1 Galant Developer Documentation for Version 5.3

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

- algorithm execution (Section 1.1)
- macro expansion and compilation (Section ??)
- text editing and file management (Section ??)
- graph editing and the graph window (Section ??)

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.

1.1 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 latter eventually results in an exit from the busy-wait loop, but not until the algorithm has had a chance to clean up.

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.

- A step forward button press or arrow key results in 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.