WOOL: A Workflow Programming Language

Geoffrey C. Hulette Matthew J. Sottile Allen D. Malony
Department of Computer and Information Science, University of Oregon
{ghulette,matt,malony}@cs.uoregon.edu

Abstract

Workflows offer scientists a simple but flexible programming model at a level of abstraction closer to the domain-specific activities that they seek to perform. However, languages for describing workflows tend to be highly complex, or specialized towards a particular domain, or both. WOOL is an abstract workflow language with human-readable syntax, intuitive semantics, and a powerful abstract type system. WOOL workflows can be targeted to almost any kind of runtime system supporting data-flow computation. This paper describes the design of the WOOL language and the implementation of its compiler, along with a simple example runtime. We demonstrate its use in an image-processing workflow.

1 Introduction

There has been a great deal of recent interest in workflows as a tool and paradigm for scientific computing [11]. Domain-specific scientific computations are relatively easy to express as workflows because, as a programming model, they closely match scientists' conceptual level of abstraction. In addition, workflow specifications are artifacts suitable for reuse and sharing, and they are naturally amenable to the identification and exploitation of concurrency [6]. Other new and useful features, such as automated generation tools and runtime support for management of data provenance through the computational process, are becoming the norm [10].

Workflows are an attractive program design paradigm partly because they reflect an intuitive approach to program construction similar to the common and intuitive "boxes and arrows" style of sketching out a program during the conceptual phase of its creation. However, using lower level languages for workflow programming can obscure the natural workflow abstraction because developers are forced to work with language primitives and details unrelated to the domain-specific concerns. On the other hand, programming languages for describing workflows tend to be highly complex, or specialized towards a particular domain, or both.

In this paper, we present a language to address the complexity and portability issues that are found in existing systems. Our solution, called WOOL (for Workflow Language), preserves the properties of most workflow languages for coordinating activities and data flow, but places a high priority on independence from specific runtime environments and a design that emphasizes ease of use and human writability without the need for GUI-based or other assistive tools.

1.1 Why are workflows useful?

Workflows represent an intuitive and simple design paradigm. The components that are composed together to form workflows often represent high-level abstractions closely tied to domain-specific analyses or models. As such, workflow programming gives scientific users a programmatic analogue to the sorts of diagrams they often sketch on a whiteboard or in a paper to describe the steps they take in solving their problem.

In most existing workflow systems, however, descriptions are complicated by details that imply a specific instance or class of runtime systems. This obscures the meaning of the workflow with implementation details, and reduces the ability of tools to retarget the abstract workflow to systems that deviate from the assumed model. WOOL avoids this specificity in the language, allowing the workflow paradigm to be used in constructing programs that span targets from a single, standalone executable on a laptop to an extensive grid-based distributed environment. The interesting problem is how to design a language that spans this range without imposing uncomfortable requirements on either endpoint, such as requiring a full grid node configuration for a standalone laptop that will never participate in a grid environment.

Workflows facilitate code reuse by dictating a component-based model for activities that execute as part of a workflow, and defining workflows themselves in such a way that allows them to be treated as a component. To make this possible they must have well defined semantics and types. Without these the meaning of the workflow is unclear and often relies on implicit assumptions that a specific workflow language or environment makes. Well-defined semantics and types facilitate retargetability and remove the need for assumptions that bind a workflow to a specific environment.

1.2 Workflows as a general programming model

Workflows are already important for scientific programming. We believe they have a role to play in more general purpose programming as well. The emergence of widespread multi-processing has created a need for languages that can help programmers identify and exploit parallelism in their programs. Workflows often exhibit natural parallelism in the form of pipeline and fork-join patterns. Unfortunately, workflow programming languages tend to be geared toward particular architectures and runtime systems. There is a need for an abstract workflow programming language that preserves the essential features of the workflow model while emphasizing retargetability of the actual execution environment.

Workflow programming can also benefit from text-based representations to facilitate portability, ease of editing, and compilation. While existing workflow languages are based on human readable XML, they are fall short in terms of human writable syntax. A simpler representation fits well with standard programming environment tools, such as version control. Graphical workflow representations can always be generated for purposes of presentation, and graphical workflow creation tools can also generate text-based forms.

2 Related Work

There are many existing languages and tools for describing workflows. Most are delivered as complete systems that include languages for describing workflows, a way of programming activities, and a runtime engine for executing a completed workflow.

Many earlier workflow projects were targeted towards the needs of business users, and employed coordinated web services as a enactment back-end. Examples of this style of workflow system include BPEL4WS [2] and WSFL [14]. These languages were designed to support and coordinate activities accessed through XML-based web services, and so while they are good at describing and enacting workflows in this particular domain, they are not very useful for abstract workflow representation.

Other workflow systems have focused on scientific workflows (SWFs). These are intended for use by scientists who want to focus on their problem domain and leave the low-level details to the SWF system. SWF tools delegate to the workflow language and runtime system the often-tedious programming needed to connect and orchestrate series of computational steps.

Triana [5][15] uses a visual SWF language that includes both data- and control-flow constructs. Triana, like other XML web service-oriented workflow systems, has a type system based on XSD schema datatypes [3]. XSD datatypes are powerful and flexible, but introduce additional complexity.

Taverna [16], part of the myGrid project, is a SWF system focused on supporting life sciences experiments. Activities are implemented either as web services or Java classes. Taverna relies on an XML-based language called SCUFL for workflow specification. SCUFL has a type system, but data types are restricted to MIME-types, names from the myGrid bioinformatics ontology, and free form text.

VisTrails [4] is another SWF system, interesting because it keeps provenance not only for data, but for workflows themselves. This allows VisTrails to treat the workflow itself as a kind of scientific notebook, documenting the evolving scientific process. VisTrail uses a visual workflow language, and is focused on workflows intended to be executed immediately and interactively.

Kepler [1] inherits a visual environment and the Modeling Markup Language (MoML) from Ptolemy [12], and adds SWF features like the ability to test a workflow without needing to completely program all its activities, distributed execution with a web-services framework or Globus grid, database access, and other specialized actors.

Other workflow systems are designed to let users easily harness the power of grid computing [9]. One example is WFEE [17], which uses a relatively simple workflow description language (called xWFL) with grid-specific constructs. WFEE features support for workflow parameterization using filenames, ranges of number, and constants, which is important for scientific workflow applications. Another example is GSFL [13], designed for Globus OGSA-based grids.

The Abstract Grid Workflow Language (AGWL) [7] is the closest project in spirit to WOOL. Like WOOL, AGWL was designed to specify workflows in a way that balances abstract representation with enough information to execute the workflow in a real environment. Unlike WOOL, AGWL makes parallelism an explicit construct in the language. This allows for a high degree of programmer control, but at the expense of the abstractness of the resulting workflow. Explicit parallelism may also increase the required level of sophistication for workflow programmers. Since parallelism is inherently implicit in data-flow programming models anyway, WOOL eschews explicit parallel constructs, instead opting only to have users to explicitly state when parallelism should be avoided.

Unlike WOOL, AGWL does not feature a robust abstract type system for validating connections between activities or identifying instances where implicit iteration or aggregation over sets of data items should be performed.

AGWL workflows are executed on a portable back-end system called CGWL [8]. CGWL must be ported to a particular platform, and acts as an interface between the platform and the workflow. WOOL could theoretically use either AGWL or CGWL as a target platform.

3 Language Design

The WOOL programming language was designed to describe workflows, and deliberately excludes orthogonal information related to the runtime system. It has an intentionally simple syntax and semantic interpretation. WOOL workflows are composed of “activities,” which are basic units of computation. Each activity has a type which assigns it a set of input and output ports and other properties. Connections between the ports on activities, from outputs to inputs, establish data-flow relationships. WOOL workflows can be composed hierarchically, with sub-workflows treated as activity types in a higher-level workflow. Activity ports use a rich type system to describe the primitive data items flowing in or out of the port, check connections for validity, and define the semantics of valid connections.

WOOL also includes both primitive and a standard library of activity types, available to all workflows and providing helpful functionality such as control flow. The WOOL language provides syntactic sugar to make certain control flow idioms easier to type and read. These are normalized at compilation time to the equivalent sequence of basic language primitives.

3.1 WOOL Syntax

There are two types of files used by the WOOL compiler. The first is called the “target” file, in which primitive data types and activity types are defined. Typically, target files are written for a particular domain. An auxiliary map is required that relates activity names in the target file to runtime-specific identifiers, such as Java class names. This name mapping is not defined to be part of WOOL, as it is specific to each runtime system to map abstract activities to concrete implementations.

The second type of file is the "workflow" file. These reference a target file for their types, and then instantiate and connect activities to form workflow graphs. Workflow files may import other workflow files in order to use their contents as complex activity types, but all the workflow files collected in this way must share the same target.

Figure 1 shows an example target file. The example defines two primitive types: *string* and *number*. It also defines two activity types: `Multiply` and `Split`. The `Multiply` activity type defines three ports, namely the two input operands and the one output result. The `Split` activity type has a similar structure. Notice that one of the outputs from `Split` is a sequence. Also notice that both of these activity types are declared `stateless`. This property allows the compiler to parallelize these activities when possible.

Listing 1. Example target file

```
# Target file: example.wft
# Define two primitive types
type string, number;

# Multiplies two numbers togther
Multiply {
  stateless;
  input leftOp:number,
        rightOp:number;
  output result:number;
}

# Split a string into characters
Split {
  stateless;
  input origString:string;
  output characters:string seq;
}
```

Now examine the workflow syntax example in Listing 2. First note that the target file from Listing 1 is referenced at the top. This imports the types defined in that target. Next, a workflow named `MyWorkflow` is defined. The workflow contains two activities named `mult1` and `mult2`, both of which use the `Multiply` activity type from the example target. The workflow uses external connection to define its ports, and connects the remaining outputs and inputs of the activities it declares (some connections are omitted for brevity).

Listing 2. Example workflow file

```
# Reference the target file
target example;

MyWorkflow {
  mult1:Multiply;
  mult2:Multiply;

  extern.op1 -> mult1.leftOp;
...
  mult1.result -> mult2.leftOp;
...
  mult2.result -> extern.result;
}
```

This very simple example is shown to give the flavor of the WOOL language; the full draft language specification can be found online.

3.2 WOOL's Type System

WOOL's type system ensures that it will reject workflow graphs that connect two incompatible ports. For example, the type system prevents a programmer from accidentally connecting a string output to a number input, or an output to another output.

The type system is very simple. Connections may only be established between ports that share the same primitive type, and are of opposite direction. Implicit casting between different primitive types is not supported.

WOOL has a notion of "external" connections, which are ports that are published by a workflow. These external ports are simply named aliases for unconnected ports within the workflow, and serve as the interface between the workflow and the outside world (or, if the workflow is used within another workflow, the external ports are the interface between the two). They are typed in the same way as the ports they alias. So, an external port that connects to an input port of type `string` will also be an input of type `string`.

A port may have a wildcard type instead of a primitive type. A wildcard is a way to tell the compiler that the activity does not care which primitive type is used for that port, it will work equally well with any of them. These are most often used for control flow activities which simply pass through data, such as aggregator or iterator. A wildcard type has a name whose scope is the containing activity, and which is used to "bind" the wildcard. A wildcard name is bound when a port using that wildcard name is connected to a second port whose type has already been resolved. A port's type is considered resolved if either it has a primitive type, or if it has a wildcard type that has already itself been bound. Once bound to a primitive type, all ports (either inputs or outputs) within the same activity having the same wildcard name will be bound to that type. A type error is reported if a conflict is identified during wildcard resolution.

The aggregate types augment the type information provided by either primitives or wildcards. The default aggregate type is the unit, meaning only a single data item of the primitive type. WOOL also supports sequences and sets as aggregate types. Sequences imply a group of primitive data items with a particular order, while sets are a group with no particular order. In general, connected ports must have the same aggregate type. However, connecting a set output to a sequence input is allowed, and will induce an order on the set. The exact ordering is undefined. Connecting a sequence output to a set input is also allowed, and will simply remove the ordering information from the sequence. Finally, connecting a sequence or set output to a unit input is allowed. This will cause the group to be serialized into individual data items. For example, consider a port that outputs a group of strings to an input that takes unit strings. Writing a single aggregate value with five elements to the output will cause five individual strings to be pushed into the input queue. This facilitates implicit iteration in a workflow.

3.3 Runtime Semantics

WOOL is designed to describe abstract workflows, as opposed to workflows tailored to a particular architecture. As such, it adopts a minimal set of assumptions about the semantics of its data-flow execution model. We believe that this should make WOOL workflows portable to and executable on almost any workflow execution system.

In WOOL, data is moved from outputs to inputs. Data delivered to an input must be queued in the order that it was received. Activities must eventually execute once there is data available on all their inputs. Execution consumes one data item from each of the activity's input ports, and produces zero or one data items on each of its output ports. Aggregate data types generally count as a single data item, with the exceptions outlined below.

An output can be connected to more than one input, in which case the data written to the output is copied to each of the connected input queues. Similarly, an input can be connected to more than one output, in which case both outputs feed their data into the single input queue. If two inputs arrive at the same input port at the same time, their order in the input queue is undefined.

External connections are simply aliases for the ports they connect to. Reading or writing values to an external port is the same as reading or writing it to the connected port.

The runtime system must respect aggregate types. That is, if a port is marked as having an aggregate type, then single values written or read from that port are treated as groups. In the case of sequences, the group must also have an ordering that is maintained between ports.

Aggregate types count as single values. That is, they occupy one space in input queues, and must be moved between ports as a group. The only exception is if an output port of aggregate type (a set or sequence) is connected to a port of unit type. In this case, the output port writes the group of values as a series of single values. These values will arrive at the input individually. If they are from a sequence, they will also arrive in the sequence order.

3.4 Transformations

WOOL supports special syntactic constructs that enable transformations on the final workflow graph. These transformations leverage a set of language primitives available to all WOOL workflows, and recombine them with user-defined activities. In this way, transformations provide simple syntax for common workflow idioms.

WOOL's map connection syntax is an example of a transformation. In a map connection, a subgraph that implements a unit-to-unit filter (i.e. an activity or sub-workflow that takes a unit input and produces a unit output) is rewired into an equivalent group-to-group filter. Map connections are implemented as a graph transformation — the incoming group is serialized, filtered individually, and then aggregated back into a group using activities from WOOL's standard library. This is analogous to the map or fold primitives available in most functional languages.

Transformations of this kind are WOOL's approach to control-flow constructs. Rather than rely on special language extensions, simple syntax combined with graph transformations allows for powerful and customizable control-flow expressions.

4 Implementation

WOOL is implemented as a two-stage compiler. The first stage consists of a parser and optimizer. In this stage, the WOOL program is validated for syntax and semantics, including type-checking, and an in-memory workflow graph is produced. The graph is currently implemented as a collection of Java objects conforming to our `Workflow` object interface. The graph is optimized, if possible, and any parallelizable regions are identified and notated. For example, map connections are expanded, and if their interior graph is stateless it is marked as parallelizable.

The second compiler stage transforms the in-memory graph into a form that may be executed by a workflow runtime system. Naturally, the exact nature of the output depends on the system being targeted, and so our system places no restrictions on what may be output. The transformation code must be implemented as a Java object conforming to the `Generator` interface, which accepts as input the graph generated in the first stage. The user may indicate which generator object to use at runtime by passing its class name as a command-line option to the WOOL compiler.

For testing and demonstration purposes, we also implemented a simple, Java-based workflow execution engine and wrote a `Generator`-conforming object that targets it. A secondary runtime based on a simplified tuple-space distributed computing model is currently being completed.

5 Application

The application of WOOL demonstrated for this paper was made in the context of medical image processing. Image processing workflows commonly take the form of pipelined processes, in which images flow through a sequence of operations that transform, identify, and measure features of interest. It is natural to define a workflow for a specific imaging problem that can be reused over time as new images are produced, and embedded in other workflows when more complex processing is desired. In this application we consider a simple workflow in which segmentation is performed based on the classical k-means clustering algorithm. This workflow will be embedded in a larger workflow to measure the variation in segmentation output as parameters on the k-means algorithm are varied. Figure 1 shows the overall workflow (center) and the two sub-workflows (left, right). The shaded components are user-defined, while the unshaded ones are built in.

The images used for this demonstration are histology slides related to the study of acute inflammation of placental tissue during fetal development[1]. Segmentation is used to compute geometric properties of the images that are indicators of infection, and a common question to ask is what variation is expected in the segment assignment as parameters are varied in order to quantify uncertainty due to algorithmic side effects.

[1]The images were provided by Dr. Carolyn Salafia of Placental Analytics, LLC. and NYU School of Medicine.

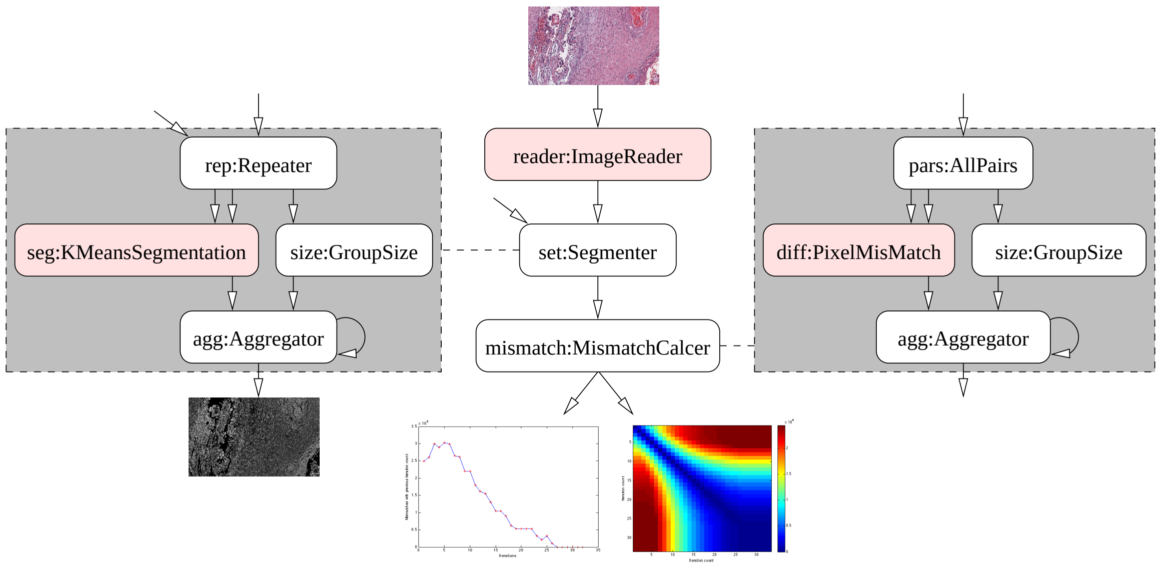

Figure 1. Segmentation parameter study workflow and sub-workflows.

5.1 Segmentation of a single image

The base workflow that we will embed in a larger context is that of simple image segmentation (Figure 1 left). The k-means segmentation algorithm takes two parameters: the segment count k, and the number of refinement iterations that it performs, i. This workflow takes as input an image filename and segmentation parameter, invokes the k-means algorithm, and produces the resulting segmented image as its output.

5.2 Parameter study

The larger workflow in which the segmentation workflow is embedded (Figure 1 center) explores the effect of changes to a segmentation parameter on the stability of the resulting image. We are interested in the effect of fixing k and varying i over a small range, such as $i = \{5, ..., 15\}$, and determining the number of pixels that change segment membership between subsequent iteration counts.

This workflow will take a set of parameters as input, and invoke the segmentation workflow which executes the segmentation component repeatedly, once for each parameter. The result will be a set of segmented images. A second sub-workflow is provided that takes a set of segmented images, and computes the mismatch in segment assignment at each pixel for two different parameter choices (Figure 1 right). The full set of mismatch counts between all combinations of iteration counts is shown as the right output of the main workflow. A plot of how mismatches change between iteration counts of i and $i + 1$ is shown in left output.

WOOL enables the use of sub-workflows to partition the domain functions (k-means segmentation and pixel mismatch) from the overall pipeline. Note that the `Aggregator`, `Repeater`, `GroupSize`, and `AllPairs` activity types are drawn from WOOL's standard library. These activities exist in the workflow to implement common control flow idioms, and can be introduced with the transformation syntax described in Section 3.4.

6 Conclusion

The WOOL system provides a simple but effective abstract workflow language with human-readable syntax and intuitive semantics. It is general enough to specify workflows targeted to almost any workflow runtime. The language includes a type system that allows workflows to be verified independent of a particular runtime. We implemented a compiler and sample generator. Finally, we have shown that WOOL is a viable and useful language for structuring a relatively involved image-processing workflow.

WOOL is a work in progress, and there are several areas that we would like to investigate further. Activities in WOOL should be able to choose aspects of their input and output semantics. For example, what happens to inputs as they arrive at a port? Currently they are queued in the order received, but alternatives might be to discard messages or to have some kind of priority queue. Providing workflow designers with choices for activity semantics might enable very concise and powerful specifications for complex workflows.

The application example demonstrated in this paper targets a very minimal, single-threaded workflow runtime. Another area for future work will be to target different kinds of runtime systems. For example, we are currently working on a parallel workflow system based on tuple spaces. It should be possible to leverage WOOL's unique type system to aggressively optimize workflows targeted to such a system for very high performance.

Finally, we note that exceptions are hugely important in workflow systems, particularly for complex scientific codes that run for very long periods of time. We are looking at ways to introduce robust exception handling into the WOOL language, while still maintaining runtime independence.

References

[1] I. Altintas et al. Kepler: an extensible system for design and execution of scientific workflows. In *Proc. of the 16th Intl. Conference on Scientific and Statistical Database Management*, pages 423–424, 2004.

[2] T. Andrews et al. Business Process Execution Language for web services version 1.1, May 2003.

[3] P. V. Biron and A. Malhotra. XML schema part 2: Datatypes second edition. W3C recommendation, World Wide Web Consortium, October 2004.

[4] S. P. Callahan et al. Managing the evolution of dataflows with VisTrails. In *22nd Intl. Conference on Data Engineering Workshops*, pages 71–75, Los Alamitos, CA, USA, 2006.

[5] D. Churches et al. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[6] E. Deelman and Y. Gil. Final report of NSF workshop on challenges of scientific workflows, May 2006.

[7] T. Fahringer, S. Pllana, and A. Villazon. A-GWL: Abstract Grid Workflow Language. In *4th Intl. Conference on Computational Science*, Lecture Notes in Computer Science, pages 42–49, June 2004.

[8] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with AGWL: an abstract grid workflow language. In *Proc. of the Fifth IEEE Intl. Symposium on Cluster Computing and the Grid*, volume 2, pages 676–685, Washington, DC, USA, 2005.

[9] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, June 2002.

[10] J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engineering*, 10(3):11–21, May–June 2008.

[11] Y. Gil et al. Examining the challenges of scientific workflows. *Computer*, 40(12):24–32, December 2007.

[12] C. Hylands et al. Overview of the Ptolemy project. Technical report, University of California Berkeley, 2003.

[13] S. Krishnan, P. Wagstrom, and G. von Laszewski. GSFL: A workflow framework for grid services, 2002.

[14] F. Leymann. Web Services Flow Language (WSFL 1.0). Technical report, IBM, May 2001.

[15] S. Majithia, M. Shields, I. Taylor, and I. Wang. Triana: A graphical web service composition and execution toolkit. In *Proc. of the IEEE Intl. Conference on Web Services*, pages 514–522, Los Alamitos, CA, USA, 2004.

[16] T. Oinn et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

[17] J. Yu and R. Buyya. A novel architecture for realizing grid workflow using tuple spaces. In *Proc. of the Fifth IEEE/ACM Intl. Workshop on Grid Computing*, pages 119–128, Washington, DC, USA, 2004.