ID ;
        toQ.ID = id;
        toQ.transToThis = activeTransitions(i);

        toQ.depth = States(fromQ.ID).depth + 1 ;

        toQ.marking= calculateTransition(toQ.parent_marking ,
subtractTransitions(activeTransitions(i),1:end) ,
addTransitions(activeTransitions(i),1:end));
        if length(extraInitialValues)>0

toQ.extraValues=fromQ.extraValues+extraTransitionValues(activeTransitions(i) , :);
        end
        toQ.history=[fromQ.history activeTransitions(i)]; % a history of all
transitions up to this point

        % if one of the values in the marking exceeds the limit, this
        % state will not be put into the queue

        for j = 1:length(toQ.marking)
            tmp=char(toQ.marking(j));
            if valueLimit< str2num(tmp)
                interrupt=true;
            end
        end
        if interrupt==true
            interrupt=false;
```

```
            id = id-1;
            continue
        end

        % check to see if duplicate exists
        for j = 1:length(States)
            tmp = toQ.marking == States(j).marking;
            if all(tmp(:) > 0)
                match = true;
                break
            end
        end
        for j = 1:length(queueStructs)
            tmp = toQ.marking == queueStructs(j).marking;
            if all(tmp(:) > 0)
                match = true;
                break
            end
        end

        if match
            toQ.type ='D' ;% duplicate
        else
            toQ.type ='N' ;% normal
        end
        queueStructs=[queueStructs toQ];
    end

end
Result1=States;
Result2=maxSpaceUsed;
end
```

```
function Result =maxStateSpaceUsed(marking, peviousMax)
% calculate the space that is needed to display a marking as a numerical value, and
compares it with a previous maximum value
markinglenght=0;
spaceUsed=0;
elementCount=0;
for ii =1:length(marking)
    z=  str2num(char(marking(ii)));
    zz=char(marking(ii));

    if zz=='W'
        elementCount=elementCount+1;
        if elementCount== 1
            spaceUsed=spaceUsed+3;
        elseif elementCount> 1
            spaceUsed=spaceUsed+6;
```

```
        end
        markinglenght = markinglenght+2;
    end

    if z >0
        elementCount=elementCount+1;
        if elementCount== 1
            spaceUsed=spaceUsed+3;
        elseif elementCount> 1
            spaceUsed=spaceUsed+6;
        end
        markinglenght = markinglenght+ length( char(marking(ii))  );
    end
end

tmp= markinglenght + spaceUsed;

if peviousMax>tmp
    Result=peviousMax;
else
    Result=tmp;
end

end
```

```
function Result = findActiveTransitions(marking,subTransMatrix,excludeList)
% takes an array of strings as the marking.
% uses a marking and a matrix of the subtraction values of all
% transitions, and returns the index values of all possible transitions
% from the given marking

intMarking=[];
sizeinput =size(marking);
for i=1:sizeinput(2)
    if char(marking(i)) == 'W'
        intMarking = [intMarking inf];
    else
        intMarking = [intMarking str2num(char(marking(i)))];
    end
end

transitions=[];
for i=1:size(subTransMatrix)

    x=( intMarking + subTransMatrix(i,1:end));
    y = x >= 0 ;
    y= all(y(:) > 0);

    if length(excludeList)>0
```

```
        if ismember(i, excludeList(:))
            continue
        end
    end

    if y > 0
        transitions = [transitions i];
    end
end

Result = transitions   ;
end
```

```
function Result = DecideType(marking ,States,queueStructs)
%returns the type as a char: N or  D (normal , duplicate)
%takes a marking and a list of the currently queued structs
type ='N';
match = false;
for j = 1:length(States)
    tmp = marking == States(j).marking;
    if all(tmp(:) > 0)
        match = true;
        break
    end
end
for j = 1:length(queueStructs)
    tmp = marking == queueStructs(j).marking;
    if all(tmp(:) > 0)
        match = true;
        break
    end
end
if match
    type ='D' ;% duplicate
else
    type ='N' ;% normal
end
Result=type;

end
```

```
function Result = calTransWithCoverCheck(listOfPrevMarks,
parent_marking,subTransVal,addTransVal,coverType,transitionMaxSub)
% takes an array of strings as the marking
% takes matrices of the subtraction and addition values of all transitions
% returns an array of strings as the new marking
```

```
intMarking=zeros(size(parent_marking));
sizeinput =size(parent_marking);
for i=1:sizeinput(2)
    if char(parent_marking(i)) == 'W'
        intMarking(i)= inf;
    else
        intMarking(i) = str2num(char(parent_marking(i)));
    end
end

intNewMarking=intMarking+subTransVal+addTransVal;

%adds inf(omega) if applicable


prevMarksDim = size(listOfPrevMarks);

for i = 1: prevMarksDim(1)

    intPrevMark=listOfPrevMarks(i,:);

    tmp= intNewMarking>=intPrevMark;


    if all(tmp(:) >0)
        for j=1:length(intPrevMark)

            if strcmp(coverType,'extra')
                if (intNewMarking(j) > intPrevMark(j)) &
(intNewMarking(j)>=transitionMaxSub(j) ) % new ad
                    intNewMarking(j)= inf;
                end
            else

                if (intNewMarking(j) > intPrevMark(j))

                    intNewMarking(j)= inf;
                end
            end

        end
    end

end

newMarkingStrings=[];
for i=1:sizeinput(2)
    if intNewMarking(i) == inf
        newMarkingStrings=[newMarkingStrings string('W')];
    else
        newMarkingStrings=[newMarkingStrings string(num2str(intNewMarking(i)))];
```

```
    end
end

Result=newMarkingStrings;
end
```

```
function Result = calculateTransition(marking,subTransVal,addTransVal)
% Takes three inputs, produces one output, calculats new marking from previous marking
and the transition marices
% takes an array of strings as the marking
% takes matrices of the subtraction and addition values of all transitions
% returns an array of strings as the new marking

intMarking=zeros(size(marking));
sizeinput =size(marking);
for i=1:sizeinput(2)
    if char(marking(i)) == 'W'
        intMarking(i)= inf;
    else
        intMarking(i) = str2num(char(marking(i)));
    end
end
newMarking=intMarking+subTransVal+addTransVal;
newMarkingStrings=[];
for i=1:sizeinput(2)
    if newMarking(i) == inf
        newMarkingStrings=[newMarkingStrings string('W')];
    else
        newMarkingStrings=[newMarkingStrings string(num2str(newMarking(i)))];

    end
end

Result=newMarkingStrings;
end
```

```
function drawGraphReachLimContinue(States ,subtractTransitions, addTransitions,
limitNumber,...
    limitType,excleList,TransExtraValues,maxSpaceUsed,transistionNames )
% Takes nine inputs, makes a figure
% this method is called from petNetCoverTreeGUI.m and itself.
% The inputs need to be in a specific order
% input 1 is a list of structs that are made by makeStatesCover.m or
% makeStatesDepthLim.m or makeStatesPlaceValLim.m
% input 2 is a matrix where each row represents the tokens removed from
% each place by a trasition. Contains only negative values or 0
% input 3 is a matrix where each row represents the tokens added to
% each place by a trasition. Contains only positive values or 0
% input 4 is used to make new states by the continueFromSelectedMarking
% internal method. it is a number.
% input 5 is used to make new states by the continueFromSelectedMarking
% internal method. it is a string to a switch statement.
% input 6 is a list of index numbers for transitions to ignore when making
% new states. used in continueFromSelectedMarking
% input 7 is a matrix, containing the extra costs for each transition.
% input 8 is a result from makeStatesCover.m or makeStatesDepthLim.m or
makeStatesPlaceValLim.m
% it is used to decide the size of the state rectangles
% input 9 is an array of strigs that need to have as many elements as there
% are rows in input 2 and 3. It is the names fo each transition.
% makes use of makeStatesCover.m, makeStatesDepthLim.m and makeStatesPlaceValLim.m



tnames=transistionNames;
% stae measurements
len= 5+ maxSpaceUsed*12 +5; % the lenght of a state rectangle
heig = 50; % % vertical height of the boxes
VertSpace=150; % vertical space between levels of states
horizSpace=20; % horizontal space bettween states

transitionArrow_headsize_ajuster=500;

adjustment= 8000; % used for ajusting the relative size of text in states

transNameMaxSize=0; % get the biggest transition name
for i =1:length(transistionNames)
    tmp=string(transistionNames(i));
    if length(char(tmp))>transNameMaxSize
        transNameMaxSize=length(char(tmp));
    end
end

tmp= size(TransExtraValues);
numberOfExtraValuesPerState=tmp(2);
```

```
singleExtraLen=50; % the lenght of one extra state rectangle
totExtraLen = numberOfExtraValuesPerState*singleExtraLen;

numberOfLevels=0; % find the max depth of the tree
for i = 1 :length(States)
    if States(i).depth > numberOfLevels
        numberOfLevels=States(i).depth;
    end
end

numberOfStatesAtEachLevel= zeros(1,numberOfLevels);  % get number of states at each
level
for i = 1 :length(States)

numberOfStatesAtEachLevel(States(i).depth)=numberOfStatesAtEachLevel(States(i).depth)+
1;
end
[valOfMaxLev , indexOfMaxLev] =max(numberOfStatesAtEachLevel);

% horisontally the graph is centered on the zero coordinate

maxLen= valOfMaxLev*(len+horizSpace+totExtraLen);

spaceOfLevel= zeros(1,numberOfLevels);
for i = 1 :length(spaceOfLevel)
    if numberOfStatesAtEachLevel(i) == valOfMaxLev
        spaceOfLevel(i)=horizSpace ;
    else
        adj=((numberOfStatesAtEachLevel(i)-1)*(len+horizSpace+totExtraLen))/100 ;
        spaceOfLevel(i)=horizSpace + (maxLen/(adj)) ;
    end
end

horisontalCoordinate= zeros(1,numberOfLevels);
for i = 1 :length(horisontalCoordinate)
    if numberOfStatesAtEachLevel(i) >1
        totalLengthOfDepthLevel= (numberOfStatesAtEachLevel(i) -
1)*(len+spaceOfLevel(i)+totExtraLen);
        horisontalCoordinate(i)= 0 - (totalLengthOfDepthLevel/2 );
    end
end


f =figure;
movegui(f,'northeast');
axes('Units', 'normalized', 'Position', [0 0 1 1])

% get axes size for text zoom function
axis([ (min(horisontalCoordinate)-len) , (abs(min(horisontalCoordinate))+len) , (-
(numberOfLevels*300)-heig) , (-300+heig) ]);
```

```
ax = axis;


%each state is given coordinates
for i = 1 :length(States) % adds central coordinates to eache state
    if numberOfStatesAtEachLevel(States(i).depth) ==1
        States(i).coordinates=[0 -States(i).depth*(heig+ VertSpace)] ; % if there is
only one state at his level, the horizontal coordinate will be 0
    elseif numberOfStatesAtEachLevel(States(i).depth) >1
        States(i).coordinates=[horisontalCoordinate(States(i).depth) -
States(i).depth*(heig+ VertSpace)] ;
        horisontalCoordinate(States(i).depth)= horisontalCoordinate(States(i).depth)
+(len+ spaceOfLevel(States(i).depth)+totExtraLen);
    end

end


hold on;
for i = 1 :length(States) % draw rectangles
    switch States(i).type % choose colour based on type
        case 'N'
            colour= [0.5 1 0.5];
        case 'D'
            colour= [1 1 0];
        case 'E'
            colour= [1 0.5 0.5];
        otherwise
            disp('error')
    end
    rek=rectangle('Position',[States(i).coordinates(1)-(len/2) ,
States(i).coordinates(2)-(heig/2)...
        , len , heig ],'FaceColor',colour , 'ButtonDownFcn' ,@markStateMarking );

    rek.UserData = States(i).ID;
    States(i).drawStateRectangle=rek;

    States(i).exValRecktangles=[];
    States(i).exValTxt=[];

    tmp3= size(TransExtraValues);
    for j= 1:tmp3(2)
        tmpRek=rectangle('Position',[States(i).coordinates(1)+(len/2)+(j-
1)*singleExtraLen , States(i).coordinates(2)-(heig/2)...
            , singleExtraLen , heig ],'FaceColor','none' , 'ButtonDownFcn'
,@markStateMarking );
        tmpRek.UserData = States(i).ID;
        States(i).exValRecktangles=[States(i).exValRecktangles tmpRek];

        tmptxt=text( States(i).coordinates(1)-(len/2)+(j-1)*singleExtraLen + len+5 ,
States(i).coordinates(2) ,...
```

```
            char(string(States(i).extraValues(j) )) , 'ButtonDownFcn'
,@markStateMarking );
        tmptxt.FontSize= ((adjustment/1.1)/(ax(4)-ax(3)));
        tmptxt.UserData=States(i).ID;
        States(i).exValTxt=[States(i).exValTxt tmptxt];

    end

    txtChar='';
    elementCount=0;
    for j = 1:length(States(i).marking)
        tmp= str2num ( char(States(i).marking(j)) );
        if char(States(i).marking(j)) == 'W'
            elementCount=elementCount+1;
            if elementCount ==1
                txtChar= strcat( txtChar , '\omega' , 'p_' , num2str(j) ) ;
            elseif elementCount >1
                tmp2=strcat({' + '} , {'\omega'}, {'p_' }, {num2str(j)}  ) ;
                txtChar=strcat(txtChar ,tmp2 );
            end
        elseif tmp>0
            elementCount=elementCount+1;
            if elementCount ==1
                txtChar= strcat( txtChar , char(States(i).marking(j)) , 'p_' ,
num2str(j) ) ;
            elseif elementCount >1
                tmp2=strcat({' + '} , { char(States(i).marking(j)) }, {'p_' },
{num2str(j)}  ) ;
                txtChar=strcat(txtChar ,tmp2 );
            end
        end
    end

    txt=text( States(i).coordinates(1)-(len/2)+5 , States(i).coordinates(2) ,...
        char( txtChar ) , 'ButtonDownFcn' ,@markStateMarking );

    txt.FontSize= ((adjustment/1.1)/(ax(4)-ax(3)));
    txt.UserData=States(i).ID;



    States(i).drawStateText=txt;

end


% drawing transition elements
% the vertical level of the textboxes for duplicate transitions are a little
% different than the other transition textboxes. This is to detect
% overlapping boxes when showing redirected transitions.
transGroup1=[];% arrows that point to normal states, these are always visible
```

```
transGroup2=[];% arrows that point to duplicate states, these are visible to begin
with
transGroup3=[];% redirected arrows that point to normal states, these are made hidden
to begin with
% all groups contain a list of structs that contain, a quiver arrow, a
% rectangle and text
for i = 1 :length(States) % draw transition arrows
    if ~isnan(States(i).parent_ID); % only draw is the state has a parent state
        p1 = States(States(i).parent_ID).coordinates; % starting point
        p1(2) = p1(2)-heig/2;
        p2 = States(i).coordinates; %ending point
        p2(2) = p2(2)+heig/2;
        dp = p2-p1; % difference
        arrw=quiver(p1(1),p1(2),dp(1),dp(2),0,'color',[0 0 1]);
        % quiver arrows have different sizes depending on the lenght of the arrow
        % divide the quiver property MaxHeadSize by the length of the arrow, then the
heads will be the same size.
        x=[p1(1) p2(1)];
        y=[p1(2) p2(2)];
        d = diff([x(:) y(:)]);
        total_length = sum(sqrt(sum(d.*d,2)));
        arrw.MaxHeadSize =
(arrw.MaxHeadSize/total_length)*transitionArrow_headsize_ajuster;

        tRekLen=transNameMaxSize*12; % the lenght of a transition rectangle

        rek=rectangle('Position',[p1(1)+(dp(1)/1.4)-tRekLen/2 ,  p1(2)+(dp(2)/1.4)-15
...
            , tRekLen , 32 ],...
            'FaceColor','w');

        transText = char(transistionNames(States(i).transToThis));

        drawnTxt=text( p1(1)+(dp(1)/1.4)-tRekLen/2+3 , p1(2)+(dp(2)/1.4)+15-15 ...
            ,transText , 'Color', 'k' );
        % choose what group to put them in
        switch States(i).type
            case 'N'
                arrw.Color =      [0 0.3 0];
                drawnTxt.Color =  [0 0.3 0];
                trans.arrow= arrw;
                trans.rektangle= rek;
                trans.text= drawnTxt;
                transGroup1=[transGroup1 trans];
            case 'E'
                arrw.Color =      [0.3 0 0];
                drawnTxt.Color =  [0.3 0 0];
                trans.arrow= arrw;
                trans.rektangle= rek;
                trans.text= drawnTxt;
                transGroup1=[transGroup1 trans];
```

```
            case 'D' % for every duplicate, make two arrows,
                arrw.Color = [0.5 0.5 0];
                drawnTxt.Color =  [0.5 0.5 0];
                trans.arrow= arrw;
                trans.rektangle= rek;
                trans.text= drawnTxt;
                transGroup2=[transGroup2 trans];

                p1 = States(States(i).parent_ID).coordinates; % p1 will be the same
regardless
                p1(2) = p1(2)-heig/2;
                colour=[0.5 0.5 0];
                for j = 1 :length(States)
                    tmp = States(i).marking == States(j).marking;
                    if (all(tmp(:) > 0)) &&  (States(j).type == 'N' | States(j).type
== 'E')
                        p2 = States(j).coordinates;
                        p2(2) = p2(2)+heig/2;

                        break
                    end
                end
                dp = p2-p1;
                arrw=quiver(p1(1),p1(2),dp(1),dp(2),0,'color',colour); % draws the
arrow
                x=[p1(1) p2(1)];
                y=[p1(2) p2(2)];
                d = diff([x(:) y(:)]);
                total_length = sum(sqrt(sum(d.*d,2)));
                arrw.MaxHeadSize =
(arrw.MaxHeadSize/total_length)*transitionArrow_headsize_ajuster;

                rek=rectangle('Position',[p1(1)+(dp(1)/1.8)-tRekLen/2 ,
p1(2)+(dp(2)/1.8)-15 ...
                    , tRekLen , 32 ],...
                    'FaceColor','w');

                transText = char(transistionNames(States(i).transToThis));

                drawnTxt=text( p1(1)+(dp(1)/1.8)-tRekLen/2+3 , p1(2)+(dp(2)/1.8)+15-15
,...
                    transText , 'Color',[0.5 0.5 0] );

                arrw.Visible='off';
                rek.Visible='off';
                drawnTxt.Visible='off';
                trans.arrow= arrw;
                trans.rektangle= rek;
                trans.text= drawnTxt;
                transGroup3=[transGroup3 trans];
```

```
            otherwise
                disp('error')
        end
    end
end

% buttons for the bottom row
hideDupButton = uicontrol('Style', 'pushbutton', 'String', 'hide duplicates',...
    'Position', [10 10 120 30],...
    'FontSize', 11,...
    'Callback', @hideDuplicateButton);

showDupButton = uicontrol('Style', 'pushbutton', 'String', 'show duplicates',...
    'Position', [130 10 120 30],...
    'FontSize', 11,...
    'Callback', @showDuplicateButton);

searchButton = uicontrol('Style', 'pushbutton', 'String', 'search',...
    'Position', [250 10 80 30],...
    'FontSize', 11,...
    'Callback', @searchForMarking);

continueButton = uicontrol('Style', 'pushbutton', 'String', 'continue',...
    'Position', [330 10 80 30],...
    'FontSize', 11,...
    'Callback', @continueFromSelectedMarking);

axis equal

h = zoom; % get handle to zoom utility
set(h,'ActionPostCallback',@zoomCallBack);
set(h,'Enable','on');

ax = axis;

for ii =1:length(States)

    set(States(ii).drawStateText,'FontSize',adjustment/(ax(4)-ax(3)));

    for jj= 1: length(States(ii).exValRecktangles)
        set(States(ii).exValTxt(jj),'FontSize',adjustment/(ax(4)-ax(3)));
    end
end

for ii =1:length(transGroup1)
    set(transGroup1(ii).text ,'FontSize',adjustment/(ax(4)-ax(3)) );
end
for ii =1:length(transGroup2)
    set(transGroup2(ii).text ,'FontSize',adjustment/(ax(4)-ax(3)) );
end
for ii =1:length(transGroup3)
```

```
    set(transGroup3(ii).text ,'FontSize',adjustment/(ax(4)-ax(3)) );
end

%--------------------functions-----------

% everytime you zoom , this function is executed
    function zoomCallBack(~, evd)

        ax = axis(evd.Axes); % get axis size


        for ii =1:length(States)

            set(States(ii).drawStateText,'FontSize',adjustment/(ax(4)-ax(3)));

            for jj= 1: length(States(ii).exValRecktangles)
                set(States(ii).exValTxt(jj),'FontSize',adjustment/(ax(4)-ax(3)));
            end

        end

        for ii =1:length(transGroup1)
            set(transGroup1(ii).text ,'FontSize',adjustment/(ax(4)-ax(3)) );
        end
        for ii =1:length(transGroup2)
            set(transGroup2(ii).text ,'FontSize',adjustment/(ax(4)-ax(3)) );
        end
        for ii =1:length(transGroup3)
            set(transGroup3(ii).text ,'FontSize',adjustment/(ax(4)-ax(3)) );
        end

    end

    function hideDuplicateButton(btn,event) % use to hide group 2 and show group 3

        for ii = 1 :length(transGroup2)
            % arrows that point to duplicate states, these are visible to begin with
            transGroup2(ii).arrow.Visible='off';
            transGroup2(ii).rektangle.Visible='off';
            transGroup2(ii).text.Visible='off';
        end
        for ii = 1 :length(transGroup3)
            % redirected arrows that point to duplicate states, these are made hidden
to begin with
            transGroup3(ii).arrow.Visible='on';
            transGroup3(ii).rektangle.Visible='on';
            transGroup3(ii).text.Visible='on';
        end

        for ii = 1 :length(States)
            if States(ii).type == 'D'
```

```
                States(ii).drawStateText.Visible='off';
                States(ii).drawStateRectangle.Visible='off';
                for jj= 1: length(States(ii).exValRecktangles)
                    States(ii).exValRecktangles(jj).Visible='off';
                    States(ii).exValTxt(jj).Visible='off';
                end
            end
        end

    end

    function showDuplicateButton(btn,event) % use to hide group 3 and show group 2

        for ii = 1 :length(transGroup3)
            % arrows that point to duplicate states, these are visible to begin with
            transGroup3(ii).arrow.Visible='off';
            transGroup3(ii).rektangle.Visible='off';
            transGroup3(ii).text.Visible='off';
        end
        for ii = 1 :length(transGroup2)
            % redirected arrows that point to duplicate states, these are made hidden
to begin with
            transGroup2(ii).arrow.Visible='on';
            transGroup2(ii).rektangle.Visible='on';
            transGroup2(ii).text.Visible='on';
        end

        for ii = 1 :length(States)
            if States(ii).type == 'D'
                States(ii).drawStateText.Visible='on';
                States(ii).drawStateRectangle.Visible='on';
                for jj= 1: length(States(ii).exValRecktangles)
                    States(ii).exValRecktangles(jj).Visible='on';
                    States(ii).exValTxt(jj).Visible='on';
                end
            end
        end
    end


markingOfClickedRectangle=[0]; % variables for selecting states on mouseclick
lastClickedState=States(1)

idOfPrevRectanglesClicked=[];
    function markStateMarking(src,event ) % changes the outline of state rectangles
when clicked

        States(src.UserData).drawStateRectangle.LineWidth = 2;
        for ii = 1:length(idOfPrevRectanglesClicked) % set old markings to default
value
            States(idOfPrevRectanglesClicked(ii)).drawStateRectangle.LineWidth = 0.5;
```

```
        end
        disp(idOfPrevRectanglesClicked)
        idOfPrevRectanglesClicked=src.UserData;
        lastClickedState=States(src.UserData);

        markingOfClickedRectangle=States(src.UserData).marking;
    end

    function searchForMarking(src,event)
        % changes the outline of state rectangles based on full search

        defaultInput=markingOfClickedRectangle(1);
        for ii = 2: length(markingOfClickedRectangle)
            defaultInput=strcat(defaultInput,{' '},markingOfClickedRectangle(ii));
        end
        dialougeAnswer = inputdlg({'input marking to search for'},'search input',[1
40],{char(defaultInput)});

        for ii = 1:length(idOfPrevRectanglesClicked) % set old markings to default
value
            States(idOfPrevRectanglesClicked(ii)).drawStateRectangle.LineWidth = 0.5;
        end

        searchFor = strsplit(char(dialougeAnswer))

        for ii = 1:length(States)
            tmp = searchFor == States(ii).marking;
            if all(tmp(:) > 0)

                States(ii).drawStateRectangle.LineWidth = 2;
                idOfPrevRectanglesClicked=[idOfPrevRectanglesClicked ii];
                markingOfClickedRectangle=States(ii).marking;
            end
        end

    end

    function continueFromSelectedMarking(src,event)
        % method for continuing tree generation from the selected state

        if length(markingOfClickedRectangle)>1
            % convert to int array of values
            intMarking=zeros(size(markingOfClickedRectangle));
            sizeinput =size(markingOfClickedRectangle);
            for i=1:sizeinput(2)
                if char(markingOfClickedRectangle(i)) == 'W'
                    intMarking(i)= inf;
                else
                    intMarking(i) = str2num(char(markingOfClickedRectangle(i)));
                end
            end
```

```
            switch limitType
                case 'coverability'
                    coverType='normal'

                    [States2,maxSpaceUsed2]=
makeStatesCover(intMarking,subtractTransitions,addTransitions,...
                        excleList,coverType);
                    drawGraphReachLimContinue(States2 ,subtractTransitions,
addTransitions, limitNumber,limitType,...
                        excleList,TransExtraValues,maxSpaceUsed2, tnames )

                case 'coverability_extra'
                    coverType='extra'

                    [States2,maxSpaceUsed2]
=makeStatesCover(intMarking,subtractTransitions,addTransitions,....
                        excleList,coverType);
                    drawGraphReachLimContinue(States2 ,subtractTransitions,
addTransitions,...

limitNumber,limitType,excleList,TransExtraValues,maxSpaceUsed2,tnames )

                case 'reachabilityWithDepthLimit'
                    if isfield(lastClickedState ,'extraValues' )
                        extra_initial2=lastClickedState.extraValues;
                    else
                        extra_initial2=[];
                    end

                    %the new home marking will be denoted as depth 1, no need to
change the limit
                    [States2,maxSpaceUsed2]= makeStatesDepthLim(intMarking ,
subtractTransitions, addTransitions, limitNumber,excleList ,...
                        TransExtraValues,extra_initial2 ,lastClickedState.history )

                    drawGraphReachLimContinue(States2 ,subtractTransitions,
addTransitions, limitNumber,...
                        limitType,excleList,TransExtraValues,maxSpaceUsed2,tnames  )


                case 'reachabilityWithValueLimit'

                    if isfield(lastClickedState ,'extraValues' )
                        extra_initial2=lastClickedState.extraValues;
                    else
                        extra_initial2=[];
                    end

                    % add the imit to the highest value in the new home marking
                    newLimitNumber=limitNumber+ max(intMarking);
```

```
                    [States2,maxSpaceUsed2]=makeStatesPlaceValLim(intMarking ,
subtractTransitions, addTransitions,...
                        newLimitNumber, excleList , TransExtraValues,extra_initial2
,lastClickedState.history);

                    drawGraphReachLimContinue(States2 ,subtractTransitions,
addTransitions,...

limitNumber,limitType,excleList,TransExtraValues,maxSpaceUsed2,tnames )
            end

        end

    end

end
```

Published with MATLAB® R2016b