SAND2015-0425C

INCORPORATING WORKFLOW FOR V&V/UQ IN THE SANDIA ANALYSIS WORKBENCH

Dr. E. Friedman-Hill, E. Hoffman, M. Gibson, Dr. R. Clay
(Sandia National Laboratories[1], USA)

K. Olson
(SAIC, USA)

1. Introduction

Numerous teams at Sandia National Laboratories develop in-house codes for problem setup and computation in multiple domains of physics and engineering. These codes are actively developed, continuously evolving, and require significant, ongoing learning on the part of users. The Sandia Analysis Workbench (SAW) is a family of desktop applications that was developed to provide an integrated interface to many of these codes, and to improve their ease of use by providing contextual information and tools that simplify and streamline common operations. Using the Workbench, users can build models, submit and manage HPC jobs, visualize results, share and store models and results in context with versioning and dependency tracking, and more. Because a diverse array of in-house and commercial tools must be integrated using limited resources, we use a component-based, data-driven approach in which tools are treated generically and customization is handled as much as possible by configuration files. This approach allows developers to concentrate on specific areas with high value for the end user.

While many of the codes integrated into the Workbench have scripting capabilities, overall sequencing of code execution – or *workflow* -- has until now been handled manually. A Workbench user invokes operations like model building, job submission, and post processing as distinct, deliberate actions. While sufficient for simple use cases,

[1] Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

manual workflow becomes burdensome for more complex situations, and does not allow users to repeat, archive, and share workflows.

Recently, we began to recast our architectural components as nodes in an explicit workflow graph. The addition of a robust, configurable workflow engine supports validation, verification, and uncertainty quantification (V&V/UQ) activities by allowing combinations of Workbench components to be used for repeatable, automated executions of parameterized models, enabling sensitivity analysis, optimization, and other compound operations with minimal user effort.

In this paper, after first presenting some motivation from the literature, we will discuss the SAW architecture and give an overview of the data-driven, component-based strategies we use for integrating tools. We'll also discuss our planned architecture for adding a workflow capability, our initial implementation, and the challenges we will face as we scale the system up for future simulations in the exascale regime.

2. Motivation

Page, Canova, and Tufarolo (1997) noted three categories of modelling and simulation (M&S): live, virtual, and constructive, where "live" referred to simulations involving live individuals, virtual referred to individuals interacting with simulators, and constructive referring to simulations invoked by executable computer programs. For the purposes of this paper, only "constructive" M&S is fully considered, and is frequently referred to as *computational simulation*. This distinction is important, as considerations of iterative, executable workflows and maturity models may have limited applicability to the other categories.

Sargent (2013) provided a useful general (though simplified) description of the M&S development process. It contained three major components: the problem entity (the system to be modelled), the conceptual model (a logical representation of the problem entity), and the computerized model. Each of these three entities is connected through data, and provides for iterative feedback between them. This approach also implicitly overlaid a lifecycle on the model, in that at any given time a model may be considered being in one of the component areas. Furthermore, Sargent specifically noted the need for V&V activities at each of the component areas. This particular process does not directly consider cases where the problem entity may have more than a single conceptual model, or the conceptual model may have more than one computerized model. In such a case, the lifecycle "state" would be tied to a particular expression. Such an expanded multi-faced expression, however, re-emphasizes the need for good V&V/UQ activity tracking in order to ensure than a given conceptual or computerized model remains consistent with the problem entities' requirements. In addition, it is clear that diverse expressions of the conceptual or computerized model may entail different assumptions. Maintaining full traceability between these assumptions, the V&V/UQ evidence, and the original problem entity is critical in order to support the credibility of the final results.

Conwell, Enright, and Stutzman (2000) extended Sargent's M&S process model, by emphasizing the V&V/UQ activities necessary to support the process. The starting point for the problem entity in this model is a requirements definition specifying the operational needs in terms of functionality, fidelity, and credibility. From this starting point, requirements traceability through the V&V/UQ activities is the cornerstone of this enhanced process. The authors also augmented the model with capability maturity model (CMM) points. They made explicit that certain key process areas backed with requirements traceability can result in an increased and documentable CMM level. In general, more repeatable and documented processes following identified key process activities (KPAs) will result in a higher CMM level. As Osman Balci, Nance, Arthur, and Ormsby (2002) noted, it is likely that an M&S executed by an CMM Level 3 organization will have more credibility than developed by a Level 1 organization. In addition to "cost and effort reduction that may be available once standardized practices are available" (Osman Balci et al., 2002), it is also implied that such processes will reduce the full potential scope of V&V/UQ activities. It is an open question as to whether organizations that have great diversity in the M&S activities can standardize across the entire organization, and to a certain extent accept the implied or explicit constraint of V&V/UQ activities.

Restriction of the tools, V&V/UQ selection, and how they operate in concert is not necessarily an impediment. Allen, Shaffer, and Watson (2005) noted that pulling modelling tools into an integrated environment (IDE) can reduce risk by encouraging particular tool and process usage. In addition, undertaking M&S activities within a defined IDE can directly support model branching while maintaining traceability. Such an IDE also supports the oft-neglected aspect of *traceability*, in which storage of the artifacts from a V&V/UQ activity provides the evidence that ultimate lends credibility to the model. In other words, a world in which an individual can choose any tool and any V&V/UQ activity at any point requires that individual to be a supreme librarian. IDEs which provide validating and structured editors, workflow execution, and artifact management (which may be considered a part of an overall configuration management requirement for the evolving and potentially multiple models) are likely to be more effective at ensuring V&V is incorporated throughout the lifecycle.

The ability to select particular V&V/UQ activities at a given point through an IDE suggests the ability to realize the suggestions of Wang (2013). As this author noted, "conducting model verification and validation requires systematic selection and application of different V&V techniques throughout the M&S life cycle" (p. 1233). The selection criteria, however, are not necessarily obvious in the absence of guidance. As Wang noted, there are more than 100 potential V&V activities, but in practice only a limited number are utilized. Vegas and V. (2005) suggested that lack of theoretical or empirical knowledge is often the basis for this limited number. O. Balci (1998) proposed a taxonomy for V&V/UQ activities that had four groups: informal, static, dynamic, and formal. Wang (2013) noted that a V&V/UQ activity is dependent not only on the characteristic of the technique, but also the context of its application. In addition, Wang provided an explicit cost component to the V&V activity selection. As suggested above, model iterations will require re-execution of a V&V/UQ activity in order to maintain the application of V&V throughout the lifecycle. As a result, selection of a V&V activity must be cost-effective in reference to model as well.

Executable workflows tied to V&V/UQ activities (as opposed to the overall model lifecycle) may be especially important given the inherent iterative nature of model development. Traceability provides the guidance for when an assumption (or requirement) at one point drives changes throughout the entire model chain. V&V/UQ executable workflows allow re-executing V&V/UQ tasks in a structured, repeatable fashion to ensure the model remains credible within the existing requirements.

3. The Sandia Analysis Workbench

Given the best practices discussed in the preceding section, we chose an implementation strategy based on the idea of an integrated development environment for model development and execution. Our organization's structure as a loose federation of science and engineering groups led us to adopt an architecture based on independent and highly separable components. Some components are self-contained, while others are *wrappers* for in-house or third party tools.

Our range of integrated applications and the architecture they are built on are collectively referred to as the Sandia Analysis Workbench (SAW). We build a flagship "Enterprise Edition" desktop application which incorporates almost all of our components, and also a number of smaller, lighter-weight editions that each include tools useful in specific narrower user domains. In this paper we'll primarily discuss the general architecture we've used to create this collection of modular components.

Eclipse. Our architecture is based on the Eclipse framework. The Eclipse IDE was originally developed at IBM as a Java development environment. The core of Eclipse (itself written in the Java language) was later extracted and became the basis of a general application framework for building modular applications. The Eclipse Platform, built on the OSGi component framework, provides a complete set of primitives for managing the lifecycles and interactions of a system of separate but complementary components. The Eclipse Rich Client Platform (RCP) implements a graphical interface on top of that framework, Applications built using the Eclipse Framework and the RCP are portable to many platforms and include both graphical desktop applications and headless server applications. Our set of integrated applications contains both graphical and non-graphical instances, as will be discussed.

The fundamental unit of composition in Eclipse is the OSGi *plugin*. Each plugin is a separately loadable software unit. A minimal plugin can contain nothing but declarative data stored in a manifest file, but most plugins contain Java code, Java or native libraries, images, scripts, and other data. A plugin can have no user interface, or optionally it can have a graphical user interface (GUI) that appears within the IDE "workbench." It can also contribute menu items or other additions to customize the GUIs offered by other plugins. As used in SAW, each tool to be integrated is implemented as one or more plugins. Typically this small group of plugins – which we loosely refer to as a *component* -- will have compile-time dependencies among themselves, but will not directly depend directly on plugins supporting other tools.

A plugin can depend on and interact explicitly with other plugins, but ideally plugins interact more abstractly through the use of Eclipse *extension points.* An extension point is a declarative (XML) description of a service that one plugin can offer to another. A plugin satisfies an extension point by implementing an *extension*. It is possible to query the Eclipse framework for all the extensions that implement a particular extension point. In this way, a consumer of a service (as defined by an extension point) can do so without compile-time knowledge of any plugins that provide that service. This means that multiple applications can be composed by selecting from a set of components, according to user needs. Each component can discover at runtime the providers or consumers of any services it involves. Component developers concentrate on delivering specific services and need not worry about how those services will be combined.

This architecture based on plugins and extension points allows us to use a technique called a *bridge plugin* to allow individual teams to develop separate but interacting plugins while operating with a great deal of autonomy and minimizing the need for communication between groups. A bridge plugin is a plugin A that implements an extension point declared in a plugin B in terms of the capabilities of a third plugin C. In this way plugins B and C (which generally interface to different tools and are created by different development teams) can directly interact even though the plugins have no interdependencies and the teams implementing them may in fact be completely unaware of one another.

Our set of components includes wrappers for the SIERRA suite of analysis codes as well as a few other analysis codes, for the CUBIT meshing and geometry library (Owen, 2009), the DAKOTA optimization library (Adams, et al, 2014), and other tools. It also includes components for general model building, parameter management, response extraction, data management, requirements management, remote computational job submission and monitoring, remote visualization, and more.

Declarative Component Definition. Many of our wrapper components are quite detailed and contain significant information about an external tool. While users want to see a GUI that exposes all the capabilities of the wrapped tool, hard-coding the necessary information (often in the form of input file syntax) would be both prohibitively expensive and very fragile as the codes evolve and syntax is added and removed. As a result, whenever possible we use a data-driven or declarative approach in which the syntax of a code is described in a data file, and graphical interfaces are generated at runtime from that description. Besides the reduction in implementation effort, ancillary benefits include a consistent appearance of generated GUI panels and easier and more complete testing and validation.

In some cases the developers of the wrapped code create the syntax definition file or can maintain it themselves, but in other cases that task falls to the integration team. It is still often better to use a declarative approach because changes in the wrapped code are easier to track and test.

We do not mandate a format for declarative description of input syntax, but rather try to accommodate formats created by various other development teams. Most tools use some form of XML. SAW includes several code generators that are driven by these various formats.

It is often the case that specific features of a code suggest a unique graphical interface presentation that cannot be specified within a simple general-purpose description format. As way of maximizing both optimal user presentation and simplicity of the format, we provide escape mechanisms for these cases in which hand-coded GUI panels can replace generated ones for specific syntax features. We have found this to be an optimal compromise between generated and hand coded GUIs.

Data Management. One central component of the Sandia Analysis Workbench is the Workbench Server that stores data in a commercial product data management (PDM) system. Our data management component (which interfaces to the Workbench Server) provides versioned storage, maintains relationships between artifacts, and is the basis for data security. Our data management model stores everything in a *project*. A project has an associated team that can access those files; access can be granted to individuals outside the team using access control lists integrated with our Laboratory's directory system.

Our data management system was originally developed for interactive use, but over time have been interfaced to other tools in the Workbench including job submission and requirements management; in both cases the interface to data management adds useful capabilities to the other tools. All parts of a model and all related resources can be stored in context in our data management system.

4. Adding a Workflow Engine

A previous section of this paper discussed the value of an IDE for modelling and simulation, and feel that we have created a capable IDE for M&S at our organization, to the extent that our collection of components provides easy access to a range of tools and aids in their use both singly and in combination. To take the next step we are adding automated workflows that orchestrate the use of all of our components in a robust and repeatable way. By doing so we hope to ease the introduction of more rigorous V&V/UQ into our M&S process as well as to improve quality and traceability.

To that end, we have begun to map our set of components onto workflow elements that can be composed into workflows and executed under the control of a formal workflow engine. In this section we'll discuss the capabilities of a workflow engine, discuss our planned architecture, and describe the current status of our implementation.

Definition of Workflow. If we adopt the definition of an *action* as something that can be executed, a *process definition* as a directed graph of interconnected actions that need to be performed, and a *process* as an actual instantiation of a process definition, then a *workflow engine* is just software which can load a process definition and from it, generate and execute a process. A workflow engine will generally offer the ability to connect actions in various ways, to force some actions to wait for the completion of others, and to retry or restart failed actions. A robust workflow engine will be able to persist the state of a process and report on it as it runs. Typically a workflow engine will include ancillary software to assist in creating process definitions, often in the form of a graphical builder.

Individual actions generally need to communicate: they must communicate their status with the workflow engine, and often must communicate data with other actions. In the typical directed graph illustration of workflow (see Figure 1) it is important to realize that the edges represent sequencing of actions and do not necessarily represent data flow. The two kinds of communications (with the engine, and with other actions) can and often do occur via separate channels. Communication with the workflow engine is often implicit; an action can communicate its status simply by completing, as when a launched program exits.

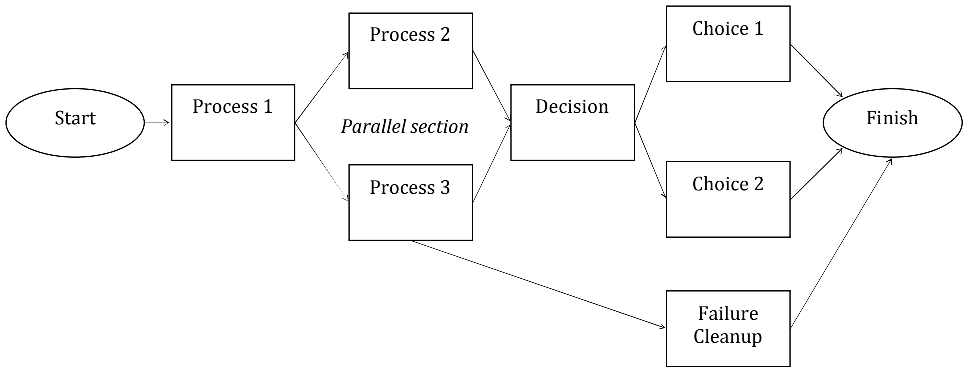

Figure 1: A diagram of a simple workflow showing various arrangements of interconnected nodes, including parallel tasks, branches, and error handlers. It is important to realize that the edges represent sequencing of actions and do not necessarily represent data flow.

Proposed Architecture. Notionally, our proposed architecture for M&S workflows is simple. Individual existing components within the SAW framework implement a Java interface defining the characteristics of an action, including required inputs and outputs. A new workflow component then makes these actions available to an embedded workflow engine. The engine executes workflows by invoking the components as actions, and each component is responsible for communicating in its own way with any wrapped external tool, launching and monitoring it locally or remotely as required and reporting its status back to the workflow engine.

The specification of each input and output will of necessity be fairly rich. In addition to a name, each input and output must have a data type (integer, character string, JPEG file) and a description of acceptable communication channels. For example, some tools may accept inputs via files on disk, others via UNIX pipes, and others by command-line arguments; some may be able to use more than one channel. Any information about the specified communication channel to be used by a process is communicated to the components; it is then up to the components to establish the channel and transfer the required data.

Process execution should be possible both in a graphical interactive mode and in a detached mode where the engine runs unattended on a server. Therefore neither the actions themselves nor the workflow engine must require the presence of a GUI.

5. Current Status

As the first step in realizing the architecture described in the previous section, we have embedded a third-party workflow engine into our framework. At this time we are using Google Sarasvati, although we are not tied to this product and may change it in the future (other candidates include Activiti and jBPM). We provide a standard interface for an action, and currently have several implementations based on existing plugins in our framework. Our action interface is still evolving as we gain experience with our implementation.

Currently, we don’t expose a process description editor to the user. Our process description is generated internally based on a simulation model as built by the user, and is generally quite simple (see figure 2). There is a sequence implicit in the combination of codes invoked as part of the model, and so our process definitions are based around this simple linear process definition. The main process definition is then augmented by the addition of *response* elements added by the user, which are basically probe calculations that can be attached to other components.

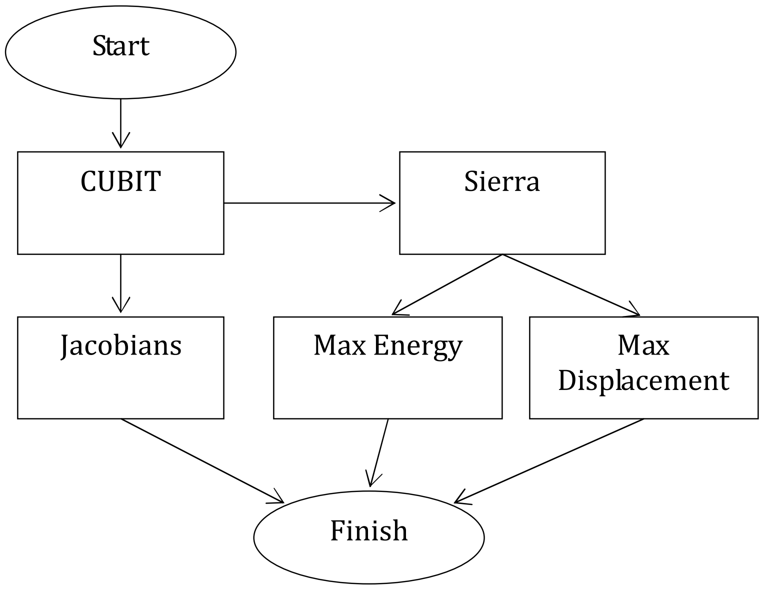

Figure 2: A typical process definition generated by our current workflow component. A simple one- or two- step meshing/simulation workflow is augmented by the addition of an arbitrary number of "response" elements (here, computing Jacobians for the mesh and extracting two quantities from the simulation results) created by the user. These workflows can be run singly or as part of a parameter study or optimization loop using DAKOTA.

The communication channel type for many of our components is implicitly file-based. Components in the Workbench marshal files for their wrapped tools and ensure that appropriate paths are passed from one action to the next. This works because all of the wrapped tools are inherently file-based as well.

Our system can run multiple concurrent processes. Because many actions actually consume little or no local CPU during execution (since they are merely a wrapper for remote execution of a simulation code,) this presents so significant problems. However, our complete workflow system cannot currently be run in a non-graphical mode; it must execute in the context of an interactive application. This places a practical limit on the length of time that any one process can run, and also on the number of concurrent processes that can be executed.

Even given the simple nature of our generated workflows, we are already seeing some real benefits. It has now become a simple task to use our DAKOTA component to create and execute a wide range of sensitivity studies and optimizations that combine parameterized mesh generation with a solid mechanics, thermal or structural dynamics simulation. Our current implementation provides an excellent laboratory for testing strategies and techniques for robust execution.

6. Future Work

As previously stated, our workflow system is in its infancy, but is already producing useful results. Our current work centers on removing many of the limitations of our existing implementation. For example, we are adding the ability for users to create custom workflow components, as well as to build complete workflows of their own design. We are also working toward a non-graphical host for the workflow actions and engine.

One significant challenge that lies ahead is scalability to larger and larger simulations. In the near future (or indeed, even now,) storing simulation results in serial files will become impractical or impossible. Our architecture specifically addresses this by abstracting the notion of a communication channel such that a process description contains only a specification of the type of channel to be used, and processes augment that specification with only some channel parameters. It is expressly not the responsibility of the workflow engine to marshal data from one action to the next, but only to provide them with the information needed to communicate. As scalable I/O channels are created to enable exascale communication between tools, the workflow action input and output specifications can offer those mechanisms as options in building a process definition. Our data management system will also need to use a broader definition of data sets which can account for “virtual” data sets which represent a link in a dependency or provenance chain but which are too large to be preserved.

7. References

Adams, B., Ebeida, M., Eldred, M., Jakeman, J., Swiler, L., Stephens, J., Vigil, D., Wildey, T., Bohnhoff, W., Dalbey, K., Eddy, J., Hu, K., Bauman, L., and Hough, P., (2014), Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.1 User’s Manual, Sandia Technical Report SAND2014-4633.

Allen, N.A., Shaffer, C.A., & Watson, L.T. (2005). Building modeling tools that support verification, validation, and testing for the domain expert. *Proceedings of the 37th conference on Winter simulation*. Orlando, Florida (pp. 419-426).

Balci, O. (1998). Verification, validation, and testing. In J. Banks (Ed.), *Handbook of simulation* (pp. 335-393): John Wiley & Sons.

Balci, O., Nance, R.E., Arthur, J.D., & Ormsby, W.F. (2002). Improving the model development process: Expanding our horizons in verification, validation, and accreditation research and practice. *Proceedings of the 34th conference on Winter simulation: exploring new frontiers*. San Diego, California (pp. 653-663).

Conwell, C.L., Enright, R., & Stutzman, M.A. (2000). Capability maturity models support of modeling and simulation verification, validation, and accreditation. *Proceedings of the 32nd conference on Winter simulation*. Orlando, Florida (pp. 819-828).

Owen, S (2009) "CUBIT 10.2 Documentation", Sandia Technical Report SAND2006-7156P.

Page, E.H., Canova, B.S., & Tufarolo, J.A. (1997). A case study of verification, validation, and accreditation for advanced distributed simulation. *ACM Trans. Model. Comput. Simul., 7*(3), 393-424. doi: 10.1145/259207.259375

Sargent, R.G. (2013). An introduction to verification and validation of simulation models. *Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World*. Washington, D.C. (pp. 321-327).

Vegas, S., & V., B. (2005). A characterization schema for software testing techniques. *Empirical Software Engineering, 10*(4), 437-466.

Wang, Z. (2013). Selecting verification and validation techniques for simulation projects: A planning and tailoring strategy. *Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World*. Washington, D.C. (pp. 1233-1244).