The WOOL Workflow Programming Language
DRP Project

Geoffrey C. Hulette

1 Introduction

There has been a great deal of recent interest in workflows as a tool and paradigm for scientific computing [13]. 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 [8]. 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 [12].

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 covers 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 often 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 systems. Graphical workflow representations can always be generated for purposes of presentation, and graphical workflow creation tools can also generate text-based forms. So-called "graphical" languages, however, often must include ancillary information to support their pictorial representations, such as layouts for components. Such information is helpful for presentation, but extraneous to the task of representing a workflow program.

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 [3] and WSFL [16]. 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 [7][17] 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 [5]. XSD datatypes are powerful and flexible, but introduce additional complexity.

Taverna [18], 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 [6] 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 [2] inherits a visual environment and the Modeling Markup Language (MoML)

from Ptolemy [14], 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.

Apple's Quartz Composer [4] is not specifically marketed as a SWF system, but shares many of the same features. Quartz Composer allows users to build workflows describing data-flow between activities, and uses an object-oriented type system to validate connections. A graphical environment makes workflow construction straightforward for non-programmers. Workflows in Quartz Composer are tied to the software architecture (Cocoa and Objective-C) of the Mac OS X platform.

Other workflow systems are designed to let users easily harness the power of grid computing [11]. One example is WFEE [20], 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 [15], designed for Globus OGSA-based grids.

The Abstract Grid Workflow Language (AGWL) [9] 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 [10]. 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 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 activity ports, 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.

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. 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.

Listing 1 shows an example target file. The example declares two primitive types: *string* and *number*. WOOL uses the types to enforce that connections between ports are valid; it is up to the runtime system to ensure that WOOL's abstract types are mapped consistently and correctly.

The example also defines a couple of activity types. The `Multiply` activity has three ports. The first two are the inputs, named `leftOp` and `rightOp`. Both of the input ports are designated as having a primitive type of `number`. The single output port is named `result`, and also has the type `number`. Note that while the intended function of the `Multiply` activity is clearly to take two numeric operands and produce their product, this is not explicitly coded in WOOL. The activity types in WOOL simply specify inputs, outputs, and some meta-information – the functionality is the responsibility of the runtime system.

The other activity types in the example follow a similar pattern. Notice that the `Split` activity makes use of aggregate types; one of its outputs is declared as a sequence. Also notice that the activity types in the example are all declared to be `stateless`. This property tells the compiler that the activities carry no stateful information between invocations, which may be helpful during the compiler's optimization phase.

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;
}

# Cast a number to a string
NumberToString {
        stateless;
        input num:number;
        output str:string;
}

# Print out a string (to some default output device)
PrintString {
        stateless;
        input str:string;
}
```

Now examine the workflow syntax example in Listing 2. First notice that the target file from Listing 1 is referenced at the top. This imports the types defined in that target for use in this workflow. Next, a workflow named `MultiplyThreeNumbers` is declared and defined. The workflow instantiates two activities named `mult1` and `mult2`, both of which are typed with the `Multiply` activity from the target. Next, the workflow publishes three of the four input ports from the two `Multiply` activity instances, mak-

ing them available to the outside world. Notice that the published names need not be the same as the names of the internal ports (e.g. `mult1.leftOp` is published under the alias `firstOp`). Next, a connection is declared between the output of the first multiplication and input second. Finally, the workflow publishes the output of the second multiplication. So, logically the `MultiplyThreeNumbers` workflow takes three numeric inputs, multiplies them all together, and pushes the product to its single output. This workflow is trivial but complete.

Listing 2. Example workflow file

```
# Reference the target file
target example;

MultiplyThreeNumbers {
  mult1:Multiply;
  mult2:Multiply;

  publish mult1.leftOp as firstOp;
  publish mult1.rightOp as secondOp;
  publish mult2.rightOp as thirdOp;

  mult1.result -> mult2.leftOp;
  publish mult2.result as result;
}
```

Next, lets examine how workflows can be composed hierarchically. In Listing 3, we define another workflow file. This workflow uses the `import` keyword to reference the workflow `MultiplyThreeNumbers` from Listing 3. Once imported, a workflow is available for use as an activity in the new, higher-level workflow, and may be used in the same way as any other activity. Listing 3 declares a workflow named `MySuperWorkflow`, which instantiates an instance of the `MultiplyThreeNumbers` workflow named `mult`. It also declares two other basic activities that were defined in the target from Listing 1: `num2str` is of type `NumberToString`, while `printer` is of type `PrintString`. `MySuperWorkflow` publishes the three inputs to the multiplication, but feeds the output through the other two activities, first converting the numeric output to a string, and then printing the string.

Listing 3. Example super-workflow file

```
# Reference the target file
target example;

import MultiplyThreeNumbers

MySuperWorkflow {
  mult:MultiplyThreeNumbers;
  num2str:NumberToString;
  printer:PrintString;

  publish mult.firstOp as firstOp;
  publish mult.secondOp as secondOp;
  publish mult.thirdOp as thirdOp;

  mult.result -> num2str.num;
  num2str.str -> printer.str;
}
```

This very simple example is shown to give the flavor of the WOOL language; the full draft language specification is included in Section 4.

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.

It is important to understand that WOOL's primitive types do not constrain the implementation of the runtime system. Primitive types, such as `number` and `string` in Listing 1, have no explicit meaning outside of the WOOL compiler – they are simply placeholders for types that exist in a targeted workflow runtime system. So, one runtime system may map the `number` type to a 32-bit integer, while another may use a 64-bit double or even a higher level type like `java.lang.Integer`. It is up to the implementer of the workflow runtime and its associated WOOL generator to ensure that types are mapped consistently from WOOL, and that the system uses the correct types at runtime. This flexibility ensures that WOOL can target any existing workflow system, while retaining the

ability to type-check its abstract workflows.

WOOL has a notion of "published" ports, which are ports that are made available outside the workflow in which they are defined. Published ports are simply named aliases for ports within the workflow, and serve as the interface between the workflow and its external environment. If a workflow is used within another workflow, the published ports of the sub-workflow are visible from the super-workflow, and may be connected just like any other port. Published ports are typed in the same way as the ports they alias. So, a published input port that is of type `string` within the workflow will also appear as an input of type `string` from outside.

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 an 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 Basic 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.

Published ports are simply aliases to ports within a sub-workflow. Reading or writing values to a published port is the same as reading or writing it to the corresponding underlying 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.

A more complete discussion of WOOL's semantics is given in Section 4.

4 The WOOL Language

WOOL is a typed workflow definition language. It allows a programmer to define workflows from basic activities, combine workflows to create hierarchical workflows, and to ensure that the data flowing between activities is consistently typed. WOOL workflows are based on data–flow relationships between "activities." An activity receives a set of inputs, performs some computation on those inputs, and then writes a set of outputs. Connections between activities (from outputs to inputs) describe a workflow.

This guide describes the WOOL language, syntax, and type system.

4.1 Terminology

The following terms are used throughout this guide. They are presented here to give an overview, but are defined more completely in later sections.

Primitive Data Type The type of a value that can be sent to an output or read from an input. Examples of primitive data types include numbers and text strings. In WOOL, instances of primitive data are not represented - only the types are important.

Activity Type An abstract representation of a particular computation. Activity types define a set of ports for input and output, as well as properties that can help the WOOL compiler during optimization. Addition is a trivial example of an activity type - the computation has two numeric input ports (the operands) and one numeric output port (the sum). Activity types are sometimes called "basic" activity types to distinguish them from complex activity types (described below).

Complex Activity Type An activity type where the computation is not abstract, but instead is defined by a workflow. Using complex activity types allows the programmer to build workflows hierarchically, composing simple workflows into larger ones.

Activity Type Property A descriptive tag that may be applied to a basic activity type. Properties are intended to inform the WOOL compiler about an activity so that it can properly optimize the workflow. For example, the "stateless" property tells the compiler that an activity carries no state from one invocation to the next, and this trait may allow the activity to be parallelized.

Activity An instance of an activity type that can be connected into a workflow. There may be zero, one, or many activities that share a single activity type. Each activity has its own set of input and output ports, which do not share state with other instances of the same activity type. We say an activity is "invoked" when it is caused to execute its computation, consuming a set of inputs and producing a set of outputs.

Port Describes an input or output of an activity. Ports have names and are typed as either a "wildcard" (described below) or with a primitive data type. Port types may also specify that they output a single piece of data or a collection (see "Aggregate Port Type" below). Instantiating an activity will also instantiate its corresponding set of ports, as defined by its activity type. The output from a port may only be connected to an input port with a matching type. The type system is described more fully later in this guide.

Aggregate Port Type Usually, a port's type indicates that it will read or write a single unit of that primitive type. It is also possible to specify that an aggregate group of data items will be output, instead of just one. Groups may be either unordered sets or ordered sequences. Generally, the port grouping type (either unit, set, or sequence) must match for two ports to be connected, but it is permitted to connect an output port that writes a group (of some primitive type) to an input that accepts only a singleton (of that same type). This causes the group to be "serialized," or written one at a time, to the input.

Wildcard Port Type Instead of specifying a particular primitive type, a port can accept a "wildcard," or any type. This is useful for activities that implement control-flow constructs, such as loops and conditional branches. Wildcards must be named, and the type system ensures that if an activity has a wildcard input port named "alpha" that is connected to a concrete type (e.g. a string), then the all other wildcard ports named "alpha" for that activity must be connected to the same concrete type. We say that connecting a concrete type to a wildcard "binds" the type of the wildcard.

Connection An edge in a workflow graph between the output port of one activity and the input port of another. Connections are established when the workflow is created, and cannot be changed at runtime. Connections must be between ports that have compatible types, as described later in this guide.

Published Port Exposes a port inside a workflow to the environment outside the workflow. Generally, the activities and ports within a workflow are concealed, known only to the workflow itself. Publishing a port allows either the runtime system or another workflow can push data in and get data out.

Map Connection A map connection takes an aggregate output (a set or sequence of some type) from a particular port, and runs each element through a "filter" or transformation. Then, the transformed elements are assembled back into an aggregate group and output to the next activity in the data–flow graph. Map connections are interesting because they provide a way to specify implicitly parallel operations, and because they can be implemented using a graph transformation (see Section 7.2).

4.2 Semantics of a WOOL Program

WOOL is designed to describe abstract workflows, as opposed to workflows tailored to a particular architecture. As such, it tries to assume very little about the workflow runtime system. Nevertheless, there are some (hopefully minimal) features that must be provided by the runtime.

1. The runtime must provide a way to push data into a workflow's published inputs, and to retrieve it from published outputs.

2. Data delivered to an input that is not immediately processed must be queued, in the order that it was received.

3. Activities will eventually execute once there is data available on all their inputs.

4. When an activity executes, one data item is removed from each of its input queues. These values are passed to the activity and consumed during execution.

5. After execution, an activity will write zero or one values to each of its output ports (aggregate groups count as one value).

6. If, after executing (and consuming a set of input values), an activity still has values in all its input queues, it must eventually be scheduled for execution again.

7. A value written to an output port must eventually be moved out of the output port and into input ports that are connected to it.

8. An output can be connected to more than one input ports, in which case the data is copied to both input queues.

9. An input can be connected to more than one output ports, in which case both outputs feed their data into the single input queue. If two inputs arrive at the same time, their order in the input queue is undefined.

10. Published ports are simply aliases for the ports they connect to. Reading or writing values to a published port is the same as reading or writing it to the underlying, internal port.

11. 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 really groups. In the case of sequences, the group must also have an ordering.

12. Aggregate types count as single values. That is, they occupy one space in input queues, and must be propagated as a group. The only exception is if an output port of aggregate type 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, in order if they are from a sequence.

4.3 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 basic type system is very simple. Connections may only be established between ports that share the same primitive type, and are of opposite direction. For example, an output port with a string primitive type may be connected to an input port of a string primitive type, but not to an output port (of string or otherwise), and not to an input port of numbers (or of any primitive type other than string).

Published ports are typed in the same way as the ports they alias. So, a published input port of type string will also appear to the external environment as an input of type string.

A port may be of 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. A wildcard type has a name, 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 input or output) of the same activity that share the wildcard name will be bound to that type as well.

The aggregate types augment the type information provided by either primitives or wildcards. The default aggregate type is a single unit, meaning only a single data item of the port's 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, the following connections are also permissible:

1. Connecting a set output to a sequence input is allowed, and will induce an order on the set. The exact ordering is undefined (or, defined by the runtime).

2. Connecting a sequence output to a set input is allowed, and will simply remove the ordering information from the sequence.

3. 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 (grouped) value with five elements to the output will cause five individual strings to be pushed into the input queue.

All ports have a concrete aggregate type, either the default unit type or one of the types set or sequence.

4.4 WOOL Syntax

This section explains the syntax and file types involved in creating a WOOL workflow.

4.4.1 File Types

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. Target files are typically constructed so as to cover the needs of a particular domain. 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.

4.4.2 Basic Syntax

Both target and workflow files share the following syntax:

1. Comments start with a # and continue to end of line, and can be placed anywhere.
2. Identifiers must start with an alpha, and contain only alphas, numbers, and underscores.
3. Whitespace is ignored.

4.4.3 Target File Syntax

Target files define the primitive data and activity types that are used and shared by a workflow or set of workflows. Target filenames must use the suffix ".wft".

Target files are divided into two parts. The first part declares the primitive data types, and the second part declares the activity types. Formally, target files have the following syntax:

```
TARGET_FILE ::= TYPE_DECLS ACTIVITY+
```

The formal syntax for `TYPE_DECLS` and `ACTIVITY` are given below.

Primitive types declarations have the following syntax:

```
TYPE_DECLS ::= TYPE_DECL*
TYPE_DECL  ::= 'type' ID (',' ID)* ';'
```

where `ID` is a valid identifier, and each `ID` declares a primitive type with that identifier as its name. It is an error to declare a primitive type more than once.

After the primitive types, the target file contains one or more activity type definitions. An activity type definition has the following syntax:

```
ACTIVITY             ::= ACTIVITY_ID '{' DECLARATION* '}'
DECLARATION          ::= (INPUT | OUTPUT | PROPERTY) ';'
INPUT                ::= 'input' PORT (',' PORT)*
OUTPUT               ::= 'output' PORT (',' PORT)*
PORT                 ::= PORT_ID ':' AGGREGATE_PORT_TYPE
AGGREGATE_PORT_TYPE ::= PORT_TYPE | (PORT_TYPE ('seq' | 'set'))
PORT_TYPE            ::= PRIM_PORT_TYPE | WILDCARD_PORT_TYPE
PRIM_PORT_TYPE       ::= ID
WILDCARD_PORT_TYPE   ::= '*' ID
PROPERTY             ::= 'stateless'
```

where `ACTIVITY_ID` is an identifier denoting the name of the activity type, and `PORT_ID` is the name of an input or output port. It is an error to declare more than one activity type of the same name in the same target file, and an error to declare more than one port of the same name in a single activity type. `PRIM_PORT_TYPE` denotes that the port is of a primitive data type, and must be an identifier matching one of the primitive types defined at the top of the target file. `WILDCARD_PORT_TYPE` denotes a wildcard type name. `AGGREGATE_PORT_TYPE` specifies the aggregate type by using the keyword `seq` for a sequence, `set` for a set. If neither of those two aggregate types is specified, the unit aggregate type is used.

Note that the only property currently supported is statelessness, which is specified with the `stateless` keyword.

4.4.4 Workflow File Syntax

A workflow file defines one or more workflows based on a set of types in a target file. A single workflow consists of a set of activities (basic or complex), and connections between the ports of those activities. A workflow also declares a set of published ports for moving data in and out, and also possibly a set of properties.

The syntax for a workflow file is:

```
WORKFLOW_FILE ::= TARGET IMPORTS WORKFLOW*
TARGET        ::= 'target' TARGET_ID ';'
```

where TARGET_ID is an identifier referencing a target file. Target files are referenced by their filename, minus the ".wft" suffix. The target file must be in the same directory as the workflow file, or in a user-defined search path.

Workflow files may import other workflow files, which can be helpful for modularizing code. Imported workflow files must have the same target as the original workflow file. The syntax is:

```
IMPORTS ::= IMPORT*
IMPORT  ::= 'import' ID (',' ID)* ';'
```

where ID is an identifier referencing a workflow file. The rules for locating imported workflow files are the same as for target files. Imported workflows are in the same namespace as the workflows defined in the original file.

Workflows have the following syntax:

```
WORKFLOW   ::= (WORKFLOW_ID '{' STATEMENT* '}')*
STATEMENT  ::= (DEFINITION | CONNECTION | PUBLISH) ';'
DEFINITION ::= ACT_ID (',' ACT_ID)* ':' ACT_TYPE_ID
CONNECTION ::= PORT '->' PORT (',' PORT)*
PUBLISH    ::= 'publish' PORT 'as' ID
PORT       ::= ID '.' ID
```

where WORKFLOW_ID is a unique identifier for the complex activity type defined by the workflow. Definitions instantiate activities, with the locally scoped names given by ACT_ID and the activity type by ACT_TYPE_ID.

Connections are defined from a source port (on the left hand side) to one or more sink ports (the right hand side). Ports can be either regular ports, which are specified with two IDs - the first identifies the activity instance, which must have been defined previously in the workflow, and the second identifies the port within that activity. The port name be valid (i.e. specified by the activity type), and the connection is subject to the typing rules.

Ports can be published, in which case the published name can be any valid identifier (that has not already been used as a published port name). A port within a workflow can only be published under a single name.

5 Implementation

WOOL is implemented as a two-stage compiler. The first stage transforms a set of text files containing valid WOOL syntax into a Java object-based intermediate graph representation. The intermediate representation is either stored in memory for immediate processing, or written out to disk using Java's standard object serialization protocol. Additionally, in this stage the WOOL program is validated for syntax and semantics, and the graph is passed through an optimizer. This phase includes routines for static type-checking of connections between activities.

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.

The complete documentation for the APIs discussed in this section may be found online in Javadoc format [1].

5.1 Lexer and Parser

WOOL's grammar and tokenizing code are described using ANTLR [19]. ANTLR was chosen for use in WOOL over several alternatives because of its clean syntax and integration of lexer and parser grammar specification. ANTLR also supports generation parsers that are easily callable from Java, a feature that we considered essential since the rest of WOOL is written in Java.

The lexer and parser together produce an `Environment` object (see Section 5.3) representing workflows and metadata from the source code. This object is then passed through a type-checking phase, which validates each connection within the workflow by its declared types. Next, this complete, valid `Environment` object is passed to the optimizing phase.

5.2 Optimizer

At the time of this writing, the optimizer phase does not do anything with the `Environment` object – it simply passes the object through unchanged. We are currently working, however, on a number of graph optimizations (see Section 7.3) that will be implemented as modular `Optimization` objects. These objects can be plugged into the optimizer phase

[1] `https://trac.nic.uoregon.edu/wool`

in any order. Each `Optimization` object is nothing more than a graph transformation, i.e. an `Environment` object is passed to the optimization, which transforms it in place. Each optimization is executed in a serial fashion, and the types and order of optimizations may be set by the user.

The optimized graph is typically written out as a binary object using Java's standard `java.io.Serializable` interface, although our current front-end also keeps the graph in memory for use by a `Generator` (see Section 5.4).

5.3 Intermediate Representation

WOOL's intermediate workflow representation consists of a set of Java interfaces that allow a programmer to query the structure of the workflow graph. Note that, for the time being, the interfaces do not specify methods to modify the graph – it is assumed that the IR is to be used primarily to generate code for a runtime system, a task for which read-only access should be sufficient.

The result of a successful WOOL compilation is a single `Environment` object. From this single object, the attributes of the compiled WOOL program can be accessed. An `Environment` provides access to a single `Target` object, which in turn provides access to the available primitive and complex types. The `Environment` also contains a list of named `Workflow` objects. Each `Workflow` object is a graph representation of activities, ports, and connecting edges. Since WOOL workflows may be hierarchical, a (higher-level) workflow in an `Environment` may refer to another workflow by using an activity with a complex type. The complex type name will correspond to the lower-level workflow. Starting from the `Environment`, the complete graph of any hierarchical workflow in the WOOL program may be traversed.

5.4 Generator

Once the WOOL compiler has transformed the source code into the intermediate graph representation, the next step is to traverse the graph in order to generate an executable workflow. Since WOOL does not assume any particular runtime system, the exact form of the executable is not be coded into the compiler. Instead, WOOL provides a modular system for transforming the graph IR into an executable, via its `Generator` interface. This Java interface provides a single method that takes a graph in WOOL's IR form (an `Environment` object, to be precise), and also an optional string parameter indicating the "primary" workflow that should be generated. To target a particular runtime system, an object conforming to the `Generator` interface is created that traverses and outputs the IR graph in an appropriate executable form.

5.5 Example Runtime and Generator

For testing and demonstration purposes, we implemented a simple, Java-based workflow execution engine and a corresponding `Generator` object. The runtime engine is quite simple. Workflow execution is single-threaded, and there is no runtime type-checking. Workflow components (i.e. WOOL's "activities") are simple Java objects that conform to an `Activity` interface. This interface provides a single method, called `execute`, that is called when the runtime executes the activity. The method takes two standard Java `HashMap` objects as parameters – the first represents the objects arriving on the activities input ports, and the second represents the set of data leaving via the output ports. The output data are initially empty, but filled during the course of the `execute` method. The runtime manages details such as queuing inputs, moving data from output ports to input ports, and semantics for sets and lists.

The runtime system requires a way to map WOOL activities to their corresponding Java objects. This is currently accomplished via a simple `HashMap` that is passed into the runtime system when it is instantiated.

To target this runtime, we created a `Generator`-conforming object. This object takes the WOOL intermediate graph representation and produces sub-classes of certain generic objects from the runtime system. These sub-classes are specialized so that they instantiate the appropriate activities and move data according to the compiled graph.

Implementing the `Generator` object for our example runtime was a valuable experience. In particular, we found that it was difficult to traverse hierarchical workflow graphs using just the information in the intermediate graph representation. WOOL's graph IR treats sub-workflows as complex activities; that is, just like a simple activity, but with the implementation given by a named sub-workflow rather than an atomic computation. Traversal of such a graph requires that the `Generator` implement data structures to keep track of which level of the workflow it is currently in. Worse, since port names are only unique within a single level of a hierarchical workflow, the `Generator` must keep track of the local namespace. This proved to be quite a burden. We are currently working to improve WOOL's graph representation by incorporating the data structures and methods for hierarchical traversal into the WOOL API.

6 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

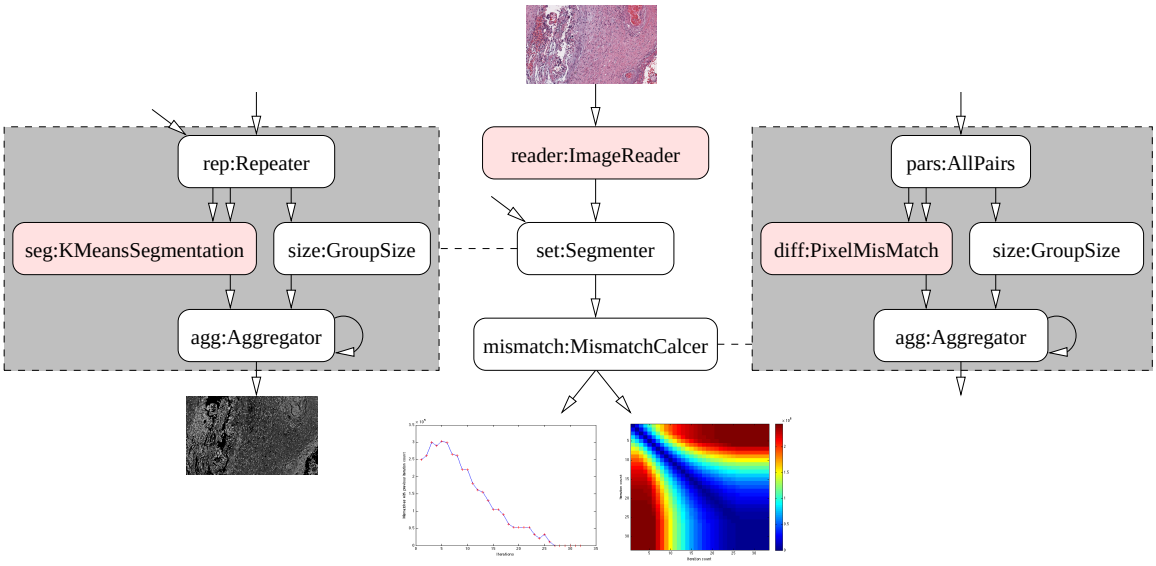


Figure 1. Segmentation parameter study workflow and sub-workflows.

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[2]. 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.

6.1 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 work-

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

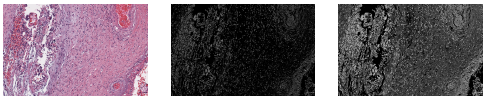
flow takes as input an image filename and segmentation parameter, invokes the k-means algorithm, and produces the resulting segmented image as its output.

6.2 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. An example image and two such parameterizations of the segmentation algorithm are shown in figure 6.2.

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 for each pixel over the entire image 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 7.2.

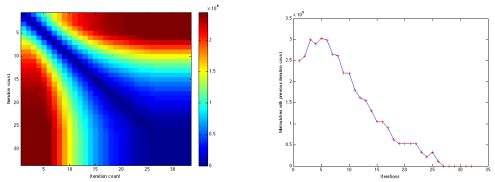


(a) Original (b) $i = 5$ (c) $i = 14$

Figure 2. Sample image and results.

7 Future Directions

Research and development on the WOOL language and tools are ongoing. There are several areas that we would like to investigate further.



(a) All pairs (b) i vs $i + 1$ pairs

Figure 3. Pixel mismatches vs. iteration count.

7.1 Standard Types and Activities

Currently, WOOL is complete agnostic as to how primitive and activity types get mapped into a particular runtime system. In a WOOL program there may be a `number` type, but it is completely up to the user-defined generator phase and the targeted runtime as to whether it equates to a C++ `int`, or a Java `java.lang.Number`, or something else entirely. While this arrangement allows WOOL a very high level of flexibility, it comes at the cost of not having the most basic types that users expect from a programming language available.

One advantage of adding basic primitive types to WOOL is that we could then add a standard library of activities that act on these types. For example, we could add a standard activity that converts an integer into a string, or vice-versa, or common arithmetic activities. Or, going even further, we could include certain control-flow constructs that are currently difficult to express in WOOL. These standard activities would make WOOL much more usable "out of the box," and should be trivial to implement in almost any workflow runtime system.

The idea of adding standard types and activities has been driven by our experience in developing the image processing workflow described in Section 6.

In adding standard types, we want to achieve a balance between ease-of-use for workflow programmers and amount of support that WOOL requires from a targeted runtime. We are currently investigating how much flexibility WOOL would lose if we were to add basic integer, floating-point, boolean, and string types to the language, along with a complementary basic set of standard activities. It is our feeling that this fairly minimal set of fundamental types and activities would be sufficient for this purpose, and enable more complex workflows to be built much more quickly.

7.2 Transformations

We are investigating ways to have WOOL support special syntactic constructs that enable transformations on the final workflow graph. Transformations would use a set of

language primitives to help users build common, idiomatic graph structures. Additionally, semantic information available from these primitives may be used to normalize the resulting workflow graph. Transformations, perhaps combined with the standard types discussed in Section 7.1, could provide a very simple and flexible syntax for common control-flow idioms.

Map connections are an example of a transformation that might be useful in WOOL. In a map connection, a subgraph that implements a unit-to-unit filter (i.e. an activity or subworkflow that takes a unit input and produces a unit output) is rewired into an equivalent group-to-group filter. Map connections would be 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. Figure 4 shows how the map connection syntax alters the workflow.

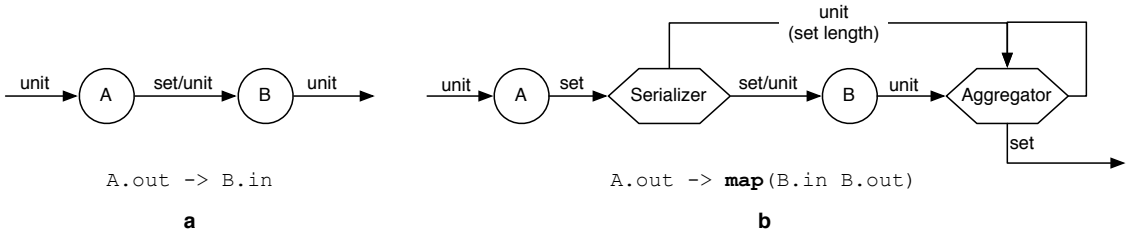


Figure 4. Example of a map connection *(a) A connection is established from activity A to activity B, where the output from A is a set, and the input to B is a unit. The connection is valid, and the set will be serialized as it is delivered to B. The output from B is a unit. (b) The same activities (A and B) are used, but the connection is a map. This causes the connection to be transformed with the addition of the Serializer and Aggregator activities. Note that the final output from the subgraph is a set.*

We envision that transformations of this kind would function as WOOL's approach to control-flow constructs. Rather than rely on special language extensions, simple syntax combined with graph transformations would allow for powerful and customizable control-flow expressions.

7.3 Optimizations

Graph optimizations are another interesting kind of graph transformation that we are currently investigating. In particular, it is possible to identify regions of the graph that can be executed in parallel. These arise in areas of the graph where activities are marked

with the `stateless` keyword and there are no loops. How and under what conditions to transform the graph to facilitate these parallel regions is a topic of continuing research.

7.4 Testing with Other Runtime Systems

WOOL is intended to be abstract, in the sense that it can be targeted to multiple different workflow runtimes while retaining the same data-flow semantics for a particular workflow. To date, however, we have only tested WOOL with our simple, single-threaded runtime system. To demonstrate that WOOL is effectively abstract, we would like to implement runtime generators capable of transforming WOOL workflows to other runtime systems. One candidate for a target is another abstract workflow language, such as AGWL [9]. We would also like to try a mainstream scientific workflow system, such as VisTrails [6] or Kepler [2].

In addition, we have begun work on a new distributed workflow system that moves data between activities using tuple spaces [1]. It uses a scheme, similar to the example runtime in Section 6, to encode activities based on Java objects and `HashMaps` for inputs and outputs. This workflow runtime system promises to be fast and easy to install and run either on a single processor, or in parallel on a multicore processor or a group of networked machines. We anticipate that it will also make an interesting target for WOOL.

7.5 User-defined Runtime Semantics

Activities in WOOL should allow modification of 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. Another example is an output that is connected to multiple inputs. Data written to that output could be broadcast to each input (as it is now) or distributed in a round-robin or randomized fashion. Providing workflow designers with choices for activity semantics might enable very concise and powerful specifications for complex workflows.

7.6 Other Enhancements

WOOL does not currently support workflow parameterization, but this feature is important for scientific of workflows. Scientists often must specify that a workflow is to run over a range of numbers or files, or uses constants on some of inputs, and so on. But, parameters are usually specified as concrete data, and WOOL is intended to be abstract and independent of particular data types. How to reconcile these goals is a topic of continuing

work. One idea is to introduce a language mechanism to specify that a port will be a parameter without specifying the exact data. This approach might facilitate some interesting optimizations, as well as enabling passing better information to the runtime.

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.

8 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.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management 2004*, pages 423–424, 2004.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for web services version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/, May 2003.

[4] Apple Inc. *Quartz Composer Programming Guide*, 2007.

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

[6] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Managing the evolution of dataflows with VisTrails. In *22nd International Conference on Data Engineering Workshops (ICDEW'06)*, pages 71–75, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[7] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience*, 18(10):1021–1037, 2006.

[8] E. Deelman and Y. Gil. Final report of NSF workshop on challenges of scientific workflows. http://vtcpc.isi.edu/wiki/images/b/bf/NSFWorkflow-Final.pdf, May 2006.

[9] T. Fahringer, S. Pllana, and A. Villazon. A-GWL: Abstract Grid Workflow Language. In *4th International Conference on Computational Science (ICCS 2004)*, Lecture Notes in Computer Science, pages 42–49. Springer Berlin / Heidelberg, June 2004.

[10] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with AGWL: an abstract grid workflow language. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 676–685, Washington, DC, USA, 2005. IEEE Computer Society.

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

[12] 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.

[13] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, 40(12):24–32, December 2007.

[14] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng. Overview of the Ptolemy project. Technical report, University of California Berkeley, 2003.

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

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

[17] S. Majithia, M. Shields, I. Taylor, and I. Wang. Triana: A graphical web service composition and execution toolkit. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, volume 0, pages 514–522, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[18] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, November 2004.

[19] T. J. Parr and R. W. Quong. Antlr: a predicated-ll(k) parser generator. *Software – Practice & Experience*, 25(7):789–810, 1995.

[20] J. Yu and R. Buyya. A novel architecture for realizing grid workflow using tuple spaces. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 119–128, Washington, DC, USA, 2004. IEEE Computer Society.