Galant Developer Documentation for Version 5.3

Matthias F. Stallmann*

December 6, 2016

The purpose of this documentation is to provide future developers a road map of the most important parts of the Galant implementation. These can be divided into several main aspects of Galant functionality:

- the graph data structure (Section 1)
- algorithm execution (Section 2)
- macro expansion and compilation (Section 3)
- text editing and file management (Section 4)
- graph editing and the graph window (Section 5)

In each case we consider sequences of events that take place as the result of specific user or software actions, pointing out Java classes and methods that handle the relevant functionality.

1 Graph Structure and Attributes

A Graph is the container for all attributes of a graph, whether read from a file, edited or manipulated by an algorithm. The source code for all objects related to the structure of a graph is in graph.component. Naturally, there is a list of nodes and a list of edges. These lists are not altered in the obvious way during editing or algorithm execution. New nodes and edges are appended to the respective lists but deletions are virtual: a node or edge has a deleted attribute. New edges are also appended to the incidence lists of their endpoints: incidentEdges in Node. A graph has attributes not directly derived from its nodes or edges. These are name (specified by an external source), comment (ditto), directed (toggled by the user but not during algorithm execution), layered (more details later) and banner (message banner displayed at the top of the graph window during algorithm execution – see method display() in Algorithm).

A node has a unique id so that it can be referred to as an endpoint of an edge in the GraphML representation. The map nodeById in class Graph maps each integer id to a Node object. The latter has an id as one of many potential attributes. The id, along with a position, x and y attributes in GraphML, are required for each Node. All other attributes are optional. An Edge object is required to have a source and a target. If the graph is undirected, the two are interchangeable.

Both Node and Edge are subclasses of GraphElement – see Fig. 1. At any point in time a GraphElement is in a particular GraphElementState and that state determines values of all attributes except for the fixed ones: id, x and y for Node; source and target for Edge. A GraphElement has a list of states to keep track of changes during algorithm execution. Each state is given a *time stamp*[1] (the integer state) to mark the point during algorithm execution at which the state is effective. Currently, the time stamp is 0 unless an algorithm is running, but could be used in a future implementation to allow undo operations during editing.

*North Carolina State University, email: mfms@ncsu.edu

[1]This terminology will be used hence to distinguish state (as time stamp) from state as GraphElementState

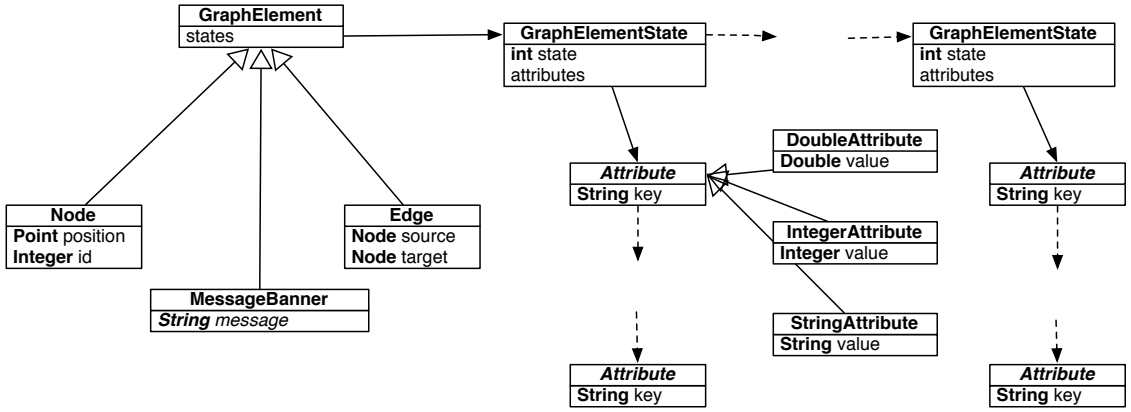

Figure 1: A UML diagram showing the structure of a GraphElement.

State changes

A GraphElement changes state whenever any attribute changes value, either during editing or algorithm execution. Methods of the form
set(String *key*, **type** *value*)
all do the following – see GraphElement.

- create a new state newState
- set the attribute *key* to *value* in the attribute list for newState
- the setter returns true if an attribute *key* was already present, i.e., if an existing attribute was modified rather than a new one added, false otherwise – see AttributeList; this Boolean value is available for use throughout, including by the algorithm animator
- add the new state to the list of states in one of two ways – see addState()

    1. if no state with the current time stamp exists, append the new state to the list of states; it becomes the most recent one, the one returned by latestState(); if there is an algorithm running, the state change triggers the end of a step – pauseExecutionIfRunning() in GraphDispatch
    2. if there is a state with the same time stamp as the current one, simply replace it; this can happen during editing – time stamp is 0, or if the element undergoes more than one state change during a single algorithm step

The list of states can be used to determine the attributes at any given time (stamp) via the getters that have a state argument. These access the last state on the list whose time stamp precedes the given one, the state argument.

States of existence

A node or edge may be created or deleted while editing or during algorithm execution. To determine whether or not a GraphElement exists at a given point in time Galant checks its state – the inScope() method. The simpler part is isDeleted(), the predefined attribute DELETED = deleted, which is set whenever the user or the algorithm deletes the object.

To determine whether the element has already been created (this makes a difference on in the context of algorithms that create new nodes and/or edges) – method isCreated(int state). Here the element exists at the given state if there is a state on its list whose time stamp precedes state, i.e., if getLatestState() returns a non-null value. Since an element is initialized with a single state on its

list – see the constructor, one whose time stamp *stamp* is the time of creation, a null indicates that state $<$ *stamp*.

There are three contexts for Node creation, each requiring a different approach. All the relevant methods are in Graph.

parsing – when reading GraphML text to create the internal representation of a graph, the Node has already been initialized (its attributes read from the file) and it needs only to be added to the graph; this is accomplished by the method addNode(Node); the new node will become the rootNode if none exists, accessible via getStartNode() and sometimes used as a start node for algorithms that require one

editing – in edit mode a call to addInitialNode(Integer,Integer) specifies x- and y-coordinates of the node; a new node is created accordingly and assigned the smallest available id; the new node becomes the rootNode if one is needed

algorithm execution – the algorithm has presumably calculated a desired position for the node; a call to addNode(Integer,Integer) has the same effect as one to addInitialNode, but it also initiates a new algorithm step if appropriate

Types of attributes

As already noted, Node objects have mandatory attributes id, x and y (position in the window in pixels); Edge objects must have source and target. These must be present at all times, whether in a GraphML representation, during editing or during algorithm execution. The exception is the position of a Node, which may not be specified in a GraphML file – if not, it is assigned randomly within window boundaries when the file is read.

Any attribute can be specified in a GraphML file, whether or not it is meaningful to Galant – it might be used by an algorithm or have meaning in a context outside Galant (so should not be discarded). It is also possible for the animator to access and modify values of any attributes to, for example, record the status of a node or edge during algorithm execution. There are some predefined attributes that have an impact on the display of a graph; a subset of these can be modified during editing as well. All of these are defined as constants (in upper case) at the beginning of GraphElement. For Boolean attributes, the absence of the attribute in an AttributeList is synonymous with it being false.

Attributes that affect the display of a GraphElement are (those marked with * can also be edited).

id (of a node) – displayed inside the circle representation of the node if it is large enough; user can set the radius as a preference

weight * – a floating point number that can be used for sorting or as a key in a priority queue (built into the compareTo() method); displayed above and right of a node, in a box in the middle of an edge

label * – a string, displayed below and right of a node, in a box in the middle of an edge, to the right of the weight

color * – a string of the form `#rrggbb`; each pair of symbols after the # is a hexadecimal number indicating the strength of red, green or blue, respectively; predefined constants for a variety of colors are in Algorithm; an edge with a defined color is thicker than one without; color, for a node, applies to the boundary, which is also thicker (thickness set by user preference) if the node has a color

deleted – if true, the element does not exist

highlighted – if true, the element is colored red; the color is determined by HIGHLIGHT_COLOR in GraphPanel; also accessible via setters and getters for Selected

marked (node only) – if true, the interior of the node is shaded using MARKED_NODE_COLOR in GraphPanel

hidden – if true, the element does not appear on the display

hiddenLabel – if true, the label of the element does not appear

hiddenWeight – if true, the weight of the element does not appear

2 Algorithm Execution

Algorithm execution is initiated when the user presses the Run or the Compile and Run button when focused on an algorithm in the text window. The sequence of method calls is

- run() in gui.editor.GAlgorithmEditorPanel; this initializes the current algorithm, the graph on which it will be run and the AlgorithmSynchronizer and AlgorithmExecutor that will be used to coordinate the algorithm with the GUI, respectively.
  The AlgorithmExecutor manages the master thread, i.e., the one associated with the gui, and the AlgorithmSynchronizer manages the slave thread, the one executing the algorithm on behalf of the user.
- Method startAlgorithm() in algorithm.AlgorithmExecuter is called to fire up the algorithm (slave). At this point the gui and the algorithm behave as coroutines. The gui cedes control to the algorithm in algorithm.AlgorithmExecuter.incrementDisplayState() and enters a busy-wait loop until the algorithm is done with a *step* (clarified below) or it is terminated.
  The algorithm cedes control to the gui in algorithm.AlgorithmSynchronizer.pauseExecution, where it either indicates that a step is finished (resulting in an exit from the busy-wait loop) or responds to a gui request to terminate the algorithm – the gui has set terminated – by throwing a Terminate pseudo-exception.

The gui controls algorithm execution, the user's view thereof, using the buttons stepForward, stepBackward and done, defined in gui.window.GraphWindow, or their keyboard shortcuts right arrow, left arrow and escape, respectively. Fig. 2 gives a rough idea of the interaction between the two threads (AlgorithmExecutor and AlgorithmSynchronizer) that are active during algorithm execution.

- A step forward button press or arrow key effects a call to performStepForward() in GraphWindow, leading to an incrementDisplayState(). First, there is a test, hasNextState() in AlgorithmExecuter, false only if the display shows the state of affairs after the algorithm has taken its last step.
- incrementDisplayState() does nothing (except increment the displayState counter) if the algorithm execution is ahead of what the display shows (as a result of backward steps).
- If the display state is current with respect to algorithm execution, the algorithm needs to execute another step – the gui cedes control and enters its busy-wait loop. At this point the algorithm performs a step, described in more detail below.
- A step back button press or array key effects a call to performStepBack() in GraphWindow, leading to a decrementDisplayState(). The latter simply decrements the displayState counter. If the display state corresponds to the beginning of algorithm execution – hasPreviousState() in AlgorithmExecutor is false – decrementDisplayState() is not called.
- The methods performStepForward() and performStepBack() also control the enabling and disabling of the corresponding buttons in the graph window, based on hasNextState() and hasPreviousState(). And they call updateStatusLabel() to display the current algorithm and display states to the user. An algorithm state corresponds to a step in the algorithm.
- A done button press or escape key leads to performDone(), which in turn calls stopAlgorithm() in the AlgorithmExecutor. Here things get interesting. The AlgorithmSynchronizer is told that the algorithm is to be stopped via a stop() method call and the AlgorithmExecutor cedes control to it. The algorithm is expected to yield control back to the executor, at which point the latter does a join() to wait for the algorithm thread to finish. Algorithm and display states are then reinitialized to 0.

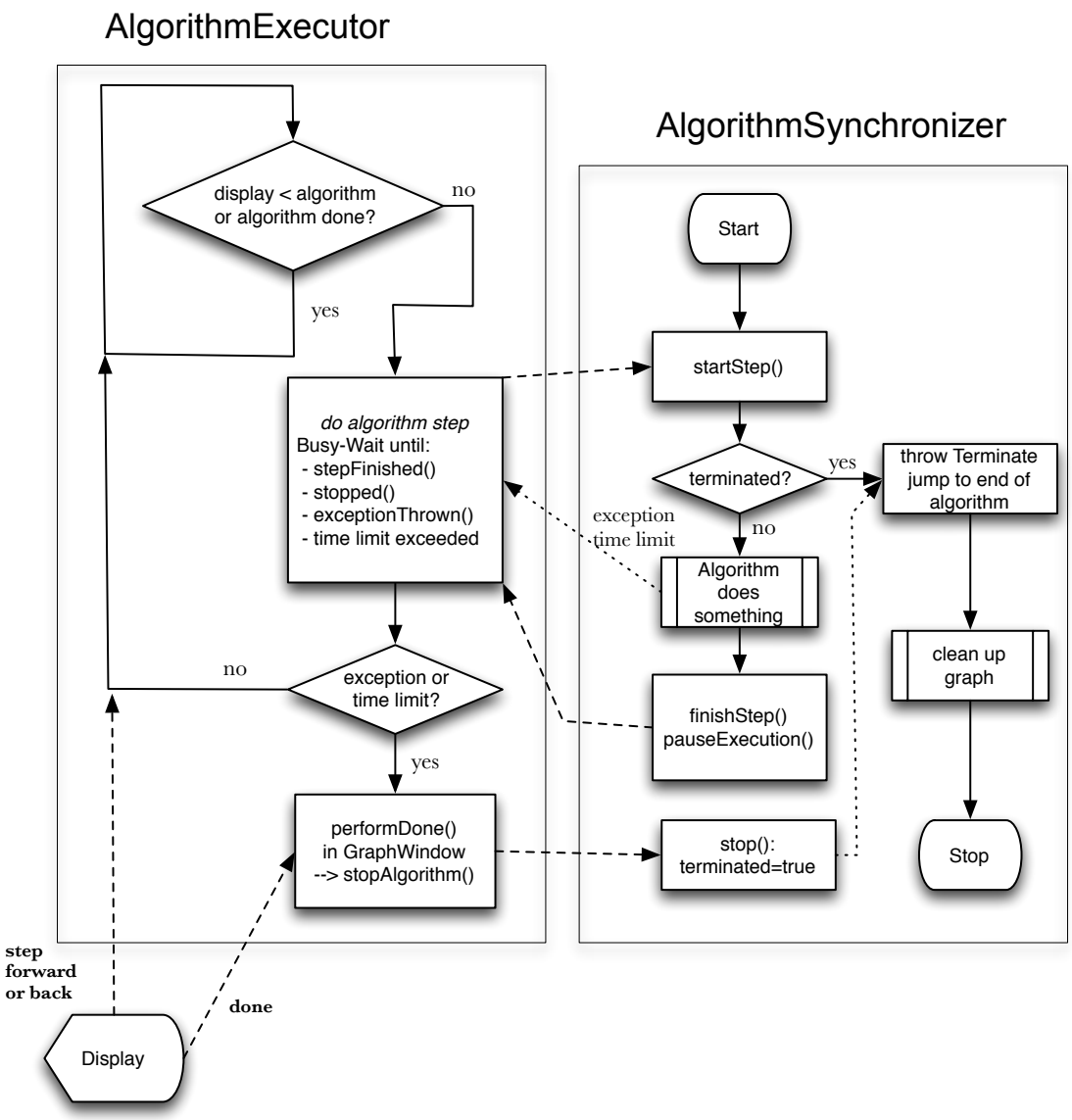

Figure 2: Interactions between two threads during algorithm execution.

- Complications arise with stopping the algorithm because the user may terminate the algorithm at any time, not just when the algorithm has run to completion. If it has not run to completion, the algorithm, at any attempt to execute the next step, checks whether it has been terminated. If so it throws Terminate, an exception that is caught at the very end of the compiled algorithm. The effect is that of a "long jump" to the end of the algorithm.

Some complications that require extra care are:

- The algorithm could throw an exception. If this is a GalantException the constructor informs the AlgorithmSynchronizer via a call to reportExceptionThrown(). Other exceptions may cause Galant to hang. The ultimate goal is to avoid these entirely. In the Algorithm class, which defines all the procedural-style method calls, potential null pointer exceptions are caught before the underlying graph methods are called. The AlgorithmExecutor, when in its busy-wait loop (or before entering it), checks whether an exception has been thrown – exceptionThrown() in AlgorithmSynchronizer – and terminates the algorithm if so.
- The algorithm could get into an infinite loop. Unfortunately, under the current setup, an interrupt initiated by a performDone() does not appear to work; so the user is not able to terminate the algorithm. The workaround is a time limit of 5 for the busy-wait loop, after which the algorithm is terminated.
- The join() used by the AlgorithmExecutor to wait for the algorithm to finish up and the thread to terminate could cause Galant to hang if (i) there was an exception; (ii) there was an infinite loop; or (iii) the animation is waiting for the user to respond to a query window. In cases (i) and (ii) the algorithm thread is allowed to die without being waited on – see stopAlgorithm(). Case (iii), the query window, is handled by having the dispatcher maintain a reference to any query window that might be open – setActiveQuery() and getActiveQuery() in GraphDispatch. The queries (all in gui.util) are responsible for setting and clearing (on successful completion of the query) the reference. If there is an active query, as with an exception or infinite loop, the join() is avoided. Also, the query window is closed.

3 Macro Expansion and Compilation

4 Text Editing and File Management

5 Graph Editing and the Graph Window