Domain-Specific Languages for Developing and Deploying Signature Discovery Workflows

Ferosh Jacob | CareerBuilder
Adam Wynne | Pacific Northwest National Laboratory
Yan Liu | Concordia University
Jeff Gray | University of Alabama

Domain-agnostic signature discovery supports scientific investigation across domains through algorithm reuse. A new software tool defines two simple domain-specific languages that automate processes that support the reuse of existing algorithms in different workflow scenarios. The tool is demonstrated with a signature discovery workflow composed of services that wrap original scripts running high-performance computing tasks.

A signature is a "unique or distinguishing measurement, pattern, or collection of data that detects, characterizes, or predicts a target phenomenon (object, action, or behavior) of interest."[1] Signatures are valuable in a wide range of application domains—such as medicine, network security, and explosives detection—for anticipating future events, diagnosing current conditions, and analyzing past events. However, current approaches have limitations on reusing existing algorithms, tools, and techniques across application domains and scientific disciplines, which requires expertise in programming language tools and the application domain.

At the Pacific Northwest National Laboratory (PNNL), we're developing the Signature Discovery Initiative (SDI), a generalized signature development methodology that's applicable to any signature discovery problem.[1] Under the SDI, we've been working closely with scientists who have developed or applied algorithms using a wide range of programming languages and tools. We've found that signature discovery facilitates scientific investigation across multiple disciplines through the reuse of existing algorithms, which can be written in any programming language for various hardware architectures (including desktops, commodity clusters, and specialized parallel hardware platforms). Reusing these algorithms requires a common architecture that can integrate analytical software components created by scientists from different disciplines.

A common architecture to support reuse of scientific workflow algorithms requires support in two essential areas:

- a service-oriented software framework that lets scientists in specific domains register and share their algorithms, so that they can offer those algorithms as reusable service components; and
- the creation of new signature discovery workflows in other domains using the created service components.

This further raises an engineering challenge: Web services must be generated for heterogeneous algorithms, so that they become service components and can be registered and composed into a scientific workflow environment.

Here, we introduce two domain-specific languages (DSLs):[2]

- Service Description Language (SDL) describes key elements of a signature discovery algorithm and generates the service code.

Related Work in Automating Web Service and Workflow Creation

Several related efforts focus on code generation for Web services and workflows. We mention a few here, along with a discussion of related work in workflow automation and application composing.

Workflow engines, such as Java Business Process Management (JBPM),[1] also provide GUIs for designing and deploying workflows. Other related works include languages[2,3] that are designed for composing computation-intensive applications. There are also many tools available for creating applications by composing Web services from different vendors.[4,5] Most of these tools assume that the Web services are available.

In a domain-independent workflow like Taverna, Web services can be of different types; Web Services Description Language (WSDL) services are just one example. To keep the uniformity, every time a WSDL processor is created, users have to add transformations to parse the XML input and output. Our framework handles these details automatically through service and workflow definition languages. Moreover, because the output is generated as a Taverna workflow file, it can be viewed, edited, and executed in Taverna's full-fledged workflow development environment. As Ian Gorton and Yan Liu demonstrated,[6] reusing open source component-based software frameworks significantly improved the development of domain-specific scientific workflows. Our framework configures the workflow definition file that declares how to compose services wrappers created by our framework. This approach reduces the complexities of encapsulating domain-specific computing procedures to general-purpose workflows.

The work described in this article is focused on helping scientists in developing signature discovery workflows. The code generation for Web services and workflows is separated from the Domain Specific Language (DSL) design to facilitate alternate implementations for the Web services and workflows. Our tool's role in support of Signature Discovery Initiative (SDI) workflows is similar to tools in other scientific collaboration environments, such as Rappture (for rapid application infrastructure) with nanoHub.org,[7] where Rappture generates a GUI based on the description of inputs and outputs of a simulator that can be deployed on a hub platform.

References

1. M. Cumberlidge, *Business Process Management with JBoss jBPM*, Packt Publishing, 2007.
2. I.J. Taylor et al., *Workflows for e-Science: Scientific Workflows for Grids*, Springer-Verlag, 2006.
3. M. Wilde et al., "Swift: A Language for Distributed Parallel Scripting," *Parallel Computing*, vol. 37, no. 9, 2011, pp. 633–652.
4. E.M. Maximilien et al., "A Domain-Specific Language for Web APIs and Services Mashups," *Proc. Int'l Conf. Service-Oriented Computing*, 2007, pp. 13–26.
5. M. Pruett, *Yahoo! Pipes*, O'Reilly, 1st ed., 2007.
6. I. Gorton et al., "Build Less Code Deliver More Science: An Experience Report on Composing Scientific Environments Using Component-Based and Commodity Software Platforms," *Proc. Int'l ACM Sigsoft Symp. Component-Based Software Eng.*, 2013, pp. 159–168.
7. M. McLennan and R. Kennell, "HUBzero: A Platform for Dissemination and Collaboration in Computational Science and Engineering," *Computing in Science & Eng.*, vol. 12, no. 2, 2010, pp. 48–52.

- Workflow Description Language (WDL) specifies the services pipeline and generates deployable artifacts for a workflow management system.

Our contribution is the new capability that emerges from combing SDL and WDL in a service-oriented approach that enables the interoperability and reuse of existing algorithms. Here, we demonstrate our approach with a software tool that removes the complexity associated with the engineering requirements in the signature discovery process. We also describe two specific engineering challenges and how our solution addresses them using an example scenario focused on the Basic Local Alignment Search Tool (BLAST) execution workflow (see http://blast.ncbi.nlm.nih.gov/Blast.cgi for details on BLAST). In addition, we include several SDL and WDL examples to show usage of the tool.

Motivating Scenario: BLAST Execution Workflow

The primary goal of our signature discovery research is to develop a methodology and an associated set of software tools, called the *Analytic Framework*, that allow scientists and analysts to apply algorithms and techniques across multiple domains. For example, after we find a technique for developing and detecting a signature in one domain, we aim to abstract the specific techniques used and investigate whether they can be applied to additional datasets and problems outside of their normal usage. Because these software tools are being directed outside their usual context, the users might be unfamiliar with them. We thus need to make every effort to make what are sometimes highly technical tools easy to use. We selected BLAST for this motivating scenario because it illustrates this process well.

In systems biology, scientists often use BLAST to find regions of local similarity between biological sequences. This has a wide range of practical uses, including addressing the homology problem, wherein researchers try to determine the similarity between genes—which are composed of sequences of nucleotides—in order to imply common evolutionary origin. BLAST's abstracted purpose is to solve an inexact string-matching problem, which has been applied successfully to the computer security domain to develop signatures of "families" of related software code by representing the code as sequences of subroutine calls or processor operations.[3] SDL/WDL offers an easier way for scientists to construct services and workflows that include sequence-based signatures like those used in BLAST. One drawback of our approach is the additional layer of tooling that must be maintained and updated as the underlying platform evolves.

Practically speaking, BLAST is a long-running job with input and output. The workflow is usually executed in three steps:

1. Submit a BLAST job in a cluster using the Simple Linux Utility of Resource Management (SLURM) job scheduler (see https://computing.llnl.gov/linux/slurm).
2. Check the job's status.
3. Download the output files upon job completion.

Note that a UNIX command utility (such as `sh`) and/or a script file (such as SLURM) identifies each step. We'll now use this BLAST workflow identified from the SDI to illustrate three engineering challenges when reusing signature discovery algorithms from different domains.

Accidental Complexity of Generating Service Wrappers

In keeping with our common architecture's goal, we aim to make existing BLAST programs available so that scientists can share them. To address this objective, we provide Web service wrappers for every executable binary. We adopt the legacy wrapper pattern[4] to encapsulate existing algorithms, while providing a standard interface so that they can be orchestrated with other services to create reusable workflows. Informally, we refer to these services as *wrappers*.

Creating a wrapper for an existing script or executable typically follows a common set of steps:

- identify the input files and output files in the program;
- retrieve the input files from the data management system;
- execute the program; and
- upload the output files to the document management system as part of the existing signature discovery software framework.

We've observed that the process for converting a script often results in significant extra code and manual effort. For example, `checkJob` is a simple BLAST service for checking the job status, with a single input (`JobId`) and output (`status`). However, it requires a service wrapper with 121 lines of Java code (in four Java classes) and 35 lines of XML code (in two files). This extra code might increase the complexity and development overhead significantly from a scientist's perspective.

Our goal is to raise the level of abstraction from these general-purpose languages to the signatures domain, such that scientists describe only the specifications for the executable binary—in this case, signature algorithms—that's to be shared as a service component using a DSL. The tools associated with our DSLs will generate and execute the code required for creating a wrapper.

Coupling between Workflows and Services

Once the executable binaries are accessible as a service component, scientists still must orchestrate the services to define the signature discovery workflow. To make any Web service available in a domain-independent workflow engine such as Taverna (www.taverna.org.uk), a user must:

- manually add each operation as a service to the workflow designer, and
- provide XML generators and parsers.

In a Web service, the input and output are expressed in XML. Hence, an XML generator is required before passing any workflow parameters to the service. Similarly, an XML parser is required to process the service output after executing the service. Therefore, these two processes (that is, generating a service wrapper and registering a service for orchestration) are coupled. Figure 1 shows this BLAST execution workflow in the Taverna workbench.

As Figure 1 shows, the workflow consists of three services: submitting a job, checking the job's status, and obtaining the result. In the figure, boxes in light blue represent workflow inputs and outputs. Other boxes correspond to processors, where processors performing similar functions are identified by the same color (for example, purple boxes represent processors that handle XML parsing). Within the `checkJob` service, `jobID` is wrapped inside an XML descriptor by `jobStatusIn` before passing to the actual service processor `jobStatus`. After executing the service processor, the output XML is passed to the `jobStatusOut` processor, which parses the XML and passes the status to workflow output port `status`.

In case a service output is an object, as in the case of `submitBlast`, multiple XML parser processors are applied (such as `submitBlastOut` and `submitBlastOut_submitBlastResults`). Hence, the type and number of XML parsers and generators required to make a Web service available in a workflow orchestration depend on the structure—that is, the type and number of the input and output—of the Web service itself. Given this, scientists need comprehensive knowledge to use these services in a workflow because they must know the input and output format of every service. The scientists would also need to provide correct parsers and generators for each service. This is a challenging task for most scientists, and is one full of accidental complexities.

Lack of End-User Environment Support

Because many scientists are unfamiliar with service-oriented software technology, they're forced to seek the help of software developers to make Web services available in a workflow environment. On the other hand, software developers find it difficult to work with scientific scripts and tools, which aren't usually their area of expertise. This technology barrier can degrade the benefits of sharing signature discovery algorithms, because any changes or bug fixes to an algorithm require both a dedicated software developer and a scientist to navigate through the engineering process.

Solution Approach Using DSLs

We introduce a new approach to simplify the integration of signature discovery algorithms into a common architecture. In this approach, two sets of DSLs[2] are defined that contain

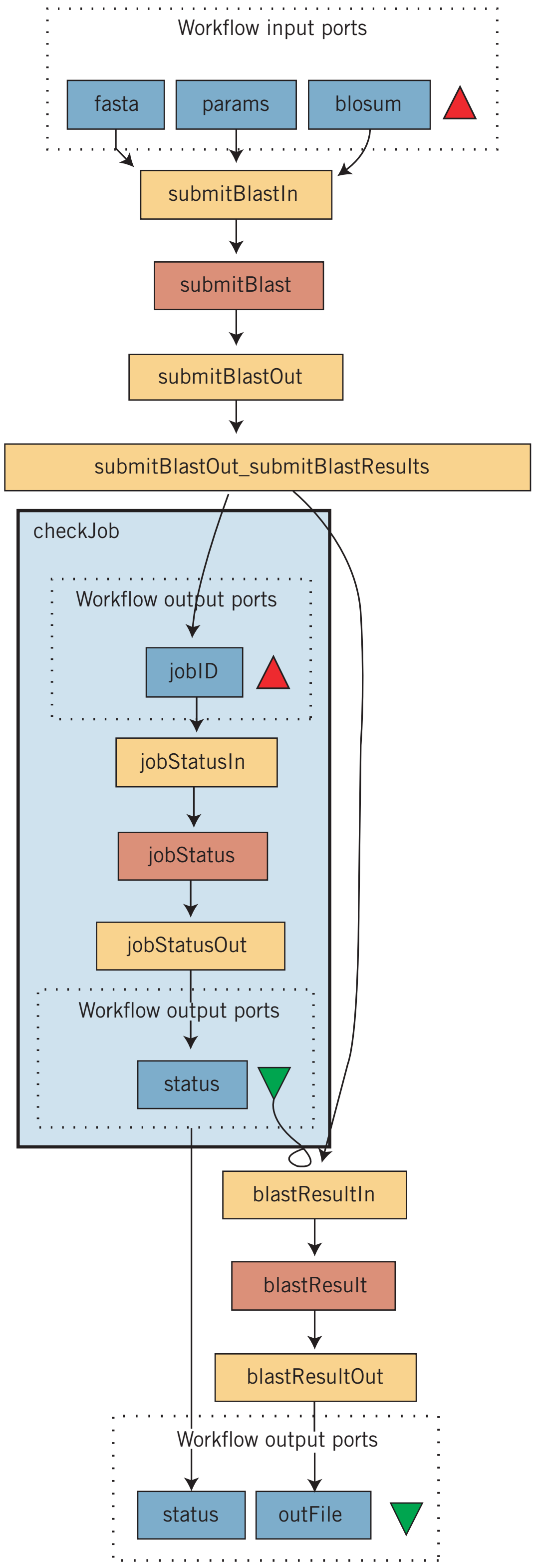

Figure 1. An example Basic Local Alignment Search Tool (BLAST) workflow. The two key processes—generating a service wrapper and registering a service for orchestration—are coupled.

```
//Remote machine configuration
define ssh_oly as "sdi" at "olympus.pnl.gov"
  with-key "/home/jaco181/.ssh/id_rsa"

/*
 *  Creating a service to submit a blastJob
 */
service submitBlast {
  use ssh_oly;
  cmd "sh runJob.sh";
  resource "jobScript.sh", "runJob.sh";
  in doc blossum, params, fasta;
  out jobID, outDir;
  /*
   *Inside run.sh
   *echo "jobID=\$JOBID" > .properties
   *echo "outDir=\$deployDir" >> .properties
   */
}
/*
 *  Creating a service to upload a remote file
 */
service blastResult {
  use ssh_oly;
  cmd "cp $outDir$/test_all_v_all_m8.out outFile";
  in outDir;
  out doc outFile;
}
/*
 *  Creating a service to check status of a job
 */
service jobStatus {
  use ssh_oly;
  cmd "sh checkStatus.sh";
  resource "checkStatus.sh";
  in jobID;
  out status;
}
```

(a)

```
use "SigQuality.sdl"
/*
 *  Main workflow
 */
workflow BlastSearch (in blosum, in params,
  in fasta, out outFile, out status){

  //Passing values to service
  blosum->submitBlast.blossum
  params -> submitBlast.params
  fasta -> submitBlast.fasta

  //Calling sub-workflow
  call checkJob
    till status="Done"
    with  submitBlast.jobID,  status

  //Configuring execution
  submitBlast.outDir ->blastResult.outDir after
      ↪checkJob

  //Passing to workflow output
  blastResult.outFile->outFile

}
/*
 *Sub-workflow checkJob
 */
workflow checkJob (in wf_jobID,  out wf_status){
  wf_jobID->jobStatus.jobID
  jobStatus.status->wf_status

}
```

(b)

Figure 2. BLAST execution workflow using Service Description Language (SDL) and Workflow Description Language (WDL). (a) The code shown here can create three Web services that act as wrappers for remote executables that will upload and download files before and after execution, respectively. (b) It also creates an executable workflow in the workflow engine (here, Taverna).

- an SDL that end users can use to specify the user credentials, executable path, script file, and the input and output of any script; and
- a WDL that specifies the interactions of these services and takes input from one or more service description files.

The WDL's syntax is mapped to the APIs of a workflow environment. Thus, we can generate the glue code to orchestrate the service components generated from SDL.

For example, Figure 2 shows SDL and WDL files for a BLAST execution workflow. In Figure 2a, three services—`submitBlast`, `jobStatus`, and `blastResult`—are defined, along with their input, resources (script files), and connection details. In Figure 2b, a main workflow is defined with input and output (we explain the code used in these two files later).

The few lines of code shown in Figure 2 are powerful enough to

- create three Web services, where each service is a wrapper for a remote executable that will upload all the required files to the remote server before execution and download the files after execution; and
- create an executable workflow in a workflow engine such as Taverna (Figure 1).

Figure 3 shows the overall approach to generating wrappers and workflows. The input to the process are scripts (or commands with template variables) and the SDL and WDL files. Using these input files, Web service wrappers and a workflow file deployable to a workflow engine (such as Taverna) are generated. Each Web service wrapper

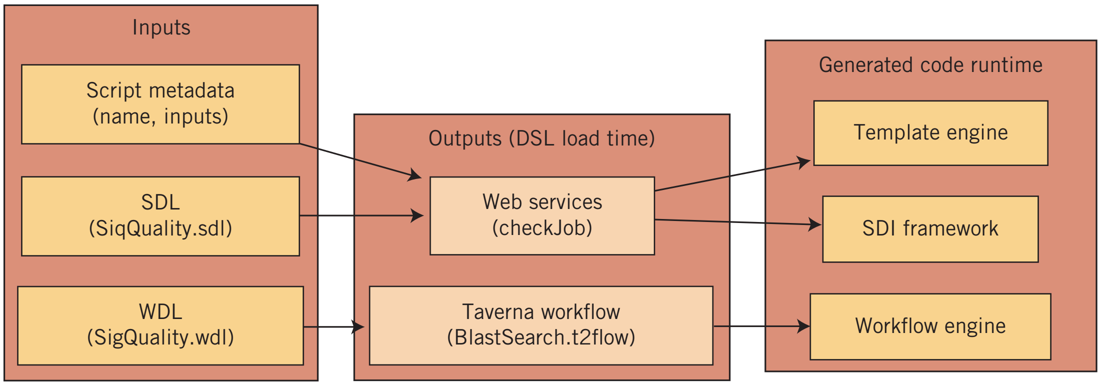

Figure 3. A block diagram showing the framework's implementation. The process inputs are scripts and the SDL and WDL files.

is created from scripts and the associated SDL files. The workflow file is created using both SDL and WDL files. Code generation occurs in two stages:

- *Web application creation*. The tool creates Web services from the SDL files that describe a script's key elements.
- *Workflow creation*. The SDL file from the first stage defines a service. It's then passed together with the WDL file to create the workflow constructs. These constructs are the basic elements for the Taverna engine to create a workflow as defined in WDL.

During the loading of an SDL file, a Web application with Web service wrappers corresponding to each `service` in the SDL is created. Similarly, a "t2flow" (Taverna workflow executable) file is created during the load time of WDL. When the t2flow is executed, the wrapped script(s) and command(s) are executed in the remote host through an SSH session with the help of existing signature discovery libraries, which are part of SDI. These libraries are responsible for making the input files available before execution and uploading the output files to a dedicated data management system after execution. We use a template engine to access runtime values of variables inside services.

SDL Definition and Wrapper Generation

An SDL file has a list of services, and each service has a connection parameter (SSH details), an execution command parameter (command to execute), resources (additional scripts required to execute the command), and a set of inputs and outputs required for the service execution. If the connection parameter isn't specified, a service will be executed in the server in which the application is deployed.

We now describe the code-generation details of SDL input and output.

Input and output strings. *String* is the default parameter type in SDL. Any parameter defined without a modifier other than "in" and "out" are treated as string parameters. The "in" and the "out" modifiers define the directions of the parameters. Each input string parameter is treated as a template variable and is applied to the scripts and commands; any occurrence of the variable will be replaced by its runtime value before execution. If there's an output string variable, code is generated to read a `.properties` file after execution.

List of documents or string. Code is generated to apply the document property or string property for all elements in the list. As shown in Figure 4a, while using the template variables and list, users must be aware that a single variable can be substituted with many values.

Wrapper generation. We generate a Web service wrapper for each SDL file using the file name. For every service, two artifacts are generated: the interface class and the corresponding implementation class. In addition, the framework generates a helper class, called `SSHHelper`, that automates the connection with remote computers. For example, Figure 2 shows the code generated for the `submitBlast` service, including the interface class and its implementation.

SDL Execution

We've created service wrappers for each step. Figure 2a shows the job submission service. A BLAST job is submitted using two script files: "jobScript.sh" (a SLURM file) and "runJob.sh" (a BASH file). The script file "runJob.sh" executes the SLURM

```
service classifier_Training {
  use ssh_exe;
  cmd "R CMD BATCH training.r training.out";
  resource "training.r";
  in doc trainXFile , trainYFile;
  in algorithm;
  out doc modelFile;
}
```

(a)

```
1 service aggregate{
2   use ssh_exe;
3   cmd "cat $inputs;  separator=\" \"  $ >
        ↪aggregatedFile" ;
4   in list doc inputs;
5   out doc aggregatedFile;
6 }
```

(b)

Figure 4. More SDL examples. (a) Wrappers for R scripts using SDL. (b) A utility wrapper service that can vertically merge a list of files.

file and writes `jobID` and `outDir` to the `.properties` file. In Figure 2a, `submitBlast` has two outputs, the execution directory (`outDir`) and the job identifier (`jobID`). Both outputs are declared as the default type; hence, the framework generates code for the string outputs. The generated code downloads and reads their values from a `.properties` file. Other services (`blastResult` and `checkJob`) are command SDL service wrappers (no script files) and are not shown. Service `checkJob` checks the status of a given `jobID` and returns status as "Running," "Pending," or "Done." Service `blastResult` downloads the files from a given directory.

At runtime, values and fields inside the script files must be exposed as the input or output of service wrappers. This is achieved by treating the scripts as templates and their runtime values as template variables. The template variables are enclosed inside the "$" character. Before executing any script files in the server, the scripts are passed through a template engine. The template engine substitutes the scripts with runtime values. Template substitution isn't necessary for the files, because their runtime names are the same as their load time names. As an example, in Figure 2 (line 25), the `blastResult` service is defined with a template variable `outDir`. When the `blastResult` is executed, the `$outDir$` in the command (line 25) will be replaced by the runtime value of the variable `outDir`.

We use the Antlr StringTemplate (www.stringtemplate.org) for the template implementation. Hence, we can use many advanced features of a template engine. As an example, to implement a Web service that can aggregate all of the input files, we defined an SDL file (Figure 4b) with an input as a "list'" of documents named `inputs` (line 3). Because the variable type is a list, the template is applied for all the values. Using the StringTemplate "separator" keyword, we separate the individual values with a space. In the case of a text output, the code is generated to read the `.properties` file and return the specified property value. Hence, a user has to make sure that the output text values are written to the `.properties` if they want the service to return the value.

As Figure 2a shows, after submitting a BLAST job, the job identifier and the output directory are written to the `.properties` file (commented lines 16 and 17). If there are multiple outputs, a new object return type is created with fields as the outputs specified and returned.

WDL Definition and Workflow Generation

A WDL file creates a Taverna workflow based on the descriptions specified by the user. A WDL workflow involves communication and interactions among various service wrappers and between service wrappers and other workflows. A WDL file can have many workflows, but the topmost workflow is considered the main workflow with all other workflows treated as subworkflows. Code is generated for a subworkflow only if it's called inside the main workflow.

Workflows have declarations of elements and connections. Figure 5 shows the Xtext (www.eclipse.org/Xtext) grammar for WDL. Elements can be of any three types: subworkflow, services, and strings. Strings must be initialized, along with their declaration. Connections also have three types: workflow input to service port, service port to service port, and service port to workflow output or subworkflow calls.

A subworkflow call in WDL introduces loops and abstractions (function calls) to another WDL. It can connect a subworkflow with the connection ports. In that case, the subworkflows, services, and strings can be used without explicit declaration; their names will be used as their identifiers. The main workflow uses a "call-till-with" structure for communicating with subworkflows. As lines 13–16 in Figure 5 show, using this structure, the main workflow can iteratively "call" a subworkflow "till" it meets a condition "with" main workflow ports.

```
grammar gov.pnl.sdi.WDL with org.eclipse.xtext.common.Terminals
generate wDL "http://www.pnl.gov/sdi/WDL"

WorkflowModel:
  'use' servicefile=STRING workflows+=Workflow+;

Workflow:
  'workflow' name=ID '(' parameters=Parameters? ')' '{'
        (definitions+=Definition*)
        (stringConstants+=StringConstant*)
        (portlinks+=PortLink|serviceLinks+=ServiceLink|workflowCalls+=WorkflowCall)+'}';

WorkflowCall:
  'call' workflowID=ID
   ('till' criterion=Criterion)?
  'with' argument=Port (',' moreArguments+=Port)*;

Criterion:
  port=ID op=OPERATOR value=STRING ;

OPERATOR:
  '='|'<'|'>';

StringConstant:
  'String'  stringAssignments=StringAssignments;

StringAssignments:
  assignment=StringAssignment(',' moreAssignments+=StringAssignment)*;

StringAssignment:
  name=ID'=' value=STRING;

Definition:
  'workflow'|'service') name=ID services=Services;

Services:
  service=ID (','  moreServices+=ID)*;

ServiceLink:
  service1=ID'|' service2=ID;

PortLink:
  (port1=Port|text=STRING)'->' port2=Port;

Port:
  serviceName=ID('.'portName=ID)? ('after' afterServiceName=ID)? ;

Parameters:
  parameter=Parameter (',' moreParameters+=Parameter)*;

Parameter:
  type=('in' |'out') variable=ID;
```

Figure 5. Xtext grammar for WDL. The top-level components are the SDL file locations and the workflows using the services defined in the SDL.

Hence, the "call-till-with" structure replaces function calls and loops in WDL.

We use a WDL specification file to define workflows. A WDL file can contain many workflows. The first workflow in the file is the main workflow. Accordingly, a Taverna workflow file will be generated with the same name as the main workflow, with an added t2flow extension. The other workflows used inside the main workflow are subworkflows. The input for the services in a SDL specification file can be defined in two ways:

- *Direct service specification*—a service name followed by the operator "." and the input variable name specified in the SDL file (Figure 2, lines 9–11).
- *Indirect service specification*—an identifier defined as a service, followed by operator "." and the variable name (Figure 6, line 13).

The indirect service specification is required when there's more than one occurrence of the same service. The "->" operator can be used for connecting any of the three cases, such as connecting

```
use "SigAnalysis.sdl"
workflow algo_classify (in algo, in trainX, in trainY,
    in testX, in testY, out outFile){

  trainX ->classifier_Training.trainXFile
  trainY ->classifier_Training.trainYFile

  algo->classifier_Training.algorithm
  algo->classifier_Testing.algorithm

  classifier_Training.modelFile -> classifier_Testing.modelFile


  testX->classifier_Testing.testXFile
  testY->classifier_Testing.testYFile

  classifier_Testing.outFile ->outFile
}
```

Figure 6. Landscape classification using WDL. This workflow uses two services: `classifier_training` and `classifier_testing`. The output of `classifier_training` links to the input of `classifier_testing`.

a workflow input to a service input, connecting a service output to another service input, or connecting a service output to a workflow output. We introduced the call-till-with construct to implement loops and abstraction in the WDL. Using the structure, subworkflows can be included into the main workflow. Using the optional "till" construct lets us specify the termination condition for a subworkflow. The subworkflow will be executed only once, as shown in the landscape classification example in Figure 7.

The support for a conditional expression is restricted by the underlying workflow engine. Hence, for the expression, the first operand must be a subworkflow port, and the second operand must be a string. WDL supports three logical operators: "=," "<," and ">." Using the "with" construct, the subworkflow's input and output are specified similar to a function call's arguments. If the arguments are defined as an "out" type in a subworkflow, the subworkflow writes to the port after execution. Otherwise, it reads from the port before the execution.

WDL Execution

Figure 2b shows the WDL file for the BLAST workflow. The WDL specification has two workflows: `BlastSearch`, the main workflow, and `checkJob`, the subworkflow. The main workflow `BlastSearch` has the following steps:

- Submit a BLAST job after passing inputs to the `submitBlast` service (lines 9–11).
- Pass the `jobID` to the subworkflow (line 16) and the output directory to the `blastResult` service and wait for the subworkflow to continue execution until it meets a criterion on finishing the subworkflow (lines 14–16).
- Download the output file using service `blastResult` and pass it to the workflow output (line 22).

In Figure 2b, the `checkJob` workflow is defined (lines 28–32). The workflow takes one input (`wf_jobID`) and returns one output (`wf_status`). The workflow input `jobID` is passed to the service, `jobStatus` (line 29). The `jobStatus` service is defined using SDL to find the status of the specified job. The workflow output `status` is fetched from the service output `status` (line 30). The order of execution is determined by the Taverna engine, which executes all the services whose outputs are available. In some cases, scientists might want to control the order, because it might not be obvious to the Taverna engine.

As an example, for the BLAST execution, the `blastResult` service that downloads the output files must wait for the BLAST job to complete. To allow a service to be executed only after the specified workflow or service, the "after" keyword is provided and can be added to any service invocation

```
1  workflow Classifier (in trainX, in trainY, in testX,
2     in testY, out finalOut) {
3
4   workflow algo_classify lda_class, knn_class, svm_class
5
6   service aggregate agg
7
8   call lda_class
9   with "lda", trainX, trainY, testX, testY, agg.inputs
10
11  call knn_class
12  with "knn", trainX, trainY, testX, testY, agg.inputs
13
14  call svm_class
15  with "svmRadial", trainX, trainY, testX, testY, agg.inputs
16
17
18  agg.finalOut ->accuracy.inputFile
19
20  accuracy.outputFilePrefix -> finalOut
21
22 }
```

Figure 7. Classification accuracy using Workflow Description Language (WDL). The output of each workflow `algo_classify` is written to the input of `agg` (lines 9, 12, and 15). Hence, `agg` (`finalOut`) has the aggregated results of all three classifications.

(Figure 2b, line 19). More than one service can be specified in the "after" clause. The language design and the associated software tools are available as an Eclipse plugin for PNNL scientists working in the SDI project.

We define the syntax of service descriptions and workflows using Eclipse Xtext. The syntax is defined according to Taverna workflow elements and processes. The code generation uses Apache CXF APIs for creating Web service wrappers for script files. (For more on the CFX open source services framework, see http://cxf.apache.org).

SDL Service Examples

Figure 4 shows two examples of the services we created using SDL for signature discovery workflows:

- Figure 4a shows wrappers for R (www.r-project.org) scripts using SDL.
- Figure 4b shows a utility wrapper service that can perform a vertical merge for a list of files.

Using the SDL service shown in Figure 4a, a Web service wrapper is created that executes an R script (`training.r`) with input files (`trainXFile` and `trainYFile`). The R scripts can be executed using the command "`R CMD BATCH scriptfile`" as shown in Figure 4a (line 3). Similarly, the UNIX utility cat command is used in the aggregate service to `aggregate` files (line 3 in Figure 4b).

Table 1 shows an overview of 10 services carefully selected to highlight the core code-generation features of the SDL parser. In the table, "lines of code" indicate the additional LOC generated to include the service. This doesn't include abstract classes or class definitions if the class already exists. The SDL code generation is comprised of the following steps:

- adding a method in a Java RMI interface;
- adding a method in a Java Service Endpoint Interface (SEI) for creating an XML-based Web service;
- implementing a method in a Java class through a helper class;
- creating the helper class for the service; and
- creating a new Java bean, if there's more than one output.

The generated code affecting multiple Java classes (such as five, if there's more than one output) for

Table 1. An overview of SDL code generation.

No.	Service	Utilities/Script	[Inputs(type)] [Outputs(type)]	LOC*	Total LOC (files)
1	`echoString`	`Echo`	[0][1 (doc)]	10 + 13 + 1 + 6	30(4)
2	`echoFile`	`Echo`	[1 (String)] [1 (doc)]	10 + 14 + 1 + 6	31(4)
3	`aggregate`	`Cat`	[1(List doc)] [1 (doc)]	10 + 20 + 1 + 7	38(4)
4	`classifier_Training`	`R`	[2 (doc), 1 (String)] [1 (doc)]	11 + 24 + 2 + 8	45(4)
5	`classifier_Testing`	`R`	[3 (doc), 1 (String)] [1 (doc)]	12 + 29 + 2 + 8	51(4)
6	`accuracy`	`R`	[1 (doc)] [1 (doc)]	11 + 19 + 1 + 6	37(4)
7	`submitBlast`	`SLURM, sh`	[3 (doc)] [2 (String)]	17 + 27 + 2 + 8 + 18	72(5)
8	`jobStatus`	`SLURM, sh`	[1 (String)] [1 (String)]	10 + 14 + 1 + 6	31(4)
9	`blastResult`	`Cp`	[1 (String)] [1 (doc)]	10 + 14 + 1 + 6	31(4)
10	`mafft`	`Mafft`	[1 (doc)] [1 (doc)]	10 + 18 + 1 + 6	35(4)

*LOC stands for lines of code.

each service wrapper is shown in the table's LOC column.

A WDL Case Study: Landscape Classification

We now demonstrate how our approach can be used to define and deploy signature workflow through a landscape classification example. In the landscape classification workflow, the goal is to compare accuracies of three landscape classification algorithms. Each landscape classification algorithm is developed as R scripts and involves four stages:

- *Training stage.* Image data, along with their actual landscape classification values, are given as input. A model (a function approximation, generalized from the input training patterns) is created, and the model is returned as the output.
- *Testing stage.* The image data and the estimated model from the training stage are passed as inputs. The estimated classification type is returned as output.
- *Aggregate stage.* We perform the first two stages for three different algorithms: Linear Discriminant Analysis (LDA), K-Nearest Neighbor (KNN), and Support Vector Machine (SVM). The results of the algorithms are merged to a single output file. This can be implemented using a simple cat command.
- *Accuracy stage.* Based on the estimated and actual values, another R script is also available that can calculate a classification algorithm's accuracy.

Figure 4 shows SDL service definitions for the training and aggregate stages. Similarly, we can define service wrappers for the testing and accuracy stages.

The next step is to define the workflow. Figure 6 shows a workflow `algo_classify` that imports the service definitions and connects the training and testing stages through their inputs. The workflow inputs are passed to `classifier_Training` (lines 5–8) and `classifier_Testing` (lines 9, 14, and 15). The output of `classifier_Training` is passed to `classifier_Testing` (line 11), and output of `classifier_Testing` is passed to the workflow output `outFile` (line 17).

The workflow shown in Figure 6 can return the classification results for a set of inputs and an algorithm. In Figure 7, a workflow `Classifier` is written with the same inputs as `Classifier` excluding the `algo` input. Three workflows (`lda_class`, `knn_class`, and `svm_class`) are defined as workflow type `algo_classify` (line 4). The `Classifier` workflow is called for values "`lda`," "`knn`," and " `svmRadial`" (lines 8–9, 11–12, and 14–15). A service of type aggregate `agg` is also defined (line 6).

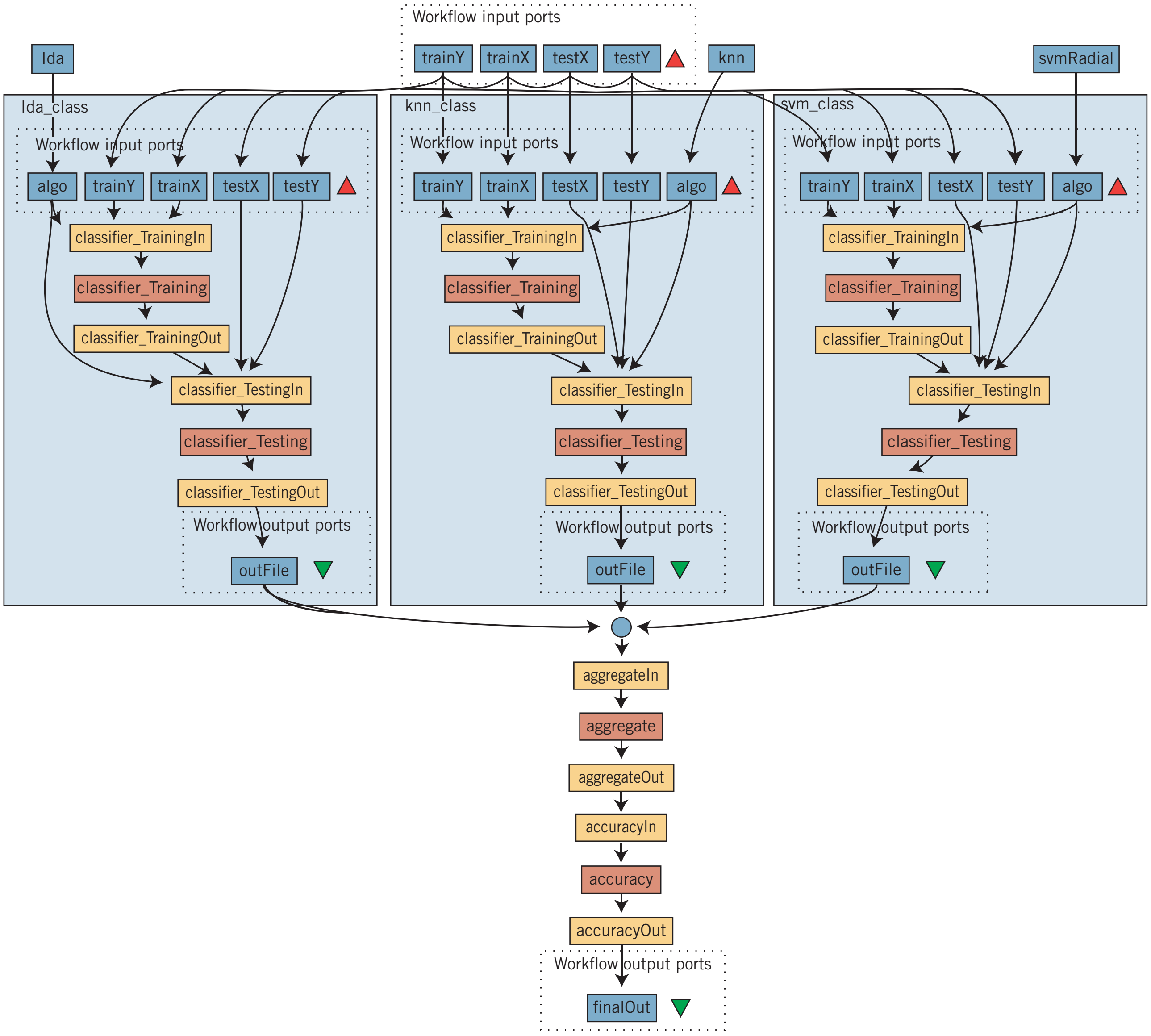

Figure 8. Taverna workflow for classification accuracy generated by WDL. The workflow has 50 processors and a similar number of connections, making it difficult to maintain using a manual approach.

The output of each workflow `algo_classify` is written to the input of `agg` (lines 9, 12, and 15). Hence, the output of `agg` (`finalOut`) has the aggregated results of all three classifications. This is given to the input of the accuracy `inputFile` (line 18), and the output of the service is given to workflow output (line 20). Figure 8 shows the final output workflow executable.

As Figure 8 shows, it would be challenging to maintain a workflow in this form using a manual approach. This workflow has 50 processors and a similar number of connections. Using our approach, scientists don't have to deal with the engineering processes to make executables globally available and accessible in a mature workflow engine like Taverna—and our approach offers all the advantages and features of executing the workflow in the Taverna workbench.

Modern scientific computing in general, and signature discovery in particular, require the integration of multiple pieces of code into reusable programs and workflows. We've taken a modern, service-oriented approach to enabling the interoperability and reuse of these codes. Our approach is extending the usefulness and value of scientific models and analytics that have been developed during the life of our research, and we hope it will enable their use in contexts in which they weren't originally envisioned.

Our service-oriented integration framework enhances the software capabilities in our signature discovery efforts by reducing the engineering complexity. Our framework can be extended to general workflows and is currently being integrated with other projects beyond signature discovery. This success might be hampered, however, by the ability to integrate new codes as quickly as they are needed. Therefore, we see tools and languages such as SDL and WDL being integral to the expansion of our service-oriented analytics platform, because they have the potential to ease the burden of integrating code into the framework, while also ensuring that it's done using standard design patterns.

As such, in our future work, we plan to evaluate the usefulness of these tools by scientists as well as engineers inside and outside the signature discovery area. For example, our framework is now being applied to the smart grid domain via PNNL's Future Power Grid Initiative (http://gridoptics.pnnl.gov) to augment the GridOPTICS[5] data integration framework with comprehensive model integration capabilities and service-oriented data sharing. This domain also needs to integrate a wide range of tools quickly into an enterprise-strength integration platform.

References

1. N. Baker et al., "Research Towards a Systematic Signature Discovery Process," *Proc. Int'l Conf. Intelligence and Security Informatics*, 2013, pp. 301–308.
2. M. Mernik, J. Heering, and A.M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, 2005, pp. 316–344.
3. C.S. Oehmen, E.S. Peterson, and J.R. Teuton, "Evolutionary Drift Models for Moving Target Defense," *Proc. 8th Annual Cyber Security and Information Intelligence Research Workshop*, 2012, article no. 37.
4. T. Erl, *SOA Design Patterns*, Prentice Hall, 2009.
5. I. Gorton et al., "GridOPTICS A Novel Software Framework for Integrating Power Grid Data Storage, Management and Analysis," *Proc. Int'l Conf. on System Sciences* (HICSS), 2013, pp. 2167–2176.

Ferosh Jacob is a software engineer at CareerBuilder. His research interests include software modeling, high-performance computing, and machine learning. Ferosh has a PhD in computer science from the University of Alabama. Contact him at ferosh.jacob@careerbuilder.com.

Adam Wynne is a senior research engineer in the National Security Directorate at the Pacific Northwest National Laboratory. His research interests include software architecture, middleware, and model-driven engineering. Adam has an MS in computer science from Western Washington University. Contact him at adam.wynne@pnnl.gov.

Yan Liu is an associate professor in the Faculty of Engineering and Computer Science, Concordia University, Canada, and was a senior scientist at Pacific Northwest National Laboratory from 2009 to 2012. Her research interests include cloud computing, distributed systems, software architecture, and model-driven development. Liu has a PhD in computer science from the University of Sydney. Contact her at yan.liu@concordia.ca.

Jeff Gray is an associate professor in the Department of Computer Science at the University of Alabama. His research interests include software engineering, model-driven engineering, software maintenance, and computer science education. Gray has a PhD in computer science from Vanderbilt University. Contact him at gray@cs.ua.edu.