Proceedings of the Tenth IEEE International Conference on Semantic Computing, Irvine, CA, February 3-5, 2016.

Dynamically Generated Metadata and Replanning by Interleaving Workflow Generation and Execution

Yolanda Gil and Varun Ratnakar
Information Sciences Institute
University of Southern California
gil@isi.edu, varunr@isi.edu

***Abstract*—Workflow engines typically plan an entire workflow and then submit it for execution, and have limited replanning capabilities when the workflow execution fails. This paper presents an approach for interleaving planning and execution. The approach supports the incremental submission of partial workflows for execution until completion. As new metadata is generated dynamically during execution for all new data products, the workflow system can incorporate that dynamically generated metadata in the workflow planning process. The approach also supports replanning in case a resource is no longer available and in case of failure, not just by reassigning resources but also by redesigning the plan by replacing components that may fail to execute. The aproach is implemented and integrated with the WINGS workflow system, and is being used for a medical application.**

***Keywords*—**Scientific workflows, distributed workflow execution, scientific metadata generation, dynamic workflow replanning, interleaving workflow generation and execution.

I. INTRODUCTION

A variety of workflow systems have been developed to manage complex scientific computations [Taylor et al 2007]. An important area of research has been to allow users to specify workflow computations in a manner that is independent from the execution environment, where the workflow system automatically maps the codes to whatever execution resources are available at run time. This can be seen as managing a separation between the physical layer and the logical layer of the computation. A workflow specification at the logical layer has a specification of the codes to be executed, and there is no mention of the actual resources where the execution will take place. A workflow specification at the physical layer does mention execution resources that are to be used to run the computations. This separation between the logical and the physical layers is also common in web services frameworks and has been adopted in some workflow systems. A workflow specification at the logical layer is sometimes called an "abstract workflow". Workflow systems automatically map the workflow specification at the logical layer to a workflow specification at the physical layer. This is an important benefit, as it enables users to run their workflow in different execution environments, bringing flexibility to their applications.

However, workflow representations are still very tied to the execution environment because they specify the application codes that are to be run at each step. For example, depending on the workflow system a step may specify the MATLAB routine or Java code to run, or the signature of the service that needs to be invoked. In this respect, workflows are still tied to particular application codes and software environments. When published in workflow repositories [De Roure et al 2009], this limits their reuse by others who may use different software. It also limits their validity over time when code becomes obsolete and no longer runs. Ideally, workflow specifications would be independent of the particular code and software environment, specifying only the domain task to be carried out rather than what application codes to run. Previous work has focused on interoperability of workflow systems and workflow representations [Kozlovszky et al 2012], but not on creating more abstract representations that address the domain layer.

We have developed an approach to represent *semantic workflows* that express domain tasks rather than the application codes that implement those tasks. We have implemented this approach in the WINGS semantic workflow system [Gil et al 2011a; Gil et al 2011b; Gil 2014], and extended WINGS to demonstrate the mapping of semantic workflows into alternative workflow execution engines, including Pegasus/Condor and Apache OODT, and how to generate alternative workflow candidates when many alternative implementations of workflow steps are possible [Gil 2013a; Gil 2013b].

In this paper, we present an approach for interleaving workflow generation and execution that take advantage of the ability of WINGS to represent workflows of domain tasks. This approach enables a workflow system to take into account metadata that is dynamically generated by the worfklow and adjust the workflow accordingly. It also enables users to change the resource availability even during execution, and the workflow is adjusted accordingly. The approach supports the incremental submission of partial workflows for execution until completion. As new metadata is generated dynamically during execution for all new data products, the workflow system can incorporate that dynamically generated metadata in the workflow planning process. The approach also supports replanning in case a resource is no longer available and in case of failure, not just by reassigning resources but also by

We gratefully acknowledge support from the US Air Force Office of Scientific Research (AFOSR) with grant number FA9550-11-1-0104 and the US National Science Foundation under grant number ICER-1440323.

redesigning the plan by replacing components that may fail to execute. The approach is implemented in the WINGS workflow system, and is being used for a medical application. In this application the metadata of the intermediate results of the workflow is used to select data sources in some of the steps of the workflow, as described in [Zheng et al 2015].

II. Overview of the WINGS Workflow System

The WINGS workflow system is an end-to-end workflow system, which spans the timeline of a workflow from describing high-level workflow templates to creating concrete instantiations of these workflow templates, to executing these workflow instantiations in a diversity of execution environments, tracking the execution provenance, and finally supporting workflow reuse. WINGS aims to provide a user-friendly and standards-based tool to allow reusability, repeatability and modularity in conducting computational experiments via a workflow system. This section gives a brief overview of WINGS, more details can be found in [Gil et al 2011a; Gil et al 2011b; Gil et al 2009; Gil 2014].

The WINGS workflow system builds on semantic web technologies and provides RDF serializations of templates, component descriptions, data descriptions, executions, and provenance information. The RDF graphs are stored in a Jena TDB triple store, and an RDF endpoint is available for authenticated users to query the store.

WINGS includes a Data Catalog, a Component Catalog and a Workflow Catalog. WINGS data types and data are exposed via the Data Catalog, which provides an API to add new data types and data, or to edit existing data types and data. The data types are organized hierarchically, and each data type can be associated with some user defined metadata properties. Data can be uploaded under a particular data type, and its metadata property values can be filled in.

WINGS components are accessible via the Component Catalog, which provides an API to add new components and component types, or to edit existing components and component types. Component inputs and outputs (IO) can be defined to have certain data types already defined in the Data Catalog. A component type (or an abstract component class) provides the skeletal IO for that category of components, whereas a concrete component could expand on or specialize the skeletal IO of its component type. A concrete component would also contain a code implementation of that component type. This implementation could be binary code or a script that can be run on the host machines. Components and component types can also include semantic constraints that are implemented as rules. This distinction is illustrated in Figure 1. In order for WINGS to be able to run the code implementation of a component, it generally needs to wrap the code in a "run" script. This script is a shell/batch script, which follows a basic pattern as specified by WINGS; it parses the input file paths to the script as specified by the component description, sends them to the code appropriately, and makes sure that the outputs are moved to the output file paths sent to the run script by WINGS.

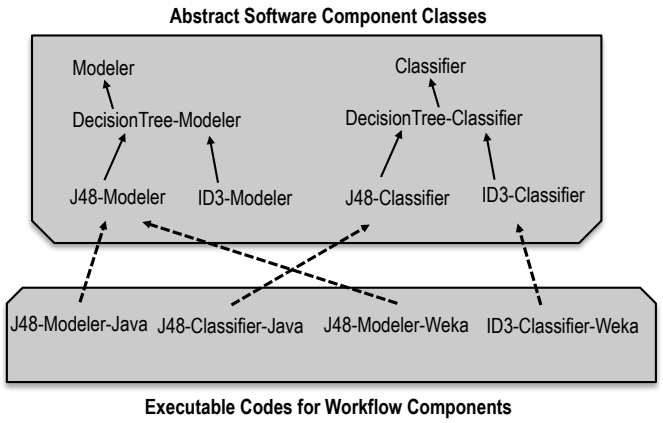

Fig 1. Abstract software component classes, shown at the top, can be used to specify a step in a workflow. The workflow generation algorithm selects executable codes, shown at the bottom, for those workflow components before submitting the workflow for execution.

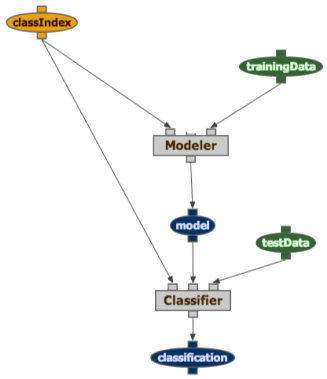

Fig 2. A workflow with abstract software component classes for its two steps.

Workflow templates in WINGS are stored in the Workflow Catalog. Workflows are represented as graphs of nodes and links which both have variables associated with them. Links can be input, output, or in-out links. Links contain data or parameter variables and may connect two nodes (in-out) or just end or originate at a single node (input or output link). Data variables could be bound to any data from the data catalog, whereas parameter variables could be bound to a basic data-type value (like string, integer, etc). Nodes in the workflow graph contain component variables, which can be bound to any component or component type from the Component Catalog.

Figure 2 shows an example of a workflow where the steps are component types that need to be specialized into executable codes during workflow generation.

III. Dynamically Generated Metadata

An important capability that can be accomplished by interleaving workflow generation and execution is the ability to have the metadata that is dynamically generated during execution shape how the rest of the workflow is created. For example, the first few steps of the workflow may do a lot of filtering on a dataset, and once the filtering is executed we can

have specific metadata (e.g., the size of the file) affect what algorithms are selected in the latter part of the workflow.

We achieve this capability in WINGS by 1) modifying components to generate metadata during execution and store it in specific metadata files, and 2) adding breakpoints in the workflow.

A. Met Files: Dynamic Metadata Generation

A component can be written to dynamically generate metadata during execution for any of of its data products. The metadata is included in a metadata file named with a ".met" extension, as in [output-file].met. The file consists of multiple lines of [property]=[value] pairs, such as:

```
hasSize=12592
numberOfLines=130
```

Plain property names are used instead of URIs in order to make things simpler for component developers, and also to allow portability of components and workflows while moving from one domain/user/system to another.

These metadata files are hidden from end users. In WINGS, they do not appear in the user interface.

B. Breakpoint Variables: Controlling Execution to Fetch Dynamically Generated Metadata

Execution can be paused by marking a data variable in a workflow as a breakpoint to indicate that dynamically generated metadata should be fetched at that point before proceeding. A workflow can have several breakpoints can be specified for any given wokflow.

When the workflow execution algorithm reaches a breakpoint, the execution is suspended. The entire workflow, both the originally planned workflow and the ongoing workflow execution state are captured. Workflow generation is re-invoked. The workflow generation algorithm checks if a met file exists for the data binding of the breakpoint variable. If a met file does not exist, then the algorithm creates a truncated workflow to submit to for execution that has all the nodes up to this breakpoint as well as breakpoints in other branches. The truncated workflow will then be executed, at which point the met file will be generated. The workflow will then be sent back to workflow generation. When a met file exists, the dynamically generated metadata from the met file overrides predicted metadata for that data binding. This allows the system to communicate dynamically generated metadata during execution to the workflow generation process.

To support this, we extended the workflow ontology with the property "breakPoint", which can be added to any workflow data variable. For example:

```
Output1
  a wflow:DataVariable ;
  wflow:breakPoint true .
```

Figure 3 illustrates how this process is set up by the user in WINGS. Figure 3(a) shows how a variable can be marked as a breakpoint from the user interface to the workflow system. Figure 3(b) shows the initial truncated workflow up to the

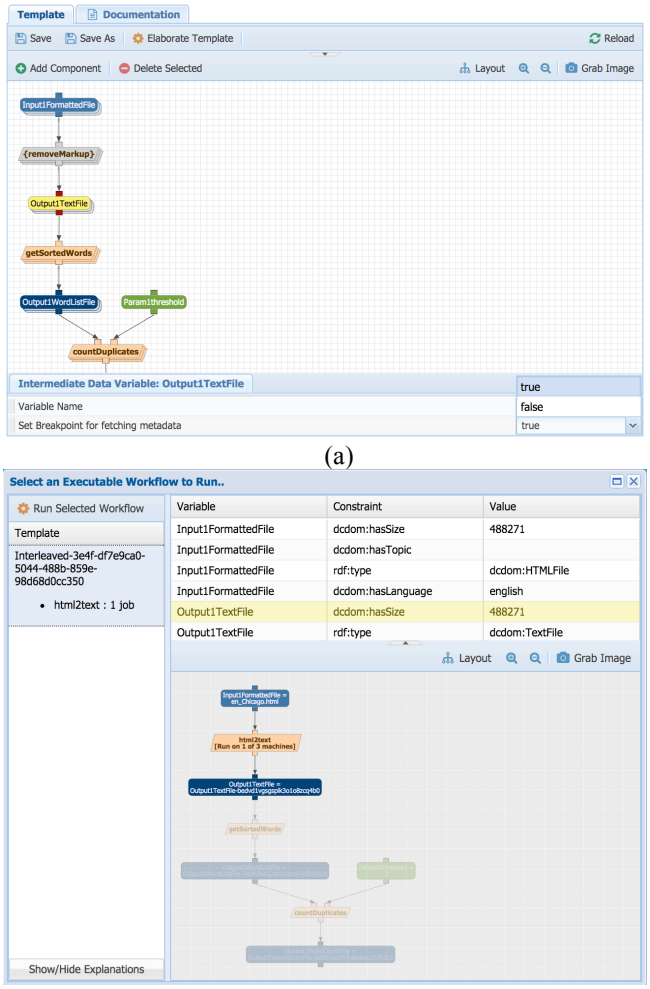

Fig 3. Incorporating dynamically generated metadata into the workflow generation process: (a) marking a variable as a breakpoint, (b) a truncated workflow with all the nodes up to the breakpoint will be submitted for execution, when the breakpoint is reached then the workflow is sent back to the workflow generation algorithm and the dynamically generated metadata will be fetched.

breakpoint that will be submitted for execution. Note that the whole workflow is generated, but only the truncated workflow will be initially executed. Once the breakpoint is reached, dynamically generated metadata for that variable's data will be available as a .met file and used to (re-)generate the next truncated workflow up to the next set of breakpoints.

IV. Dynamic Workflow Generation Based on Resource Availability

Another important capability that requires interleaving of workflow generation and execution is to allocate or deallocate resources dynamically. A user may want to indicate that a resource has become unavailable for a particular worklow, or that a resource suddendly has become available. This can be done during workflow generation or during workflow execution. It causes the system to reassign resources for

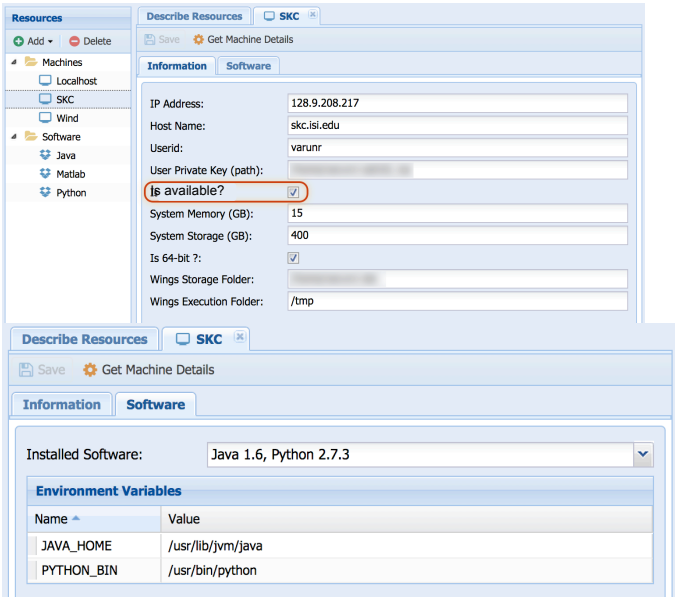

Fig 4. Changing the availability of a hardware resource can be done by the user even during workflow execution, and as a result the system regenerates the workflow and reassigns resources.

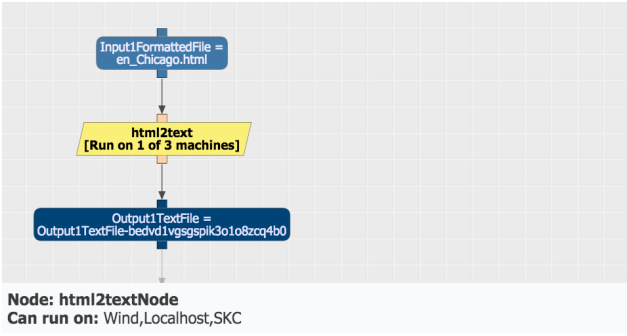

Fig 5. After resource selection, the user can see what are the possible execution resources for each component of the workflow.

execution. Many workflow execution systems have the ability to handle these changes in resource availability during execution. WINGS has a unique capability in being able to do this also during workflow generation.

In addition, the system is able to give the user an idea of the feasibility of their workflow given the selected resources. For example, one of the steps may require execution resourecs that are not available, and the system would let the user see that the workflow is unfeasible. Users may generalize a step in the workflow to include an abstract component so that more executable components may be candidates for that workflow step. This gives the user more flexibility in designing the workflow and managing resource allocation.

Figure 4 shows the WINGS interface to accomplish this. Each execution resource is described in terms of its hardware capabilities as well as the software installed in it. The user can indicate whether an execution resource is available or not.

In order to support this capability, we extended WINGS with a Resource Catalog that contains all the information about resource capabilities as well as their availability.

Figure 5 shows the interface to allow a user to see what resources have been assigned to a workflow. The user can go back and change the availability of resources at any time.

V. Interleaving Workflow Generation and Execution

This section describes the algorithms for interleaving workflow generation and execution used in WINGS. The pure workflow generation algorithms are quite complex and described formally in [Gil et al 2011b]. We do not present them here for lack of space, but they provide context for the modifications to support interleaving with execution. The algorithms are also complex because they handle the parallel processing of *data collections* and *component collections* described in compact form in the input workflow [Gil et al 2009]. Workflows generated with WINGS can be executed in a variety of execution engines [Gil 2013a; Gil 2013b].

A. Interleaving Workflow Generation, Resource Allocation, and Execution

Table 1 shows a high-level description of the algorithm for interleaving workflow generation, resource allocation, and workflow execution.

The algorithm starts with a *seeded workflow* consisting of a workflow template which may include abstract steps (e.g. a component class whose instances are executable components), data bindings for some or all of the input variables, and value bindings for some or all of the input parameters.

A workflow generation step generates a set of *expanded workflows* from a given seeded workflow. Expanded workflows have executable components for all the steps, bindings for all the input data variables, values for all the input parameters, and individual nodes to process each item in any collections in the workflow (data collections and component collections). The algorithm creates new nodes for each dataset being processed in any data collections. If a variable is marked as a breakpoint, then the algorithm checks if the workflow has been executed to that point and if so it retrieves the met file and merge it with the predictive metadata obtained from the Data Catalog. If the workflow has not been executed to that breakpoint, then the algorithm marks the workflow so the execution will stop at that breakpoint, and continues to expand the rest of the workflow but marking all the rest of the nodes as "inactive". This forms a *truncated workflow* to be submitted to execution. As the workflow undergoes iterations of the workflow generation algorithm interleaved with execution, a new truncated workflow is generated as the breakpoints are reached.

The next step is to select resources for each of the expanded workflows. This is done based on the software and hardware requirements of the components and taking into account the execution resources available. There may be many possibilities, and one may be selected by the user or by the system (for example based on minimizing execution time).

Next, the selected workflow is executed. Met files are dynamically generated as the workflow is executed. If it is a truncated workflow, it will only proceed until the breakpoints.

Table 1. Top-level algorithm for interleaving workflow generation and workflow execution.

```
Algorithm: INTERLEAVED-WF-GENERATION+EXECUTION
Input: seeded-workflow (workflow-template + input-data + input-parameters)
Output: execution-outputs

expanded-workflows ← WORKFLOW-GENERATION
                        (seeded-workflow)
resourced-workflows ← {}
for each expanded-workflow ∈ expanded-workflows
    workflow ← SELECT-RESOURCES(expanded-workflows)
    resourced-workflows ← resourced-workflows ∪ workflow
resourced-workflow ← WF-SELECTION(resourced-workflows)

execution-outputs ← WORKFLOW-EXECUTION-WINGS
                        (resourced-workflow)

while resourced-workflow.is-incomplete do
    expanded-workflows ← WORKFLOW-GENERATION
                            (seeded-workflow)
    new-resourced-workflows ← {}
    for each expanded-workflow ∈ expanded-workflows
        workflow ← SELECT-RESOURCES(expanded-workflows)
        new-resourced-workflows ← new-resourced-workflows ∪ workflow
    new-resourced-workflow ← resourced-workflow
    /* Select the first workflow that has more steps than current workflow */
    while num-steps(new-resourced-workflow) =
            num-steps(resourced-workflow)
        new-resourced-workflow ← dequeue(new-resourced-workflows)
    if new-resourced-workflow = resourced-workflow then
        mark-error(resourced-workflow)
        break
    end if
    resourced-workflow = new-resourced-workflow
    execution-outputs ← WORKFLOW-EXECUTION-DISTRIBUTED
                            (resourced-workflow)
end while

return execution-outputs
```

The algorithm then iterates the workflow generation, resource selection, and workflow execution until there is an execution failure, there are no possible resources to assign to some workflow task, or there are no more nodes in the workflow to execute. The algorithm also supports replanning in case a resource is no longer available, not just by reassigning resources but also by redesigning the workflow.

B. *Dynamic Generation of Metadata*

The workflow generation step is responsible for generating metadata as well as execution breakpoints. This means taking a seeded workflow and setting up parameter values as well as handling the generation of multiple nodes for parallel processing of data collections. The algorithm processes the input links of the initial bound workflow, and for each link it finds the port of the destination node that the link is connected to. The node can have several binding rule expressions to handle and combine data collections, such as cross-product, n-ways, and n-shift. It also handles dimensionality of the collections, for example if a component merges a collection of collections (e.g., a 2-dimensional collection would have m items appearing n times). The algorithm then creates as many nodes as needed to handle each item in the data collection. The handling of collections is described in detail in [Gil et al 2009]. Next, the algorithm calls the Component Catalog to get constraints on each node, which include constraints that set the values of each parameter.

At that point the algorithm can fetch the dynamically generated metadata for the input data of the workflow if it exists which would appear in the met file for that dataset, and merge that dynamically generated metadata with the predicted metadata. The dynamically generated metadata supersedes the predicted metadata when they differ. Next, the algorithm sets up whether node collections should be each processed in separate workflows (i.e., one workflow for each data item in the collection) or the same workflow depending on port binding rules (e.g., "Use all input data in the same workflow"). As always, if the algorithm detects inconsistent constraints then no configured workflows are returned.

VI. CONCLUSIONS

This paper describes an approach for interleaving planning and execution, which supports the incremental submission of partial workflows for execution until completion. As new metadata is generated dynamically during execution for all new data products, the workflow system uses that dynamically generated metadata to support dynamic planning and replanning of the workflow. The approach is implemented and integrated with the WINGS workflow system.

VII. ACKNOWLEDGMENTS

We would like to thank Shannon McWeeney and Christina Zheng of Oregon Health and Sciences University for valuable discussions of requirements and metadata management that were very valuable for this work.

REFERENCES

[1] De Roure, D; Goble, C.; Stevens, R. "The design and realization of the myExperiment Virtual Research Environment for social sharing of workflows". Future Generation Computer Systems, 25 (561-567), 2009.

[2] Gil, Y. "Mapping Semantic Workflows to Alternative Workflow Execution Engines." Proceedings of the Seventh IEEE International Conference on Semantic Computing (ICSC), Irvine, CA, 2013.

[3] Gil, Y. "Towards Task-Centered Network Models through Semantic Workflows." Proceedings of the IEEE Conference on Intelligence and Security Informatics (ISI), Seattle, WA, 2013.

[4] Gil, Y. "Intelligent Workflow Systems and Provenance-Aware Software." Proceedings of the Seventh International Congress on Environmental Modeling and Software, San Diego, CA, 2014.

[5] Gil, Y.; Groth, P.; Ratnakar, V.; and Fritz, C. "Expressive Reusable Workflow Templates." Proceedings of the Fifth IEEE International Conference on e-Science (e-Science), Oxford, UK, 2009.

[6] Gil, Y.; Ratnakar, V.; Kim, J.; González-Calero, P.; Groth, P.; Moody, J.; Deelman, E. "WINGS: intelligent workflow-based design of computational experiments." IEEE Intelligent Systems 26 (1), 2011.

[7] Gil, Y.; Gonzalez-Calero, P. A.; Kim, J.; Moody, J.; and Ratnakar, V. "A Semantic Framework for Automatic Generation of Computational Workflows Using Distributed Data and Component Catalogs." Journal of Experimental and Theoretical Artificial Intelligence, 23(4), 2011.

[8] Kozlovszky, M., Karoczkai, K., Marton, I., Balasko, A., Marosi, A., Kacsuk, P. "Enabling Generic Distributed Computing Infrastructure Compatibility for Workflow Management Systems." Computer Science, 13(3), 2012.

[9] Taylor, I., Deelman, E., Gannon, D., and M. Shields (Eds), "Workflows for e-Science: Scientific Workflows for Grids," Springer, 2007.

[19] Zheng, C.L., Ratnakar, V., Gil, Y., and S. K. McWeeney. "Reproducibility or Bust: Semantic Workflows for Clinical Omics". Genome Medicine, 7(73), 2015.