contained jobs are completed. Up to this point, no processing has taken place; the programmer only declared what needs to be done in what order. Once the whole sequence is set up, the user queues it into the application-global job queue (a lazy initialized singleton). The sequence is automatically executed by worker threads.

3. When the `done()` signal of the sequence is received, the data is ready to be displayed.

Two aspects are apparent. First, the individual steps are all declared in one go and then executed. This alone is a major relief to GUI programmers because it is a nonexpensive operation and can easily be performed in an event handler. Second, the usual issues of synchronization can largely be avoided by the simple convention that the queuing thread only touches the job data after it has been prepared. Since no worker thread will access the job data anymore, access to the data is serialized, but in a cooperative fashion. If the programmer wants to display progress to the user, the sequence emits signals after the processing of every individual job (signals in Qt can be sent across threads). The GUI remains responsive and is able to dequeue the jobs or request cancellation of processing.

Since it is much easier to implement I/O operations this way, ThreadWeaver was quickly adopted by programmers. It solved a problem in a nice, convenient way.

Core Concepts and Features

In the previous example, job sequences have been mentioned. Let us look at what other constructs are provided in the library.

Sequences are a specialized form of job collections. Job collections are containers that queue a set of jobs in an atomic operation and notify the program about the whole set. Job collections are composites, in the way that they are implemented as job classes themselves. There is only one queuing operation in ThreadWeaver: it takes a Job pointer. Composite jobs help keep the queue API minimal.

Job sequences use dependencies to make sure the contained jobs are executed in the correct order. If a dependency is declared between two jobs, it means that the depending job can be executed only after its dependency has finished processing. Since dependencies can be declared in an m:n fashion, pretty much all imaginable control flows of depending operations (which are all directed graphs, since repetition of jobs is not allowed) can be modeled in the same declarative fashion. As long as the execution graph remains directed, jobs may even queue other jobs while being processed. A typical example is that of rendering a web page, where the anchored elements are discovered only once the text of the HTML document itself is processed. Jobs can then be added to retrieve and prepare all linked elements, and a final job that depends on all these preparatory jobs renders the page for display. Still, no mutex necessary. Dependencies are what distinguish a scheduling system like ThreadWeaver from mere tools for parallel processing. They relieve the programmer of thinking how to best distribute the individual suboperations to threads. Even with modern concepts such as futures, usually the programmer still needs to decide on the order of operations. With ThreadWeaver, the worker