25

Expressing Workflow in the Cactus Framework

Tom Goodale

25.1 Introduction

The *Cactus Framework* [15, 73, 167] is an open-source, modular, portable, programming environment for collaborative HPC computing. It was designed and written specifically to enable scientists and engineers to perform the large-scale simulations needed for their science. From the outset, Cactus has followed two fundamental tenets: respecting user needs and embracing new technologies. The framework and its associated components must be driven from the beginning by user requirements. This has been achieved by developing, supporting, and listening to a large user base. Among these needs are ease of use, portability, the ability to support large and geographically diverse collaborations and to handle enormous computing resources, visualization, file IO, and data management. It must also support the inclusion of legacy code, as well as a range of programming languages. It is essential that any living framework be able to incorporate new and developing cutting edge computation technologies and infrastructure, with minimal or no disruption to its user base. Cactus is now associated with many computational science research projects, particularly in visualization, data management, and Grid computing [14].

Cactus has a generic parallel computational toolkit with components providing, e.g., parallel drivers, coordinates, boundary conditions, elliptic solvers, interpolators, reduction operators, and efficient I/O in different data formats. Generic interfaces are used (e.g., an abstract elliptic solver API), making it possible to develop improved components that are immediately available to the user community. Cactus is used by numerous application communities internationally, including numerical relativity e.g. [26], climate modeling [118, 404], astrophysics [32], biological computing [211], computational fluid dynamics (CFD) [239], and chemical engineering [74]. It is a driving framework for many computing projects, particularly in Grid computing (e.g., GrADS [12], GridLab [175], GriKSL [176], ASC [32, 57]).

Also, due to its wide use and modular nature, Cactus is geared to play a central role in general dynamic infrastructures.

Although Cactus is distributed with a unigrid MPI parallel driver, codes developed in it can also already use multiple *adaptive* mesh-refinement drivers *with minimal or no changes to the code*, including Carpet [80, 380], PAGH GrACE [170]), and SAMRAI [282, 375].

25.2 Structure

As with most frameworks, the Cactus code base is structured as a central part, called the "flesh" which provides core routines and components called "thorns" .

The flesh is independent of all thorns and provides the main program, which parses the parameters and activates the appropriate thorns, passing control to thorns as required. It contains utility routines that may be used by thorns to determine information about variables and which thorns are compiled in or active, or perform non-thorn-specific tasks. By itself, the flesh does very little apart from move memory around; to do any computational task the user must compile in thorns and activate them at runtime.

A thorn is the basic working component within Cactus.[1] All user-supplied code goes into thorns, which are, by and large, independent of each other. Thorns communicate with each other via calls to the flesh API plus, more rarely, custom APIs of other thorns. The Cactus component model is based upon tightly coupled subroutines working successively on the same data, although recent changes have broadened this to allow some element of spatial workflow.

The connection from a thorn to the flesh or to other thorns is specified in configuration files that are parsed at compile time and used to generate glue code that encapsulates the external appearance of a thorn. Two thorns with identical public interfaces defined in this way are equivalent.

At runtime, the executable reads a parameter file that details which thorns are to be active, rather than merely compiled in, and specifies values for the control parameters for these thorns. Inactive thorns have no effect on the code execution. The main program flow is shown in Figure 25.1.

25.3 Basic Workflow in Cactus

In most existing workflow systems component composition is achieved by specifying the components and their connections in some workflow language or

[1] Thorns are organized into logical units referred to as "arrangements." Once the code is built, these have no further meaning — they are used to group thorns into collections on disk according to function, developer, or source.

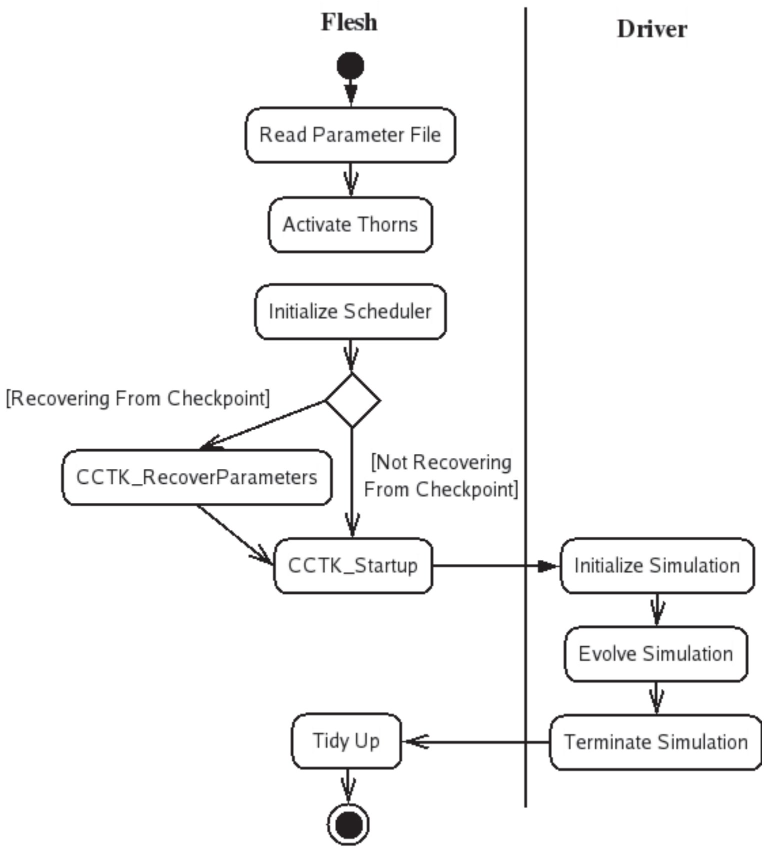

Figure 25.1: The main flow of control in the Cactus framework. The flesh initializes the code and then hands control to the driver thorn (see Section 25.3.2). The actions in the driver swimlane are detailed in Figures 25.2, 25.3, and 25.4.

through a graphical user interface, a paradigm familiar to most users. Cactus component composition however, is a function of the flesh, guided by rules laid down by developers when they develop their thorns.

Cactus defines a set of coarse scheduling bins, as shown in Figures 25.1–25.4[1]; routines from a thorn are scheduled to run in one of these bins relative to the times when other routines from this thorn or other thorns are run. The thorn author may also schedule groups within which routines may be

[1] Future versions of Cactus will allow thorns to specify additional top-level scheduling bins.

scheduled; these groups are then scheduled themselves at time bins or within schedule groups analogously to routines.

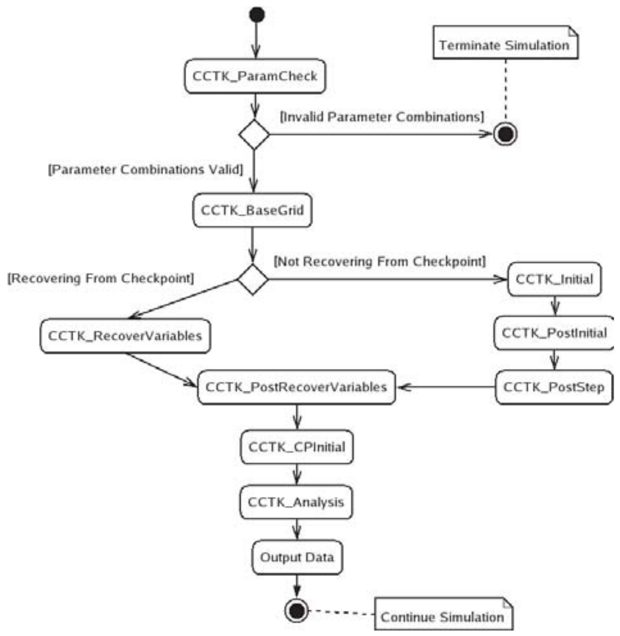

Figure 25.2: Initialization action diagram (corresponding to the *initialize simulation* action in Figure 25.1). All activities prefixed with "CCTK_" are schedule bins (see Section 25.3).

Routines (or schedule groups — for scheduling purposes they are the same) scheduled from thorns may be scheduled *before* or *after* other routines from the same or other thorns and *while* some condition is true. In order to keep the modularity, routines may be given an alias when they are scheduled, thus allowing all thorns providing the same implementation to schedule their own routine with a common name. Routines may also be scheduled with respect to routines that do not exist, thus allowing scheduling against routines from thorns or implementations that may not be active in all simulations. Additionally, the `schedule.ccl` file may include `if` statements which only register routines with the scheduler if some condition involving parameters is true.

Once all the routines have been registered with the scheduler, the before and after specifications form a directed acyclic graph, and a topological sort is carried out. Currently this is only done once, after all the thorns for this simulation have been activated and their parameters parsed.

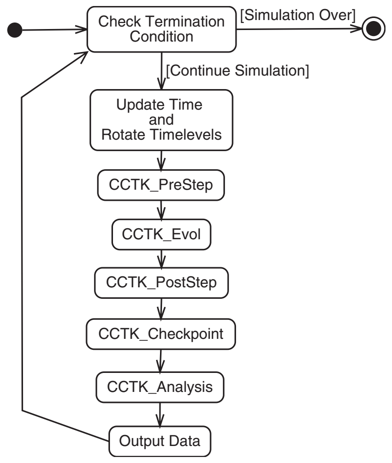

Figure 25.3: Evolution action diagram (corresponding to the *evolve simulation* action in Figure 25.1). All activities prefixed with “CCTK_” are schedule bins (see Section 25.3).

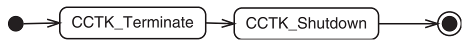

Figure 25.4: Termination action diagram (corresponding to the *terminate simulation* action in Figure 25.1). All activities prefixed with “CCTK_” are schedule bins (see Section 25.3).

This scheduling mechanism is rule-based as opposed to script-based. There are plans to allow scripting as well; see Section 25.4.3 for further discussion of this.

25.3.1 Conditional Scheduling and Looping

The scheduling of routines may currently be made conditional by two methods: at schedule creation time based on parameters having certain values, using `if` statements in the CCL (Cactus Configuration Language) file; and at schedule execution time by use of a `while` statement. More sophisticated flow control will be possible in the future.

An `if` clause in the schedule specification controls whether a routine is registered with the scheduler for running at any point. These `if`s are based upon algebraic combinations of the configuration parameters read in at program startup and thus are only evaluated once.

The `while` specification allows for a degree of dynamic control and looping within the schedule, based upon situations in the simulation, and allows looping. A routine may be scheduled to run while a particular integer grid scalar is nonzero. On exit from the routine this variable is checked and if still true, the routine is run again. This is particularly useful for multistage time integration methods, such as the method of lines, which may schedule a schedule group in this manner.

25.3.2 Memory Management

The language (CCL) used to define the scheduling also defines the variables passed between the routines invoked by the scheduler. This allows memory and parallelization to be managed by a central component (the *driver*) and also provides support for legacy codes written in languages, such as FORTRAN 77, that lack dynamic memory support.

When scheduling routines, the developer specifies which variables require memory allocated during the course of a particular routine. Memory for variables may be allocated throughout the course of the simulation or just during the execution of a particular scheduled routine or schedule group; specifying memory just for a group has no effect if the memory was already allocated for that variable.

25.3.3 Spatial Workflow

Thorn authors may define specific functions that they provide to other thorns or expect other thorns to provide. This provides an aliasing mechanism whereby many thorns may provide a function that may be called by another thorn with a particular name, with the choice of which one is actually called being deferred until runtime.

A thorn using such a function may state that it requires the presence of the function or that this function is optional. If it is required, the flesh will produce an error and stop code execution after all thorns have been activated if none of the activated thorns provides the function; if it is optional, the thorn using it must make its own check before calling the function.

25.4 Extensions

Cactus defines a basic scheduling mechanism for tightly coupled simulations. It is, however, possible to use Cactus in more loosely coupled situations. This section describes two such applications: large-scale distributed task farming and the use of Cactus within other frameworks.

25.4.1 Task Farming

Many problems are amenable to a *task farming* approach, whereby a large number of independent tasks are started and their results collated;

e.g. Monte Carlo simulations and parameter searches. The Cactus task farming infrastructure allows many independent processes to be started across a heterogeneous set of resources; these processes need not be Cactus applications.

In order to deal efficiently with startup costs on remote resources, such as security and batch queues, the Cactus task farming infrastructure makes use of a three-level approach, in contrast with a classical master–slave two-level approach. The user starts a Task Farm Manager (TFM) on a machine that has good connectivity to the outside world, or at least to the potential resources, and this TFM then finds resources and starts slave TFMs (e.g., by submitting to a batch queue) on the resources; in principle, this can be repeated, so we number the generation of TFMs—TFM0 is the master, TFM1s are the first-generation child TFMs, etc. When a TFM1 starts, it contacts the TFM0 and requests tasks, based upon the resources it has allocated to it. For example we may wish to run 500 two-processor tasks. The TFM finds two resources — a 100 processor queue on one machine (A) and a 400 processor queue on another machine (B) — and queues TFM1s for these machines. When the TFM1 on machine A starts, it requests 50 tasks from the TFM0.

A TFM is just a running instance of Cactus containing two particular components: a core TFM thorn providing the application-independent part of the task farming, such as choosing resources and starting child TFMs or tasks themselves, and an application-specific part, referred to as a `logic manager`, which provides the application-specific information. Figure 25.5 shows the logical relationship of the thorns and shows the interface that a logic manager must expose to the TFM:

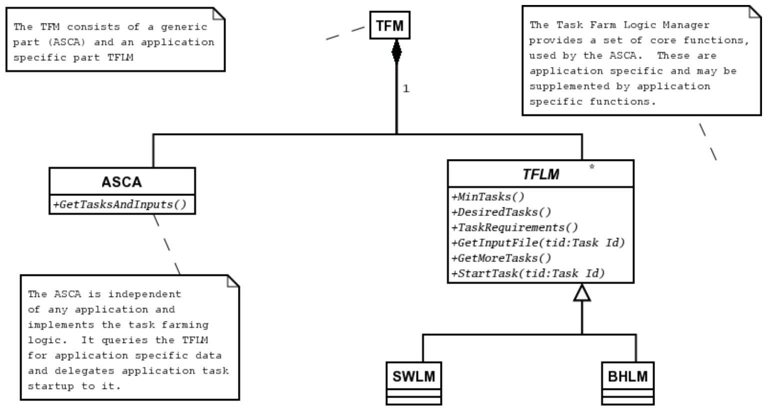
The TFM consists of a generic part (ASCA) and an application specific part TFLM
TFM
1
The Task Farm Logic Manager provides a set of core functions, used by the ASCA. These are application specific and may be supplemented by application specific functions.
ASCA
+GetTasksAndInputs()
TFLM
*
+MinTasks()
+DesiredTasks()
+TaskRequirements()
+GetInputFile(tid:Task Id)
+GetMoreTasks()
+StartTask(tid:Task Id)
The ASCA is independent of any application and implements the task farming logic. It queries the TFLM for application specific data and delegates application task startup to it.
SWLM
BHLM

Figure 25.5: The “classes” making up a Task Farm Manager. SWLM and BHLM are application specific logic managers.

- *MinTasks.* Returns the minimum number of tasks that must be run simultaneously. For many applications, this would be 1; however for applications where tasks exchange data between them, such as a distributed Smith–Waterman algorithm, a larger number is necessary.
- *DesiredTasks.* In an ideal world, it would be possible to specify the maximum number of tasks that will be run, search for the required resources, and run them all simultaneously; in practice, however, there are not infinite resources, and, for a guided parameter search, the number may not be known in advance. This function returns the application's (or application programmers') best guess for reasonable resource requirements.
- *TaskRequirements.* This function returns the number of processors and amount of memory required by each task.
- *GetInputFiles.* Given a task ID, this function returns the command-line arguments for starting the task and a list of URLs of files that must be staged to the task's working directory before it starts.
- *GetMoreTasks.* The TFM uses this to retrieve more task IDs from the logic manager whenever tasks finish.
- *StartTask.* Some tasks require to be started up in special ways (e.g., using mpirun); this function allows the logic manager to customize the startup. This is invoked on the TFM1.

The first three functions are used by the TFM0 to determine the characteristics of the tasks before searching for resources; `GetMoreTasks` and `GetInputFiles` are used by the TFM0 whenever a TFM1 requests more tasks; and `StartTask` is used by the TFM1 to start individual tasks. See Figure 25.6 for a diagram showing the interaction sequence of the various thorns and processes.

25.4.2 Connection to Other Frameworks

The Cactus framework is designed around the needs of tightly coupled, high-performance simulations. The majority of the other frameworks included in this book deal with large-scale distributed workflows, and Cactus can easily be integrated as a component within such a workflow.

Triana

Within Cactus, users typically want to view or analyze certain files to monitor the progress of the application or derive scientific results. Cactus can output data in many formats, such as HDF 5 files, JPEGs, and ASCII formats suitable for visualising with common tools such as X-Graph or GNUPlot. A user would typically want the flexibility of being able to choose, at runtime, the files he or she wishes to view or analyze in an interactive fashion. For example, a user may notice from the JPEG images that a simulation of a system consisting

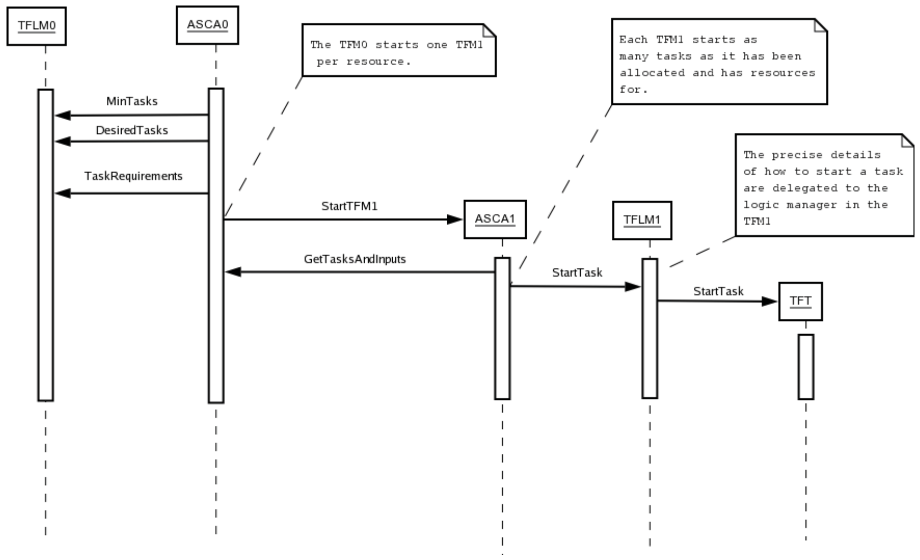
TFLM0
ASCA0
The TFM0 starts one TFM1 per resource.
Each TFM1 starts as many tasks as it has been allocated and has resources for.
The precise details of how to start a task are delegated to the logic manager in the TFM1
MinTasks
DesiredTasks
TaskRequirements
StartTFM1
ASCA1
TFLM1
GetTasksAndInputs
StartTask
StartTask
TFT

Figure 25.6: The sequence of actions performed in the startup of an application using the Cactus task farming infrastructure.

of two orbiting sources is showing the sources coalescing; this user may then wish to verify these findings by retrieving the detailed simulation data and passing them to other analysis tools or even converting the output to an audio format and listening to the acoustic waveform directly. Our protocol therefore supports the dynamic notification necessary for such interactions. When a file is created, the Web service deployed within a Triana unit is notified, and at each time step, the Web service is contacted and can choose to receive any of the files that are available. By default, the application only sends differences in text files since the last time the Web service received part of the file, thus reducing bandwidth; binary files are transferred in their entirety. If something interesting happens, the Web service can select and receive a different set of files in the next iteration.

This is aided by the use of the Triana problem-solving environment, which allows components to be dynamically added/removed as the application runs. Within Triana, a unit was created to host the Web service representing the underlying protocol. This is shown in Figure 25.7. The unit upon initialization uses WSPeer [187] to dynamically create and deploy the Web service within the Axis environment and create the necessary WSDL file representing the methods within the protocol. The actual protocol is quite simple. It involves a notification and selection procedure but is carefully designed so that it is completely application (i.e., Cactus) driven. This ensures that we do not run into firewall issues.

In our initial development, Triana and the Cactus application are deployed and instantiated independently, which is a useful model for occasional monitoring of an application's progress, as it allows a user to make a later decision to use Triana to monitor the output. In the full usage scenario we envisage, however, a Triana unit would also be used to deploy Cactus on a remote resource on demand, thus allowing Triana to manage the full life cycle of the workflow, as is done in other workflow management systems.

In this current stage of deployment, we are using one Cactus Triana unit per Cactus instance running on the Grid. This approach is not scalable in the visual sense (e.g., imagine trying to visualise several thousand Triana units) or in the networking sense (e.g. having thousands of local instances of the same Web service would be impractical for hosting environments). To address these issues, we are currently planning on building a scalable Cactus unit that allows many instances to be mapped internally within one Web service instance. We imagine that this would build around the Triana dynamic scripting or looping implementation (to hide the visual complexity) and then such instances would be mapped using proxies to a Cactus Triana unit instance for that script or loop. This would allow the connection of possibly hundreds of instances.

If more instances are needed, then we can use the Triana distributed mechanisms [408] to segregate the workflow and run it across several Triana GAP services across the Grid, allowing potentially many thousands of instances. However, the algorithmic problem of how these results are analyzed would be application-specific. Within one scenario involving Cactus, we imagine that Triana would be monitoring the output of its results to see if something interesting had happened (e.g., the apparent horizon of a black hole simulation). Then Triana would invoke a separate workflow to farm off many independent Cactus simulations to investigate this phenomenon more closely and then analyse the results upon completion. The user would only wish to view when a certain optimization level has been reached.

A prototype of this protocol has been demonstrated in SC2004 and SC2005, where we showed the visualisation of a 3D scalar field produced by two orbiting sources. This was accomplished by using this protocol to connect a Cactus simulation, running on an HPC resource, and Triana, running on a users workstation. Triana received notifications of the files created by Cactus and then selected the ones it wished to visualize. The result was that the user could see real-time JPEG images from the remote application, representing the three dimensions of the scalar field, as the simulation progressed.

25.4.3 Future Directions

Another enhancement would be to allow scripting as an alternative to the current scheduling mechanism. The current mechanism allows thorns to interoperate and for simulations to be performed with the logic of when things happen encapsulated in the schedule CCL file; other frameworks do the same thing by providing a scripting interface, which gives more complete control

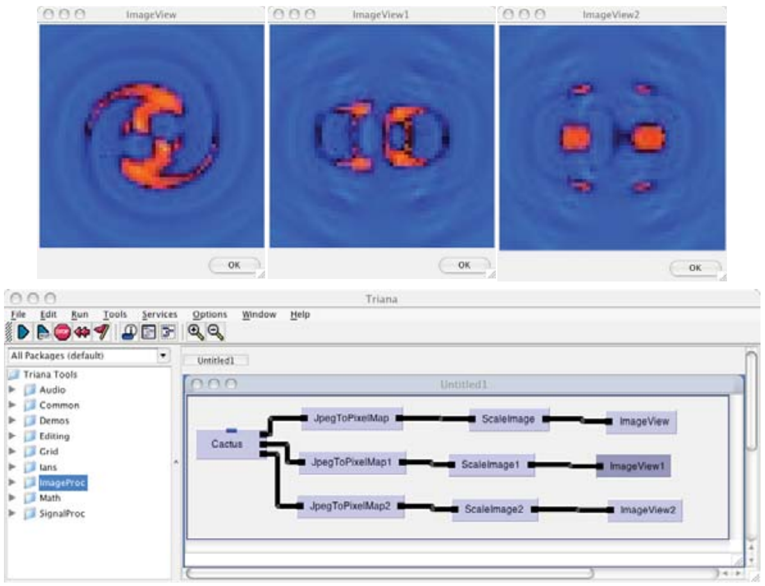

Figure 25.7: The resulting visualization from a Cactus simulation of the evolutions of a 3D scalar field.

of the flow of execution, at the expense of the user needing to know more of the internals. Both schemes have advantages and disadvantages. In the future we would like to allow users to script operations using Perl, Python, or other scripting languages.

Automated Composition of Workflows

While in Cactus composition of workflow currently consists of activating the requisite thorns, as the size and complexity of workflows increase it may becomes difficult or impossible for a human to create the workflow explicitly. Future versions of Cactus will address this problem by providing components with semantic information that can be used to automatically compose workflows and allow automated recomposition to be triggered on demand. We plan to provide the ability to take a set of software components and a task specification and determine the appropriate composition and configuration of these components.

Distributed Component Level Debugging

Debugging large distributed applications is hard. On single systems or clusters, tools such as TotalView [136] are very useful; however, these do not scale well to the wide-area heterogeneous component-based simulations currently being developed. There are currently plans to develop the component interfaces in Cactus to allow single stepping at the component level and tracing data flow into and out of components, and to add features to allow debugging through familiar mechanisms, such as breakpoints, trace variables, stepping through workflows, or even dynamically reconfiguring workflows.