Galant Developer Documentation for Version 5.3

Matthias F. Stallmann[*]

December 5, 2016

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)
- 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 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. 1 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

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

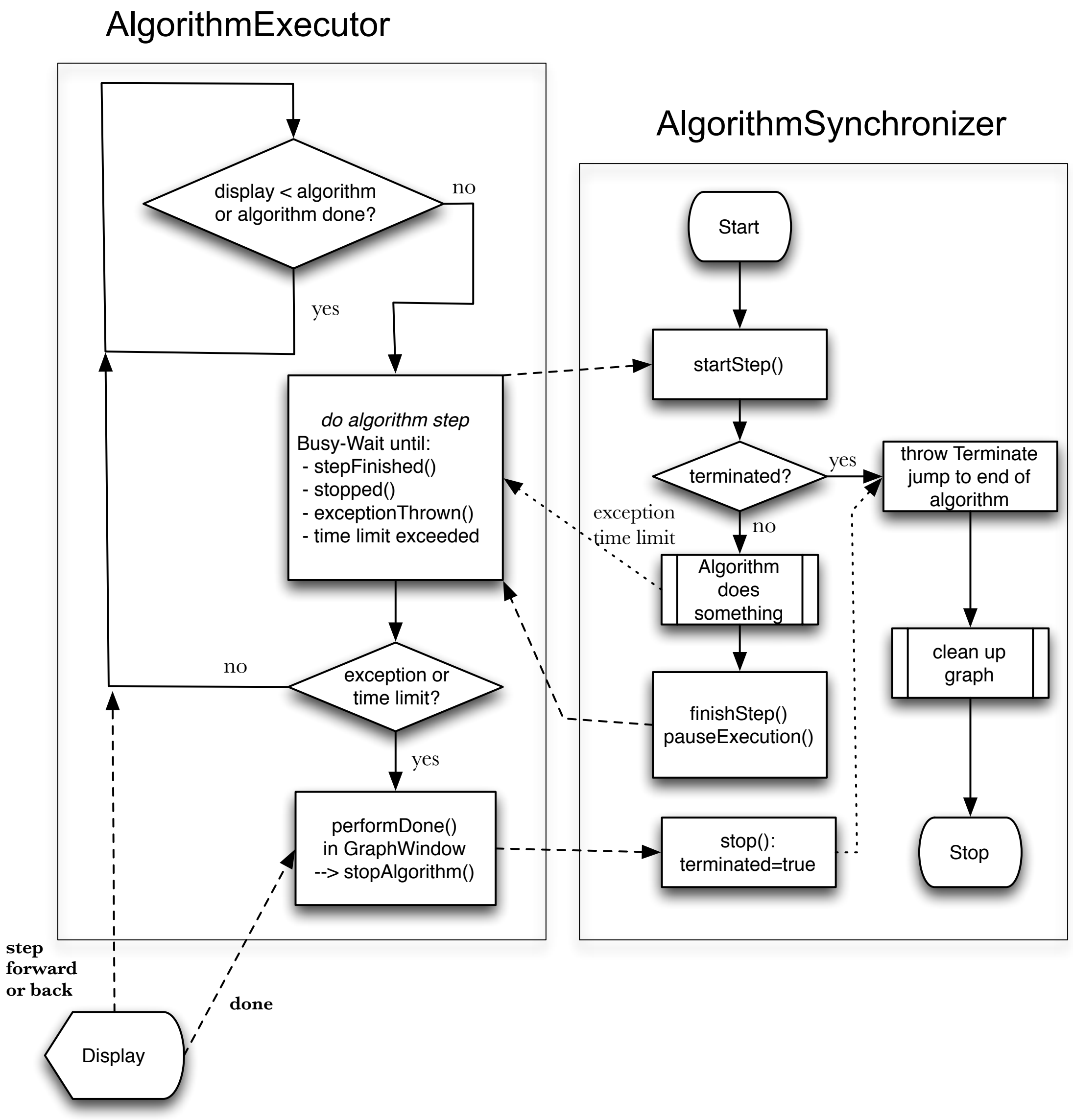

Figure 1: Interactions between two threads during algorithm execution.

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.