Theme Article

Middleware Building Blocks for Workflow Systems

Matteo Turilli
Rutgers, The State University of New Jersey

Vivek Balasubramanian
Rutgers, The State University of New Jersey

Andre Merzky
Rutgers, The State University of New Jersey

Ioannis Paraskevakos
Rutgers, The State University of New Jersey

Shantenu Jha
Rutgers, The State University of New Jersey
Brookhaven National Laboratory

***Abstract*—This paper describes a building-block approach to the design of scientific workflow systems. We discuss RADICALCybertools as one implementation of the building blocks concept, showing how they are designed and developed in accordance with this approach. This paper offers three main contributions: (i) showing the relevance of the design principles underlying building-block approach to support scientific workflows on high performance computing platforms; (ii) illustrating a set of building blocks that enable multiple points of integration, "unifying" conceptual reasoning across otherwise very different tools and systems; and (iii) case studies discussing how RADICAL-Cybertools are integrated with existing workflow, workload, and general purpose computing systems, and used to develop domain-specific workflow systems.**

Sophisticated and scalable workflows have come to epitomize advances in computational science. To the credit of workflow systems initially developed for "big science" projects, such as those in high-energy physics, many advances were made when the scientific distributed computing infrastructure and software ecosystem was missing important features and was relatively fragile when compared to today's ecosystem. Many successful workflow systems evolved to support the end-to-end execution of workflows.

The landscape of scientific application requirements and software infrastructure has changed. Although high-throughput execution of tasks—the original driver of "big science" workflows—is still important, it is joined by other functional and automation requirements. New application scenarios involve the time-sensitive integration of experimental data from large-scale instruments and observation systems with high-performance computing (HPC). Workflows are also becoming more pervasive

Digital Object Identifier 10.1109/MCSE.2019.2920048
Date of publication 30 May 2019; date of current version 24 June 2019.

Published by the IEEE Computer Society

across application types, scales, and communities. Scientific insight typically requires computational campaigns with multiple distinct workflows, heterogeneous tasks, and distinct runs. For example, an application may involve distinct phases of parameter exploration and optimization, sensitivity analysis, and uncertainty quantification.

Previously missing software infrastructural capabilities that necessitated the development of end-to-end workflow systems are now relatively more reliable, better supported, and more consistently available. The emergence of diverse Python-based task distribution and coordination systems, Apache data analysis tools, and container technologies provide useful examples.

This paper advocates a building-block approach to the design and development of scientific workflow systems.

An important and increasingly prevalent trend is that application developers tend to develop their own workflow solutions, tailored to the requirements of their applications. The list curated by Crusoe *et al.*[1] enumerates in excess of 230 purported workflow systems: some partial, others closer to being end-to-end; some specific to a workload or functionality, others general-purpose; some standalone, others designed to be integrated with other systems.

The proliferation of workflow systems raises many questions for users and developers: 1) how to support the agile development and composition of workflow systems that share capabilities, while not constraining functionality, performance, or sustainability; 2) how to lower the barrier for leveraging existing software infrastructure; 3) how to provide a sustainable ecosystem of both existing and new software components from which tailored workflow systems can be composed. These are set against trends of increasing functional requirements and sophistication of workflows.

This paper advocates a building-block approach to the design and development of scientific workflow systems. We postulate building-blocks leverage emerging trends in software and distributed computing infrastructure, and the approach supports a sustainable ecosystem of both existing and new software components from which tailored workflow systems can be composed. Building blocks enable expert contributions while lowering the breadth of expertise required of workflow system developers. They render obsolete a focus on developing a workflow system that purports to be "better" than other workflow systems, and emphasizes if not incentivizes the community toward development of collective capabilities.

After a brief description of the building-block approach and its four design principles of self-sufficiency, interoperability, composability, and extensibility, the "RADICAL-CYBERTOOLS" section discusses how we used the building-block approach to develop RADICAL-Cybertools to enable the execution of workflows from diverse scientific domains on HPC platforms. RADICAL-Cybertools are a set of software systems that can be used independently and integrated into middleware, among themselves, and with third-party systems. We introduce a four-layered view of high performance and distributed systems and describe how each system implements distinctive functionalities for each layer.

The "BUILDING BLOCKS, RADICAL-Cybertools AND WORKFLOWS SYSTEMS" section discusses how RADICAL-Cybertools complement and contribute to existing workflow systems and middleware. We present three case studies integrating RADICAL-Cybertools with end-to-end workflow systems (Swift), workload management systems (PanDA), i.e., WMS, and general-purpose computing frameworks (Spark and Hadoop), and a case study discussing the development of domain-specific workflow (DSW) systems (ExTASY, RepEx, HTBAC, and ICEBERG). These case studies have enabled diverse scientific applications, involving high-throughput computing (HTC), multiprotocol simulations, adaptive execution, data-intensive simulations, and image processing.

We conclude with a discussion of the practical impact of the case studies as well as the lessons learned by testing the validity and feasibility of the building-block approach. We highlight the benefits of implementing new capabilities into existing workflow systems by

integrating the RADICAL-Cybertools. We also outline the limitations of our contributions as well as some open questions.

RELATED WORK

We classify existing workflow systems into three categories, focusing only on those with the highest adoption and ongoing development. All-inclusive workflow systems, such as Kepler, Swift, Fireworks, and Pegasus, provide full-featured, end-to-end capabilities that include application creation, execution, monitoring, and provenance. General-purpose workflow systems, such as Ruffes, COSMOS, and GXPMake, enable end-to-end execution but prioritize the simplicity of their interfaces, limiting the range of capabilities. Finally, DSW systems, such as Galaxy, BioPipe, and Copernicus, provide interfaces tailored to the requirements of specific domain scientists.

The decomposition of workflow systems into systems with high cohesion and low dependency supports decoupling of independent software development efforts and promotes the use of standardized interfaces. These systems are implemented in monolithic or modular fashion to support specific capabilities, and have been used to develop multiple workflow systems by integration. For example, Spark, Hadoop, and MapReduce can be integrated—with or without pipelining tools such as Luigi, Toil, Airflow, Azkaban, or Oozie—to create special-purpose workflows systems.[2,3] Nonetheless, these tools are specifically tailored to data-oriented workflows, face several performance bottlenecks when ported to HPC machines, and require dedicated deployment.[4] Research in interoperability of HPC systems with data-parallel frameworks is ongoing and provides and extends middleware to efficiently support data-oriented workflows on HPC. A few examples are Pilot-Hadoop and Pilot-Spark, Twister, or Pilot-Streaming.

Modularity, in software deployment, has evolved from chroot, jails, and Solaris zones and, more in general, to what is called the "UNIX philosophy" into modern day service-oriented architecture (SOA) and its microservice variants.[5] These approaches evolve from the concepts of component-based software engineering[6] (CBSE) where computational and compositional elements are explicitly separated.[7–9]

We build upon CBSE and SOA concepts, investigating modularity at the level of standalone software systems and not at the level of modules or routines of a single system. In this context, we underline the benefits of CBSE-like concepts when applied to workflow systems for scientific computing executed on HPC resources. AirFlow, Oozie, Azkaban, Spark Streaming, Storm, or Kafka are examples of tools that have a design consistent with the proposed approach. Different from the CBSE and SOA approaches, we make the internal states and events of each module accessible and employ connectors and translation layers between interfaces.

BUILDING-BLOCK APPROACH

Each building block has a set of entities, a set of functionalities that operate on these entities, and a set of states, events, and errors for each entity. Architecturally, the building-block design requires: first, a well defined and stable interface for input and output, which enables clean separation between computational and compositional features; second, one or more conversion layers capable of translating across diverse representations of the same type of entity; third, one or more modules that implement the functionalities to operate on these entities and expose higher level abstractions for their composition.

In our adaptation, the building-block approach is based on four design principles: self-sufficiency, interoperability, composability, and extensibility. Self-sufficiency and interoperability depend upon the choice of both entities and functionalities. Entities have to be general enough so that specific instances of that type of entity can be reduced to a unique abstract representation. Accordingly, the scope of the functionalities of each building block has to be limited exclusively to its entities. In this way, interfaces can be designed to receive and send diverse codifications of the same type of entity, while functionalities can be codified to consistently translate those representations and operate on them.

Composability depends on whether the interfaces of each building block enables communication and coordination. Blocks communicate information about the states, events, and errors of their entities, enabling the coordination of their functionalities. Due to the requirement of self-sufficiency, the coordination among blocks cannot be assumed to happen implicitly but has to be codified on the base of an explicit model of the entities' states. The sets of functionalities of a block need to be extensible to enable the coordination among states of multiple and diverse blocks. Note that extensibility remains bound by both interoperability and self-sufficiency.

Each design principle of the building-block approach poses unique challenges when applied to software systems that can be used both standalone and integrated with third-party systems. Choosing entities and scoping functionalities to enable self-sufficiency requires expanding the design phase and, therefore, longer development iterations. Furthermore, interoperability requires system-level interfaces to become a first-order concern and to be based on well-defined, general-purpose abstractions. The coordination protocols that enable composability require generalization of variable access, dataflow, and procedure calls. Extensibility also requires shared coding convention and documentation.

The building-block approach does not reinvent modularity; it applies it at the system level to enable composability among independent software systems. As an abstraction, modularity enables separation of concerns by encapsulating discrete functions into semantic units exposed via a dedicated interface. As such, modularity can be used both at function and system level. Modularity at function level depends on the programming paradigm and the facilities offered by programming languages. Modularity at system level depends on the interface exposed by each system.

Traditionally, components of software systems independently designed by third party organizations have been difficult to integrate outside the well-defined scope of an operating system like, for example, Unix. While interfaces can hide implementation details, working as implementation-independent specifications of capabilities, integration still requires semantic uniformity across interfaces. Obtaining such uniformity is challenging and largely unsupported by specific constructs both at specification and programming level. Furthermore, integrating independent systems poses challenges in language heterogeneity, error handling, input/output validation, effective documentation, and comprehensive testing.

The building-block approach contributes to address integration challenges across independent systems by specifying state, event, and error models for each block. Following best practices in application program interface design, entities are explicitly specified and implemented in the block's interface and used as input for each exposed functionality. Each entity has a set of associated states, events, and errors. The order of the state is guaranteed by the implementation (e.g., a task cannot be executed before being scheduled and scheduled before being bound to a resource) while events are unordered but always contained between two defined states. Errors are always associated to an entity, state, and event. Communication is decoupled from coordination and independent from the implementation of communication channels.

Even when applied at system level, the building blocks approach presents at least two major trade offs. Building systems as blocks increase design and implementation effort, making unfeasible an unstructured but rapid development approach. While unstructured approaches are counterproductive for long-term maintenance, short-term solutions would pay an unpractical overhead to the building-block approach. Furthermore, integration of systems that are independently developed imposes sharing responsibility of software reliability across multiple stakeholders. Often, this can be undesirable as users attribute all the responsibility to the stakeholder of their immediate interface. This problem can be mitigated by system-level fault tolerance but it remains an element to carefully evaluate when considering the building-block approach.

RADICAL-CYBERTOOLS

RADICAL-Cybertools are software systems designed and implemented in accordance with

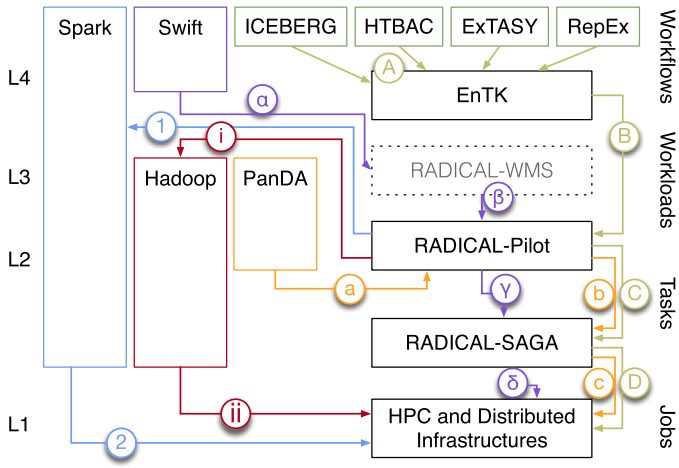

Figure 1. Composition of RADICAL-Cybertools (black) with DSW systems (green, A–D), workflow system (purple, α–δ), WMS (orange, a-c), framework for distributed data processing (red, i-ii), and a unified analytics engine (blue, 1–2). Numbered layers on the left; names of entities on the right. Solid colored lines indicate various integrations points with RADICAL-Cybertools; dashed boxes indicate tools still under development.

the building-block approach. Each system is independently designed with well-defined entities, functionalities, states, events, and errors. Figure 1 shows three existing RADICAL-Cybertools systems: RADICAL Ensemble Toolkit (hereafter simply referred to as EnTK), RADICAL-Pilot (RP), and RADICAL-SAGA. RADICAL-WMS is a WMS still under development.

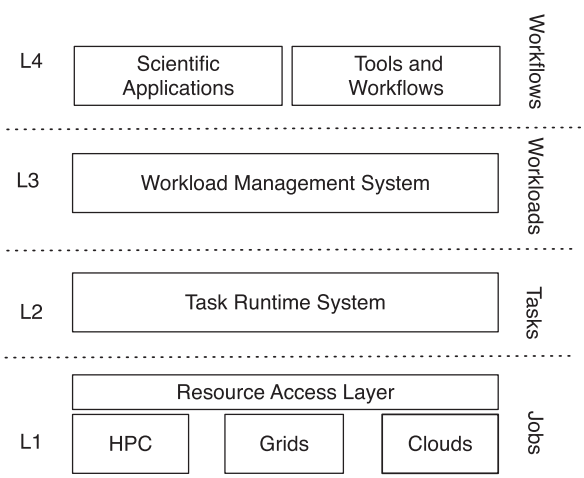

Figure 2. Primary functional levels. The diagram supports an analysis of the functional requirements for workflow systems, and the primary entities at each level, agnostic of the applications and resources.

Individual RADICAL-Cybertools are designed to be consistent with a four-layered view of distributed systems for the execution of scientific workloads and workflows on HPC resources (see Figure 2). Each layer has a well-defined functionality and an associated "entity." The entities are *workflows* (or applications) at the top layer and resource specific *jobs* at the bottom layer, with *workloads* and *tasks* as intervening transitional entities in the middle layers.

Workflow and Application Description Level (L4): Requirements and semantics of an application described in terms of a workflow.

Workload Management Level (L3): Applications devoid of semantic context are expressed as workloads, which are a set of tasks that can be executed concurrently. The workload management layer is responsible for: first, the selection and configuration of available resources for the given workload; second, partitioning the workload over the selection of suitable resources; third, binding of tasks to resources.

Task Runtime Level (L2): L3 delivers tasks to L2, which is responsible for their execution on the selected resources. L2 is a passive recipient of tasks from L3 but includes functionalities to acquire the indicated HPC resources, schedule the given tasks over available resources, and execute these tasks with the indicated data and number of cores.

Resource Layer (L1): The resources used to execute tasks are characterized by their capabilities, availability and interfaces. Different resources present inconsistency in the way capabilities are provisioned but advances in syntactically uniform resource access layers enable task execution across resources.

Currently, in RADICAL-Cybertools each task defines an executable, e.g., python, GROMACS, AMBER, SPECFEM, or any other executable programs. Each task description contains the arguments to pass to the executable, the type of parallelism required (e.g., message passing interface (MPI) and open multi-processing (OpenMP)), the type and number of processing units (CPU or GPU), the amount of memory, and the data staging requirements. RADICAL-Cybertools implement full task isolation, enabling the concurrent or sequential execution of heterogeneous, dependent, or independent executables

on a given set of acquired resources. RADICAL-Cybertools are agnostic of the operations performed by each executable: for each task, RADICAL-Cybertools satisfy its dependencies, set up its environment, spawn its execution, and wait for the executable to return in a final state. Therefore, RADICAL-Cybertools do not have access to the operations performed by each executable.

RADICAL-Cybertools conform to the principles of self-sufficiency, interoperability, composability, and extensibility. EnTK exposes an application programming interface (API) for the description of scientific applications as static or dynamic sets or sequences of pipelines. Pipelines are sequences of stages that, in turn, are sets of tasks. Sequences and sets formally define the relationship of priority among task executions: the tasks of a stage execute concurrently, tasks of different stages of the same pipeline execute sequentially, and pipelines execute concurrently. Resources are acquired and managed via a third-party runtime system that executes tasks on the acquired resources.

EnTK is self-sufficient as it enables necessary and sufficient functionalities for its set of entities, independently from third-party software systems. EnTK is interoperable because different representations of a workflow [e.g., directed acyclic graph (DAG)] can be converted to pipelines of stages of tasks, and because it is agnostic toward runtime systems and the type of resources on which they execute tasks. EnTK is also composable because it enables arbitrary coordination protocols (e.g., push/pull or master/worker) by explicitly defining the state model of its entities. Finally, EnTK is also extensible as new capabilities can be implemented for its entities, e.g., adaptivity of both workflows structure and task specifications at runtime.

RP is a pilot system that exposes an API to enable the acquisition of resources on which to schedule tasks for execution. The design of RP includes pilot and compute unit as entities. Capabilities are made available to describe, schedule, manage, and execute entities. Pilots, units, and their functionalities abstract the specificities of diverse types of resource, enabling the use of pilots mainly on single and multiple HPC machines, but also on HTC and cloud infrastructures. A pilot can span single or multiple compute nodes, resource pools, or virtual machines. Units of various size and duration can be executed, supporting MPI and non-MPI executables, with a wide range of execution environment requirements.

The design of RP is: self-sufficient because, as with EnTK , it independently implements the necessary and sufficient set of functionalities for its entities; interoperable in terms of type of task, resource, and execution paradigm; extensible as new properties can be added to the pilot, unit, and resource descriptions, and more functionalities can be implemented for these entities. Currently, composability is partially designed and implemented: while the API can be used by both users and other systems to describe generic tasks for execution, RP requires RADICAL-SAGA to interface to HPC resources. A prototype interface to cloud resources based on LibCloud is available and a general-purpose resource connector component is underdevelopment.

RADICAL-SAGA exposes a homogeneous programming interface to the queuing systems of HPC resources. SAGA—an open grid forum standard—abstracts away the specificity of each queue system, offering a consistent representation of jobs and of the capabilities required to submit them to the resources. The design of RADICAL-SAGA is based on the job entity and its functionalities enable job submission and jobs' requirement handling (self-sufficiency). Both entities and functionalities can be extended to support, for example, new queue systems, or new type of jobs (extensibility). The SAGA API resolves the differences of each queue system into a general and sufficient representation (interoperability), exposing a stable set of capabilities to both users and/or other software elements (composability).

Currently, data staging capabilities are implemented in each RADICAL-Cybertools via third-party tools, such as SFTP, SCP, and Globus Online. Nonetheless, we do not have a dedicated Cybertool for managing data storage, provenance, and archiving. Users can enable these capabilities by integrating third-party systems into EnTK workflows and RP workloads, creating their own data management steps. Integration of third-party systems is facilitated by implementing task and compute unit in RADICAL-Cybertools as wrappers for self-contained programs.

Data management tools can be executed or accessed independent from the coordination, communication, and runtime environment requirements of RADICAL-Cybertools.

BUILDING BLOCKS, RADICAL-CYBERTOOLS, AND WORKFLOW SYSTEMS

RADICAL-Cybertools as a whole are not an end-to-end workflow system. Each cybertool is an independent system that can also be integrated with other systems (RADICAL-Cybertools or otherwise) to form tailored middleware solutions. For example, several independent communities directly utilize RADICAL-SAGA alone, with RP or other pilot systems like, for example, PanDA Pilot. Other communities integrate all RADICAL-Cybertools with or without third-party systems to support the execution of diverse types of scientific workflows. Thus, RADICAL-Cybertools are not posed to replace existing workflow systems: RADICAL-Cybertools' novelty is to enable the integration across systems independently developed and not necessarily designed to integrate. Crucially, this includes existing workflow, workload, and computing frameworks, alongside their components.

> When integrated, RADICAL-Cybertools simplify the codification of workflows, lowering the barrier to adoption, maintenance, and reuse.

We believe an ecosystem in which end-to-end workflow systems and building blocks coexist and, when useful, are integrated helps to avoid both lock-in and fragmentation. Such an ecosystem would allow scientists with specific and stable requirements to use an end-to-end system while others to aggregate existing capabilities into tailored solutions. Inversely, nonintegrable systems built with slightly different capabilities fragments the user experience and forcing scientists to learn to use multiple systems, depending on the context in which they have to operate.

As building blocks, RADICAL-Cybertools offer several benefits when used to describe and execute scientific workflows. Among these benefits, the most relevant is isolating scientists from job management (L1), task management (L2), and workload management (L3). These capabilities are further abstracted away in L4, letting scientists to exclusively focus on workflow description and application logic. Note that while this isolation is offered by other systems, RADICAL-Cybertools is agnostic toward which software tools and systems are integrated at each layer L1–4.

When integrated, RADICAL-Cybertools simplify the codification of workflows, lowering the barrier to adoption, maintenance, and reuse. When using EnTK, workflows are codified as pipelines in a general-purpose language (Python) and application-specific constructs (Task, Stage, Pipeline, and AppManager). As programs, workflows can be maintained following diverse approaches: from keeping a simple script on a scientist's workstation to sharing a more complex application among multiple scientists via a collaborative version control system. Codifying workflows as code but without a dedicated domain language offers the opportunity of reusing a portion of code in the form of methods, classes, and modules. Furthermore, scientists have the option to grow the code as needed, typically starting from a small script and growing it into an application as the research advances, alone, or with the help of other scientists and software engineers.

Interoperable, extensible, and logically self-contained software blocks alongside lower technical barriers to their composability allow designing workflows as domain-specific applications. These type of applications solves classes of scientific problems, not issues of resource and execution management. Domain-specific applications, alongside the blocks they use, become sustainable because they can be understood and maintained by diverse, invested communities. This is the sustainability model of successful open source software, including some of the existing solutions for certain types of workflows and resources, e.g., the Apache Hadoop ecosystem.

Supporting the development and maintenance of domain-specific applications is becoming increasingly important to enable scientific workflows. Alongside large communities in which the same workflow is used for many years

(e.g., the LHC community), many research fields increasingly require running rapidly-evolving workflows with relatively short computation campaigns. These workflows depend on simulations and analysis procedures that evolve during the campaign, integrating new models and methodologies. As such they require a software ecosystem with independent systems that can be easily integrated and extended, depending on evolving scientific requirements.

Integrating End-To-End Workflow Systems

End-to-end workflow systems enable a wide range of capabilities on several types of computing resources. Often, their adoption and deployment require investing sizable amount of resources, developing system-specific code, and establishing dedicated processes. Extending this type of workflow systems requires advanced development knowledge both at system and resource level, and taking into account the requirements of a widely used and production-grade code base. Integrating building blocks with these end-to-end system may lower the amount of resources needed to implement new functionalities while requiring moderate development skills.

As an example of how the building-block approach can be utilized in other systems, we map the primary functional levels described in Figure 2 to Pegasus,[10] one of the most adopted end-to-end workflow system. Scientific applications are described as abstract workflows using the HubZero API, or workflow composition tools such as Wings and Airvata. These interfaces correspond to the application layer (L4) shown in Figure 2.

The abstract workflow is transformed to a concrete workflow by the Mapper component. The transformation takes into account the availability of software, data, and computational resources required for execution, and can restructure the workflow to optimize performance. A concrete workflow with several interdependent jobs, each consisting of several interdependent tasks, is passed to a workflow engine. Pegasus utilizes different engines, depending on the target resource: first, lightweight execution engine for local resources; second, HTCondor DAGMan and HTCondor Schedd for clusters and HPC platforms; third, HTCondor with Glide-in WMS for grids. Functionally, Mapper, workflow engine, and local scheduler correspond together to the workload management layer (L3) given in Figure 2.

Pegasus supports three modes of job execution, depending on the execution environment and architecture of the remote machine: first, PegasusCluster, a single-threaded engine that submits one task at a time; second, PegasusLite, for handling tasks input and output data on resources with no shared filesystem; third, Pegasus MPICluster, for systems with a shared filesystem where MPI is used to implement a master–slave layout for task binding and execution. Collectively, these three remote execution engines correspond to the task runtime layer (L2) shown in Figure 2.

Pegasus uses GlobusGRAM and CREAM-CE to submit jobs directly to remote batch-queuing systems and resource managers, such as Simple Linux Utility for Resource Management, Portable Batch System, Platform Load Sharing Facility, and Sun Grid Engine. These tools correspond to the resource access layer (L1) shown in Figure 2.

Following this mapping, end-to-end workflows systems, or some of their component can be integrated with RADICAL-Cybertools as building blocks to enable new capabilities. Together with the Swift development team, we used this approach to integrate Swift[11] with RP and RADICAL-SAGA. Swift has a long development history, with several versions that supported diverse case studies. Swift also integrated pilot systems of which Coasters is actively supported. The design of Swift is modular and it relies on connectors to interface with third-party systems.

In Swift, the language interpreter and the workflow engine are tightly coupled but connectors can be developed to stream the tasks of workflows to other systems for their execution. As seen in Section "RADICAL-CYBERTOOLS," RP can get streams of tasks as an input and submit these tasks to pilots for execution.

We integrated Swift with RP to enable the distributed and concurrent execution of Swift workflows on multiple HPC platforms and HTC infrastructures (see Figure 1, purple α–δ). The distributed scheduling capabilities of RP offered the

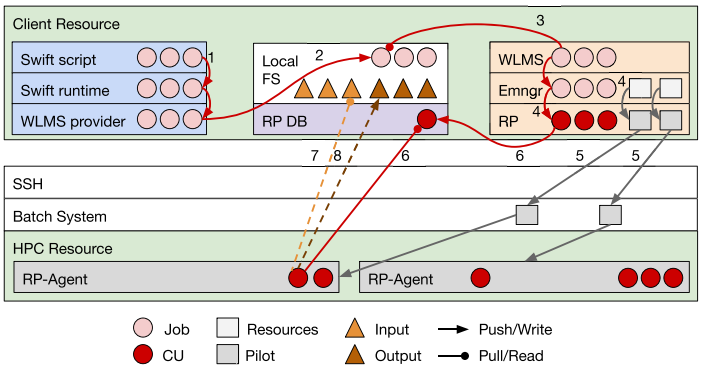

Figure 3. Integration between Swift and RP. The two systems exchange task descriptions via a local filesystem. RADICAL-WMS derives the size and duration of the pilots from the task requirements, independently from Swift.

possibility to minimize the time for completion of tasks execution, obtaining both qualitative and quantitative improvements.[12] Qualitatively, RADICAL-Cybertools -enabled Swift to execute workflows concurrently on both HPC and HTC resources via late binding of both tasks to pilots and pilots to resources. Quantitatively, the time for completion of workflows was improved by leveraging the shortest queue time among all the target resources.

The integration with RP required the Swift team to develop a dedicated connector by iterating on the already available shell connector (see Figure 3). We used the opportunity to prototype a distributed workload management (RADICAL-WMS) as means of research. The connector enabled saving task descriptions on the local filesystem from where RADICAL-WMS was able to load and parse these descriptions without needing any added functionality. This type of integration was made possible by sharing the task entity semantics between the two systems and by isolating distinct functionalities operating on that entity in two distinct software systems. Note how these two systems were not designed to be integrated and were developed by independent teams.

Integrating a WMS

Often, workflow managers are developed to support specific resources, workloads, projects, and communities. Extending their capabilities can be difficult, especially when the new capabilities do not serve the intended core use cases. Instead of developing yet another domain-specific workload manager, integration with existing building blocks can represent an economic and viable solution. This was true for PanDA, a WMS designed to support execution of independent tasks on grid computing infrastructures like WLCG[13] and, to a lesser extent, leadership-class HPC platforms.

Executing large number of small jobs on leadership HPC platforms presents two main challenges: first, using a queue system that privileges large MPI jobs; second, accessing untapped resources without disrupting the overall utilization of the machine. Pilots can address both challenges but pilot systems are difficult to deploy on HPCs. The main problem is efficiently managing the concurrent and sequential execution of small heterogeneous jobs at scale.

We developed an interface to RP called Next Generation Executer (NGE). NGE enables WMSs designed for HTC to execute workloads on pilots instantiated on leadership-class HPC platforms (see Figure 1, orange a–c). As part of their WMS, the PanDA team developed Harvester, a job broker to support the execution of part of the ATLAS Monte Carlo workflow on Titan. Harvester misses pilot capabilities and the PanDA team developed an Harvester connector to NGE instead of implementing a new pilot system. In this way, event simulations of the ATLAS project can be executed both concurrently and sequentially on the resources acquired by submitting a single job to Titan's queue.

Harvester uses NGE to exchange information about tasks descriptions and resources requirements while RP behaves like a resource queue for Harvester (see Figure 4). Both systems require no modifications to be integrated but the development of a coordination protocol to pull/push information about entities and their states. As with Swift, PanDA Broker, and RP are independently developed and their integration was performed when the two stacks were already in production.

Integrating General-Purpose Computing Frameworks

Computing frameworks such as Hadoop and Spark offer specific capabilities, programming models, and a large ecosystem of related software modules. As with end-to-end workflow

systems, users tend to produce code and processes that depend on these frameworks that need to be maintained while scaling and runtime requirements evolve. Deploying and using these frameworks on HPC resources is challenging, especially when considering machines that are not built and configured as Hadoop or Spark clusters. Integration with the RADICAL-Cybertools building blocks addresses these challenges and offer the same multitask programming interface across both frameworks.

Hadoop exposes an API for users and other software systems (composability) to describe distributed applications, mostly in terms of the MapReduce programming model, supporting distributed filesystem capabilities. Accordingly, Hadoop implements the necessary and sufficient functionalities of a WMS (self-sufficiency). Hadoop can aggregate and manage diverse storage resources via a master/worker subsystem composed of multiple Namenode and DataNode instances (interoperability), and supports diverse runtime systems such as Mesos and YARN, to schedule and execute tasks on computing resources (extensibility).

Spark can be considered as a self-sufficient implementation of a workflow system. It enables necessary and sufficient workflow functionalities, for machine learning, iterative analytics, and streaming, independent of the underlying task runtime system. Spark enables interoperability by supporting different execution engines, such as Hadoop and MPI. Spark exposes an API that can be used to develop distributed applications (composability). Spark can be extended to support different types of workflows.

We integrated Hadoop, Spark, and RP into a framework dedicated to HPC machines[14] (Figure 1, red i–ii and blue 1–2). The integration addresses one of the major problems of using Hadoop and Spark on HPC resources, avoiding the need for dedicated deployment and customizations while retaining the full functionalities of both systems. RP configures, starts, and manages a Hadoop/Spark cluster, and then executes a user's Hadoop/Spark application on that cluster. Once done, RP shuts down the cluster and cleans up the environment.

We used RPs integration with Spark to parallelize MDAnalysis and characterize its performance. MDAnalysis is a Python library that provides a comprehensive environment for filtering, transforming, and analyzing molecular dynamics simulation trajectories.[15] Currently, we use RPs integration with Spark to support diverse imagery analysis algorithms for geological and polar sciences.

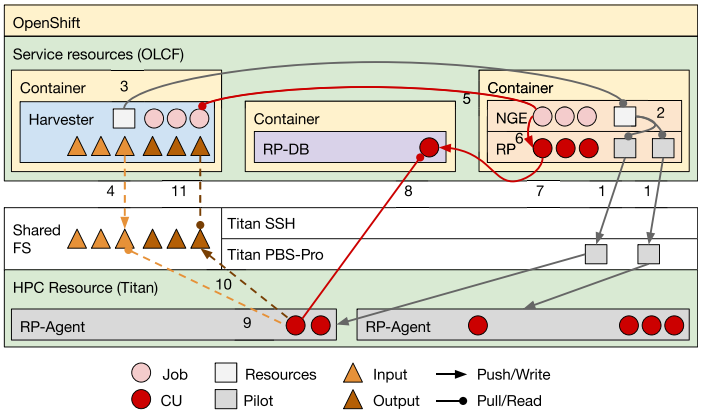

Figure 4. Integration between PanDA and RP via the NGE REST interface. All systems execute on OLCF service resources within containers. Pilots are exposed to PanDA as an aggregation of available resources (steps 2 and 3).

DSW Systems

We call a workflow system that provides a specific higher level functionality a DSW system. We aggregated RADICAL-Cybertools, developing four DSW: ExTASY, RepEx, HTBAC, and ICEBERG (see Figure 1, green A–D). Driven by specific application needs, each DSW is characterized by a unique execution and coordination pattern and can serve multiple applications.

Our DSW systems use EnTK to support ensemble-based workflows. EnTK is agnostic to the details of the specific executables run by the ensemble members, and the system used to manage their execution. Figure 5 shows how EnTK couples with RP to execute the ensembles via pilots on HPC resources. Note that, in principle, EnTK could use a different runtime system and type of computing infrastructure.

ExTASY provides the simulation-analysis execution pattern to support several biomolecular sampling methods such as DM-d-MD and CoCo.[16] RepEx enables multiple replica-exchange methods that vary in the coordination patterns across replicas, e.g., global synchronization

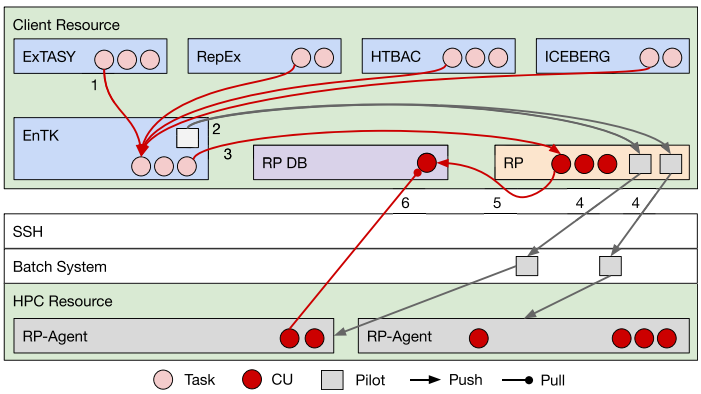

Figure 5. Integration between four DSW systems—ExTASY, RepEx, HTBAC, ICEBERG—and EnTK. Numbers indicate the execution flow. RP database can be deployed on any host reachable from the resources.

barrier, or pairwise synchronization. RepEx supports synchronous and asynchronous Pmultidimensional exchange schemes,[17] separating performance and functional layers while providing simple methods to extend interfaces. HTBAC implements multiple pipelines of heterogeneous tasks for binding free energy calculations, wherein both pipelines and tasks within a pipeline can change at runtime. All three biomolecular DSW run on several HPC platforms, including Oak Ridge National Laboratory and National Center for Supercomputing Applications leadership-class machines. ICEBERG supports scalable image analysis applications using multiple concurrent pipelines.

ExTASY, RepEx, HTBAC, and ICEBERG benefit from integrating RADICAL-Cybertools by not having to reimplement workflow processing, task management, and task execution capabilities on distinct and heterogeneous platforms. This, in turn, enables users to focus on customizing each DSW based on the requirements of specific scientific domains. DSW and RADICAL-Cybertools free scientists from developing capabilities outside their domain of expertise and from dealing with resource- and middleware-specific deployment issues.

DISCUSSION AND CONCLUSION

Traditionally, assumptions about types of applications or resources have led to software systems that, while modular, have not allowed reuse outside their original requirements. We believe this is why functionalities pertaining to entities like tasks or pilots are often reimplemented. Each system serves well the single research group or the large scientific project but not each other.

As argued in the "BUILDING-BLOCK APPROACH" section, building blocks—i.e., self-sufficient, interoperable, composable, and extensible software systems—can serve arbitrary requirements for a well-defined set of entities. For example, a workflow manager can provide methods for DAG traversing, independent of how and when that DAG is specified or where the tasks of the workflow will be executed. Analogously, a pilot manager can provide multistaging and task execution capabilities, independent of the task scheduler or the compute resources on which tasks will be executed.

Modularity is not a design principle strong enough to realize this type of software systems. Modularity needs to be augmented by API and coordination agnosticism alongside an explicit understanding of the entities that define the domain of utilization of the software system. Each system developed following this approach, implements a well-defined set of functionalities specific to a set of entities, with minimal assumptions about the system that will use these functionalities or the environment in which they will be used. Without these elements, systems developed by independent teams and not specifically designed to work together, may require major re-engineering to coordinate and aggregate their functionalities.

Systems like Celery, Dask, Kafka, or Docker are early examples of software designed by implicitly following the proposed building-block approach. These tools implement specific capabilities like queuing, scheduling, streaming, or virtualization for the domain of distributed computing. Consistently, they assume a set of core entities like workloads, tasks, pipelines, or messages, each with well-defined properties like concurrency and states. Their integration in multiple domains shows the potential of their underlying design approach.

This paper offers three main contributions: first, showing the relevance of the building-block approach for supporting the workflows of various scientific domains on HPC platforms; second,

illustrating building-block that enable multiple points of integration, resulting in design flexibility and functional extensibility, and providing a level of "unification" in the conceptual reasoning (e.g., execution paradigm) across otherwise different tools and systems; third, showing how these building blocks have been used to develop and integrate workflow systems for HPC platforms.

The "BUILDING BLOCKS, RADICAL-CYBER-TOOLS, AND WORKFLOW SYSTEMS" section highlights the practical impact of the building-block approach. All the integrations required minimal development, mainly focused on translation layers and glue interfaces. Importantly, no refactoring was required within the systems we integrated. Explicit and agreed upon engineering processes was necessary to enable the integration among systems developed by independent teams and institutions. GitHub proved to be fundamental to enable these processes and to manage the engineering process. Explicit agreement on written use case and software requirements specifications greatly increase development coordination and, ultimately, efficiency. Finally, weekly meeting among the lead developers helped the coding process and establishing a shared development culture.

The building-block approach spawns many new questions. A prominent one pertains to the issue of how we might model workflows systems and tools, so as to provide a common vocabulary, reasoning, and comparative framework. The paper by Turilli *et al.*[18] provided the architectural paradigm for pilot systems; however, it is still unclear how an analogous paradigm would complement the work done on reference architectures for workflow systems,[19,20] and whether given the very broad diversity of workflow systems and tools, we can even formulate a single architectural paradigm. This paradigm has been elusive so far, but it might be more fruitful to formulate system-level paradigms that have the properties of building blocks.

It is important to outline what this paper does not attempt to achieve. This paper presents a preliminary study focused on one approach to building blocks for workflows systems, without a quantitative analysis of its benefits. It does not provide qualitative insight or identify either the set of applications or systems where a building-block approach will surpass alternative approaches. Finally, this paper does not discuss best practices in the design, granularity, and provision of building blocks. These are all topics of ongoing investigation. Although preliminary, this work is not premature, the building-block approach is still a work in progress but we believe early reports of success are necessary.

An end-goal and intended outcome of this paper is to begin a discussion on how the scientific workflows community—end-users, workflow designers, workflow systems developers, and HPC facilities providers—can better coordinate, cooperate, and reduce redundant and unsustainable efforts. We believe the building-block approach enables an examination and investigation of design principles and architectural patterns for workflow systems that may facilitate this discussion.

ACKNOWLEDGMENTS

The authors would like to thanks their collaborating teams led by Peter Coveney (HTBAC); Kaushik De, Jack Wells, and Alexei Klimentov (ATLAS Project/PanDA); Charlie Laughton and Cecilia Clementi (ExTASY); Heather Lynch (ICEBERG). The authors also would like to thank Daniel Smith, Levi Naden, and Sam Ellis (Molecular Sciences Software Institute) for useful discussions and insight. This work was supported in part by National Science Foundation ACI 1440677 and United States Department of Energy Advanced Scientific Computing Research DE-SC0016280. The authors acknowledge access to computational facilities XSEDE resources (TG-MCB090174) and Blue Waters (NSF-1713749).

REFERENCES

1. M. R. Crusoe *et al.*, "Computational data analysis workflow systems." 2019. [Online]. Available: https://s.apache.org/existing-workflow-systems
2. J. Vivian *et al.*, "Toil enables reproducible, open source, big biomedical data analyses," *Nature Biotechnol.*, vol. 35, no. 4, pp. 314–316, 2017.
3. Z. Zhang *et al.*, "Scientific computing meets big data technology: An astronomy use case," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 918–927.

4. N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling spark on HPC systems," in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, ACM, 2016, pp. 97–110.
5. N. Dragoni *et al.*, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Berlin, Germany: Springer-Verlag, 2017, pp. 195–216.
6. G. T. Heineman and W. T. Councill, "Component-based software engineering," in *Putting the Pieces Together*. Reading, MA, USA: Addison-Westley, 2001, p. 5.
7. D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 4, pp. 355–398, 1992.
8. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch or why it's hard to build systems out of existing parts," in *Proc. 17th Int. Conf. Softw. Eng.*, 1995, pp. 179–185.
9. J.-G. Schneider and O. Nierstrasz, "Components, scripts and glue," in *Software Architectures*. Berlin, Germany: Springer-Verlag, 2000, pp. 13–25.
10. E. Deelman *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.
11. M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Comput.*, vol. 37, no. 9, pp. 633–652, 2011.
12. M. Turilli *et al.*, "Evaluating distributed execution of workloads," in *Proc. IEEE 13th Int. Conf. e-Sci.*, IEEE, 2017, pp. 276–285.
13. T. Maeno, "Panda: Distributed production and distributed analysis system for atlas," *J. Phys., Conf. Ser.*, vol. 119, 2008, Art. no. 062036.
14. A. Luckow, I. Paraskevakos, G. Chantzialexiou, and S. Jha, "Hadoop on HPC: Integrating Hadoop and pilot-based dynamic resource management," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 1607–1616.
15. N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein, "MDAnalysis: A toolkit for the analysis of molecular dynamics simulations," *J. Comput. Chem.*, vol. 32, no. 10, pp. 2319–2327, 2011, doi: 10.1002/jcc.21787.
16. V. Balasubramanian *et al.*, "Extasy: Scalable and flexible coupling of md simulations and advanced sampling techniques," in *Proc. IEEE 12th Int. Conf. e-Sci.*, 2016, pp. 361–370.
17. B. K. Radak *et al.*, "Characterization of the three-dimensional free energy manifold for the Uracil Ribonucleoside from asynchronous replica exchange simulations," *J. Chem. Theory Comput.*, vol. 11, no. 2, pp. 373–377, 2015.
18. M. Turilli, M. Santcroos, and S. Jha, "A comprehensive perspective on pilot-job systems," *ACM Comput. Surv.*, vol. 51, no. 2, 2018, Art. no. 43.
19. C. Lin *et al.*, "A reference architecture for scientific workflow management systems and the VIEW SOA solution," *IEEE Trans. Serv. Comput.*, vol. 2, no. 1, pp. 79–92, Jan.–Mar. 2009.
20. P. Grefen and R. R. de Vries, "A reference architecture for workflow management systems," *Data Knowl. Eng.*, vol. 27, no. 1, pp. 31–57, 1998.

Matteo Turilli is Assistant Research Professor at the Department of Electrical and Computer Engineering, Rutgers University, New Brunswick, NJ, USA. His research focuses on the design and performance analysis of middleware for high performance and distributed computing. Specifically, he focuses on how to enable high-throughput computing (HTC) on a federation of resources designed to support high-performance computing (HPC). He works with several HPC resources,including DoE and NSF leadership-class machines. He received the Ph.D. degree in computer science from the University of Oxford, Oxford, U.K. Contact him at matteo.turilli@rutgers.edu.

Vivek Balasubramanian is working toward the Ph.D. degree at Rutgers University, New Brunswick, NJ, USA. He received the master's degree in electrical and computer engineering and a bachelor's degree in electronics and communication engineering. His research interests include software abstractions, frameworks for scientific applications, high performance computing, distributed computing, and component-based software engineering. Contact him at vivek.bala@rutgers.edu.

Andre Merzky is the Senior Research Developer of the RADICAL-Group at Rutgers University, New Brunswick, NJ, USA. He studied Physics and Computer Science in Leipzig, Berlin and Edinburgh and has worked on topics in distributed computing such as data management, visualization, and high level APIs. He currently focuses on scalability challenges in the context of high performance computing. Contact him at andre@merzky.net.

Ioannis Paraskevakos is working toward the Ph.D. degree at Rutgers University, New Brunswick, NJ, USA. He received the Engineering Diploma (5-year studies) in computer engineering and informatics, and the M.Sc. degree in embedded systems. He has worked as a Hardware Design Engineer. His research interests include high performance computing, task parallel applications and architectures, distributed system communication protocols, and distributed data abstractions. Contact him at i.paraskev@rutgers.edu.

Shantenu Jha is on the Computer Engineering faculty at Rutgers University, New Brunswick, NJ, USA, and also leads the Data Driven Discovery department at Brookhaven National Laboratory, Upton, NY, USA. His research interests are at the intersection of high-performance distributed computing and computational & data science. He has graduate degrees in computer science and computational science from Syracuse University, Syracuse, NY, USA, and is a member of ACM SIGHPC. Contact him at shantenu.jha@rutgers.edu.