

# Galant Developer Documentation for Version 6.1.1

Matthias F. Stallmann\*

January 9, 2021

## Contents

<b>1 Overview</b>	<b>2</b>
<b>2 Graph Structure and Attributes</b>	<b>2</b>
2.1 State changes . . . . .	5
2.2 States of existence . . . . .	5
2.3 Types of attributes . . . . .	6
2.4 GraphML parsing . . . . .	7
2.5 Layered graphs . . . . .	8
<b>3 Algorithm Execution</b>	<b>8</b>
<b>4 Macro Expansion and Compilation</b>	<b>11</b>
<b>5 Text Editing and File Management</b>	<b>11</b>
5.1 Modified (dirty) files . . . . .	11
<b>6 Graph Editing and the Graph Window</b>	<b>12</b>
6.1 Editing attributes of edges and nodes . . . . .	12
<b>7 Undo/Redo</b>	<b>12</b>
<b>8 Queries</b>	<b>13</b>

## List of Figures

1 Organization and directory structure of Galant packages. . . . .	2
2 Directory listing of Galant source code: global information, algorithm and graph. . .	3
3 Directory listing of Galant source code: GUI, preferences and miscellaneous. . . . .	4
4 A UML diagram showing the structure of a <code>GraphElement</code> . . . . .	6
5 Interactions between two threads during algorithm execution. . . . .	10

---

\*North Carolina State University, email: [mfms@ncsu.edu](mailto:mfms@ncsu.edu)

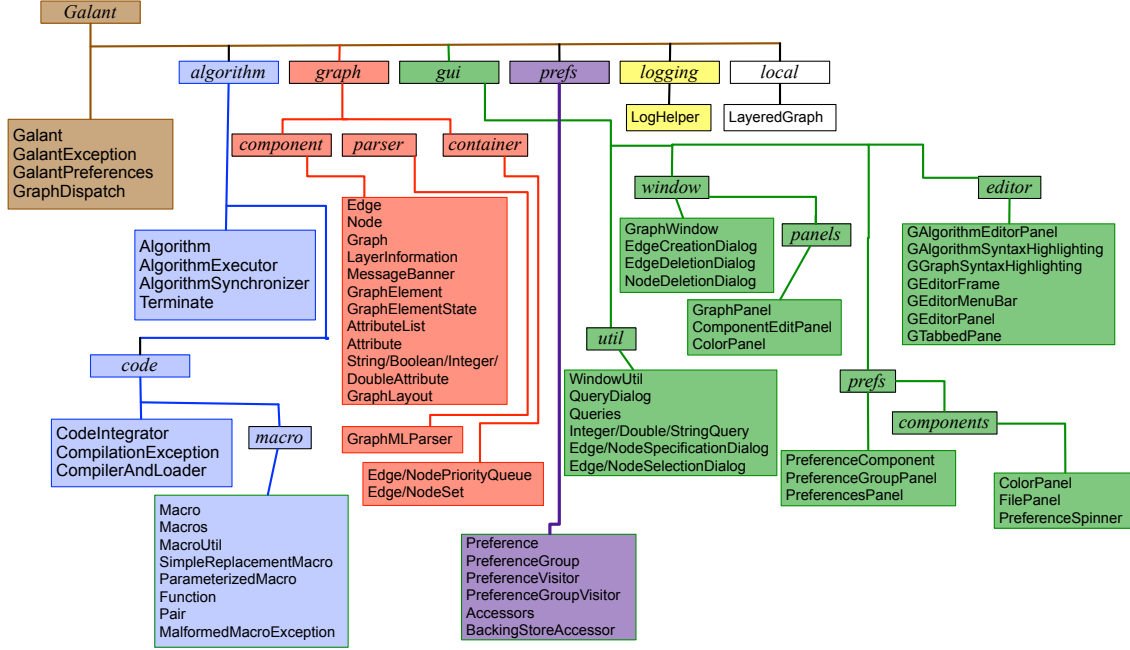


Figure 1: Organization and directory structure of Galant packages.

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 2)
- algorithm execution (Section 3)
- macro expansion and compilation (Section 4)
- text editing and file management (Section 5)
- graph editing and the graph window (Section 6)

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 Overview

## 2 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

```

Galant.java
GalantException.java
GalantPreferences.java
GraphDispatch.java
  algorithm/
    Algorithm.java
    AlgorithmExecutor.java
    AlgorithmSynchronizer.java
    Terminate.java
  code/
    CodeIntegrator.java
    CompilationException.java
    CompilerAndLoader.java
  macro/
    Function.java
    Macro.java
    MacroUtil.java
    Macros.java
    MalformedMacroException.java
    Pair.java
    ParameterizedMacro.java
    SimpleReplacementMacro.java
graph/
  component/
    Attribute.java
    AttributeList.java
    BooleanAttribute.java
    DoubleAttribute.java
    Edge.java
    Graph.java
    GraphElement.java
    GraphElementState.java
    GraphLayout.java
    GraphState.java
    IntegerAttribute.java
    Layer.java // not used
    LayerInformation.java (used by Graph.java)
    LayeredGraph.java // not used
    MessageBanner.java
    Node.java
    StringAttribute.java
  container/
    EdgePriorityQueue.java
    EdgeSet.java
    NodePriorityQueue.java
    NodeSet.java
  parser/
    GraphMLParser.java

```

Figure 2: Directory listing of Galant source code: global information, algorithm and graph.

```

gui/
  editor/
    GAlgorithmEditorPanel.java
    GAlgorithmSyntaxHighlighting.java
    GEditorFrame.java
    GEditorMenuBar.java
    GEditorPanel.java
    GGraphEditorPanel.java
    GGraphSyntaxHighlighting.java
    GTabbedPane.java
  prefs/
    PreferenceComponent.java
    PreferenceGroupPanel.java
    PreferencesPanel.java
  components/
    ColorPanel.java
    FilePanel.java
    PreferenceSpinner.java
  util/
    DoubleQuery.java
    EdgeSelectionDialog.java
    EdgeSpecificationDialog.java
    ExceptionDialog.java
    IntegerQuery.java
    NodeSelectionDialog.java
    NodeSpecificationDialog.java
    Queries.java
    QueryDialog.java
    StringQuery.java
    WindowUtil.java
  window/
    EdgeCreationDialog.java
    EdgeDeletionDialog.java
    GraphWindow.java
    NodeDeletionDialog.java
  panels/
    ColorPanel.java
    ComponentEditPanel.java
    GraphPanel.java
local/
  LayeredGraph.java
logging/
  LogHelper.java
prefs/
  Accessors.java
  BackingStoreAccessor.java
  Preference.java
  PreferenceGroup.java
  PreferenceGroupVisitor.java
  PreferenceVisitor.java

```

Figure 3: Directory listing of Galant source code: GUI, preferences and miscellaneous.

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. 4. 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*<sup>1</sup> (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.

## 2.1 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,<sup>2</sup> or if the element undergoes more than one state change during a single algorithm step, i.e., as a result of `beginStep()` and `endStep()` invocations in the algorithm code.

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.

## 2.2 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`.

<sup>1</sup>This terminology will be used hence to distinguish *state* (as time stamp) from state as `GraphElementState`

<sup>2</sup>There is now also an *edit state* that allows undo and redo operations during editing. So time stamp can depend on edit state as well. See Section 7.

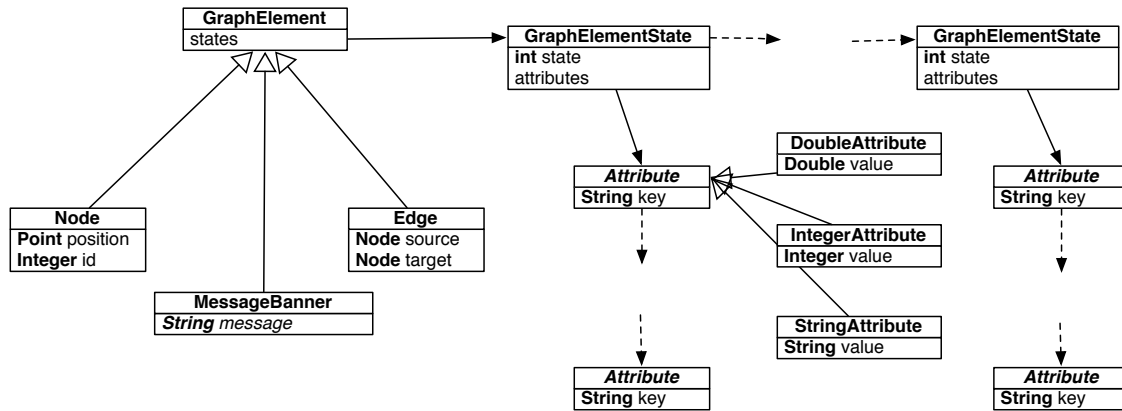


Figure 4: A UML diagram showing the structure of a `GraphElement`.

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

## 2.3 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

## 2.4 GraphML parsing

The `GraphMLParser` (in `graph.parser`) is invoked in one of three ways listed below. The corresponding graph is then the working graph returned by `getWorkingGraph()` in `GraphDispatch`. All of the three invocations are in package `gui.editor`.

1. in `GEditorFrame`, method `updateWorkingGraph()`, called when the user does a `Save` or `Save As` on the current panel and it happens to be a graph; the text in the panel has to be parsed in order for the changes to be reflected in the graph window – while any edit in the graph window is immediately pushed to the text window, the reverse is true only when the text is saved
2. in the constructor for `GGraphEditorPanel`, a subclass of `GEditorPanel`, when the user opens a GraphML file; the new panel is created in method `addEditorTab()` in `GTabbedPane`
3. in `GTabbedPane`, method `stateChanged`, invoked when the user clicks on a panel containing GraphML content (either associated with a file or with an empty, unnamed graph to be edited by the user)

The code for handling the panels in the text window is convoluted – see Section 5 – to the extent that the three classes represented above are opaquely intertwined. But now we will look at parsing in isolation.

Both `Node` and `Edge` objects have an `initializeAfterParsing()` method to make sure that the required attributes are present:

- for a `Node`: `id` (already converted to an integer) and coordinates `x` and `y` (need to be converted to integer)
- for an `Edge`: `source` and `target` – these need to be integers *and* ids of existing nodes (the parser processes nodes first); there may be an optional `id`, which is handled by graph initialization – see below – to ensure that either all edges have ids or none of them do; in either case, edges are assigned ids internally so that these can be used as array indexes.

These are then stored directly with the object rather than being part of its list of attributes. First, however, the `initializeAfterParsing()` method for the super class `GraphElement` is invoked to handle attributes that have special meaning when the graph is displayed. These are converted from strings to the appropriate type (if present): `id` (`Integer`), `weight` (`Double`) and `highlighted` (`Boolean`). A `Node` object also converts `marked`, if present to `Boolean`.

There is also an `initializeAfterParsing()` method for `Graph`. Currently its only purpose is to assign ids to the edges. The id of an `Edge` is not required except that it might be used as an array index by an algorithm. There are two cases.

1. At least one edge has an explicit id in its GraphML representation. In this case all edges will be given ids using the `getNextEdgId()` method of `Graph` and duplicates will be avoided as they are with node ids, except that here it is more convenient to do this at the graph level.
2. None of the edges have explicit ids. In this case ids are assigned sequentially. The `getNextEdgId()` method does the right thing.

In the first case the graph keeps track of the fact that explicit edge ids were present in its representation so that these will be written to the GraphML file when it is saved. In the second, the assigned ids will not appear in the GraphML. The distinction is made in the `xmlString()` method of `Edge`. The relevant attribute is `hasExplicitEdgIds` and each edge has an attribute `hasExplicitId`, set in `initializeAfterParsing`.

## 2.5 Layered graphs

The utilities in `local.LayeredGraph` are used in the crossing minimization algorithms in `Research/Layered-Graphs`. The GraphML representation of a layered graph specifies `type="layered"` and, instead of the mandatory `x` and `y` attributes for each node, there are mandatory `layer` and `positionInLayer` attributes. In the current implementation, layered graphs are awkwardly shoehorned into various parts of the code. Among these are.

- `layered` is an attribute of a graph; it might be more natural for a layered graph to be a subclass
- when parsing node, a special case for layered graphs in method `initializeAfterParsing()`; a subclass `LayeredGraphNode` could override relevant parts of this processing
- in `GraphPanel` the method `getNodeCenter()` makes a special case for layered graphs, calculating the position of a node so that the layers are distributed equally in the vertical direction and the nodes on each layer in the horizontal direction; a subclass could override a method that specifies the display position of a node appropriately

## 3 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.AlgorithmExecutor` 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.AlgorithmExecutor.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. 5 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 `AlgorithmExecutor`, 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.
- 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()`.

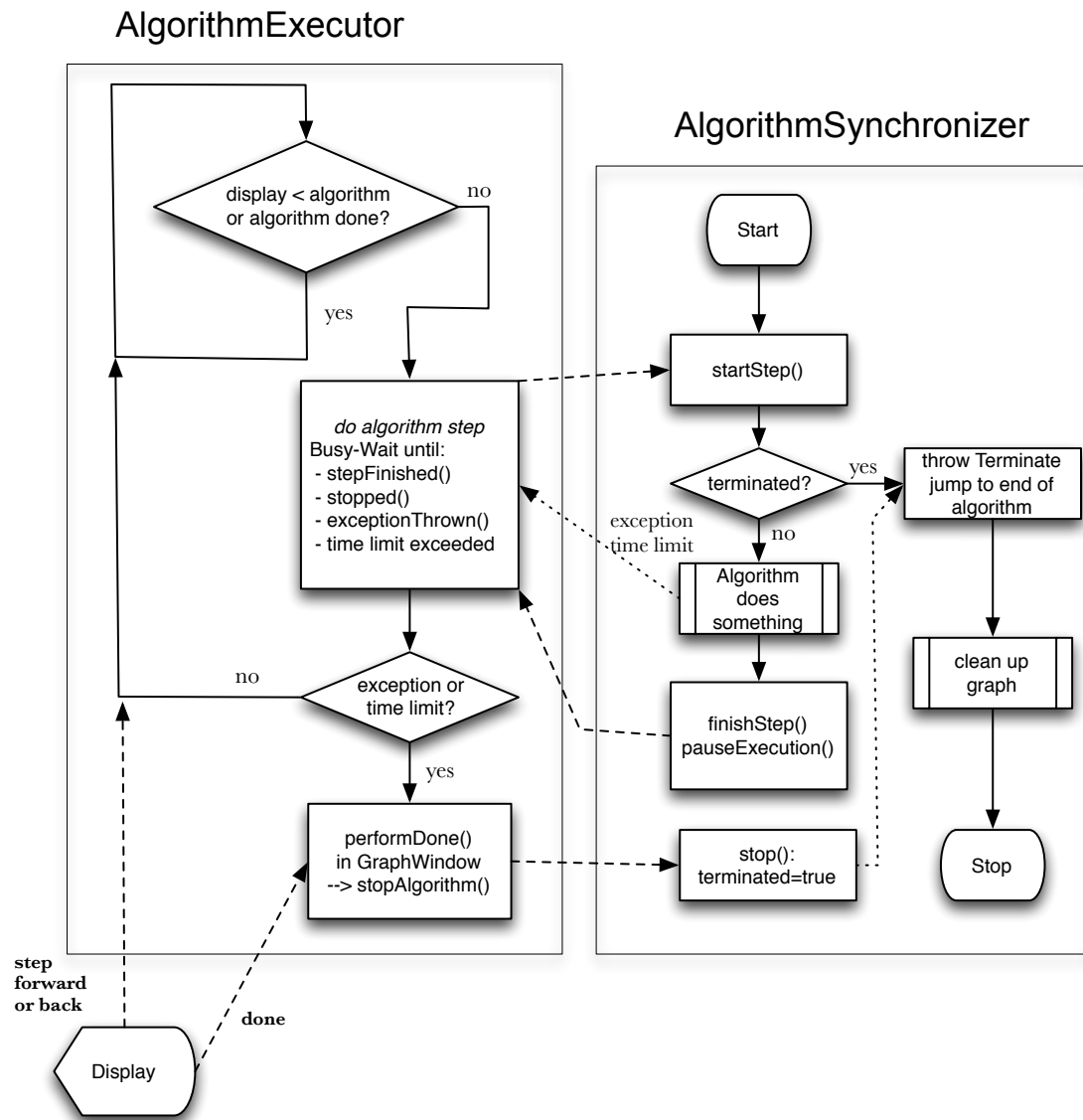


Figure 5: Interactions between two threads during algorithm execution.

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.

## 4 Macro Expansion and Compilation

Compilation is initiated in the `compile` method in `GAlgorithmEditorPanel`, which in turn calls on `integrateCode` in `CodeIntegrator`. The method `toJavaClass` does all the important work:

**removes comments** using method `removeAllComments`, being careful to preserve line breaks,

**converts the algorithm** code to the `run` method, adding initialization and cleanup code

**does macro replacement** one macro at a time, using the `applyTo` method in class `Macro`,

**assembles the code** with `package`, `import` and class declarations at the beginning and a closing brace at the end,

Finally, two methods in `CompilerAndLoader` are called: (i) the `compile` method invokes the system Java compiler on the assembled Java code, collecting any error messages, which are then sent to a `CompilationException`; and (ii) the `loadAlgorithm` method returns an instance of class `Algorithm` with the algorithm's name – the corresponding class file is stored in a directory specified by user preference.

[ Subsection on macro expansion? ]

[ Explain

- how macros are applied one at a time, each to the whole of the code
- how regular expressions are used in modify
- complications with `ParameterizedMacro` and suggestion for change: do a replacement for the macro with block first and then check for version without block – `for_nodes`, etc.

]

## 5 Text Editing and File Management

### 5.1 Modified (dirty) files

Galant keeps track of which files have been modified since the start of the edit session or the last save. The filenames on the editor tabs are marked with `*` to indicate a modification. The standard mechanism for keeping track of modifications is to monitor property changes on the panel, i.e., the overridden `insertUpdate` and `removeUpdate` events for `GEditorPanel`, via calls to `setDirty(true)`, making the file *dirty*.<sup>3</sup> This works fine if the user is editing text directly or editing a graph in the graph window and changes are pushed to the text panel via `pushToTextEditor` in `GraphDispatch`. Two situations require special care:

1. The GraphML parser can change attributes of nodes and edges, for example, to assign random positions to nodes that do have them. When parsing is done the text panel for the graph is refreshed whether or not a change has occurred. Since random positions of nodes are usually not worth saving, it makes sense to ignore changes during the refresh. A refresh also occurs at the end of animation execution.

---

<sup>3</sup> `setDirty(false)` clears the `isDirty` flag when the file is saved.

2. A change in graph directedness should not make the file dirty; a user may want to run an algorithm in each mode on the same graph with different results (e.g., Dijkstra's); or simply check the directions of the edges.

Galant, in `GraphDispatch`, keeps track of whether or not it is in edit mode. `GGraphEditorPanel` overrides the definition of `setDirty` so that it makes the file dirty only if Galant is in edit mode. Now the fun part: how to ensure that Galant is in edit mode at exactly the right times.

Edit mode is made false by default and is activated via `setEditMode(true)` in `GraphDispatch` when a change in the graph window is pushed to the text editor. As soon as the change is accomplished – `propertyChange` in `GGraphEditorPanel` – edit mode is set back to false; care is taken so that the reset occurs only in the panel corresponding to the active graph.<sup>4</sup>

A feature/bug in this approach is that the text panel will not become dirty if the user edits the GraphML text of a graph directly. Perhaps this is acceptable – the change is not reflected in the graph window until the file is saved.

## 6 Graph Editing and the Graph Window

### 6.1 Editing attributes of edges and nodes

From the graph window it is possible to edit the color, label and weight of a node or edge. The `ComponentEditPanel` class manages this process. There are three different mechanisms here, one for each editable attribute.

**label** This is a simple text field `TextField`; whatever the user types becomes the label of the element.

**weight** This is a spinner with a text field. The spinner, created with a `SpinnerNumberModel`, increments and decrements the current weight. The text field of the editor behaves like the text field of the label, except that now the text must be converted to a `Double`. An illegal number is supposed to throw an exception, but this feature is not working. If the text field is blank or something other than a number, there is a "bell", not what the code intends but also not undesirable.

**color** Colors are handled in the `ColorPanel` class, which invokes a `JColorChooser`, a fairly complex mechanism, not particularly user-friendly at this point.

Currently there is no obvious way to allow a user to erase an attribute once it has been set by the editor. There is an algorithm `strip_attributes` to get rid of all attributes that were inadvertently set. The semantics for weights are, unfortunately, difficult to define in this context. Labels are erased if they are blank (equal to "") and colors are erased if they are black (equal to "#000000"). The current implementation of `strip_attributes` erases all weights if and only if they are all 0.

## 7 Undo/Redo

The editor allows the user to undo the effects of an editing operation that modifies an attribute of a node or edge (including deletion). This is accomplished using the same state-change mechanism as one used during algorithm execution. `GraphDispatch` maintains an `editState` that is decremented by an undo and incremented by a redo. The operations are activated by corresponding buttons or by `Ctrl-Z` and `Ctrl-Y`, respectively: the relevant code is in `GraphWindow`. A subtle point here is that deletion of a node requires deletion of all incident edges. If each deleted edge involves a state change the result is edit states with "dangling edges". To mitigate this problem, there is an `atomic` property in `GraphDispatch`. When set, all attribute changes take place without a state change. The only place where this property is set/reset is in `deleteNode` in `Graph`. The

---

<sup>4</sup> The `notifyListeners(TEXT_UPDATE, null, null)` call in `GraphDispatch` is indiscriminate – all text panels are notified.

`startStepIfAnimationOrIncrementEditState` in `GraphDispatch` decides whether to increment the edit state or not when in edit mode.

## 8 Queries

Galant offers the animator functions that query the user to select a node or edge or to provide an integer, double or string. In addition, most exceptions that arise result in a popup window that gives the user the option of either continuing Galant execution or exiting from Galant. Implementations of query windows are all in the `gui.util` package.

The current implementation of these queries is somewhat opaque and does not follow standard procedure for use of Java's `JOptionPane`, but it's a case of "if it ain't broke ...". However, the class `Queries` has an implementation of a simple Boolean query that appears to work; it is invoked in `Algorithm` to make it accessible to the animator. Adapting the technique to the other query types will take more work because they need to throw exceptions and repeat the request for input if the user's response is not satisfactory (a malformed number, a non-existent node or edge).