PETRINET SIMULATOR

Program Documentation

# Contents

1 Manual

The Petrinet Simulator is able to display a petrinet and gradually build a reachability graph by firing transitions using the *GraphStream* library[1]. The following picture shows the programs surface:
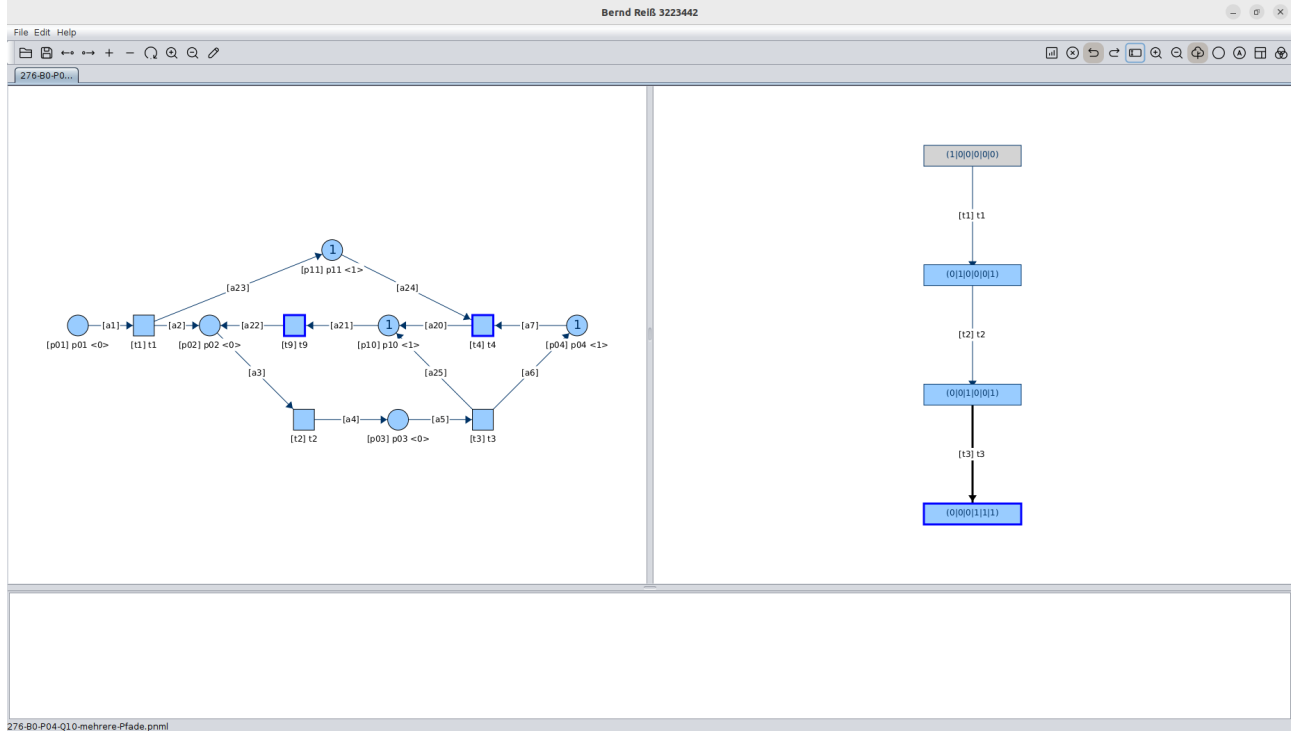
In the petrinet on the left activated transitions are marked bold, these transitions can be fired. On the right in the reachability graph the current state is marked bold. Additionally the initial state is marked gray. After performing the boundedness analysis the graphs look like this:
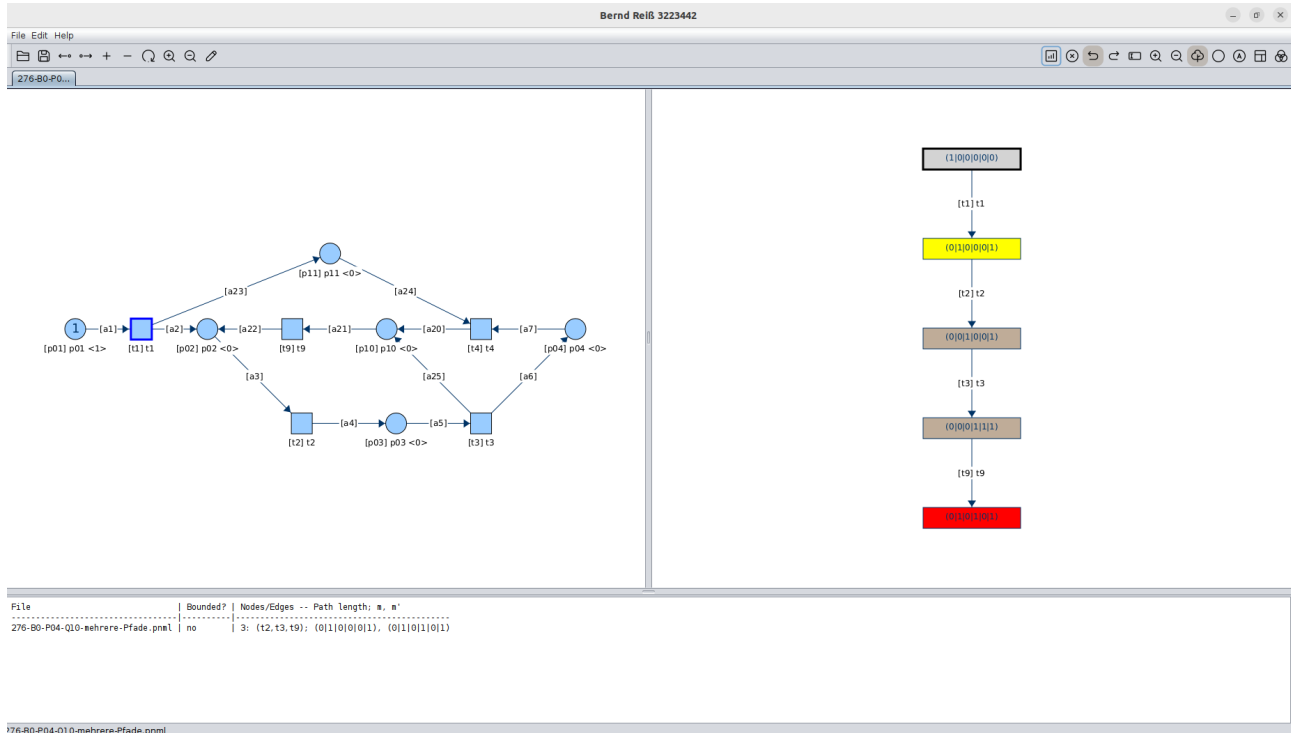
The reachability graph shows whether it encountered a state that signifies that the petrinet is unbounded

---

[1]https://graphstream-project.org/

and marks the beginning (yellow) and ending (red) nodes on the path signifying unboundedness. Any states on the paths between these nodes and between the yellow and intial node are marked also (brown).

The following sections give an overview of additional features implemented in the program and explain the menus and toolbars.

1.1 Additional features

Automatic Boundedness Check By default building the reachability check automatically checks whether the petrinet is bounded or not (see also 2.2). This can be turned off in the menu *Edit*.

Tabs A new petrinet can be opened in a new tab. This can be done via the menu *File*.

Undo/Redo The reachability graph provides undo/redo functionality.

Custom Layouts The Petrinet Simulator provides two custom layouts: tree layout and circle layout (see 2.3). Users can switch between these two and the automatic layout provided by GraphStream.

Editor Mode Petrinets can be edited in an editor mode (see 1.3.2).

1.2 Menus

1.2.1 File

The *File* menu consists of the following entries:

- **New:** opens a new empty file for editing → automatically opens the Editor view (see 1.3.2).

- **Open:** opens saved file from directory.

- **Open in new Tab:** same as open but in a new tab.

- **Reload:** reload the currently opened file.

- **Save:** save changes.

- **Save as:** save changes and choose directory/name.

- **Analyse++:** analyse many petrinets at once. Results are printed in the text area.

- **Close:** close currently opened file.

- **Exit:** close the program.

1.2.2 Edit

The *Edit* menu consists of the following entries:

- **Open Editor:** opens the Editor.

- **Close Editor:** closes the Editor and reverts back to Viewer.

- **Change Look and Feel:** change between Nimbus and Metal look and feel.

- **Enable Automatic Boundedness Check:** automatically check for boundedness when building the reachabilty graph.

- **Disable Automatic Boundedness Check:** do not automatically check for boundedness when building the reachabilty graph.

1.2.3 Help

The *Help* menu only consists of the element *Info*, which shows information about the Java version used and the user directory.

1.3 Toolbar Modes

The program has two distinct toolbar modes: viewer mode and editor mode. This section discusses the two modes and toolbar buttons associated with it. In general the toolbar is fully de- and reattachable (except for the bottom of the program being reserved for the status bar). All icons for the toolbar buttons are taken from `https://iconoir.com/`.

1.3.1 Viewer Mode

The viewer mode provides functionality for viewing petrinets and building the reachability graph. The toolbar is split into two parts: the petrinet toolbar to the left and the reachability graph toolbar to the right.

**Petrinet Toolbar**

From left to right the buttons provide the following functionality:

- **Load:** see Menu *File* 1.2.1.

- **Save:** see Menu *File* 1.2.1.

- **Previous:** open the previous file in the current directory → do nothing if current file is the first.

- **Next:** open the next file in the current directory → do nothing if current file is the last.

- **Increment Place:** increment a marked place by one token.

- **Decrement Place:** decrement a marked place by one token.

- **Reset:** reset the petrinet graph to initial state → if place was in-/decremented the initial state is being updated.

- **Zoom in:** zoom into petrinet graph.

- **Zoom out:** zoom out of patrinet graph.

- **Open Editor:** open the Editor (see 1.3.2).

Reachbility Graph Toolbar

From left to right the buttons provide the following functionality:

- **Analyse Petrinet:** analyse the petrinet currently opened → the reachability graph is built and a pop up window shows whether the petrinet is bounded or unbounded. If the petrinet is unbounded the path signifying unboundedness is marked (see 1).

- **Reset:** delete the reachability graph except for the initial state (marked gray). Also resets the petrinet.

- **Undo:** undo the last step in the reachability graph. If last step was analysis it is treated as one single step.

- **Redo:** redo the last step in the reachability graph. If last step was analysis it is treated as one single step.

- **Clear text area:** clears the text area.

- ⊕ **Zoom in:** zoom into reachability graph.

- ⊖ **Zoom out:** zoom out of reachability graph.

- ⊕ **Tree layout:** organize the reachability graph in a tree layout (see 2.3.1).

- ○ **Circle layout:** organize the reachability graph in a circle layout (see 2.3.2).

- ⊚ **Automatic Layout:** use the auto layout by GraphStream to organize the reachability graph.

- ⊞ **Reset Split Panes:** reset the split panes to the default ratio.

- ⊛ **Change Look and Feel:** change between the different look and feels (*Metal* or *Nimbus*).

1.3.2   Editor Mode

Petrinet Toolbar   ⊟ ⊟ ← → + − ⊕ ⊖ ⊕ ⊞ ◇ ⌒ ⌒ ▭ ⊙

From left to right the buttons provide the following functionality:

- ⊟ **Load:** see Menu *File* 1.2.1.

- ⊟ **Save:** see Menu *File* 1.2.1.

- ← **Previous:** see Petrinet Toolbar in Viewer Mode (1.3.1).

- → **Next:** see Petrinet Toolbar in Viewer Mode (1.3.1).

- + **Increment Place:** see Petrinet Toolbar in Viewer Mode (1.3.1)..

- − **Decrement Place:** see Petrinet Toolbar in Viewer Mode (1.3.1)..

- ⊕ **Zoom in:** see Petrinet Toolbar in Viewer Mode (1.3.1)..

- ⊖ **Zoom out:** see Petrinet Toolbar in Viewer Mode (1.3.1)..

- ⊕ **Add Place:** add a place above the left top most element.

- ⊞ **Add Transition:** add a transition above the left top most element.

- ◇ **Remove Element:** remove the marked element (place or transition).

- ⌒ **Add Edge:** choose beginning and ending node of an edge to add it → beginning and ending nodes have to be of different type (place/transition).

- ⌒ **Remove Edge:** choose beginning and ending node of an edge to remove it → the edge has to exist.

- ▭ **Add Label:** add label to marked place or transition.

- ⊙ **Close Editor:** close the Editor and opens the Viewer (see 1.3.1).

Reachability Graph Toolbar   ⊞ ⊛

From left to right the buttons provide the following functionality:

- ⊞ **Reset Split Panes:** see Reachability Graph Toolbar in Viewer Mode (1.3.1).

- ⊛ **Change Look and Feel:** see Reachability Graph Toolbar in Viewer Mode (1.3.1).

2 Design of the Program

2.1 General Structure

A Petrinet is determined by its places and transitions. The number of tokens on places represent the state of a petrinet and activated transition represent passages between states. These in order are buildings blocks for nodes and edges in the reachability graph. As places and transitions in this implementation of a petrinet simulator know their predecessors and successors they hold all the information needed for a data structure representing a petrinet. Thus the rest of the program essentially revolves around managing places and transitions [2].

On a general level it can be noted that controllers control the input provided via the GUI elements and listeners transfer changes in the data model to the GUI components. The program is split into a set of hierarchichally organized controllers. For a general overview of the flow of control please refer to Figure 1. Double rules signify a GUI component, dashed lines signify interfaces / listeners. The figure demonstrates the basic design of the program and how the essential data models, controllers and GUI components communicate with each other. Other parts of the program not shown in the figure are discussed in the text.

The following sections give a detailed description of how the data models are structured and how controllers and listeners work.

2.1.1 Data Models

Petrinet Model It handles everything belonging only to the petrinet. It has functions for modifying the petrinet (such as adding or removing elements) and checks whether an action is valid. If an action is not valid it throws an according *PetrinetException.* The petrinet model holds helper data structures *Place* and *Transition.* There is no data structure for arcs, as they are fully defined via the places and transitions connected to them [3]. Rather arcs are defined by elements preceding and succeeding places and transitions. Places contain information about the number of tokens stored in them. Transitions have a status of being activated or not activated.

Reachability Graph Model It provides methods for adding/removing states to/from the reachability graph and giving information about the graph. It contains a list of *PetrinetStates.* Such a state contains predecessor and successor states, information about the tokens on the places in the petrinet as well as a state called m, being the first state on a path that marks the given state as unbounded. When a new state is added the model checks whether the new state completes a path signifying unboudedness (see 2.2. If so, the state at the start of the path is remembered by the state and the reachability graph model remembers the last state on the path. The states also store transitions leading up to them and from them. These are stored in lists in a map.

Reachability Graph Undo Queue This queue records all the steps taken in the reachabilty graph and keeps track of the actions being taken (i.e. whether something has been added). To this end it has its own data structure called *ReachabilityGraphUndoQueueState.* The queue can go back or forward or can be rewinded/reset. If it goes backward, alls steps are undone. Going forward means redoing all steps. The reachability model has methods to stop pushing steps onto the queue. Furthermore steps can be recorded as "skippable". A step being skippable means on un-/redo it does not stop but continues going back-/forwards. This enables a multi step process like adding steps when analysing the petrinet to be treated as one big step.

2.1.2 Controllers

There are three controllers in the program: the main controller, the petrinet viewer controller and the petrinet editor controller. Additionally the *PetrinetPanelInterface* serves as an intermediary between the main controller

---

[2]Although in this sense the petinet model can be seen as a kind of controller, it is still considered to be mainly a data structure, just not as pure as places and transitions might be.

[3]For storing ids belonging to arcs a map is used.

Figure 1: General Design of the Program.

The figure shows boxes connected by arrows. From top to bottom: "Main Frame" connects to "Main Controller", which connects to "Petrinet Panel". The "Petrinet Panel" connects to "Petrinet Editor Controller" and "Petrinet Viewer Controller". These connect to "Petrinet Model". The "Petrinet Model" connects to "Transition" and "Place". Dashed arrows labeled "State Ch.", "Compenent Changed", "Token Count Ch.", and "State Ch." connect to "Petrinet Graph" and "Reachability Graph Model". The "Reachability Graph Model" connects via "State Ch." to "Reachability Graph".
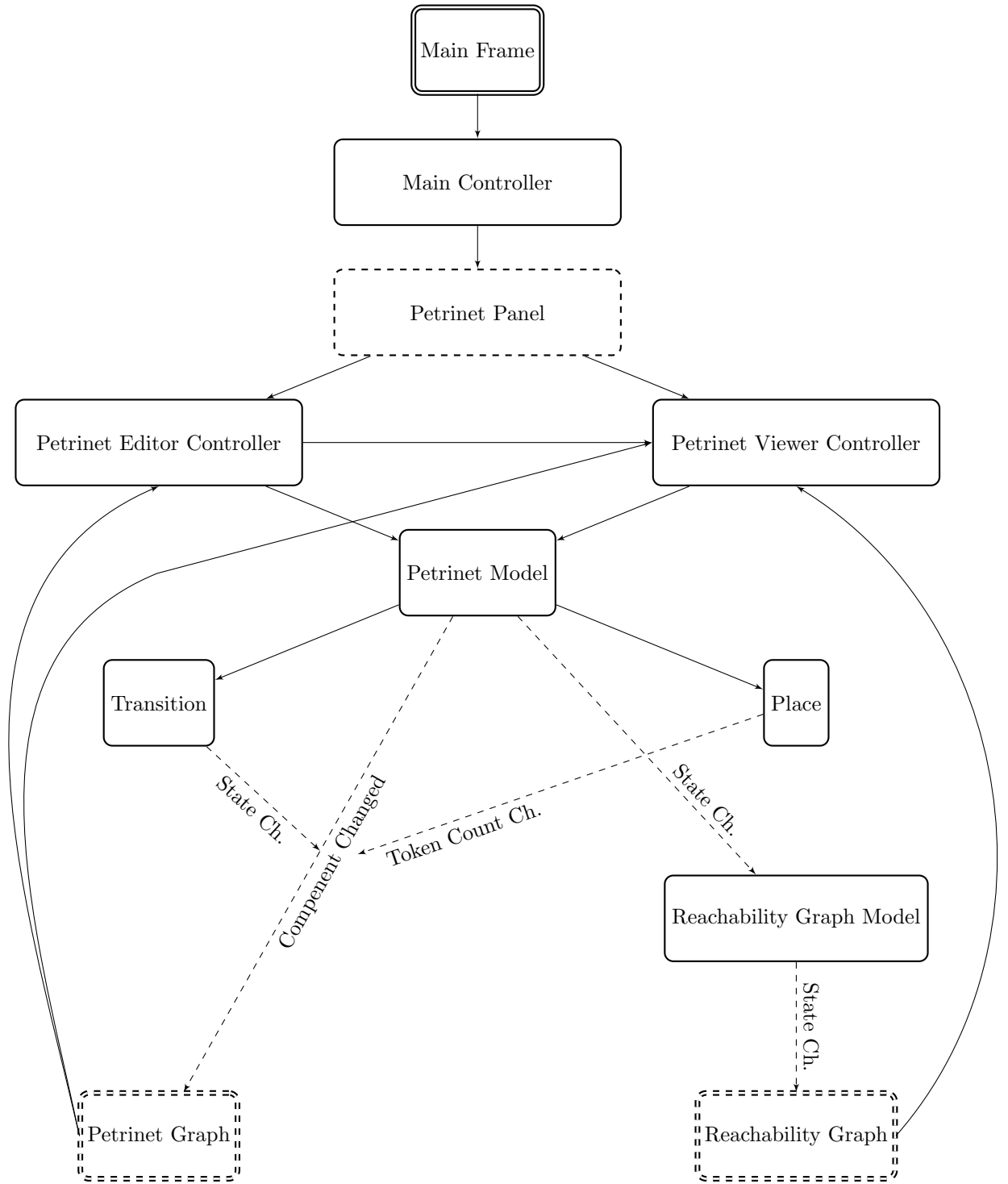
and the other controllers.

Main Controller On a general level there is a main frame and a number of petrinets in tabs. The task of the main controller is to pass user interactions with elements belonging to the main frame (menus and toolbars) to the petrinet panel that is currently active. If tabs are switched it is also the task of the main controller to restore the state of the toolbar with regards to highlighted buttons for the active petrinet or switching to editor/viewer mode (see 1.3).

Petrinet Panel Interface This interface forwards messages from the main controller. It also handles clicks and forwards them to either the viewer controller or the editor controller. Additionally it keeps track of the toolbar mode and handles GUI related actions for an instance of a petrinet.

Petrinet Viewer Controller This controller manages all clicks on the petrinet when in viewer mode and clicks on the reachability graph. It provides methods for interactions with the petrinet and reachability graph except those pertaining to editing the petrinet.

Petrinet Editor Controller This controller manages all clicks on the petrinet when in editor mode. It is tasked with providing functionality when editing the given petrinet. This controller could have been part of the petrinet viewer controller. But because the size of both controllers being combined as one would have been quite big and clicks on transitions are handled differently in viewer and editor mode this solution appeared more manageable.

2.1.3   Listeners

Petrinet State Changed Listener This listener serves as an intermediary between the petrinet model and the reachability graph model. When the petrinet changes states the reachability graph is informed. It distinguishes between states changing because a transition has been fired and changing states because the petrinet has changed → in the latter case the reachability graph and its initial state have to be reset.

Petrinet Component Changed Listener This listener communicates changes in components of the petrinet to the petrinet graph. Whenever elements are added/removed or their attributes are changed it forwards this information. Because it listens for changes in elements it also listens to the *NumberOfTokensChangedListener* and the *TransitionStateListener* and forwards their events.

Number of Tokens Changed Listener Part of the data structure *Place* it listens to changes in the number of tokens on the place. If the number changes it informs the petrinet component changed listener.

Transition State Listener Part of the data structure *Transition* it listens to changes in the activation state of the transition. If the state changes it informs the petrinet component changed listener.

Reachability State Changed Listener Listens for changes in the reachability graph model and communicates it to the reachability graph. It contains methods for adding/removing states and also methods for modifying edges.

Toolbar Changed Listener Listens for events belonging to the un-/redo buttons and the add/remove edge buttons. If events happen the buttons are highlighted accordingly.

Adjust Arrow Heads Listener This is a special listener for GraphStream. As the arrow heads don't adjust appropriately when adding/removing elements in the graph they need to be adjusted manually. This listener watches for events where arrow heads need to be adjusted.

2.2 Petrinet Analyser and the Algorithm for Checking Boundedness

The *PetrinetAnalyser* simulates clicking through the transitions of the petrinet and records the results. For this purpose it starts with the initial state and gets all activated transitions. It fires each transition one by one, gets the resulting new state and recursively calls the analyse method. At each firing of a transition the boundedness of the resulting state is checked. If it signifies the petrinet being unbouded the analyser terminates. Otherwise it terminates if it does not encounter new states. It therefore uses a depth first search approach to build the reachability graph step by step.

Algorithm 1 Check if Current State is Bounded in a Petrinet

---

   function CHECKIFCURRENTSTATEISBOUNDED
      for each predecessor state s of currentState **do**
         $state \leftarrow$ CHECKIFSTATEISBOUNDED(s, new list, currentState)
         if $state \neq null$ **then**
            return $false$
         end if
      end for
      return $true$
   end function
   function CHECKIFSTATEISBOUNDED($state$, $visitedStates$, $originalState$)
      if $state$ is in $visitedStates$ **then**
         return $null$
      end if
      Add $state$ to $visitedStates$
      if $originalState.isBiggerThan(state)$ **then**
         return $state$
      end if
      for each predecessor state s of $state$ **do**
         $newState \leftarrow$ CHECKIFSTATEISBOUNDED(s, $visitedStates$, $originalState$)
         if $newState \neq null$ **then**
            return $newState$
         end if
      end for
      return $null$
   end function

The actual algorithm for checking the boundedness of a given state is implemented in the reachability graph model itself and is represented by algorithm 1. When a state is added the reachability graph model automatically checks for unboundedness. The state is the *currentState* in the algorithm as any added state becomes the current state in the reachability graph model. As each instance of the petrinet state model knows its predecessors it can search for states that mark the current state as unbounded → the algorithm loops through them for the current state and calls the *CHECKIFSTATEISBOUNDED* method for every one of them. This medhod in turn loops through every predecessor of the state passed to it and calls itself recursively. The base case terminating the recursive loop is if the state has already been visitied or if it signifies that the given petrinet is unbounded. To this end the petrinet state model implements a method called *isBiggerThan* that takes a state as a parameter and returns whether the number of tokens on any place is bigger while the tokens on all the other places are the same or bigger also. The algorithm checks the original state (the current in the reachability graph model) and checks it against the state currently passed to the method. This means by performing a depth first search all states in the reachability graph model are checked iteratively until an m is found or all states have been checked.

Algorithm 2 shows how the *PetrinetAnalyser* works. All the analyser has to do is start from the initial state and simulate all possible configurations of the petrinet at hand. It iteratively performs a depth first approach of firing all transitions → for each state it loops over all activated transitions, fires them and calls the analyse function for the resulting new state in turn firing all activated transitions in this new state. It does so until all configurations have been exhausted, the reachability model informs the analyser that the petrinet is unbounded

---

Algorithm 2 Analyse State in a Petrinet

---
  function AnalyseState(*state*, *visited*)
    if not *bounded* **then**
      return
    end if
    if *state* is in *visited* **then**
      return
    end if
    $numberOfNodes \leftarrow numberOfNodes + 1$
    Add *state* to *visited*
    petrinet.setState(state)
    for each activated transition *t* in *petrinet* **do**
      petrinet.fireTransition(t)
      stateBounded ← reachabilityGraphModel.checkIfCurrentStateIsBounded()
      if not *stateBounded* **then**
        $bounded \leftarrow false$
        return
      end if
      $numberOfEdges \leftarrow numberOfEdges + 1$
      AnalyseState(*reachabilityGraphModel.getCurrentPetrinetState()*, *visited*)
      reachabilityGraphModel.setCurrentState(state)
      petrinet.setState(state)
    end for
  end function

---

or the petrinet has been found unbounded in another configuration (this what the variable *bounded* keeps track of). If the petrinet is unbounded the analyser updates the reachability graph by resetting it to the initial state and only firing the transitions leading up to and including the path marking the petrinet unbouded. This way only the relevant information is shown in the reachability graph.

2.3 Layout

The program supports three layouts: two custom layouts and the automatic layout[4] provided by GraphStream. Only the two custom layouts are discussed in this section.

2.3.1 Tree Layout

The tree layout organizes nodes in the reachability graph as a tree. The level in which a node goes into is defined by the length of the path from the initial node and is defined for each state in the class *PetrinetState* and therefore provided to the graph whenever a state is added. On each level the nodes are ordered by their proximity to the parent in the level above → nodes should be near below the parent node so edges don't intersect too much with other nodes. Unfortunately this does not prevent intersecting edges altogether. For example if a node has a child directly below and the child has another child directly below and the child of the child has an edge to the grandparent the edge between the grandchild and the grandparent will intersect the child. For this reason adding a new node to the tree performs two additional checks: *checkEdgeIntersection* and *spreadSprites*. The former checks whether any edges intersect with any nodes. If so the nodes are pushed to the left or right according to their position on the level. This is repeated until no edges intersect nodes anymore or until the function has looped 20 times. The latter checks three cases: parallel edges and crossing edges. On parallel edges the sprites need to be moved because they are almost guaranteed to cross each other. This is easy since the default position of sprites is in the middle and can be shifted accordingly. Crossing edges are more tricky. An additional check has to be performed to check whether the sprites actually intersect. To this end the *reachabilityGraphLayout* package provides data structures and functions to work with rectangles. The layout may not solve all issues but in most cases it provides a pleasant experience.

---

[4]see Automatic layout at `https://graphstream-project.org/doc/Tutorials/Graph-Visualisation/1.0/`

2.3.2 Circle Layout

This layout organizes nodes around a circle – or to be more precise an ellipse. Initially the nodes are ordered by level. Just putting them around a circle would lead to level clusters and most importantly edges clustering at the circles scope. To avoid clusters the initial ordering of nodes is spread evenly around the circle by a certain ratio[5]. The ratio to which nodes are spread around the circle can be modified by clicking the circle layout button repeatedly.

---

[5]For a more in depth explanation of how the algorithm works please refer to the comments in the source code.