To appear in the Journal of Experimental and Theoretical Artificial Intelligence, 2010.

A Semantic Framework for Automatic Generation of Computational Workflows Using Distributed Data and Component Catalogs

Yolanda Gil[1], Pedro A. González-Calero[2], Jihie Kim[1], Joshua Moody[1], and Varun Ratnakar[1]

[1] Information Sciences Institute, University of Southern California
4676 Admiralty Way, Marina del Rey CA 90292, United States
{gil, jihie, moody, varunr}@isi.edu

[2] Facultad de Informática, Universidad Complutense de Madrid
28040 Madrid, Spain
pedro@sip.ucm.es

December 15, 2009

Abstract

Computational workflows are a powerful paradigm to represent and manage complex applications, particularly in large-scale distributed scientific data analysis. Workflows represent application components that result in individual computations as well as their interdependencies in terms of data flow. Workflow systems use these representations to manage various aspects of workflow creation and execution for users, such as the automatic assignment of execution resources. This paper describes an approach to automating a new aspect of the process: the selection of application components and data sources. We present a novel approach that enables users to specify varying degrees of detail and amount of constraints in a workflow request, including the specification of constraints on input, intermediate, or output data in the workflow, abstract workflow component classes rather than specific component implementations, and generic reusable workflow templates that express a pre-defined combination of components. The algorithm elaborates the user request into a set of fully ground workflows with specific choices of data sources and codes to be used so that they can be submitted for mapping and execution. The algorithm searches through the space of possible candidate workflows by creating increasingly more specialized versions of the original template and eliminating candidates that violate constraints cumulated in the candidate workflow as components and data sources are selected. A novel feature of our approach is that it assumes a distributed architecture where data and component catalogs are separate from the workflow system. The algorithm explicitly poses queries to external catalogs, and therefore any reasoning regarding data or component properties is not assumed to occur within the workflow system. We describe our implementation of this approach in the Wings workflow system. This implementation uses the W3C Web Ontology Language (OWL) and associated reasoners to implement the workflow system as well as the data and component catalogs. This research demonstrates the use of artificial intelligence techniques to support the kinds of automation envisioned by the scientific community for large-scale distributed scientific data analysis.

Keywords: computational workflows, semantic workflows, workflow generation, workflow systems, planning, semantic web, OWL, distributed reasoning.

1. Introduction

Large-scale computational experimentation has an ever-increasing impact in scientific practice, producing significant advancements in almost every discipline. Massive amounts of computing power are exploited to create sophisticated simulations for weather and earthquakes (www.scec.org), to extract new results from astronomical or particle physics data (www.nvo.org, www.ligo.caltech.edu), and to link biological entities at the molecular and cellular level. Visionary roadmaps in almost every scientific discipline include increasing levels of automation and support for ever more complex software [Atkins et al 03; Washington et al 05; Nature 06]. Artificial intelligence techniques can play an important role to represent complex scientific knowledge, to automate processes involved in scientific discovery, and to support scientists to manage the complexity of the hypothesis space [Muggleton 06; Gil 09; Gil et al 07b; Deelman and Gil 06; Langley et al 87].

Workflows have emerged as a useful paradigm to describe, manage, and share complex scientific analyses [Taylor et al 06; Gil et al 07b; Deelman and Gil 06]. Workflows represent declaratively the components or codes that need to be executed in a complex application, as well as the data dependencies among those components. Workflow systems exploit those representations in order to manage the elaboration and execution of workflows that process very large datasets in a distributed environment. Several workflow systems have been developed for a variety of applications, including Taverna [Oinn et al 06], Kepler [Ludaescher et al 06], Askalon [Wieczorek et al 05], and Pegasus [Deelman et al 05; Deelman et al 03]. Some workflow systems use workflows to compose web services. Other workflow systems manage computational workflows that combine implemented codes that can be submitted for execution to different remote resources. As the use of workflows becomes more common practice, workflow libraries have been identified as a crucial mechanism for sharing and reuse that is very important for e-Science [DeRoure et al 09; Gil 06; Gil et al 07b; Goderis et al 05a; Wroe et al 07]. Workflow libraries can represent well-known methods for data analysis that are discovered once, shown to work well, and subsequently reused by other researchers. These libraries would represent standard approaches, and their reuse would provide assurance of good practices.

An important requirement for workflow systems is that they should assume a very distributed environment. In e-Science, many distributed data repositories are made available and maintained by diverse institutions. Examples are the National Virtual Observatory (www.nvo.org), the Earth System Grid (www.earthsystemgrid.org), the Biomedical Informatics Research Network (www.nbirn.org), and the Cancer Biomedical Informatics Grid (cabig.cancer.gov). Data providers may provide services to access data sources. There can be many organizations playing the role of data providers, and as a result data may be accessible in various catalogs that are in distributed remote locations. Other organizations may provide algorithms, services, models, or implemented codes that can process data and can be used as components of the workflow. These are typically distributed and provided by different organizations. Therefore, an important requirement for workflow systems in e-Science is that they must rely on distributed services to access the data and algorithms necessary for data analysis.

This paper describes a novel algorithm for automatic generation of workflows from high-level workflow templates available in a workflow library. In our approach, users can create a workflow request by selecting a workflow template from a library and providing additional constraints on the desired results and on data sources to be used. A novel feature of the algorithm is that it allows users great flexibility in terms of the level of abstraction in the information that they provide as well as in the amount of information provided. Given a workflow request, the algorithm: 1) interprets the workflow request as a set of constraints on workflow components and datasets; 2) relies on an external component catalog to reason

about the implications of those constaints on the individual components, and an external data catalog to reason about individual data constraints; 3) reasons about the overall workflow by propagating constraints throughout the workflow, and 4) considers alternative dataset, component, and parameter choices by creating possible workflow candidates; 5) generates workflows that are ready for execution submission. An important and novel feature of the algorithm is that it assumes that all the reasoning about data and components is done by external catalogs rather than within the workflow system. The paper also describes our implementation of this approach in Wings [Gil et al 10; Gil et al 09b; Gil et al 09a; Kim et al 08; Gil et al 07a; Kim et al 06], which generates workflows executable by the Pegasus workflow system [Deelman et al 05; Deelman et al 03].

To exemplify our approach we use a workflow library for machine learning and data mining since this domain may be more familiar to computer scientists than other e-Science domains. It is a complex computational domain where the theoretical characterization of learning algorithms, their convergence properties, and their relative strengths and weaknesses remains a major research topic [Mitchell 06]. As a result, it is a good domain to investigate the use of workflow templates, as some researchers suggest that templates can capture important expertise that is not captured in libraries of algorithms [Bernstein et al 05]. It is also a domain that illustrates the distributed nature of data and algorithm providers. Many algorithms for machine learning are available in libraries [Witten and Frank 05], while many machine learning datasets used by researchers are maintained in the well-known Irvine repository [Asuncion and Newman 07]. Many research groups make datasets and algorithms available on their project sites. Machine learning is being pushed by researchers in the direction of further sharing and accessibility of data and codes [Sonnenburg et al 07]. Ultimately, automating the generation of workflows composed of machine learning algorithms would open the door to a vast opportunities offered by the scientific, business, and open source data available at an increasingly larger scale.

The paper begins describing the use of workflows for data analysis and motivating the need for automating their creation. Next, it describes the requirements that automation poses on various aspects of a workflow system architecture that operates in a distributed environment. With those requirements in mind, we provide a formalization that reflects those requirements and can be used to design the workflow system and its interactions with external services. We then describe an algorithm that can elaborate workflow requests automatically, and relies in the formalization to reflect its dependencies on various components of the distributed architecture. Finally, we describe our implementation of this algorithm and show results using workflows that use machine learning algorithms and data.

2. Motivation and Requirements for Automatic Workflow Generation

The goal of our research is to automate workflow creation and execution as much as possible. A benefit for scientists is the dramatic reduction of the computational experimentation cycle, since setting up data analysis experiments can be done quickly and all the elaboration and execution details are taken care of automatically by the system. Another benefit of automation is that it paves the way for efficient experimentation and discovery, as the system could autonomously explore different combinations of codes and data that may be appropriate to accomplish high-level goals. Finally, a benefit of automation is to empower users that are not experts in the particulars of workflow constituents or available data sets, but would like to perform analyses that others have designed and added to a workflow library. This not only includes junior researchers in an area, but also scientists in a different area of research that may wish to conduct complex

analysis that cross the boundaries of their particular discipline and do not have deep understanding of the techniques used in other research areas.

Consider machine learning research and practice. Machine learning research has produced a vast amount of learning algorithms and had led to many successful applications. However, it is challenging to select the right algorithm or combination of algorithms when faced with a new problem or dataset. This is in part because understanding the relationships and applicability of different algorithms is an important research area in its own right. Libraries such as Weka facilitate sharing of codes, and more recently also small workflows. Some interactive tools for automatic generation of novel algorithm combinations have been developed to assist researchers and practitioners alike [Berstein et al 05; Morik and Scholz 04; St Amant and Cohen 98]. While it is possible to describe for some algorithms what their function is, it remains a research challenge to specify how to describe them fully or how to build valid or good combinations of a given algorithm with others [Mitchell 06]. For example, it is possible to specify that a naïve Bayes classifier and a decision tree classifier both generates classes based on their input data, and that the former takes categorical attributes as input while the latter takes numerical attributes [Bernstein et al 05]. For more complex algorithms in the literature that perform more complex learning tasks such as pattern matching and other forms of relational learning [Kubica et al 03; Adibi et al 04] it is very hard to characterize their individually differentiating factors especially when combined with other algorithms into a more complex analysis (a workflow). The choice of algorithms results in crucial tradeoffs between the quality of the result, the cost (in terms of false positives and false negatives), and the execution time. Recently, ensemble methods that combine several algorithms have better performance than each of the algorithms individually [Opitz and Maclin 99]. Particular ensemble designs could be represented as workflows composed of the algorithms selected. Therefore, a workflow library that captures successful combinations of algorithms could provide very a useful repository of machine learning software.

While recent workflow repositories have been created with machine learning algorithms, they rely on user-guided selection and sometimes execution of workflows [Cannataro et al 04]. Automating the selection and execution of workflows would make machine learning technology accessible to a broader range of applications. [Bernstein et al 05] proposes that a user could request an analysis to classify a dataset into two classes with minimal cost, to find classes with comprehensible descriptions, or to classify a large dataset as fast as possible. A user may also want to query for proof that a certain property holds for some type of objects (i.e., a pattern), and the system would have to find a workflow and to marshall appropriate data sources to provide matches for that pattern. [Bernstein et al 05] argue strongly for the value of well-known reusable compositions of algorithms that are essentially high-level workflow templates.

Figure 1 shows sketches of some very simple workflows that can be built with machine learning algorithms. The first workflow shows how to use 2007 weather data for Santa Monica to train an ID3 model, then use that model to make predictions of the weather in Pasadena using an ID3 classifier. The suffix ".csv" indicates data formatted as comma-separated values, and the parameter ClassIndex set to 5 indicates that the feature in column 5 is the one we are trying to predict. The second is a generic version of that workflow that expresses how to use two algorithms, an ID3 modeler and an ID3 classifier, to learn a decision tree model from training data and then use that learned model to classify test data. The third is also a generic workflow that uses training data to learn a decision tree model used to classify given test data. This workflow uses general classes of algorithms, since a possible decision tree algorithm could be C4.5, ID3, or LMT. Because some algorithms can only deal with discrete datasets, the workflow includes steps to discretize the initial training and test data.

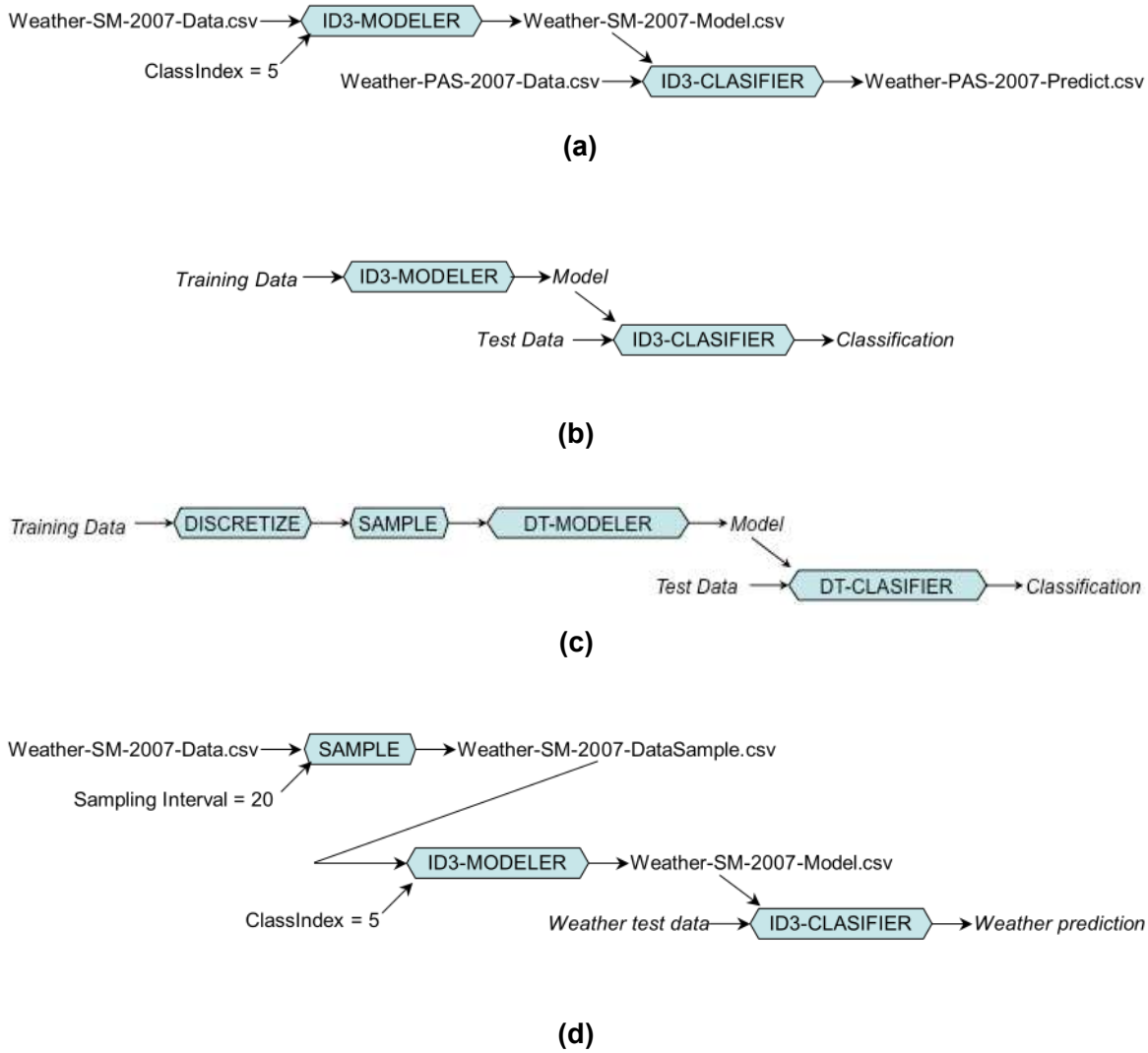


Figure 1: A high-level sketch of some workflows: (a) W_A is a workflow to process 2007 weather data from Santa Monica to make weather predictions for Pasadena; (b) W_B is a generic version of W_A that uses ID3 to learn a model from training data, then use the model to classify test data; (c) W_C is a generic workflow to use any algorithms that use decision trees to learn and classify continuous datasets after discretizing them; and (d) W_D is a generic workflow that is customized for weather prediction using ID3, and that samples the training data to obtain results faster. Components are shown in upper case, references to data and component parameters are capitalized.

In addition, the workflow includes a step that will sample the training data if it is over a certain size in order to make the learning process more efficient. The fourth workflow is also generic but it is customized for predicting weather data using ID3. Notice that it is a more specialized version of the second workflow.

The implementation of each algorithm as software code becomes a *workflow component*. A workflow is essentially a composition of these components, and indicates the order and dataflow for their execution. Workflow components can be web services, or implemented codes that can be submitted for execution to different remote resources [Deelman et al 05; Deelman et al 03; Kim et al 08; Gil 06]. We refer to either of these workflow constituent

computations as workflow components. Workflow components are registered in a *component catalog*.

Workflow components process data which is available through a *data catalog* that manages unique identifiers for *data objects*. Initially, the data catalog is populated with data that will be used as input to the workflows. As the workflows execute, the components generate new data objects that are *data products* of the workflows. For example, the execution of a Bayes learning algorithm generates a model of the training data in the form of a Bayes model. This learned model is a new data object that is added to the data catalog by the workflow system.

Our goal is to automate the process of answering user requests. Requests may include a variety of requirements, such as:

> Create a Naïve Bayes model of a dataset.
>
> Classify a dataset using a model that was created by 10-fold cross-validation of 500 instances.
>
> Classify a weather dataset, prefer faster rather than more accurate response.

Notice that the first includes requirements in terms of a desired output data product, while the second constrains what kind of input data to use. The workflow in Figure 1(c) would be appropriate for the first user goal, while the workflow shown in Figure 1(b) would not be appropriate since the models created with ID3 are not Bayes models. For the second user goal, either of those two workflows may be appropriate but the choice should take into account what kinds of models are available in the data catalog. For the third request, the workflow in Figure 1(d) would be most appropriate. Users may wish to use a range of requirements, these are some initial examples just to illustrate that variety.

In order to reason about workflows and how to use them to satisfy user requests that include such requirements on datasets, we need a rich workflow representation that allows us to reason about the properties of input and output datasets as well as the properties of algorithms. The rest of this section describes these requirements, starting off with data representations, then component representations, and using those to introduce representations of workflows and of user requests.

2.1. Representing Data: Requirements and Examples

Data objects have *metadata properties*, which are used to describe useful features of the data. An example metadata property is the size of a datasets, and whether the data is continuous or discrete. Table 1 shows some examples of the input datasets that we use in this paper and some of their metadata properties.

Metadata properties can specify the type of the data, for example whether they are instances or models, and for models whether they are a decision tree model or a Bayes model. Table 1 shows the types of data that we consider in this paper. All the datasets listed in the table are of type TrainingInstances.

An important requirement is the ability to describe new workflow data products since we need to be able to refer to data objects that do not yet exist and that will only exist once the workflow is executed. In addition, we need to anticipate the metadata properties of those objects. We refer to this as the *projected metadata* for a data object. For example, if the training dataset is of a particular kind or domain, such as weather data, then we can state that the model learned is also for weather data and that the classifier is expected to operate on test data that are also weather data. To support this, we need to be able to state not only that the domain of the training data (e.g., weather) must be the same as the domain for the given test data, but also to state or infer that the learned model is of that same domain (e.g., also weather). Notice that the projected metadata for a data object may be

different than the actual metadata obtained once the data object is created. For example, we can anticipate that a learned model for a training set of 1,000 instances will have 25 rules, but the actual learned model may end up containing exactly 26 rules. The data catalogs need to handle the fact that these new projected data products may never materialize, either because the workflow will not be selected for execution (in favor of other alternatives) or because the execution of the workflow may fail.

In summary, representing workflows requires being able to represent statements about data objects that include the ability to:

1) refer to the data objects that will be used as input data,

2) attach properties to those data objects,

3) state relations among properties of different data objects,

4) refer to new workflow data products and their properties, and

5) make inferences about properties of new workflow data products based on properties of existing input data objects and of other data products.

2.2. Representing Components: Requirements and Examples

We mentioned earlier that the implementation of each algorithm as software code becomes a workflow component. A workflow component is a model of a software code that has been encapsulated so as to have clearly defined inputs and outputs. It also must be encapsulated so that any execution requirements and dependencies on run-time libraries are explicitly stated, so the component's execution can be managed by the workflow system. A given algorithm can have several implementations, each resulting in a different workflow component. Also, a given implementation can be compiled or configured in different ways for a range of computer architectures or operating systems, which would result in different workflow components. Therefore, a workflow component uniquely identifies an algorithm or function, an implementation of that algorithm, and an executable configuration of that implementation.

Components take as *input data* datasets registered in the data catalog and produce output data that will also be registered in the catalog when created. Additionally, components may have *parameters* which are inputs that serve to configure the behavior of the component. Parameters take values from simple types that are not registered in the data catalog. We refer to the combination of input data and parameters as the set of *arguments* of a component.

An example of a component is an ID3 decision tree modeler. Given a dataset as input data, it uses the dataset as training data to learn a decision tree model that can be used to classify new data. It has an additional input argument, which is a parameter to specify which example feature is to be predicted by the learned model (this is called the class index of the dataset, which effectively will be the column to be ignored for training). These characteristics can be represented as shown in Table 2(a).

Components often require representing properties of their input or output data. For example, that a component that discretizes a dataset has as output a dataset that is discrete (instead of continuous data). This is shown in Table 2(b). Components also may require representing constraints among their input and output arguments. For example, a component that extracts a sample from a dataset by selecting every k-th element has as its output a dataset whose size is 1/k of the size of the input dataset. This is shown in Table 2(c).

Data sets:

Id	Domain	Size (K)	isDiscrete	hasMissingValues
20070730194138	contact lenses	3924	t	nil
20070731161618	cpu	2388537	nil	t
20070730195133	iris	30612	nil	nil
20070730195539	labor	588210	nil	t
20070730202315	soybean	27113	t	t
20070731101501	weather	2413	nil	nil
20070730221909	weather-nominal	27061	t	nil

Data Types:

DataObject
- *InitialData*
  - *RawData*
  - *SampledData*
  - *DiscretizedData*
- *LearnedModel*
  - *DecisionTreeModel*
  - *BayesModel*
- *Classification*
  - *DecisionTreeClassification*
  - *BayesClassification*

Components:

Weka-Component

Modeler	*Classifier*	
DecisionTreeModeler	*DecisionTreeClassifier*	
J48Modeler	J48Classifier	// Implements C4.5
ID3Modeler	ID3Classifier	// ID3: discrete only
LmtModeler	LmtClassifier	// Lmt: no missing values
BayesModeler	*BayesClassifier*	
BayesNetModeler	BayesNetClassifier	// BayesNet: discrete only
NaiveBayesModeler	NaiveBayesClassifier	// NaïveBayes: discrete only
HNBModeler	HNBClassifier	// HNB: discrete only

Preprocessor
- RandomSampleN
- Discretize

Table 1: Example datasets and components for machine learning workflows. For each dataset we show the metadata properties. The data types are organizad into a class hierarchy, indicated as indentation. All datasets shown are of type TrainingInstances. The components are organized into hierarchies, classes shown in italics. Major requirements that components have on datasets are also shown as comments on the right hand side.

(a) Basic Datatypes	(b) Properties	(c) Inter-argument constraints
Component ID: ID3-Classifier Input: d: Dataset m: DecTree-Model Params: i: ClassIndex Output: o: DecTreeClassification	Component ID: Discretize Input: d: Dataset Output: o: Dataset **is Discrete**	Component ID: SystematicSample Input: d: Dataset hasSize s:Size Params: k: SamplingInterval Output: o: Dataset **hasSize quotient(s,k)**

Table 2. Increasingly more detailed representation of the arguments of a workflow component: (a) representing only basic datatypes of the arguments, (b) representing more detailed datatypes of the output arguments, (c) representing constraints across component arguments.

(a) Abstract Component	(b) Concrete Components	
Component ID: DecTreeModeler is Abstract Input: d: Dataset hasSize s:Size Params: i: ClassIndex j: maxJavaHeapSize j <- 256x rem(s/1000) Output: o: Model is DecTree	Component ID: ID3-Modeler is DecTreeModeler is Concrete Input: d: Dataset is **Discrete** Params: i: ClassIndex j: maxJavaHeapSize Output: o: DecTree-Model	Component ID: Lmt-Modeler is DecTreeModeler is Concrete Input: d: Dataset is **NoMissingVals** Params: i: ClassIndex j: maxJavaHeapSize Output: o: DecTree-Model

Table 3. Representation of abstract and concrete workflow components. Concrete components correspond to executable codes, abstract components correspond to classes of components with common general properties.

Components have *argument identifiers* that enable the workflow system to refer to particular arguments of the component. For example, the component ID3-Modeler has an input dataset whose identifier is "d".

Components in a component catalog may be *abstract* or *concrete*. Concrete components model pieces of software that can actually be executed while abstract components are descriptions of the common features of a set of concrete components, in a similar sense to the way an abstract class in object-oriented programming gathers the commonalities of its subclasses. Table 3(a) shows an example of an abstract component, which represents all common properties of decision tree modelers such as the output is in the form of a decision tree. Table 3(b) shows two examples of concrete components. The ID3-Modeler requires input data that are discrete, while the LMT-Modeler requires the training examples used as input to have no missing values for their features. The abstract component illustrates how the value of a metadata property of an input dataset (s) is used to set the value of a parameter (h).

For this paper, we consider a component catalog with the components shown in Table 1, organized into the component classes shown in italics. The catalog represents requirements that the components place on the input datasets. For example, only J48 and Lmt can process continuous data. Lmt cannot process data sets with missing values. The execution of a workflow component results in an invocation of the code that implements that component. We refer to the *invocation command* as the string that can be used to invoke

that code and that contains all the arguments in the required order and with the required prefixes and formats. The invocation command may require an identifier for the output datasets (for example a file name). It may have inputs and outputs mixed together. In order to enable dynamic assignment of execution resources to the component, the codes must be *encapsulated* so that all their execution requirements are declared explicitly, including paths to libraries and other dependencies. To execute workflow components, a workflow catalog needs to be able to provide information about where the codes are installed and what their execution requirements are.

Components can have additional general properties, such as failure rates, average execution time, or usage rates.

In summary, we have the following requirements for representing components:

1) represent input data, parameters, and output data in each with a unique argument identifier,

2) represent constraints on the values that arguments can take, including type,

3) represent constraints across argument values,

4) represent classes of components based on common properties, and

5) ability to generate an appropriate invocation command.

2.3. Workflows: Requirements and Examples

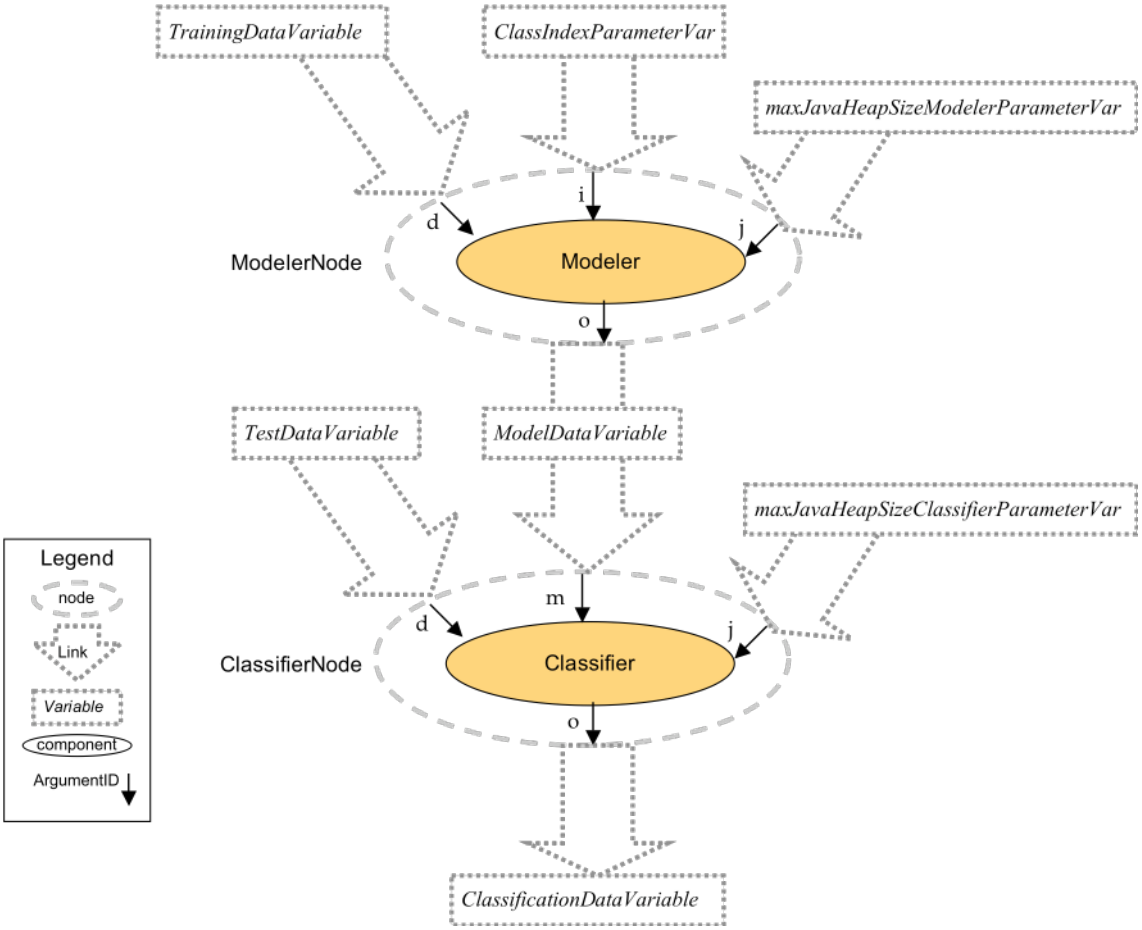
Figure 1(a) shows a workflow that refers to specific datasets and has values for all component parameters. We refer to such a workflow as a *workflow instance*. Workflow instances are specific to the given datasets and parameters. We also want to represent workflows that are generally applicable to a range of types of data or that can take in a range of parameter values. The workflows shown in Figures 1(b) and 1(c) are examples of such generic workflows. We refer to such a workflow as a *workflow template*. Workflow templates are reusable, and represent commonly used analyses.

The need for workflow templates places additional requirements for workflow representation. For example, for the workflows sketched in Figures 1(b) and 1(c) we would want to state one of the tenets of machine learning: that the training data must be a different dataset from the test data used within the same workflow. Representing such statements hinges on the ability to refer to the data objects that will be eventually selected as training data and test data and to make statements that relate those data objects. Therefore, to represent workflow templates we need to include *data variables* (instead of identifiers of particular datasets as a workflow instance would). Data variables are used to refer to input data as well as workflow data products that have not been yet identified. Thus, data variables allow us to express workflows that are variabilized and highly reusable.

Workflows have complementary representations of *structure* and *constraints*. The structure of a workflow reflects the dataflow among components, while the constraints reflect interactions among components and datasets. We explain now both aspects of the representation in more detail.

The structure of a workflow is specified as a set of *nodes*, each corresponding to a component, and a set of *links* that reflect the dataflow across components. For simplicity, we use links to specify input and output data and refer to them as input and output links, with an empty origin node and an empty destination node respectively. Similarly, we use parameter links for parameters, and give them an empty origin node. All other links are in-out links and incluye both an origin and a destination node.

Workflow Structure:



Workflow Constraints:

TrainingDataVariable ≠ TestDataVariable

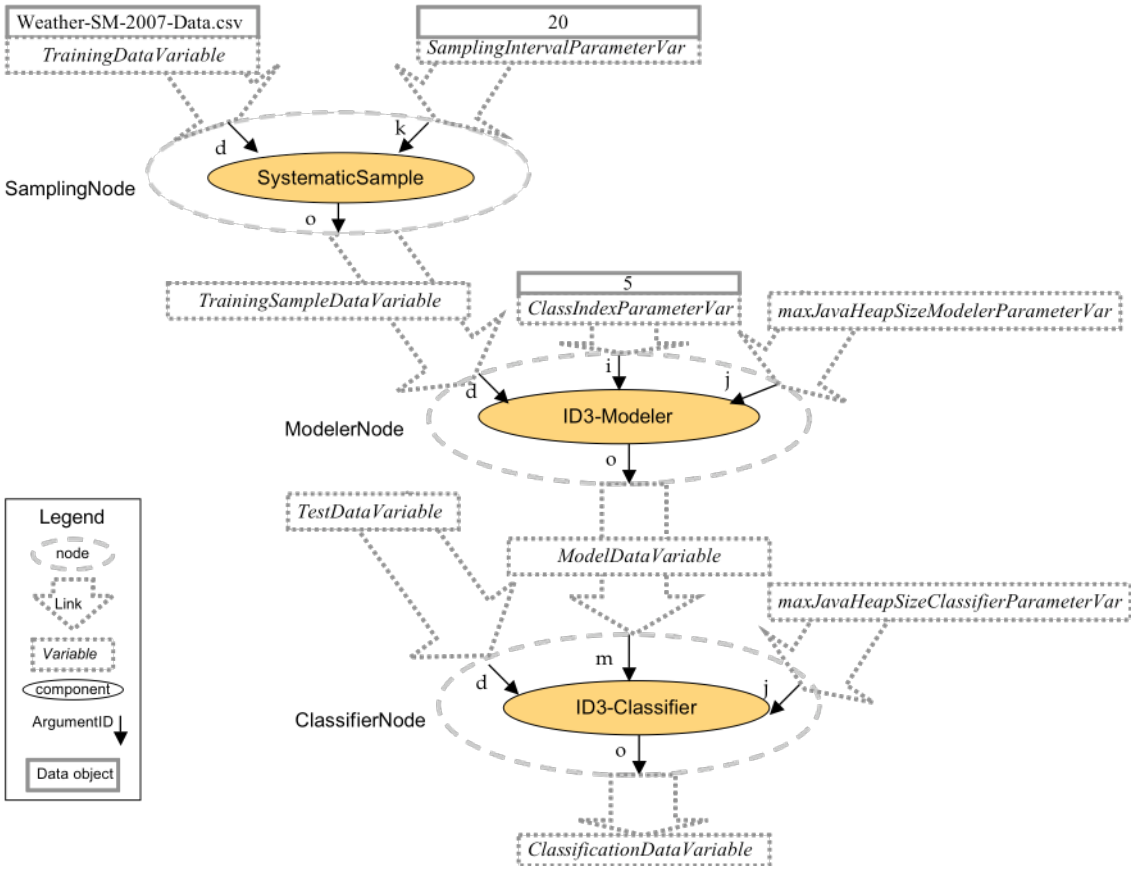
Domain of TrainingDataVariable = Domain of TestDataVariable

Figure 2: The representation of a workflow to learn a model and then classify test data. The nodes are shown in dashed ovals, links are shown in dotted arrows. The top illustrates the representation of the structure of the workflow, the bottom shows the constraints of the workflow.

The representation of constraints for a workflow consists on assertions on data variables. A constraint may restrict the kind of data that can be bound to a variable, essentially its type. Workflow constraints can relate more than one data variable. An example is the constraint mentioned earlier that the training data and the test data should not be the same.

Data variables express the cross-references between the structural representation and the constraints representation. Constraints are expressed over workflow data variables. The same data variables are used to express dataflow among components in the links of the workflow structure representation.

Workflow Structure:



Workflow Constraints:

TestDataVariable has Domain = weather
Training DataVariable = Weather-SM-2007-Data.csv
SamplingIntervalParameterVar = 20
ClassIndexParameterVar = 5
TrainingDataVariable ≠ TestDataVariable
Domain of TrainingDataVariable = Domain of TestDataVariable

Figure 3: A workflow to learn a model to predict weather. The training data is sampled first at a set rate. The workflow constraints specify bindings of data variables as well as parameter settings. Workflow constraints also express a restriction on the test dataset that it contains weather data, which avoids incorrect use of the workflow with other kinds of data.

Figure 2 shows a representation of the workflow in Figure 1(b) which has only two nodes. The dotted ovals and the dotted lines represent workflow nodes and links respectively. The ModelerThenClassifier workflow consists of two nodes (modelerNode with Modeler component and ClassifierNode with Classifier component), two input links (for modelerTrainingData and classifierInputData), two parameter links (for javaMaxHeapSize and modelerClassIndex), one in-out link (for outputModel and classifierInputModel), and one output link (for classifierOutput). In this workflow, there are several data variables shown. One is TrainingDataVariable, which refers to the input data that will be used as training data and

can be bound to any data sets available. Another data variable is ModelDataVariable, which refers to the data product generated after the Modeler component will be executed. The workflow in the figure is variabilized, and its input data variables can be bound to many posible combinations of data inputs. Links specify which argument identifier of the component is associated with the link. In Figure 2, the argument identifiers are shown next to the solid arrows inside the nodes. For example, the learned model corresponds to the "m" argument id of the Classifier component. We include here a parameter that is used in the Weka implementation and that specifies the allocation of memory to be used (indicated with a "j" argument identifier on both components) and should be set in proportion to the size of the input data sets. We will use this parameter in later sections to illustrate how the components can be automatically configured during workflow generation.

In our work, the structure of a workflow is constrained to be a directed acyclic graph (DAG). This is a very simple structure that we have found very useful in many fronts: including hiding programming constructs from users, facilitating reasoning about workflows (in particular automatic workflow generation, which is the concern of this paper), and last but not least recovery of execution when a job fails in the middle of the workflow. Many workflow languages depart from this basic structure and enable constructs such as conditionals and iterations through global variables. Using DAGs, we are able to support simple forms of iteration over data collections, as well as conditional execution based on data types [Gil et al 07a]. We have found this structure to be very manageable and to cover what was needed for a wide range of applications.

Figure 3 shows another example of a workflow customized to learn to predict weather data, as sketched in Figure 1(d). This is a workflow that has data objects assigned to some of the data variables and values assigned to some of the parameters, both done through the workflow constraints. This workflow also illustrates how constraints represent additional metadata properties of the input data. In this case, a constraint indicates that the domain of the test dataset used must be weather. This constraint will avoid the incorrect use of this workflow to make predictions over non-weather data.

The representation of the structure of the workflow is essentially syntactic in nature, as it is concerned with having a complete specification of the direction of the dataflow for all the inputs and outputs of the components. The representation of the constraints is semantic in nature, and is concerned with having a consistent specification of the nature of the data exchanged among components through the dataflow.

In summary, to represent workflows we have the following requirements:

1) represent dataflow across components,

2) represent data variables that are generic placeholders for actual datasets, so we can have reusable workflow templates,

3) represent constraints on data variables,

4) represent constraints across data variables, and

5) represent different degrees of generality in the workflows, including bindings for input datasets and values for parameters.

2.4. Workflow Requests: Requirements and Examples

Our goal is to automatically generate responses for a variety of requests from users by generating and executing workflows that satisfy the requests. Users may specify a variety of criteria in the requests, such as:

- *Functional properties*: Users often want to use workflows based on the nature of the computations performed or the desired data products. These include:

- *component-centered properties* that refer to the kinds of computations performed by the workflow components
- *data-centered properties* that refer to desired data products, or that specify that a certain type of data of interest to the user must be used in the workflow.

Example of output data properties: Create a Naïve Bayes model of labor data.

Example of input data properties: Classify iris data using a naive-bayes model with three classification classes and created from at least 500 instances.

Example of component properties: Create a model of labor data using ID3.

Example of component properties: Create a model of labor data with no sampling steps in the workflow (i.e., using the complete training data set).

- *Structural properties*: Users may provide constraints on the structural composition of the workflow concerning the relative ordering of steps. For example, a user may seek a workflow that performs data aggregation on a collection of datasets before performing clustering operations.

  Example: Sample soybean data and then create a Naïve Bayes model.

  Example: Use ModelThenClassify workflow with soybean data.

- *Non-functional properties*: These properties express user requirements regarding workflow performance and other costs. We highlight two here:
  - Execution time: A desired turnaround time for obtaining results.
  - Result quality: A threshold of quality or accuracy measured in some domain-relevant metric.

  Because some of these requirements may be in conflict, users may state additional combination functions or preferences. For example, there is typically a tradeoff between execution time and result quality, where shorter time often implies a lower quality results. A combination function may be expressed in the request when both time and quality matter.

  Example: Create a model of soybean data with maximum accuracy.

  Example: Classify iris data and minimize the response time.

- *Resource properties*: Users may have specific requirements about the execution resources to be used in executing the workflow. For example, for a workflow designed to compare the performance of a set of algorithms the user may request that all the algorithms may be executed on the same target architecture, or that the datasets used should be those existing at specific locations. Users may also request that specific resources should not be used, such as datasets generated by prior workflow executions or datasets that have not been updated for some period of time.
- *Cumulative properties*: These are properties of workflows that are derived through usage. Users may prefer to use workflows that are most frequently used by a user group, or more popular for a given function.

  Example: Create a model for soybean data using the most popular decision tree modeler.

- *Comparative properties*: Properties that are derived by comparing across possible candidate workflows. These can be used as ranking functions that drive the selection of workflows that have higher ranking.

  Example: Create a model for soybean data with minimum description length.

A language for expressing workflow requests should include constructs to specify all these kinds of properties. For this work, we focus on supporting requests that specify functional and structural properties. In order to represent user requests we need to be able to:

1) allow users to specify specific input datasets,

2) allow users to specify properties of desired input, output, and intermediate data,

3) allow users to specify values of parameters, and

4) allow users to specify a workflow template as a pattern of computation.

2.5. Workflow Generation: Requirements and Examples

An important consideration for automated workflow generation is that it must allow the user to have the flexibility to specify very little or alternatively to specify precisely what they want in the workflow request. The system has to specify all the aspects of the workflow that the user did not specify in the request. We illustrate this requirement for flexibility in the user input using the examples in Table 4. We show several examples of user requests, which refer to the workflows shown in Figure 1. For each request, the table shows whether workflow, component, parameter, and data are specified by the user or by the system. In user request R_A a workflow template is provided that specifies all the components to be executed and the datasets to be used. However, the request only specifies one parameter setting, so the system has to generate values for other parameters and as it turns out they depend on the parameter value that the user provided. In user request R_B only one dataset is specified, but constraints on another input dataset (training data) are given (domain must be weather and area must be Pasadena). The system has to select a dataset that satisfies those constraints. The system also has to select values for all the component parameters, and those values are determined by what datasets are chosen. All the components are specified in R_B so the system does not have to select them. R_C shows an example of a user request where the user does not specify the components to be used, but provides constraints on an intermediate dataset (the model learned must be a C4.5 model) and the output dataset (it must be of labor data). The system also has to choose values for parameters not provided by the user, which it could do by assuming default values. R_D is a user request where all the components, data, and parameters are provided by the user, so there are no requirements for the system in terms of workflow generation. In the other extreme is R_E where the user does not specify a workflow template to follow, and only specifies one dataset to be used. The system would have to either retrieve a matching workflow template or generate a workflow from scratch. When retrieving a workflow template, the system would have to map the dataset constraints provided by the user to data variables in the template. Then the system would have to select the remaining datasets, components (if the template has abstract components), and parameters.

These examples illustrate that user requests can range in terms of the specificity and the completeness of the information provided. The system must be able to use the information provided in the request and automatically generate a completely specified workflow that can be submitted for execution.

In terms of user interaction, the requirements for the workflow generation system are that it should provide flexibility to the user in two important ways:

1. Users should have the ability to specify functional or structural properties concerning any component or dataset in the workflow. The system should then use that information to generate an executable workflow from that high level guidance.
2. Users should have the flexibility to specify some or no information concerning a specific type of functional or structural property. For example, users should be allowed to issue workflow requests that do not specify what components should be used but that specify what datasets to use.

User request	Workflow selection	Component selection	Parameter selection	Data selection
$\boldsymbol{R_A}$: *"Use W_A as is"*	User selects a workflow template.	User selects a workflow that has specific components.	User specifies class index 5 for the modeler but no other parameters.	User specifies all datasets.
	System does not have to determine dataflow among components.	System does not have to select any components.	System has to select: • Class index for the classifier, which has to be set to the same value as the class index for the modeler • Heap size, which could be set to a small size since those weather datasets are large	System does not have to select any datasets.
$\boldsymbol{R_B}$: *"Use W_B to classify 2007 Santa Monica weather data using Pasadena weather as training data"*	User selects a workflow template.	User selects a workflow that has specific components.	User specifies no parameter values.	User specifies one of the input dataset to be used and some of the characteristics of another dataset (the training dataset).
	System does not have to determine dataflow among components.	System has to select any components.	System has to select all the parameter values, some using defaults and some constrained by the datasets chosen (eg heap size will depend on the size of the dataset chosen).	System has to select: • A training dataset with the required characteristics of containing Pasadena weather data
$\boldsymbol{R_C}$: *"Use $W_{C\text{-}NEW!!}$ to generate a classification of labor data on class index 7 using a C4.5 model"*	User selects a workflow template.	User selects a workflow template but does not select specific components.	User specifies class index 7 for the classifier but no other parameters.	User specifies characteristics of the output dataset, which should determine the choice of input datasets.
	System does not have to determine dataflow among components.	System has to select appropriate decision tree modelers and classifiers, inferring that only J48 is possible: • ID3 cannot be used as classifier since it is a continuous dataset, and cannot be used as modeler because its output is not a C4.5 model • LMT is not possible since labor data have missing values	System has to select: • Class index for the modeler, inferring that it has to be set to the same value as the class index for the classifier • Heap size, which has to be set to a large size since that labor dataset is large • A default value for the sampling rate • A default value for the discretizer bins	System has to select: • A test dataset in the labor domain as specified in the request • A training dataset, inferring that it would have to be in the labor domain so that the model created is in the labor domain and can be used for a labor classification
$\boldsymbol{R_D}$: *"Use W_D to classify Pasadena 2007 weather using class index 5 and heap size 512M"*	User selects a workflow template.	User selects a workflow that has specific components.	User specified all parameter values.	User specifies all datasets.
	System does not have to determine dataflow.	System does not have to select any components.	System does not have to select any parameter values.	System does not have to select any datasets.
$\boldsymbol{R_E}$: *"Generate a classification of 2007 Santa Monica weather data using 2006 Pasadena weather as training data"*	User does not specify a workflow template, components or a dataflow.		User specifies no parameter values.	User specifies some datasets.
	System has to: • Retrieve a matching workflow that satisfies the constraints in the user request • Select components if the matching workflow has any abstract components		System has to select all the parameter values once a workflow is chosen.	System has to: • Assign each dataset in the request to specific workflow data variables • Select other datasets that may be needed by the workflow retrieved

Table 4. Examples of user requests with descriptions of what requirements they pose to the system in terms of automated workflow generation as well as requirements to the workflow request language.

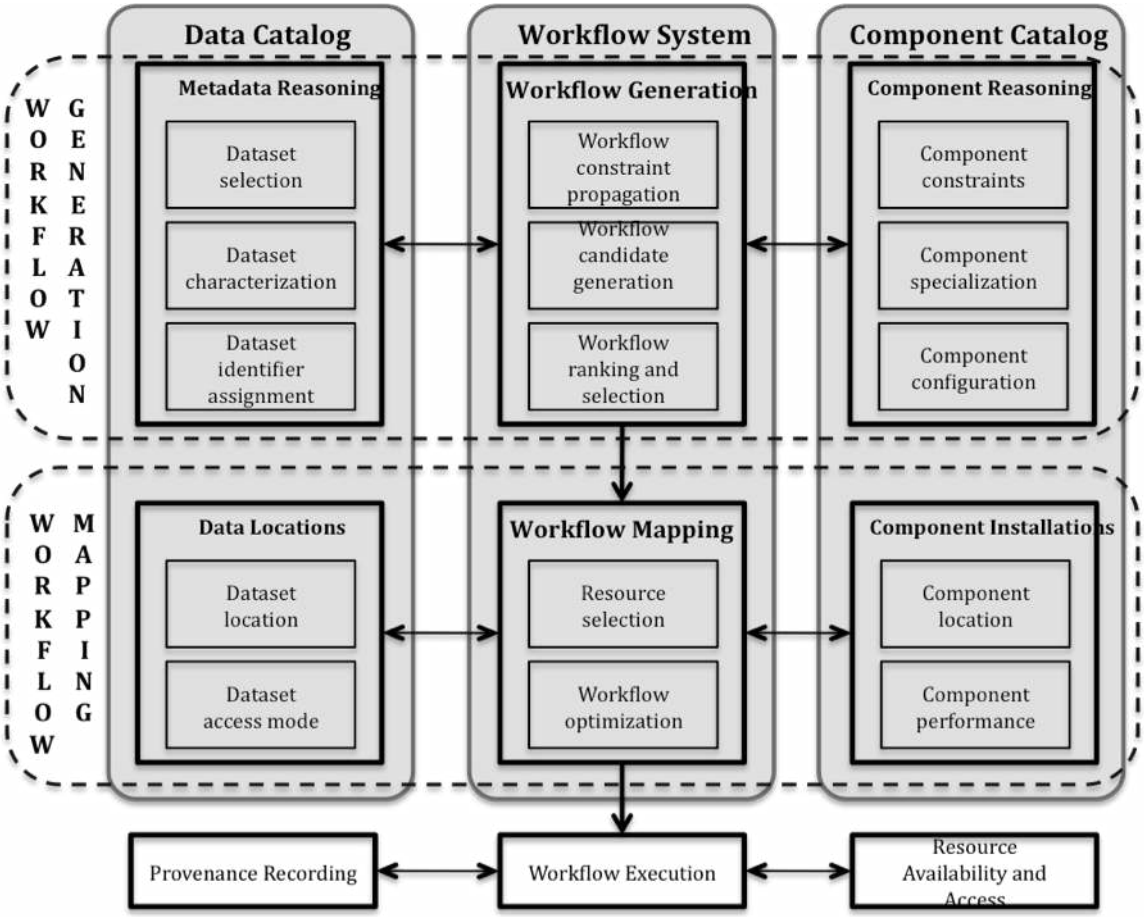


Figure 4. Workflow systems use distributed services to access external data, component, and other catalogs for workflow creation and execution.

In the remainder of this article, we assume that user requests always specify a workflow template to be used in combination with a *seed* that specifies additional constraints. If a workflow request does not include a workflow template, the seed could be used to search for relevant workflows in the library that could be used to accomplish the request. Matching requests and workflow templates is a unification problem [Baader and Narendran 01]. There is a large body of work on matching in the case-based reasoning literature [Ashley and Aleven 97; Bergmann and Stahl 98; Champin and Solnon 03; Forbus et al 94] as well as in matching in first-order and in description logic [Li and Horrocks 03; Baader et al 05; Hull et al 06]. Graph matching techniques have been used to retrieve workflows based on structural properties [Goderis et al 06]. For this paper, we assume that a workflow request is formed by a workflow template and a seed that specifies additional constraints and do not address workflow matching issues. Another alternative to responding to a workflow request that does not include a template is to use a planning algorithm to generate a workflow using the seed as a goal statement [Blythe et al 04a; Blythe et al 04b; Sirin et al 04] or as user guidance [Lin et al 08]. However, even though those approaches would be useful to express user requests such as R_E they would not be able to express some of the other types of requests exemplified above. Users cannot refer to intermediate datasets in the workflow, or to specific combinations of components and dataset characteristics. A wider range of user requests presented here could be addressed by approaches to planning from sketches [Myers et al 03] using hierarchical task network planning, although those

approaches have not been used for workflow systems. In our approach, by having a workflow template that contains a handle on what data variables and what components are to be used and what their ordering is, the user has great flexibility to refer to specific constituents of the workflow and relate them as they wish.

Another important set of requirements stems from the fact that workflow systems must assume a highly distributed architecture. Figure 4 illustrates the distributed nature of the environment where workflow systems must operate. Workflow generation is not a typical function of deployed workflow systems, and we will come back to describe this function shortly. Rather, workflows are typically created by selecting components and data by hand using a workflow editor [Taylor et al 06; Oinn et al 06; Ludaescher et al 06]. Workflow editors access distributed catalogs of components and datasets, and allow users to compose workflows out of those elements. Once a workflow is specified, an important function of workflow systems is mapping workflows to execution resources available in the environment. A well-known workflow mapping and execution system that we use in our work is Pegasus [Deelman et al 03; Deelman et al 05]. Pegasus assigns execution resources to each workflow component depending on its requirements stated in the component catalog, adds nodes to the workflow to account for movement of data to execution locations, adds nodes for data registration in catalogs, reduces the workflow by removing computations whose data products already exist, and performs several optimizations by restructuring the workflow. Pegasus submits the final workflow to the Condor Directed Acyclic Graph Manager (DAGMan) for execution [Thain et al 05]. Other examples of workflow execution systems include Askalon [Wieczorek et al 05] and the Java CoG Kit [von Laszewski et al 06]. Workflow execution is customized and optimized to the available resources. Once workflows are executed, provenance catalogs store detailed records of the execution process [Moreau and Ludaescher 08; Kim et al 08; Zhao et al 08]. Provenance services can be accessed by any other sercice in the architecture. For example, the component catalog services may use provenance data to analyze which components in its catalog are not used in practice and as a result can extend or improve the representations of those components.

Data and component catalogs are distributed and are external to workflow systems. Data catalogs typically focus on offering services that enable distributed access to data in an efficient and scalable manner [Moore et al 01; Singh et al 03; Tuchinda et al 04; Gil et al 05; Szalay and Gray 06]. The data resides in various distributed locations, and metadata services contain descriptions of the data. Runtime workflow systems typically access data catalogs to find out the necessary access protocols and physical locations of the data. Component catalogs may be service registries, typically using a registry based on the standard Universal Description Discovery and Integration (UDDI). Component catalogs can also be code repositories. Components are typically described today in terms of simple input and output data types [Fan and Kambhampati 05; Al-Masri and Mahmoud]. Some recent research addresses the development of catalogs with semantic service descriptions [Bussler et al 02] and their use to validate workflows created by users [Belhajjame et al 08].

For workflow generation, we must assume that the components and data catalogs are distributed and external to the workflow system. These catalogs will be typically provided by third parties that naturally want to have control over their contents. Data catalogs must be able to support workflow generation by reasoning about characteristics of datasets. Component catalogs must support reasoning about constraints and requirements of components.

The workflow system should be responsible for any reasoning that concerns the workflow itself while invoking the data and component catalogs to perform aspects of the

reasoning that are specific to data and components. At a minimun, in order to support the kinds of user requests discussed above, the workflow system should be responsible for:

- Merging off-the-shelf workflow templates with additional seed constraints expressed in the request,
- Reasoning about end-to-end compositions by managing the propagation of the effects that the requirements expressed in the request pose on other components and datasets of the whole workflow, and
- Exploring choices of datasets, components, and parameters to generate an executable workflow such that all choices are consistent with the information provided by the user in the workflow request.

To accomplish this, the workflow system should rely on the following reasoning services provided by external catalogs:

- Reasoning about data constraints should be performed by data catalogs
- Retrieval of datasets should be supported by data catalogs
- Assignment of unique indentifiers for any new datasets created by the workflow must be performed by data catalogs
- Reasoning about any constraints on input and output data characteristics that are required by components should be done by component catalogs
- Reasoning about the criteria for component specialization should be performed by component catalogs
- Assignment of component parameter values should be supported by component catalogs

Therefore, the distributed environment where workflow systems operate leads us to the requirement that workflow generation be driven by the workflow system while relying on reasoning functions provided by data and component catalogs that are external to the workflow system. The distributed reasoning that occurs during workflow generation is illustrated in Figure 4. Other existing research on workflow generation [McIlraith and Son 01; McDermott 02; Narayanan and McIlraith 02; Blythe et al 03; Kim et al 04; Ambite and Kapoor 07; Sirin et al 04; Lin et al 08] does not comply with this requirement. Those approaches assume a distributed architecture but the reasoning about data and components is centralized in the workflow generation system. In that work, the descriptions of components and data may come from external distributed services and in that aspect they assume a distributed architecture. However, the descriptions are imported into the workflow system, which then reasons about all data and component constraints so that all the reasoning is centralized.

The next section introduces a formalism that captures the requirements on the various distributed services needed to support automatic workflow generation.

3. Formalization

This section shows a formalization of our framework to satisfy the requirements discussed above. The formalization describes how a workflow system can model data, components, and workflows. We also show the basic functions that the workflow system needs in order to access external data, component, and workflow catalogs. Although our own implementation uses a particular representation formalism we use a more general formalization here, since we aim to be inclusive of data and component catalogs that do not use logic but are developed with other formalisms in mind (e.g., [Moore et al 01; Singh et

al 03; Tuchinda et al 04; Gil et al 05; Szalay and Gray 06]). We show later in the paper our implementation of this formalization.

Most functions in our framework take and produce metadata annotations. We use the term *data object description* (DOD) to refer to a set of metadata annotations that describe the properties of a given data object. DODs for workflow data variables effectively constrain the possible data that can be used to bind those data variables or to specify the dataset identifier that is bound to that data variable. DODs for component parameters can be used to constrain the values that the parameter can take, and to specify the value of a component parameter. DODs are in effect sets of constraints, and we often refer to them as *constraints*. DODs are sometimes obtained from reasoning about the workflow seeds, other times they represent metadata properties or *characteristics* of an existing dataset.

First, our formalism need the ability to express literal annotations and literal relational annotations. Formally, given the set E of data objects (input data, data variables or component arguments) that can be annotated with metadata, the set P of metadata properties, and the set R of relations between pairs of property values, we assume that the metadata formalism allows to write:

— *literal DODs* to specify the value of a metadata property for a given object as:
$\langle e, p, v \rangle$

where $e \in E$, $p \in P$, $v \in T_p$ and T_p is the type of metadata property p, i.e., the set of possible values for that type.

Example: For the workflow shown in Figure 3,

<TestDataVariable, has Domain, weather>

— *literal relational DODs* to specify a relation between the values of two properties of the same or different data objects as:
$\langle r, e_1, [p_{11}, \ldots, p_{1n}], e_2, [p_{21}, \ldots, p_{2m}] \rangle$

where $e_1, e_2 \in E$, $p_{11}, \ldots, p_{1n}, p_{21}, \ldots, p_{2m} \in P$, $r \in R$, and $n, m \geq 1$

Representing that relation r holds between the value at the end of the property chain $p_{11}, \ldots, p_{1n}$ starting in e_1 and the value at the end of the property chain $p_{21}, \ldots, p_{2m}$ starting in e_2. Formally:

$\exists\, x_1, \ldots, x_n, y_1, \ldots, y_m$ such that

$\langle e_1, p_{11}, x_1 \rangle \wedge \ldots \wedge \langle x_{n-1}, p_{1n}, x_n \rangle \wedge \langle e_2, p_{21}, y_1 \rangle \wedge \ldots \wedge \langle y_{m-1}, p_{2m}, y_m \rangle$ and

$r(x_n, y_m)$

Example: To add a restriction in the workflow in Figure 3 that it can only be used to make predictions with weather data that is from the same county as the training data, we can state:

< equals, TestDataVariable, [has-area, has-county],
TrainingDataVariable, [has-area, has-county]>

which represents compactly the following set of literal DODs:

< TestDataVariable, has-area, A1>
< A1, has-county, C1>
< TrainingDataVariable, has-area, A2>
< A2, has-county, C2>
< C1, equals, C2>

With these representations for DODs, we can represent workflow constraints. For example, the workflow constraints shown in Figure 3 can be expressed as the set of DODs:

M3= {< TestDataVariable, hasDomain, weather >
< TrainingDataVariable, has-value, Weather-SM-2007-Data >
< SamplingIntervalParameterVar, has-value, 20 >
< ClassIndexParameterVar, has-value, 5 >
< not-equal, TrainingDataVariable, [has-value],
TestDataVariable, [has-value] >
< equal, TrainingDataVariable, [has-domain],
TestDataVariable, [has-domain] >}

Given a set E of data objects that can be annotated with metadata, a set P of metadata properties, and a set R of relations between pairs of property values we denote as M^{EPR} the set of all possible literal and literal relational DODs, as defined above, that can be built using those sets. Whenever we do not need to specify E, P and R for the description, we will use M as an abbreviation for M^{ERP}, and whenever P and R are not relevant but E is we will use M^E as an abbreviation for M^{ERP}.

We assume two functions to extract the subset of DODs that only refer to a set of given entities E, and a function that returns the set of entities used in a set of DODs, as follows:

— **entity-DODs**: $P(E) \times P(M) \rightarrow P(M)$
which returns a subset of the given set of DODs that only refer to the given set of entities, and

— **get-entities-in-DODs**: $P(M) \rightarrow P(E)$
which returns the set of entities referred in the given set of DODs.

Given a set V_1 of entities and a set M_1 of DODs the following must hold:

entity-DODs(V_1, M_1) $\subseteq M_1$, and get-entities-in-DODs(entity-DODs(V_1, M_1)) = V_1

For example, for the set of DODs M3 above:

entity-DODs ({TestDataVariable TrainingDataVariable}, M3) ≡
{ <TestDataVariable, hasDomain, weather>
<TrainingDataVariable, has-value, Weather-SM-2007-Data >
<not-equal, TrainingDataVariable, [has-value], TestDataVariable, [has-value]>}

get-entities-in-DODs(M3) ≡
{ TestDataVariable TrainingDataVariable
SamplingIntervalParameterVar ClassIndexParameterVar }

Note that P(M) and P(E), which represent the powerset of the set M of possible metadata annotations and the powerset of the set E of entities, are the types of the arguments.

3.1. Data and Data Catalogs: Formalization

A data catalog must include functions relevant to retrieve and reason about data objects.

Formally, we say that given the set E of entities that can be annotated with metadata, a data catalog DC is described through a set of object data identifiers D_{DC} ($D_{DC} \subseteq E$) for the data objects managed by the catalog, along with the set of metadata anotations on those entities M_{DC}. Given those sets, we define the following functions:

— **obtain-DODs**: $DC \times D_{DC} \rightarrow P(M_{DC})$
Return as a DOD all the metadata properties and values of the given data object identifier.

— **assign-identifier**: $DC \times \mathrm{P}(M^E) \rightarrow E$
Provides a data object identifier based on a given set of metadata properties and values on an entity *e*. The identifier is formed based on the metadata properties and values and not on the actual entity *e*. The entity *e* may in fact not exist until the workflow is executed, and at the same time the workflow system needs an identifier to refer to it so it can prepare for the execution of the workflow.

— **assert-predicted-DODs**: $DC \times \mathrm{P}(M^E) \times E \rightarrow DC$
Register in the data catalog all the predicted metadata properties and values of the given data object identifier. As a result, the entity passed to the funtion is included in D_{DC}.

— **find-data-objects**: $DC \times \mathrm{P}(M^E) \rightarrow \mathrm{P}(\ \mathrm{P}(\ E \times D_{DC}\ )\ )$
Given an input set of DODs for several data variables, return a (possibly empty) set of data objects for all the variables in the input DODs that are consistent with the DODs. Each tuple of the form $\langle\ e \times d\ \rangle$ is a *binding* for a workflow data variable, where a variable is bound to a data object identifier.

— **combine-DODs**: $DC \times \mathrm{P}(M^E) \times \mathrm{P}(M^E) \rightarrow \mathrm{P}(M^E)$
Return a set of DODs which combines the metadata properties of two given sets, all of them on a given set of variables.

In order to be valid, an invocation to combine-DODs must verify that the sets are consistent.

3.2. Components and Component Catalogs: Formalization

In order to support the workflow generation process, a component catalog must include different functions about the components in the catalog.

Formally, we say that a component catalog *CC* is described through a set of components C_{CC}, which may be abstract or concrete, a set *I* of unique identifiers for each argument of the components (input data, input parameters, and output data), and a set M_{CC} of metadata annotations on the entities identified by I_{CC}. The invocation command for a concrete component in the catalog can be generated for data objects in a given set *D* and parameter values in a given set *V*.

We define the following basic functions:

— **inputs**: $CC \times C_{CC} \rightarrow \mathrm{P}(I_{CC})$
Return the identifiers for the input data objects of a component.

— **parameters**: $CC \times C_{CC} \rightarrow \mathrm{P}(I_{CC})$
Return the identifiers for the parameters of a component.

— **outputs**: $CC \times C_{CC} \rightarrow \mathrm{P}(I_{CC})$
Return the identifiers for the outputs of a component.

— **args**: args(*c*) is the set of *arguments* of a component $c \in C_{CC}$:
$\mathrm{args}(c) \equiv \mathrm{inputs}(c) \cup \mathrm{parameters}(c) \cup \mathrm{outputs}(c)$

— **invocation-command**: $CC \times C_{CC} \times \mathrm{P}(M_{\mathrm{CC}}) \rightarrow \mathit{String}$
Return the invocation command for a concrete component, given a set of data object descriptions that set the values (data object identifiers) for its inputs and the values for its parameters.

When the component library supports these functions for a component, and the invocation command results in a successful invocation and execution of the component, we consider the library to contain a *basic component encapsulation*. Supporting this basic

encapsulation of the underlying code does not require a component library to include semantic constraints or properties of data or components.

When a component catalog includes both abstract and concrete components, it needs to support the following functions:

— **is-concrete**: $CC \times C_{CC} \rightarrow Bool$
Determine whether a component c is abstract or concrete.

— **specialize**: $CC \times C_{CC} \times \mathrm{P}(M_{\mathrm{CC}}) \rightarrow \mathrm{P}(\ C_{CC}\ )$
Return a (possibly empty) set of abstract or concrete components that can specialize a given abstract component using a given set of DODs on the abstract component arguments. We assume that there is a one-to-one mapping between the arguments of each of the concrete components returned and the arguments of the abstract one. When this is not the case, there must be provisions for extending the workflow to account for the additional arguments, possibly to link them to data variables in the workflow, and possibly to add workflow components to the workflow to generate some of the additional arguments.

In order to be valid, an invocation of specialize(cc, c, M_1) must verify that DODs in M_1 only refer to the arguments of c: $M_1 \subseteq M^{\mathrm{args}(c)}$.

— **specialize-to-concrete**: $CC \times C_{CC} \times \mathrm{P}(M_{\mathrm{CC}}) \rightarrow \mathrm{P}(\ C_{CC}\ )$
Similar to specialize but returns only concrete components rather than subclasses.

The next functions support the automatic setting of parameters for a given component. When the component library supports these functions, we refer to the component as *self-configurable*. The functions are defined as follows:

— **is-configurable**: $CC \times C_{CC} \times \mathrm{P}(M_{\mathrm{CC}}) \rightarrow Bool$
Determine whether the parameter values for the component can be obtained from the component catalog given a set of DODs on the component arguments. Notice that this does not set the values of any parameters, it simply checks that they can be set by the component catalog.

— **configure**: $CC \times C_{CC} \times \mathrm{P}(M_{\mathrm{CC}}) \rightarrow \mathrm{P}(\ \mathrm{P}(M_{\mathrm{CC}})\ )$
Return a set of sets of DODs, each set specifying values for all the parameters of a component c given a set of DODs on the component arguments (inputs, outputs and parameters).

In order to be valid, an invocation of configure(cc, c, M_1) must first verify that:

- DODs in M_1 only refer to the arguments of c: $M_1 \subseteq M^{\mathrm{args}(c)}$ and
- is-configurable(cc, c, M) returns true.

— **is-configured**: $CC \times C_{CC} \times \mathrm{P}(M_{\mathrm{CC}}) \rightarrow Bool$
Determine whether a component c is fully configurable based on a given set of DODs on the component arguments.

We define two additional functions that component catalogs can provide in order to support workflow generation:

— **find-DODs-given-output-requirements**: $CC \times C_{CC} \times \mathrm{P}(M_{\mathrm{CC}}) \rightarrow \mathrm{P}(M_{\mathrm{CC}})$
Return additional metadata properties on the concrete component arguments using the given DODs.

In order to be valid, an invocation find-DODs-given-output-requirements (cc, c, M_1) must comply with:

- DODs in M_1 only refer to the arguments of c:
  $M_1 \subseteq M^{\text{args}(c)}$
- M_1 includes some annotations on the outputs of c:
  get-entities-in-DODs(M_1) ∩ outputs(c) ≠ ∅

Ex: find-DODs-given-output-requirements(Catalog1, ID3-Classifier,
<ID3-Classifier-o hasDomain Weather>)
→ { < ID3-Classifier-d hasDomain Weather>}

— **predict-DODs-given-input-requirements**: $CC \times C_{CC} \times \text{P}(M_{\text{CC}}) \rightarrow \text{P}(M_{\text{CC}})$
Return additional metadata properties on the concrete component arguments using the given DODs.

In order to be valid, an invocation predict-DODs-given-input-requirements (cc, c, M_1) must comply with:

- DODs in M_1 only refer to the arguments of c:
  $M_1 \subseteq M^{\text{args}(c)}$
- M_1 includes some annotations on the inputs of c:
  get-entities-in-DODs(M_1) ∩ inputs(c) ≠ ∅

Ex: predict-DODs-given-input-requirements(Catalog1, ID3-Classifier,
<ID3-Classifier-d has-value weather-2007-31-101501>)
→ {< ID3-Classifier-d number-of-instances 100>,
< ID3-Classifier-j has-value “512M”>}

When a component library supports the first function for a given component, we refer to the component as *backward-enabled*. The function is used to propagate through the workflow structure any constraints required from the output data products. When the second function is supported, we refer to the component as *forward-enabled*. This function is used to propagate through the workflow structure any properties of the input data. We also have functions to check whether the component representations in the catalog support these capabilities as follows:

— **is-backward-enabled**: $CC \times C_{CC} \rightarrow Bool$
The function **find-DODs-given-output-requirements** is defined for the component.

— **is-forward-enabled**: $CC \times C_{CC} \rightarrow Bool$
The function **predict-DODs-given-input-requirements** is defined for the component.

Finally, a function to estimate the performance of a component:

— **estimate-performance**: $CC \times \text{C}_{\text{CC}} \times \text{P}(M_{CC}) \times \text{A} \rightarrow T$
Return estimated performance as time of execution for the component for the given metadata that must include values for all the component parameters. The performance is estimated for a given reference architecture a within the set of all possible architectures A.

In order to be valid, an invocation estimate-performance (cc, c, M_1,v_1, … , v_n, a) must comply with:

- DODs in M_1 only refer to the arguments of c:
  $M_1 \subseteq M^{\text{args}(c)}$

- M_1 includes some annotations on the inputs of c:

  $$\text{get-entities-in-DODs}(M_1) \cap \text{inputs}(c) \neq \varnothing$$

- is-concrete(cc, c) returns true
- is-configured(cc, c, M_1) returns true

In later sections of the paper, we will show how all these functions are used during workflow generation.

3.3. Workflows

Given a component catalog which includes a set C of components, a data catalog with a set D of data object identifiers, a set V of possible values for component parameters and a set P of metadata properties, a workflow $\boldsymbol{w}$ is defined as a tuple of nodes, component to node mappings, data variables, parameter variables, DODs on data and parameter variables, data links, parameter links, data bindings, and parameter bindings. Formally, a workflow is specified as a tuple:

$$\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$$

where:

- N_w is the set of *nodes* in the workflow,
- σ_w is a mapping function that associate a component to a node:

  $$\sigma_w : N_w \rightarrow C$$

  Note that different nodes in the workflow may have the same associated component.
- DV_w is the set of data variables in the workflow
- PV_w is the set of parameter variables in the workflow.
- M_w, is DODs on the data and parameter variables of the workflow: get-entities-in-DODs (M_w) $\subseteq DV_w \cup PV_w$
- L_w, is the set of links in the workflow. A link l is represented as a tuple of the form:

  $$\langle n_o, o, v, n_d, i \rangle$$

  where $n_o, n_d \in N_w \cup \{\perp\}$, $o \in \text{outputs}(\sigma(n_o)) \cup \{\perp\}$, $v \in DV_w \cup PV_w$, and $i \in \text{inputs}(\sigma(n_d)) \cup \{\perp\}$.

  We refer to n_o as the origin and to n_d as the destination.

  An *input link* to the workflow is one without origin that connects a data variable to an input argument of a component: $\langle \perp, \perp, v, n_d, i \rangle$ $v \in DV_w$, while an *output link* is one without destination that connects an output argument to a data variable: $\langle n_o, o, v, \perp, \perp \rangle$ $v \in DV_w$.
- PL_w, is the set of parameter links in the workflow. A parameter link pl is represented as a tuple of the form:

  $$\langle pv, n, p \rangle$$

  where $pv \in PV_w$, $n \in N_w$, and $p \in \text{parameters}(\sigma_w(n))$.
- DVB_w is a, possibly empty, set of bindings of the data variables to data object identifiers:

  $$\langle dv, d \rangle \equiv \langle dv, \text{has-value}, d \rangle$$

where $dv \in DV_w$, $d \in D$

- PVB_w is a, possibly empty, set of bindings of the parameter variables to values:
  $\langle pv, v \rangle \equiv \langle pv, \text{has-value}, v \rangle$
  $pv \in PV_w$, $v \in V$

The parameter link indicates which parameter variable in the component corresponds to the link. No values are set to parameters in parameter links, instead the values are set through the parameter bindings.

Workflow catalogs should contain workflows that comply with the basic component encapsulation requirements for all of its nodes' components. That is, all the arguments and argument identifiers specified in the workflow nodes and links have a one-to-one correspondence with the arguments and argument identifiers defined for the nodes' components. We refer to such workflows as *well-formed* workflows. This is a syntactic property concerning the structure of the workflows, and does not concern the constraints or properties that may be defined for data variables or data objects.

We define the following types of workflows:

— *Specialized workflow*:
A workflow that contains only concrete components.
Formally, a workflow $\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$ where

- $\forall\, c \in C_w$: isConcrete(c)

where C_w denotes the set of components in the workflow:
$C_w \equiv \{\, c \in C \mid \exists\, n \in N_w : \sigma_w(n) = c \,\}$

— *Bound workflow*:
A workflow whose input data variables are bound to data objects identifiers registered in the data catalog.
Formally, a workflow $\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$ where

- $\forall\, dv \in DV_i$ that is an input data variable $\exists\, d \in D : \langle dv, d \rangle \in DVB_w$

where DV_i denotes the set of input data variables in the workflow:
$DV_i \equiv \{\, dv \in DV_w \mid \exists\, n \in N_w : \exists\, i \in I : \langle \bot, \bot, dv, n, i \rangle \in L_w \,\}$

— *Configured workflow:*
A workflow where all the parameters of its components have been assigned values.
Formally, a workflow $\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$ where:

- $\forall\, pv \in PV_w : \exists\, v \in V : \langle pv, v \rangle \in PVB_w$

— *Ground workflow:*
A workflow where all the data variables in the workflow have been assigned data object identifiers.
Formally, a workflow $\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$ where:

- $\forall\, dv \in DV_w : \exists\, d \in D : \langle dv, d \rangle \in DVB_w$

With these definitions at hand, we can make the following distinction:

— *Workflow instance*.

A specialized bound workflow which is also configured. A ground workflow is a special case of workflow instance where all data variables are bound to data objects with assigned identifiers.

— *Workflow template*.
Any workflow that is not a workflow instance.

The workflow in Figure 2 is not specialized, since both nodes contain abstract components. It is partially bound, since it has a binding for the training data variable but not for the test data variable. It is partially configured, since only some of its parameter variables have been assigned values. This workflow is a workflow template, and we will use it as a running example to show how our algorithms use it to create a workflow instance by specializing, binding, and configuring it. The workflow in Figure 3 is specialized, partially bound, and partially configured.

A workflow library containing reusable workflows can include any kind of workflow template, whether they are fully or partially specialized, bound, or configured. Different kinds of workflow templates can be reused for different purposes. For example, a fully configured workflow can be reused to process different datasets, while a fully bound workflow can be reused to explore alternative parameter settings.

In contrast with workflow templates, workflow instances are fully specified in terms of data, parameters, and components to be used. Therefore, workflow instances can be submitted to a workflow mapping and execution engine to be mapped to available execution resources and to be subsequently executed.

In order to be submitted to the workflow mapping and execution system, workflow instances need to have unique data object identifiers for each new data product as well as exact command line invocations for each component. The workflow mapping and execution system does not need the DODs and other constraints that may be included in the workflow as a result of its evolution from a workflow template to a workflow instance. It only needs to have a unique identifier for each new dataset that will result from the execution of the workflow, specific mention of codes to be executed for each component, and an invocation command to invoke each component code. We refer to these workflows as *ground workflow instances*, where only the basic structure of the workflow is given and no data variables or metadata are included.

A workflow catalog can support analogous functions to component catalogs:

- **is-configurable**(workflow)
- **configure**(workflow)
- **is-backward-enabled**(workflow)
- **find-DODs-given-output-requirements**(workflow)
- **is-forward-enabled**(workflow)
- **predict-DODs-given-input-requirements**(workflow)
- **estimate-performance**(workflow)

The functions **find-DODs-given-output-requirements** and **predict-DODs-given-input-requirements** can be used to generate requirements on inputs and predictive metadata on outputs respectively at the workflow level, as an alternative to finding requirements component by component as we will explain in detail later. Similarly to components, we refer to the workflows in the workflow catalog as *self-configurable*, *forward-enabled*, and *backward-enabled* when the corresponding functions are supported.

Later on, we will show how these functions can be used during workflow generation.

3.4. Workflow Requests: Formalization

We discussed earlier a broad range of requirements that users could provide to a workflow system. We focus here on a particular kind of requests, namely those where a workflow template is given by the user to provide functional and structural properties of the answer to be found by the system. Together with a workflow template from the library, a seed is specified that further constrains data and parameter variables. A given request may contain several pairs of templates and seeds when the user would like the system to consider several templates as a starting point to find the solution. By specifying a template, the user is providing structural properties as indicated by the relative ordering of the steps in the template. Also through the workflow template, the user can provide functional properties since the template specifies component types to be used as well as constraints on data variables.

Formally, given a workflow template library L, a component catalog C, and a data catalog D, a request WR is defined as a pair of a workflow template and a seed:

$$\langle w_r, S_r \rangle$$

where $w_r \in L$ is a workflow template defined by a tuple

$$\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$$

and the seed S_r is defined by a tuple:

$$\langle DV_r, PV_r, M_r, DVB_r, PVB_r \rangle$$

where:

- DV_r is a set of data variables for the seed. A subset of these variables may be specified to be input variables $IDV_r \subseteq DV_r$ and another subset may be specified to be output variables $ODV_r \subseteq DV_r$.
- PV_r is a set of parameter variables for the seed.
- M_r is a (possibly empty) set of DODs using the variables in DV_r and in PV_r
- DVB_r is a (possibly empty) set of bindings of the workflow data variables to data object identifiers:

  $$\langle dv, d \rangle \equiv \langle dv, \text{has-value}, d \rangle$$
  $$dv \in DV_r, d \in D$$

- PVB_r is a (possibly empty) set of bindings of the parameter variables to values:

  $$\langle pv, v \rangle \equiv \langle pv, \text{has-value}, v \rangle$$
  $$pv \in PV_r, v \in V$$

We use the following definitions:

— *Unified request*:
A request where the data and parameter variables in the seed correspond to the data and parameter variables of the workflow template specified in the request.

Formally, $DV_r \subseteq DV_w$ and $PV_r \subseteq PV_w$

— *Well-formed request*:
A request where any bindings and values in the seed for data and parameter variables do not overlap with the bindings specified in the workflow template.

Formally:

- $DVB_r \cap DVB_w = \varnothing$

- $PVB_r \cap PVB_w = \varnothing$

— *Bound request*:
A unified well-formed request where the seed and the workflow template provide bindings for all the input data variables of the workflow template, and the bindings do not overlap.

Formally:

- $DVB_r \cap DVB_w = \varnothing$
- $\forall$ dv $\in DV_w$ then $\exists$ <dv, d> $\in DVB_r \cup DVB_w$

— *Configured request*:
A unified well-formed request where the seed and the workflow template provide values for all the parameter variables of the workflow template specified in the request, and the value assignments do not overlap.

Formally:

- $PVB_r \cap PVB_w = \varnothing$
- $\forall$ pv $\in PV_w$ then $\exists$ <pv, v> $\in PVB_r \cup PVB_w$

— *Seedless request*:
A request where the seed is empty.

Table 5 shows an example request specifying the workflow to be used (ModelerThenClassifier) and providing DODs on a data variable (ClassificationDataVariable) and a binding for a parameter variable (ClassIndexParameterVar).

We define an additional type of workflow:

— *Seeded workflow*, or workflow seeded with a request:
A workflow where the DODs for the variables in the request have been combined with the DODs of the workflow template specified in the request, and the bindings and parameter values in the request have been combined with those of the workflow template specified in the request. In order to create a seeded workflow, the request has to be unified and well-formed.

Formally, a request with the workflow template:

$\langle N_w, \sigma_w, DV_w, PV_w, M_w, L_w, PL_w, DVB_w, PVB_w \rangle$

and the seed:

$\langle DV_r, PV_r, M_r, DVB_r, PVB_r \rangle$

results in the seeded workflow:

$\langle N_w, \sigma_w, DV_w, PV_w, M_n, L_w, PL_w, DVB_n, PVB_n \rangle$

where:

$DV_r \subseteq DV_w$

$DVB_r \cap DVB_w = \varnothing$

$PV_r \subseteq PV_w$

$PVB_r \cap PVB_w = \varnothing$

$M_n =$ **d:combine-DODs**(M_r, M_w)

$DVB_n = DVB_r \cup DVB_w$

$PVB_n = PVB_r \cup PVB_w$

```
<ModelerThenClassifier,
  <{ClassificationDataVariable}, {ClassIndexParameterVar},
   {<ClassificationDataVariable  hasDomain weather> <ClassificationDataVariable  hasType Classification>},
   {}
   {<ClassIndexParameterVar 5>}>>
```

Table 5. Formal representation of a request as a pair of a workflow template and a seed tuple.

Typically a request will contain a single template and a single seed. We generalize this by allowing a request to contain several template and seed pairs. We refer to the former as an *atomic seed* and the later as a *composite seed*. By specifying a composite seed, the user would intend to provide the system with a broader pool of candidate workflows to search through in generating a solution. There is another reason to support workflow generation from composite seeds. If a workflow template were not specified in the request and only the seed was, the system would have to retrieve relevant templates from the workflow library, which would often result on several matching templates. At that point, the workflow generation process would be started using a composite seed formed by each of the templates combined with the original seed.

4. Automatic Template-Based Workflow Generation

The workflow generation process starts with a request containing several template/seed pairs. We assume we start with a unified and well-formed request, that is, the variables that appear in the seed are a subset of the variables in the workflow template and any bindings specified in the seed do not overlap with the bindings specified in the workflow template.

Throughout this section, we use the following conventions. The variables of the algorithm are shown in italics. The functions shown in all capital letters are elaborated in later subsections. We assume some functions have been defined with the prefixes "get-" or "set-" on workflow and request data structures to access their individual constituents. The function calls in boldface are functions supported by external catalogs, using **c:** as a prefix for function calls to external component services, **d:** for metadata services, and **w:** for workflow services.

Since the algorithms perform function calls to external services, it includes provisions for function calls returning a special error code (the empty set) when there is either some error in the inputs to the function call or the function is undefined for those inputs. In such cases, the algorithms reject the workflow being considered as a candidate. These may be indications that the external catalogs may need to be extended to refine their models or to include additional components or data objects.

4.1. Overview of Template-Based Workflow Generation

Figure 5 shows an overview of the distinct stages during workflow generation. A pool of workflow candidates is formed from the initial request. Each stage adds increasingly more detail to each candidate workflow until they are ready to be submitted to the workflow mapping and execution engine. In this process candidate workflows can be added or eliminated. If at any point there are no workflow candidates remaining, the algorithm ends returning an empty result.

The initial request is assumed to contain template/seed pairs that are each well-formed and unified with the template variables. In the first stage, a seeded workflow is created from each template/seed pair by merging the seed with the workflow template constraints. These seeded workflows are considered to be the initial pool of candidate workflows. The next stage propagates constaints from the workflow outputs to the workflow inputs to create binding-ready workflows. Next, input data sources that satisfy the constraints imposed by the workflow are found to create a pool of candidate bound workflows. In the next stage, the properties of input data sources are propagated through each component, resulting on configured workflows. Finally, unique identifiers for workflow data products are obtained to create workflow instances, and specific command invocations are associated with each workflow component to create ground workflows. Finally, the candidate workflows are ranked and the k-best candidates are submitted to the workflow mapping and execution engine.

4.2. Top-Level Algorithm

Table 6 describes the top-level algorithm for automatic template-based workflow generation. The algorithm analyzes workflow candidates at each level on a breadth-first manner, that is, all candidates are elaborated before proceeding to elaborate workflows at the next level of detail. A depth-first search version of the algorithm is also a possible alternative. In either case, the approach we take is to generate all possible candidates, and then rank them to select the top choices. This is needed because ranking candidates properly requires that the workflow is specialized and configured.

The algorithm begins by creating a seeded workflow from each of the template/seed pair in the request. If there were any errors seeding the workflow, due to inconsistencies in the definition of the seed and the template, the call to SEED-WORKFLOW would return an empty set and the whole procedure would be terminated.

Next, the algorithm elaborates the workflows and in the process it will find the requirements on input data. For each candidate workflow, it will start from the output links and retrieve any additional constraints on the workflow variables that are required in order to produce the required output. We refer to this process as a *backward sweep*. This is done within the algorithm BACKWARD-SWEEP, which we describe in detail below. When the workflows contain abstract components, the backward sweep algorithm may find more specific component classes that are appropriate to satisfy the requirements. When this occurs, several candidate workflows are returned. At the end of the backward sweep, the original DODs of the workflows have been augmented and can be used to find datasets that match the request and workflow requirements. We refer to these as *binding-ready workflows*.

Now that there are as many constraints on the input data as could be uncovered in the backward sweep, the algorithm retrieves appropriate input data sources. This is done by the algorithm SELECT-INPUT-DATA-OBJECTS, which essentially generates bindings for all the input data variables of the workflow that are not bound in the original request. Because there may be several alternative datasets that are appropriate, several alternative bindings may be found and in that case several candidate workflows are returned. For each workflow candidate, all the properties of the input data sources that may be relevant to the request are incorporated into the workflow. At the end of this process, the workflow candidates are all bound.

Next, the algorithm elaborates the workflows by propagating the properties of input data sources through each step of the workflow. Starting from the input links, it will retrieve any additional constraints on workflow variables that result from the properties of input data sources. This process is called a *forward sweep*. It is done with the algorithm

FORWARD-SWEEP, which is described below. These workflows have augmented DODs that result from propagating the properties of the input data, and we refer to them as *elaborated workflows*. The workflow may still contain abstract components, and the forward sweep algorithm would specialize them. This results on several candidate workflows being returned. The forward sweep also assigns values to all the parameters of the workflow components based on the constraints that are known for the workflow variables at each step. At the end of this process, the workflow candidates are all specialized and configured in addition to being elaborated.

Next, all the candidate workflows are ranked based on estimates of their performance. This ranking function only takes into account a rough estimate of the execution time of a component based on characteristics of the data. It does not take into account the different performance across architectures or other characteristics of the execution host such as memory availability. It also does not take into account how the workflow performance is affected by data movements, queue wait times, and other execution delays. Such finer-grained estimates are produced by the workflow mapping and execution system and are not discussed here. The rough estimates used at this stage are generated by the algorithm ESTIMATE-PERFORMANCE, described below. The k-best workflows are returned.

The algorithm then proceeds to ground the selected workflows by assigning unique logical identifiers to variables in the workflow that are not input data variables nor parameter variables. For each intermediate and final link in the workflow, its corresponding variable will be assigned a unique identifier using the DODs that describe its properties. During this step the invocation command for each component is also formulated. This is done by the algorithm INSTANTIATE-WORKFLOW described below. All workflow candidates will then be ground and ready to be formatted for submission to the workflow mapping and execution system by extracting from the workflow instance only the information required for a ground workflow.

4.3. Generating Seeded Workflows

First, the DODs of the seed and the DODs of the workflow template of the request are combined. If the DODs of the seed and the workflow are inconsistent, the call to the metadata services to combine these DODs will indicate an error by returning an empty set. In that case, an empty seeded workflow is returned to the top-level algorithm. Next, the data variable bindings of the seed and the workflow template are combined. Finally, the parameter bindings of the seed and the workflow are combined. Since we assume that the request is unified and well-formed, no errors will occur when merging the bindings. The result of this stage is a seeded workflow.

4.4. Backward Sweep

The backward sweep can obtain the constraints on the input data variables in two different ways. One way is to use workflow services. These services would propagate the constraints at the workflow level and would not necessarily reason about constraints for the intermediate variables in the workflow. Another way is to use component services. The algorithm would have to walk though the workflow nodes and propagate constraints component by component by invoking functions implemented by the component catalog for each of the components.

Table 8 shows the algorithm for the BACKWARD-SWEEP function using the workflow services. A single function that takes the whole workflow as an argument will return any additional DODs including DODs on input data variables but may also contain DODs on intermediate data variables when appropriate.

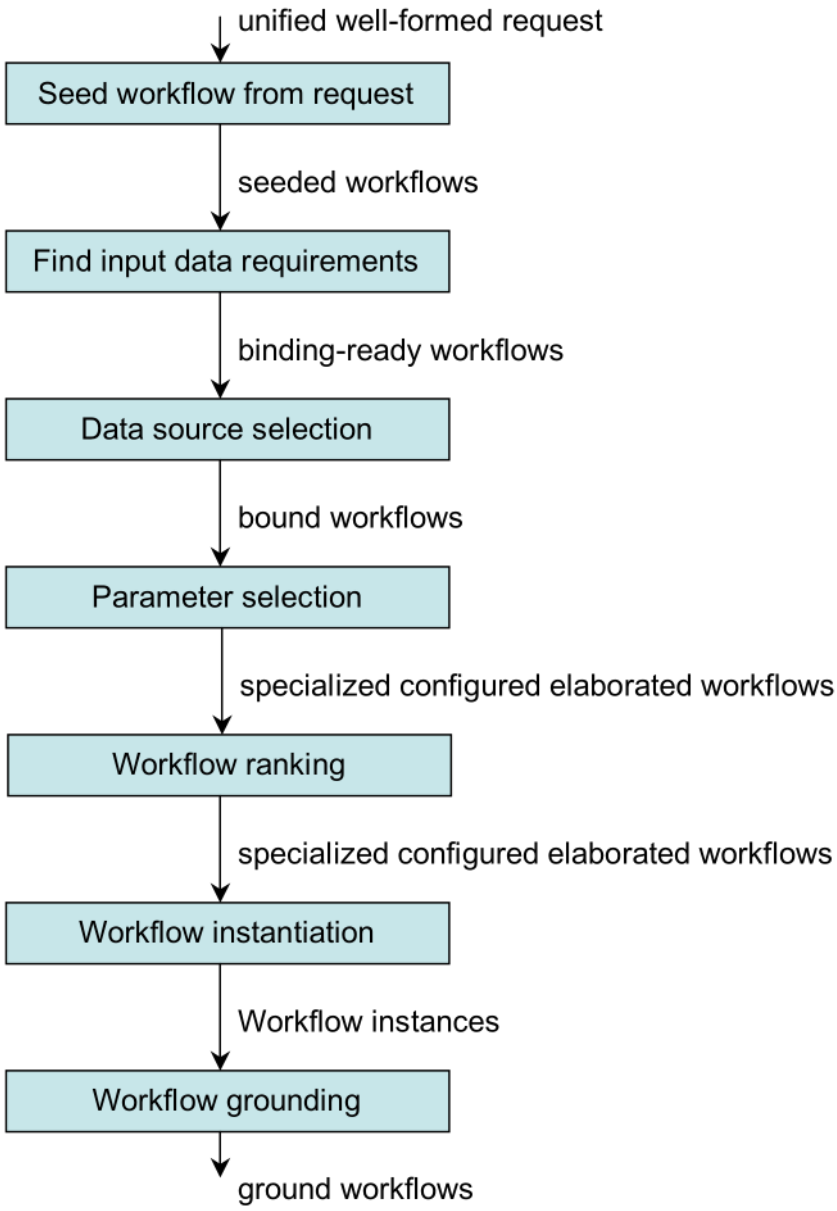


Figure 5: Stages During Workflow Generation.

Table 9 shows the algorithm for the BACKWARD-SWEEP function using the component services. For each node in the candidate workflow, it traverses the workflow from end results to initial inputs. For each of the nodes visited, the algorithm processes together all the links that have that node either as an origin or as a destination. This is because some workflow nodes are origin to more than one link. In such cases, we need to gather all the DODs on workflow variables that constraint the parameters of that node's component as we traverse the workflow. If the component is abstract, all the possible specializations from that abstract component class are obtained. These could be either more specific component classes or concrete components. When more than one specialization is returned, more than one specialized workflow will be created for the initial seeded workflow. All additional DODs that the component places on its arguments and that are returned by the component catalog are added to the workflow DODs. This may include parameter values that can be set during this step as constraints on input and output arguments of the component are introduced by the workflow.

Algorithm: **TEMPLATE-BASED-WORKFLOW-GENERATION**

Input: *request*
Output: *workflow-instances*

```
    Seed Workflows from Request
1     seeded-workflows ← {}
2     for each template-seed-pair ∈ request do
3         workflows ← SEED-WORKFLOW(template-seed-pair)
4         if (workflows ≠ null) then
5             seeded -workflows ← seeded -workflows ∪ workflows
6     if (seeded -workflows = null) then workflow-instances ← {}; return

    Find Input Data Requirements
7     binding-ready-workflows ← {}
8     for each seeded-workflow ∈ seeded-workflows do
9         workflows ← BACKWARD-SWEEP(seeded-workflow)
10        if (workflows ≠ null) then
11            binding-ready-workflows ← binding-ready-workflows ∪ workflows
12    if (binding-ready-workflows = null) then workflow-instances ← {}; return

    Data Source Selection
13    bound-workflows ← {}
14    for each binding-ready-workflow ∈ binding-ready-workflows do
15        workflows ← SELECT-INPUT-DATA-OBJECTS(binding-ready-workflow)
16        if (workflows ≠ null) then
17            bound-workflows ← bound-workflows ∪ workflows
18    if (bound-workflows = null) then workflow-instances ← {}; return

    Parameter Selection
19    configured-workflows ← {}
20    for each bound-workflow ∈ bound-workflows do
21        workflows ← FORWARD-SWEEP(bound-workflow)
22        if (workflows ≠ null) then
23            configured-workflows ← configured-workflows ∪ workflows
24    if (configured-workflows = null) then workflow-instances ← {}; return

    Workflow Ranking
25    ranked-workflows ← {}
26    for each configured-workflow ∈ configured-workflows do
27        workflow ← ESTIMATE-PERFORMANCE(configured-workflow)
28        ranked-workflows ← ranked-workflows ∪ {workflow}
29    ranked-workflows ← select-k-best(ranked-workflows)

    Workflow Instatiation and Grounding
30    workflow-instances ← {} ground-workflows ← {}
31    for each ranked-workflow ∈ ranked-workflows do
32        workflow ← INSTANTIATE-WORKFLOW(ranked-workflow)
33        workflow-instances ← workflow-instances ∪ {workflow}
34        ground-workflows ← GROUND-WORKFLOW(workflow)
354   return ground-workflows
```

Table 6. Top-Level Algorithm for Automatic Template-Based Workflow Generation.

Algorithm: **SEED-WORKFLOW**

Input: *template-seed-pair*
Output: *seeded-workflow*

workflow ← get-template*(template-seed-pair)*
seed ← get-seed*(template-seed-pair)*
Combine DODs of the seed with the workflow DODs
workflow-DODs ← get-DODs*(workflow)*
seed-DODs ← get-DODs*(seed)*
new-DODs ← **d:combine-DODs**(*workflow-DODs, seed-DODs*)
If the DODs are inconsistent, reject the current workflow
if *new-DODs* = {} then
workflow ← {}
else
set-DODs*(workflow*, *new-DODs*)
Combine the data variable bindings of the seed with the workflow data variable bindings
workflow-data-var-bindings ← get-data-var-bindings(*workflow)*
seed-data-var-bindings ← get-data-var-bindings(*seed)*
new-data-var-bindings ← *workflow-data-var-bindings* ∪ *seed- data-var-bindings*
set-data-var-bindings(*workflow*, *new-data-var-bindings*)
Combine the parameter bindings of the seed with the workflow parameter bindings
workflow-par-var-bindings ← get-par-var-bindings(*workflow)*
seed-par-var-bindings ← get-par-var-bindings(*seed)*
new-par-var-bindings ← *workflow-par-var-bindings* ∪ *seed-par-var-bindings*
set-par-var-bindings(*workflow*, *new-par-var-bindings*)
return *workflow*

Table 7: Algorithm for Seeding a Workflow Template with the Seed Given in the Request.

Algorithm: **BACKWARD-SWEEP-THROUGH-WORKFLOW**

Input: *seeded-workflow*
Output: *binding-ready-workflows*

workflow ← *seeded-workflow*
new-DODs ← **w:find-DODs-given-output-requirements**(*workflow*)
when (new-DODs ≠ {})
set-DODs(*workflow*, get-DODs(*workflow*) ∪ *new-DODs*)
binding-ready-workflows ← {*workflow*}
return *binding-ready-workflows*

Table 8: Algorithm for Backward Sweep Through Workflow.

For simplicity, the algorithm in Table 10 assumes that each node in the workflow is origin to only one link. In cases where there is more than one link with the node as origin, the algorithm will only proceed to specialize the component in a node when all links relevant to the outputs of a node have been processed. That is, it ensures that the paths from the outputs to that node have already been fully processed.

Algorithm: **BACKWARD-SWEEP-THROUGH-COMPONENTS**

Input: *seeded-workflow*
Output: *binding-ready-workflows*

```
1   workflow-queue ← seeded-workflow
2   binding-ready-workflows ← {}
3   while (workflow-queue ≠ {}) do
4     workflow ← dequeue(workflow-queue)
7     link-queue ← get-output-links(workflow)
8     while (link-queue ≠ {} & workflow ≠ {}) do
9       link ← dequeue(link-queue)
10      when (current-link ∉ get-input-links(workflow))
11        node ← get-origin(link)
      Find all links (going sideways) that have the current node as the origin node
12        links-shared-origin ← l s.t. l ∈ get-links(workflow) & get-origin(l) = node
13        link-queue ← link-queue \ links-shared-origin
      Find all links (going upstream) that have the current node as the destination node
14        links-shared-dest ← l s.t. l ∈ get-links(workflow) & get-destin(l) = node
15        link-queue ← link-queue \ links-shared-dest
      Create a set with all those links that have the current node as origin or destination
16        links-current-node ← links-same-origin ∪ links-dest-is-origin
      Find all the DODs in the workflow that are relevant to the current node
17        vars ← get-data-vars(workflow) ∪ get-param-vars(workflow)
18        vars-node ← v s.t. v ∈ vars & l ∈ links-current-node & get-variable(l) = v
19        node-DODs ← entity-DODs(vars-node, get-DODs(workflow))
20        comp ← get-node-component(node)
      Map DODs on workflow variables to DODs on arguments of the node's component
21        comp-DODs ← find-comp-DODs(node,comp,node-DODs,links-current-node)
      If the node's component is not concrete, get specializations of the component
      and create a new workflow candidate with each of the specializations obtained
22        if (not c:is-concrete(comp)) then
23          concrete-components ← c:specialize(comp,comp-DODs)
      If no specialization of the component can satisfy the requirements, reject the current workflow
24            when (concrete-components ≠ {})
25            for each cc ∈ concrete-components do
26              copy ← copy(workflow)
27              copy ← replace(comp,cc, node,copy)
28              workflow-queue ← workflow-queue ∪ copy
29        else
30          comp-input-DODs ← c:find-DODs-given-output-requirements(comp,comp-DODs)
      If no DODs can satisfy the requirements on the component, reject the current workflow
31          if comp-input-DODs = {}
32            workflow ← {}
33          else
      Map DODs on arguments of the node's component to DODs on workflow variables
34            var-DODs ← find-variable-DODs(vars,node,comp,comp-input-DODs)
35            set-DODs(current-workflow) ← get-DODs(workflow) ∪ var-DODs
36    end while over link-queue
37  when (workflow ≠ {})
38        binding-ready-workflows ← binding-ready-workflows ∪ workflow
39  end while over workflow-queue
40  return binding-ready-workflows
```

Table 9: Algorithm for Backward Sweep Through Components.

Algorithm: **SELECT-INPUT-DATA-OBJECTS**

Input: *binding-ready-workflow*
Output: *bound-workflows*

```
1   bound-workflows ← {}
2   input-links ← get-input-links(specialized-workflow)
3   input-data-variables ← get-variables(input-links)
4   input-DODs ← get-variables(input-data-variables)
5   input-bindings ← d:find-data-objects(input-DODs)
6   for each binding ∈ input-bindings do
7       workflow ← copy(specialized-workflow)
8       set-data-variable-bindings(workflow, get-data-variable-bindings(workflow) ∪ binding)
9       data-objects ← get-data-objects(input-bindings)
10      for each data-object ∈ data-objects do
11          additional-DODs ← d:obtain-dataset-characteristics(data-object)
12          set-DODs(workflow, get-DODs(workflow) ∪ additional-DODs)
13      bound-workflows ← bound-workflows ∪ {workflow}
14  return bound-workflows
```

Table 10: Algorithm for Binding Workflows by Selecting Input Data.

Algorithm: **FORWARD-SWEEP-THROUGH-WORKFLOW**

Input: *bound-workflow*
Output: *configured-workflow*

```
1   workflow ← bound-workflow
2   new-DODs ← w:predict-DODs-given-input-requirements(workflow)
3   when (new-DODs ≠ {})
4       set-DODs(workflow, get-DODs(workflow) ∪ new-DODs)
5   if is-configured(workflow)
6       configured-workflow ← workflow
7   else
8       configured-workflow ← null
9   return configured-workflow
```

Table 11: Algorithm for Forward Sweep Through Workflow.

When using the wokflow services for the backward sweep, any abstract components of the workflow will not be specialized. Therefore, when using workflow services for the backward sweep the workflow template specified in the request must be a concrete workflow.

The result of the backward sweep is a set of candidate workflows that are all binding-ready workflows.

4.5. Selecting Input Data

This algorithm starts with a binding-ready workflow that includes DODs on all input data variables. First, it finds available data objects that match those DODs. There can be several combinations of data object for input data variables, and in that case several sets of

bindings are returned. In that case, a bound workflow will be created for each set of bindings. If there are no matching data sources then the workflow is rejected and an empty workflow is returned.

Note that there is a single query to the data catalog for a given workflow, rather than a query per input data variable. This ensures that any constraints among input data variables are taken into account by the data catalog during the matching of input data sources.

Next, the algorithm requests from the metadata services all additional DODs of the selected input data objects. There may be arbitrarily many possible properties of a data object and there may be a cost to generating the values of some of the properties. Ideally, this function would be invoked in a selective and cost-sensitive manner though this is not addressed in our current work.

4.6. Forward Sweep

Like the backward sweep, the forward sweep can be approached in two different ways. One approach is to use workflow services. These services would propagate the constraints on input data variables at the workflow level and would not necessarily reason about constraints for the intermediate variables in the workflow. Another approach is to use component services. The algorithm would have to walk though the workflow nodes and propagate constraints component by component by invoking functions implemented by the component catalog for each of the components.

Table 11 shows the FORWARD-SWEEP algorithm using the workflow services. A single function that takes the whole workflow as an argument will return additional DODs on output and intermediate data variables.

Table 12 shows the FORWARD-SWEEP algorithm using the component services. For each node in the candidate workflow, it traverses the workflow from initial inputs to end results. For each of the nodes visited, the algorithm processes together all the links that have that node either as an origin or as a destination. This is because some workflow nodes are the destination of more than one link. In such cases, we need to gather all the DODs on workflow variables that constraint the parameters of that node's component as we traverse the workflow. In the table, we refer to a function find-comp-DODs that extracts all the DODs relevant to a component expressed in terms of component parameters rather than workflow data variables so that then component catalog can be invoked. If a component is abstract, all the concrete components of that abstract component class are obtained. When more than one concrete component is returned, more than one specialized workflow will be created for the initial bound workflow. In the table we use a function replace-component that replaces the relevant component by one of those returned. All additional DODs that the component places on its arguments are added to the workflow DODs.

For simplicity, as with the backward sweep, the algorithm in Table 12 assumes that each node in the workflow is the destination to only one link. In cases where there is more than one link with the node as destination, the algorithm will only proceed to specialize and find output requirements for the component in a node when all links relevant to the inputs of a node have been processed. That is, it ensures that the paths from the inputs to that node have already been fully processed.

As was the case with the forward sweep, the algorithm that uses the workflow services for the backward sweep does not specialize components. Therefore, when using workflow services for the forward sweep the workflow must be a concrete workflow.

The result of the forward sweep is a set of candidate workflows that are all configured and specialized workflows.

```
Algorithm:  FORWARD-SWEEP-THROUGH-COMPONENTS
Input:      bound-workflow
Output:     configured-workflows
1   workflow-queue ← bound-workflow
1   configured-workflows ← {}
3   while (workflow-queue ≠ {}) do
4     workflow ← dequeue(workflow-queue)
2     link-queue ← get-input-links(workflow)
8     while (link-queue ≠ {} & workflow ≠ {}) do
9       link ← dequeue(link-queue)
10      when (current-link ∉ get-output-links(workflow))
11        node ← get-dest(link)
      Find all links (going sideways) that have the current node as the origin node
12        links-shared-origin ← l s.t. l ∈ get-links(workflow) & get-origin(l) = node
13        link-queue ← link-queue \ links-shared-origin
      Find all links (going upstream) that have the current node as the destination node
14        links-shared-dest ← l s.t. l ∈ get-links(workflow) & get-destin(l) = node
15        link-queue ← link-queue \ links-shared-dest
16        links-current-node ← links-same-origin ∪ links-dest-is-origin
      Find all the DODs in the workflow that are relevant to the current node
17        vars ← get-data-vars(workflow) ∪ get-param-vars(workflow)
18        vars-node ← v s.t. v ∈ vars & l ∈ links-current-node & get-variable(l) = v
19        node-DODs ← d:entity-DODs(vars-node, get-DODs(workflow))
20        comp ← get-node-component(node)
      Map DODs on workflow variables to DODs on arguments of the node's component
21        comp-DODs ← find-comp-DODs(node,comp,node-DODs,links-current-node)
      If the node's component is not concrete, create a new workflow candidate with each specialization
22        if (not c:is-concrete(comp)) then
23          concrete-components ← c:specialize-to-concrete(comp,node-DODs)
      If no specialization of the component can satisfy the requirements, reject the current workflow
24          when (concrete-components ≠ {})
25          for each cc ∈ concrete-components do
26            copy ← copy(workflow)
27            copy ← replace(comp,cc, node,copy)
28            workflow-queue ← workflow-queue ∪ copy
29        else
      If the component is not configured, create a new workflow candidate with each configuration obtained
30        if (not c:is-configured(comp, comp-DODs)) then
31          component-configurations ← c:configure(comp,comp-DODs)
      If no configuration of the component can satisfy the requirements, reject the current workflow
            when (component-configurations ≠ {})
32            for each cc ∈ component-configurations do
33              new ← copy(workflow)
34              new ← replace-component(workflow, cc)
35              workflow-queue ← workflow-queue ∪ copy
36        else
37          comp-o-DODs ← c:predict-DODs-given-input-requirements(comp,comp-DODs)
      Map DODs on arguments of the node's component to DODs on workflow variables
38          var-DODs ← find-variable-DODs(vars,node,comp,comp-o-DODs)
39          set-DODs(current-workflow, get-DODs(workflow) ∪ var-DODs )
40    end while over link-queue
41  when (workflow ≠ {})
42      configured-workflows ← configured-workflows ∪ {workflow}
43  end while over workflow-queue
44  return configured-workflows
```

Table 12: Algorithm for Forward Sweep Through Components.

Algorithm: **ESTIMATE-PERFORMANCE-THROUGH-WORKFLOW**

Input: *ground-workflow*
Output: *ranked-ground-workflow*

```
1   workflow ← ground-workflow
2   set-performance-estimate(workflow, w: estimate-performance(workflow) )
3   return workflow
```

Table 13: Algorithm for Estimating Workflow Performance Through the Workflow.

Algorithm: **ESTIMATE-PERFORMANCE-THROUGH-COMPONENTS**

Input: *ground-workflow*
Output: *ranked-ground-workflow*

```
1   workflow ← ground-workflow
2     link-queue ← get-output-links(workflow)
3     while (link-queue ≠ {} do
4       link ← dequeue(link-queue)
5       when (current-link ∉ get-output-links(workflow))
6         node ← get-dest(link)
      Find all links (going sideways) that have the current node as the origin node
7         links-shared-origin ← l s.t. l ∈ get-links(workflow) & get-origin(l) = node
8         link-queue ← link-queue \ links-shared-origin
      Find all links (going upstream) that have the current node as the destination node
9         links-shared-dest ← l s.t. l ∈ get-links(workflow) & get-destin(l) = node
10        link-queue ← link-queue \ links-shared-dest
11        links-current-node ← links-same-origin ∪ links-dest-is-origin
      Find all the DODs in the workflow that are relevant to the current node
12        vars ← get-data-vars(workflow) ∪ get-param-vars(workflow)
13        vars-node ← v s.t. v ∈ vars & l ∈ links-current-node & get-variable(l) = v
14        node-DODs ← d:entity-DODs(vars-node, get-DODs(workflow))
15        comp ← get-node-component(node)
      Map DODs on workflow variables to DODs on arguments of the node's component
16        comp-DODs ← find-comp-DODs(node,comp,node-DODs,links-current-node)
      Get estimate of the component performance
17        set-predicted-execution-time(node, c:estimate-performance(comp,comp-DODs) )
18    end while over link-queue
19  set-performance-estimate(workflow, estimate-aggregate-performance(workflow) )
20  return workflow
```

Table 14: Algorithm for Estimating Workflow Performance Through Components.

4.7. Estimating Workflow Performance

Like the forward and backward sweeps, estimating wokflow performance can be done using workflow services or component services. Because the estimates using component services would need the DODs for intermediate data products, it is required that the forward sweep should have been done using component services as well.

Algorithm: **INSTANTIATE-WORKFLOW**

Input: *configured-workflow*
Output: *workflow-instances*

```
1   workflow-queue ← configured-workflow
2   workflow-instances ← {}
3   while (workflow-queue ≠ {}) do
4     workflow ← dequeue(workflow-queue)
5     link-queue ← get-output-links(workflow)
6     while (link-queue ≠ {} & workflow ≠ {}) do
7       link ← dequeue(link-queue)
8       when (current-link ∉ get-input-links(workflow))
9         link-DODs ← d:entity-DODs(get-variable(current-link), get-DODs(workflow))
10        id ← d:assign-identifier(link-DODs)
11        binding ← <get-link-variable(link), id>
12        d:assert-predicted-DODs(id,link-DODs)
13        set-workflow-bindings(workflow, get-workflow-bindings(workflow) ∪ binding )
14    end while over link-queue
14    set-invocation-commands(workflow)
15  workflow-instances ← workflow-instances ∪ {workflow}
16  end while over workflow-queue
17  return workflow-instances
```

Table 15: Algorithm for Instantiating Workflows.

The algorithm to estimate performance using component services is shown in Table 14. It walks through the nodes of the workflow, and for each node it gathers the DODs that are relevant to it. Using those DODs, it invokes the component services to retrieve the estimates of performance of the workflow. With the individual estimates for each node, the algorithm then calls a function that aggregates the estimates for the overall workflow (estimate-aggregate-performance). This function finds the longest path between the input links and the output links.

4.8. Instantiating and Grounding Workflows

The algorithm for instantiating workflows, shown in Table 15, traverses a workflow and gathers all the DODs on a data variable and requests from the data catalog a unique identifier for the corresponding execution data product. If the DODs are rich enough, the data catalog will be able to detect when data products are equal and therefore give them the same identifier. This enables data reuse with the benefit of saving computation time, as the workflow execution system can eliminate unnecessary computations that produce already existing data that was produced by previously executed workflows. When the DODs are not rich enough, then reuse will not be possible as each new data product will have its own identifier and there will be no way to detect when data products from different workflows are the same (unless the workflows are identical). This can happen in the case where the forward sweep proceeds through workflows rather than through components.

During grounding of workflows, the invocation command is set for each of the node's components. A function is shown in the table that applies to the whole workflow, within that function there is an invocation of the component catalog for each node's component using all the DODs that are relevant to the variables in links adjacent to the node.

The final grounding step essentially extracts a small subset of the information in the workflow instance to create a ground workflow that can be submitted to the workflow

mapping and execution engine. An example of a workflow instance and its corresponding ground workflow is shown in the next section.

4.9. Summary of Reasoning Requirements for Data and Catalog Services

Table 16 summarizes the functions for data and catalog services invoked by the workflow generation algorithm. For each function, we indicate the use of that function in the algorithm.

Table 17 highlights the main types of reasoning needed for automatic template-based workflow generation and the corresponding requirements on the data, component and workflow catalogs. It shows the kind of workflow needed to carry out the reasoning, and what kind of workflow is produced. Reasoning about component constraints is done first in the backward sweep, starting with a seeded workflow and ending with a binding-ready workflow. The calls to the component catalog obtain the constraints of each component, but the workflow generation algorithm takes care of their propagation throughout the workflow and of their integration with seed constraints. Data selection starts with a binding-ready workflow to produce a bound workflow. Notice that a binding-ready workflow contains constraints relevant to the seed and the components, however the selected datasets may have additional characteristics that can affect how the components behave. Those characteristics are obtained in the data characterization step, and result in a workflow that characterizes all its input datasets. With such a workflow, the algorithm proceeds to reason again about component constraints in the forward sweep to produce a workflow that states all the predicted characteristics for all the datasets in the workflow. This information is needed to reason about component configuration to produce a configured workflow. It is also needed for easoning about data identifier assignments, since the datasets can be described by their characteristics and assigned a unique identifier that enables future reuse in other workflows as well as retrieval by other users. We note that reasoning about component constraints and component configuration occurs jointly for each workflow component during the forward sweep. This is because the parameter settings often determine the characteristics of the output datasets for a component. For domains where this is not the case the two kinds of reasoning can be done separately.

The next section walks through the main steps of the algorithm with an example of a workflow request.

5. A Walkthrough Example of Workflow Generation

We now show an example of how workflow candidates are generated from a workflow request using the algorithm just presented. We present an example that runs end-to-end in our implemented system, and show through the representation of candidate workflows at each stage. We use different namespaces to refer to terms that are defined in different catalogs. Therefore, the workflow catalog, component catalog, and data catalog will have different namespaces. We use the W3C Ontology Web Language OWL (www.w3.org/TR/owl-features) to represent workflows and DODs, but will show the examples using N3 notation.

Table 18 shows the representation of the workflow template for `ModelerThenClassifier` shown in Figure 2. The workflow contains two nodes for a modeler and a classifier. There are six links that represent inputs and outputs of the two nodes. Note that the workflow in Figure 2 shows the heap size as a parameter of the classifier, which is not used in this example.

Reasoning required	Description	Workflow needed	Functions required from data catalogs	Functions required from component catalogs	Workflow produced
Component constraints – for backward sweep	What kinds of input data are required given output requirements	Seeded workflow	-	c:find-DODs-given-output-requirements	Binding-ready workflow
Dataset selection	How to select valid input data	Binding-ready workflow	d:find-data-objects	-	Bound workflow
Dataset characterization	What kinds of input data are needed for the workflow	Bound workflow	d:obtain-dataset-characteristics	-	Worfklow with characteristics of input datasets
Component constraints – for forward sweep	What kinds of output data will be produced given input characteristics	Worfklow with characteristics of input datasets	-	c:predict-DODs-given-input-requirements	Workflow with predicted characteristics of all datasets
Component configuration	How to set up parameter values	Workflow with predicted characteristics of all datasets[1]	-	c:is-configured c:configure	Configured workflow
Dataset identifier assignments	How to assign unique identifiers for semantically equivalent datasets	Workflow with predicted characteristics of all datasets	d:assign-identifier d:assert-predicted-DODs	-	Workflow instance

Table 17: Types of reasoning needed for automatic template-based workflow generation and the corresponding requirements placed on external data and component services to supplement what a user may not specify in the workflow request. Our algorithm combines the forward propagation of constraints with the component configuration, since parameters are often important to determine predictive characteristics of output datasets.

Function in Metadata Services	Purpose in Automatic Generation Process
d:combine-DODs	Seed workflow templates
d:assign-identifier	Create unique identifiers and properties for workflow data products so they can be reused in future workflows
d:assert-predicted-DODs	
d:find-data-objects	Selection of input data enables creation of bound workflows
d:obtain-dataset-characteristics	Propagation of input data properties in forward sweep enables component specialization and workflow candidate elimination

Function in Component Services	Purpose in Automatic Generation Process
c:invocation-command	Ground workflows to be submitted for execution
c:is-concrete	Use of abstract components in workflow templates that can be specialized in backward and forward sweep
c:specialize	
c:specialize-to-concrete	
c:is-backward-enabled	Generate binding-ready workflows in backward sweep
c:find-DODs-given-output-requirements	
c:is-forward-enabled	Generate configured workflows in forward sweep
c:predict-DODs-given-input-requirements	
c:is-configurable	
c:configure	
c:is-configured	
c:estimate-performance	Rank candidate workflows

Table 16: Summary of functions that need to be supported in the metadata services and the component services to enable automatic template-based workflow generation.

Table 19 shows an example of a representation of a workflow request. It specifies the workflow to use (ModelerThenClassifier) and provides additional DODs on an output data variable and a parameter variable. In particular, the output of the workflow should include a classification of a weather data object (i.e. the domain of the ClassificationDataVariable is Weather) and the value of the ClassIndexParameterVar is 5. This is the same request shown in Table 5.

Given this request, the seeded workflow initially generated by the algorithm is as the original template shown in Table 16 except that it includes the additional DODs on ClassificationDataVariable and ClassIndexParameterVar that are introduced by the request. Table 20 shows a relevant excerpt of the seeded workflow, where the additions to the origninal workflow template are highlighted in bold face.

```
ModelerThenClassifier     a      wflow:WorkflowTemplate ;
    wflow:hasNode classifierNode , modelerNode ;
    wflow:hasLink modelerTrainingDataInputLink, modelerJavaMaxHeapInputLink,
        classifierOutputLink , modelerOutputClassifierInputInOutLink,
        modelerClassIndexInputLink , classifierDataInputLink .

modelerNode     a      wflow:Node ;     wflow:hasComponent ac:Modeler.
classifierNode     a      wflow:Node ;     wflow:hasComponent ac:Classifier .

modelerTrainingDataInputLink     a      wflow:InputLink ;
    wflow:hasDestinationNode modelerNode ;
    wflow:hasDestinationArgument ac:d ;
    wflow:hasVariable TrainingDataVariable .

TrainingDataVariable     a      dcdm:Instance , wflow:DataVariable .

maxJavaHeapSizeModelerParameterVar     a      wflow:ParameterVariable .

modelerJavaMaxHeapInputLink     a      wflow:InputLink ;
    wflow:hasDestinationNode  modelerNode ;
    wflow:hasDestinationArgument ac:j ;
    wflow:hasVariable maxJavaHeapSizeModelerParameterVar .

ClassIndexParameterVar     a      wflow:ParameterVariable .

modelerClassIndexInputLink     a      wflow:InputLink ;
    wflow:hasDestinationNode    modelerNode ;
    wflow:hasDestinationArgument ac:i ;

ModelDataVariable     a      dcdm:Model , wflow:DataVariable .

modelerOutputClassifierInputInOutLink     a      wflow:InOutLink ;
    wflow:hasDestinationNode      classifierNode ;
    wflow:hasDestinationArgument ac:m ;
    wflow:hasOriginNode modelerNode ;
    wflow:hasOriginArgument ac:o ;
    wflow:hasVariable ModelDataVariable .

classifierDataInputLink     a      wflow:InputLink ;
    wflow:hasDestinationNode classifierNode ;
    wflow:hasDestinationArgument ac:d ;
    wflow:hasVariable TestDataVariable .

TestDataVariable     a      dcdm:Instance , wflow:DataVariable ;
    dcdm:notSameObject TrainingDataVariable;
    wflow:hasVariable modelerClassIndex .

classifierOutputLink     a      wflow:OutputLink ;
    wflow:hasOriginNode classifierNode ;
    wflow:hasOriginArgument ac:o ;
    wflow:hasVariable ClassificationtDataVariable .

ClassificationDataVariable     a      wflow:DataVariable , dcdm:Classification .
```

Table 18: A workflow example in N3 notation.

```
owl:imports ModelerThenClassifier.owl  // use ModelerThenClassifier workflow

ClassificationDataVariable a dcdm:Classification ;
    dcdm:hasDomain dcdm:weather.

ClassIndexParameterVar wflow:hasParameterValue 5.
```

Table 19: Example of workflow request in N3 notation.

```
…
ClassIndexParameterVar   a   wflow:ParameterVariable ;
    wflow:hasParameterValue 5.  // from the request
…
ClassificationDataVariable   a   wflow:DataVariable , dcdm:Classification;
    dcdm:hasDomain dcdm:weather. // from the request
…
```

Table 20: Relevant excerpts of a seeded workflow.

Table 21 shows an example of a binding-ready workflow generated as a candidate after the backward sweep. The original Modeler abstract component has been specialized into LmtModeler and the Classifier into J48Classifier. This specialization introduces some new DODs of the data objects used or created by the components, such as dcdm:hasModelType. The DODs in the original request are propagated by the backward sweep and result in additional DODs on some of the workflow data variables. For example, TrainingDataVariable, ModelDataVariable, and TestDataVariable have a new DOD with a requirement in their domain property that it be weather.

Decision Tree classifiers can use Decision Tree models only and Bayes classifiers can only use Bayes models. Assuming a component catalog that includes three Decision Tree modelers (J48Modeler, LmtModeler, ID3Modeler), three Decision Tree classifiers(J48Classifier, LmtClassifier, ID3Classifier), three Bayes modelers(BayeNetModeler, NaiveBayesModeler, HBNModeler) and three Bayes classifiers (BayeNetClassifier, NaiveBayesClassifier, HBNClassifier and six classifiers), 18 total seeded specialized workflows would be generated as candidates.

The next step of the algorithm finds available data objects for the input data variables. With the workflow in Table 21, the following query for selecting input data objects for two input data variables is generated:

```
TrainingDataVariable   a   dcdm: Instance , wflow:DataVariable;
    dcdm:hasDomain dcdm:weather.

TestDataVariable   a   dcdm:Instance , wflow:DataVariable;
    dcdm:notSameObject TrainingDataVariable;
    dcdm:hasDomain dcdm:weather.
```

LmtModelerThenJ48Classifier a wflow:WorkflowTemplate ;
wflow:hasNode classifierNode , modelerNode ;
wflow:hasLink modelerTrainingDataInputLink , modelerJavaMaxHeapInputLink , classifierOutputLink ,
modelerOutputClassifierInputInOutLink , modelerClassIndexInputLink , classifierDataInputLink.

modelerNode a wflow:Node ; wflow:hasComponent **ac:LmtModeler**.
classifierNode a wflow:Node ; wflow:hasComponent **ac:J48Classifier**.

modelerTrainingDataInputLink a wflow:InputLink ;
wflow:hasDestinationNode modelerNode ;
wflow:hasDestinationArgument ac:d ;
wflow:hasVariable TrainingDataVariable ;

TrainingDataVariable
a dcdm:Instance , wflow:DataVariable;
dcdm:hasDomain dcdm:weather.

maxJavaHeapSizeModelerParameterVar a wflow:ParameterVariable .

modelerJavaMaxHeapInputLink a wflow:InputLink ;
wflow:hasDestinationNode modelerNode ;
wflow:hasDestinationArgument ac:j ;
wflow:hasVariable j maxJavaHeapSizeModelerParameterVar .

ClassIndexParameterVar a wflow:ParameterVariable .
wflow:hasParameterValue 5. // from the request

modelerClassIndexInputLink a wflow:InputLink ;
wflow:hasDestinationNode modelerNode ;
wflow:hasDestinationArgument ac:i ;
wflow:hasVariable ClassIndexParameterVar .

modelerOutputClassifierInputInOutLink a wflow:InOutLink ;
wflow:hasDestinationNode classifierNode ;
wflow:hasDestinationArgument ac:m ;
wflow:hasOriginNode modelerNode ;
wflow:hasOriginArgument ac:o ;
wflow:hasVariable ModelDataVariable ;

ModelDataVariable
a dcdm:**LmtModel** , wflow:DataVariable ;
dcdm:hasDomain dcdm:weather ;
dcdm:hasModelType DecisionTree.

classifierDataInputLink a wflow:InputLink ;
wflow:hasDestinationNode classifierNode ;
wflow:hasDestinationArgument ac:i ;
wflow:hasVariable TestDataVariable .

TestDataVariable a dcdm:Instance , wflow:DataVariable;
dcdm:notSameObject TrainingDataVariable;
dcdm:hasDomain dcdm:weather.

classifierOutputLink a wflow:OutputLink ;
wflow:hasOriginNode classifierNode ;
wflow:hasOriginArgument ac:o ;
wflow:hasVariable ClassificationDataVariable .

ClassificationDataVariable a wflow:DataVariable , dcdm:**DtmClassification** ;
dcdm:hasDomain dcdm:weather; // from the request

Table 21: An example binding-ready workflow after the backward sweep.

```
    …
TrainingDataVariable
    a       dcdm:Instance , wflow:DataVariable;
    dcdm:hasDomain dcdm:weather;
    wflow:hasDataBinding dcdm: weather-2007-31-101501.

TestDataVariable
    a       dcdm:Instance , wflow:DataVariable;
    dcdm:hasDomain dcdm:weather;
    wflow:hasDataBinding dcdm:weather-2007-31-155754.
```

Table 22: Relevant excerpts of an example bound specialized workflow.

Other seeded specialized workflow candidates may generate different queries for finding data objects. For example, workflow candidates with BayesModeler or BayesClassifier will need DiscreteInstance as an input. A workflow with a NaiveBayesModeler and a J48Classifier will result in a query with the following data object descriptions:

```
TrainingDataVariable    a    dcdm:DiscreteInstance , wflow:DataVariable;
    dcdm:hasDomain dcdm:weather.

TestDataVariable    a    dcdm:Instance , wflow:DataVariable;
    dcdm:notSameObject TrainingDataVariable;
    dcdm:hasDomain dcdm:weather.
```

With a data catalog with four weather domain datasets (weather-2007-31-101501, weather-2007-31-101503, weather-2007-31-101656, and weather-2007-31-155754) that are all ContinuousInstances, the system will not find matching datasets for the workflows that need DiscreteInstances. In our running example, only 4 of the 18 candidate workflows with Lmt and J48 combinations (LmtModelerThenJ48Classifier, LmtModelerThenLmtClassifer, J48ModelerThen LmtClassifier, J48ModelerThen J48Classifier) will get results from the query to find matching data objects. For each candidate binding-ready workflow, the system produces twelve bindings since TrainingDataVariable and TestDataVariable should be bound to different weather datasets. That is a total of 48 candidate bound workflows generated in our running example. Table 22 shows an example of a bound workflow for LmtModelerThenJ48Classifier. For brevity, only the bindings and the DODs of the input data variables are shown.

The forward sweep sets all the parameter values of components and includes DODs for workflow data products. After that, the grounding step introduces data object identifiers for intermediate and final workflow data variables. Table 23 shows an example of a resulting workflow instance. The value of maxJavaHeapSizeModelerParameterVar is set in proportion to the size of the data sets that are bound to Training DataVariable (dcdm:weather-2007-31-101501). In particular, if the size of the data set is greater than 10,000 the parameter value is set to 1024M and if the size is less than 1,000 the value is set to 256M; otherwise it will be set to 512M.

The next step is ranking the 48 candidate workflows based on estimates of performance. For this, the predicted DODs for intermediate data sets are useful. For example, the size of intermediate data products is useful to obtain estimates on performance time for the different algorithms of the workflow components.

```
LmtModelerThenJ48Classifier  a  wflow:WorkflowTemplate ;
    wflow:hasLink modelerTrainingDataInputLink , modelerJavaMaxHeapInputLink , classifierOutputLink ,
modelerOutputClassifierInputInOutLink , modelerClassIndexInputLink , classifierDataInputLink ;
    wflow:hasNode classifierNode , modelerNode .

modelerNode  a  wflow:Node ;  wflow:hasComponent ac:LmtModeler.
classifierNode  a  wflow:Node ;  wflow:hasComponent ac:J48Classifier.

modelerTrainingDataInputLink  a  wflow:InputLink ;
    wflow:hasDestinationNode modelerNode ;
    wflow:hasDestinationArgument ac:d ;
    wflow:hasVariable TrainingDataVariable.

TrainingDataVariable  a  dcdm:Instance , wflow:DataVariable;
    dcdm:hasDomain dcdm:weather;
    wflow:hasDataBinding dcdm:weather-2007-31-101501.

maxJavaHeapSizeModelerParameterVar  a  wflow:ParameterVariable ;
    wflow:hasParameterValue “512M”;

modelerJavaMaxHeapInputLink  a  wflow:InputLink ;
    wflow:hasDestinationNode  modelerNode ;
    wflow:hasDestinationArgument ac:j ;
    wflow:hasVariable maxJavaHeapSizeModelerParameterVar .

modelerClassIndex  a  wflow:ParameterVariable .
    wflow:hasParameterValue 5;  // from the request

modelerClassIndexInputLink  a  wflow:InputLink ;
    wflow:hasDestinationNode  modelerNode ;
    wflow:hasDestinationArgument ac:i ;

modelerOutputClassifierInputInOutLink  a  wflow:InOutLink ;
    wflow:hasDestinationNode  classifierNode ;
    wflow:hasDestinationArgument ac:m ;
    wflow:hasOriginNode modelerNode ;
    wflow:hasOriginArgument ac:o ;
    wflow:hasVariable ModelDataVariable .

ModelDataVariable  a  dcdm:BayesModel , wflow:DataVariable ;
    dcdm:hasModelType DecisionTree ;
    dcdm:hasDomain dcdm:weather ;
    wflow:hasDataBinding modelerOutputModelDataVariable_1191372118140.

classifierDataInputLink  a  wflow:InputLink ;
    wflow:hasDestinationNode classifierNode ;
    wflow:hasDestinationArgument ac:d ;
    wflow:hasVariable TestDataVariable .

TestDataVariable  a  dcdm:Instance , wflow:DataVariable;
    dcdm:hasDomain dcdm:weather;
    workflow:hasDataBinding dcdm:weather-2007-31-155754.

classifierOutputLink  a  wflow:OutputLink ;
    wflow:hasOriginNode classifierNode ;
    wflow:hasOriginArgument ac:o;
    wflow:hasVariable ClassificationDataVariable .

ClassificationDataVariable  a  wflow:DataVariable , dcdm:DtmClassification ;
   dcdm:hasDomain dcdm:weather; // from the request
   wflow:hasDataBinding ClassificationDataVariable_1191372118140.
```

Table 23: A workflow instance after workflow instantiation.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- generated: Tue Oct 02 17:42:29 PDT 2007 by Wings -->
<adag xsi:schemaLocation="http://www.griphyn.org/chimera/DAX http://www.griphyn.org/chimera/dax-1.10.xsd"
xmlns="http://www.griphyn.org/chimera/DAX"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
version="1.10" count="1" index="0" name="ModelerThenClassifier-dax198d8-b4d199239c5e7f99">

<!-- part 2: definition of all jobs -->
  <job id="Job1-04bd4f3cbfd2" namespace="http://www.isi.edu/ac/dm/library.owl" name="J48Classifier" version="">
    <argument>-T <filename file="weather-2007-31-155754"/>
              -l <filename file="modelerOutputModelDataVariable_1191372118140"/>
              -O <filename file="ClassificationDataVariable_1191372118140"/> </argument>
    <uses file="modelerOutputModelDataVariable_1191372118140" link="input"/>
    <uses file="weather-2007-31-155754" link="input"/>
    <uses file="ClassificationDataVariable_1191372118140" link="output"/>
  </job>
  <job id="Job0-18917e3ec858" namespace="http://www.isi.edu/ac/dm/library.owl" name="LmtModeler" version="">
    <argument>-Xmx 512M -t <filename file="weather-2007-31-101501"/>
                -d <filename file="modelerOutputModelDataVariable_1191372120250"/> -c 5 </argument>
    <uses file="weather-2007-07-31-101501" link="input"/>
    <uses file="modelerOutputModelDataVariable_1191372120250" link="output"/>
  </job>

<!-- part 3: list of control-flow dependencies (empty for single jobs) -->
  <child ref="Job1- 04bd4f3cbfd2">
    <parent ref="Job0-18917e3ec858"/>
  </child>
</adag>
```

Table 24: Example ground workflow in Pegasus format, generated from a workflow instance.

In the final step, each workflow instance is turned into a bound workflow that can be submitted to the workflow mapping and execution system. Table 24 shows the ground workflow extracted for the workflow instance in Table 22. It shows the format used by the Pegasus workflow mapping and execution engine [Deelman et al 03; Deelman et al 05].

The next section shows an implementation of a component, workflow, and data catalog that were used to generate these workflows automatically.

6. Using Wings for Template-Based Automatic Workflow Generation

We have implemented the automated template-based workflow generation algorithm as part of the Wings workflow generation system [Gil et al 10; Gil et al 09b; Gil et al 09a; Kim et al 08; Gil et al 07a; Kim et al 06]. We also implemented a workflow catalog, a component catalog, and a data catalog that were separate components following the distributed architecture shown in Figure 4.

```
Modeler  a       ModelerClass ;
    ac:hasArgument modelerClassIndex , outputModel,
                        javaMaxHeapSize , modelerTrainingData;
    ac:hasInput javaMaxHeapSize , modelerTrainingData , modelerClassIndex ;
    ac:hasOutput outputModel ;
    ac:isConcrete "false"^^xsd:boolean .
Classifier  a       Classifier_Class ;
   ac:has Argument classifierInputData , classifierInputModel ,
                      classifierOutput ;
   ac:hasInput classifierInputData , classifierInputModel ;
   ac:hasOutput default:classifierOutput ;
   ac:isConcrete "false"^^xsd:boolean .
```

Table 25: Representation of a Modeler component and a Classifier component.

```
// properties represented as metrics (such as the property domain) are propagated backward by a Modeler
[modelerTransferBackwd:
    (?c rdf:type pcdom:Modeler)
    (?c pc:hasOutput ?odv) (?odv ac:hasArgumentID "o")
    (?c pc:hasInput ?idv) (?idv ac:hasArgumentID "d")
    (?odv ?p ?val) (?p rdfs:subPropertyOf dc:hasMetrics)
  -> (?idv ?p ?val)]

// number of classes is propagated backward by a Classifier
[classifierTransferNClass:
    (?c rdf:type pcdom:Classifier)
    (?c pc:hasOutput ?odv) (?odv ac:hasArgumentID "o")
    (?c pc:hasInput ?idvmodel) (?idvmodel ac:hasArgumentID "m")
    (?c pc:hasInput ?idvdata) (?idvdata ac:hasArgumentID "d")
    (?odv dcdom:hasNumberOfClasses ?val)
  -> (?idvmodel dcdom:hasNumberOfClasses ?val),
    (?idvdata dcdom:hasNumberOfClasses ?val)]
```

Table 26: Examples of rules used in the component services for the backward sweep.

We use the W3C Web Ontology Language OWL standard (www.w3.org/TR/owl-features) to represent components, workflows, and metadata. OWL is based on description logics and extends the W3C RDF/XML standards with an ontology language. We also use the Jena framework and its associated reasoning engines (jena.sourceforge.net). Each of the services uses its own reasoner, so the reasoning about components is separate from the reasoning about workflows. The workflow system uses the functions in the component service according to the algorithms presented in the prior section. A well-known issue of using OWL and other description logic systems is that one cannot make assert property values about classes. This is because there is a separate A-box for instances and a separate T-box for classes (i.e., for terminological reasoning). Therefore, we create instances for many items in order to be able to make assertions. For example, in order to assert properties of a component class or a file type, we create a *Skolem instance* that represents a prototypical instance of the class. A definition of a Modeler_Skolem and a Classifier_Skolem are shown in Table 25 in a N3 style notation. The workflow template representations use these Skolem instances to refer to a component.

```
// number of classes is propagated forward by a Classifier
[classifierTransferDataFwdNOfClasses:
    (?c rdf:type pcdom:Classifier)
    (?c pc:hasOutput ?odv) (?odv ac:hasArgumentID "o")
    (?c pc:hasInput ?idvmodel) (?idvmodel ac:hasArgumentID "m")
    (?c pc:hasInput ?idvdata) (?idvdata ac:hasArgumentID "d")
    (?idvmodel dcdm:hasNumberOfClasses ?val) (?idvdata dcdom:hasNumberOfClasses ?val)
   -> (?odv dcdom:hasNumberOfClasses ?val)]

// metrics data is propagated forward by a Modeler
modelerTransferFwd:
    (?c rdf:type pcdom:Modeler)
    (?c pc:hasOutput ?odv) (?odv ac:hasArgumentID "o")
    (?c pc:hasInput ?idv) (?idv ac:hasArgumentID "d")
    (?idv ?p ?val) (?p rdfs:subPropertyOf dc:hasMetrics) -> (?odv ?p ?val)]

// javaMaxHeapSize is set by the number of instances in the data object
[exampleParamSet1:
    (?c rdf:type pcdom:Modeler)
    (?c pc:hasInput ?idv) (?idv ac:hasArgumentID "d")
    (?c pc:hasInput ?ipv) (?ipv ac:hasArgumentID "j") // maxJavaHeapSize
    (?idv dcdom:hasNumberOfInstances ?x) ge(?x 10000)
    -> (?ipv ac:hasValue "1024M")]
[exampleParamSet2:
    (?c rdf:type pcdom:Modeler)
    (?c pc:hasInput ?idv) (?idv ac:hasArgumentID "d")
    (?c pc:hasInput ?ipv) (?ipv ac:hasArgumentID "j") // maxJavaHeapSize
    (?idv dcdom:hasNumberOfInstances ?x) lessThan(?x 10000)
    -> (?ipv ac:hasValue "512M")]
[exampleParamSet3:
    (?c rdf:type pcdom:Modeler)
    (?c pc:hasInput ?idv) (?idv ac:hasArgumentID "d")
    (?c pc:hasInput ?ipv) (?ipv ac:hasArgumentID "j") // maxJavaHeapSize
    (?idv dcdom:hasNumberOfInstances ?x) lessThan(?x 1000)
    -> (?ipv ac:hasValue "256M")]
```

Table 27: Examples of rules used by the component services to answer queries about components during the forward sweep.

6.1. Component Services for Workflow Generation

Table 26 shows some of the rules used to answer backward sweep queries that are used in this step. For example, for a modeler component, any values of **hasMetrics** sub-properties of the output are asserted to be the same for the input data. Table 27 shows some rules used to answer forward sweep queries.

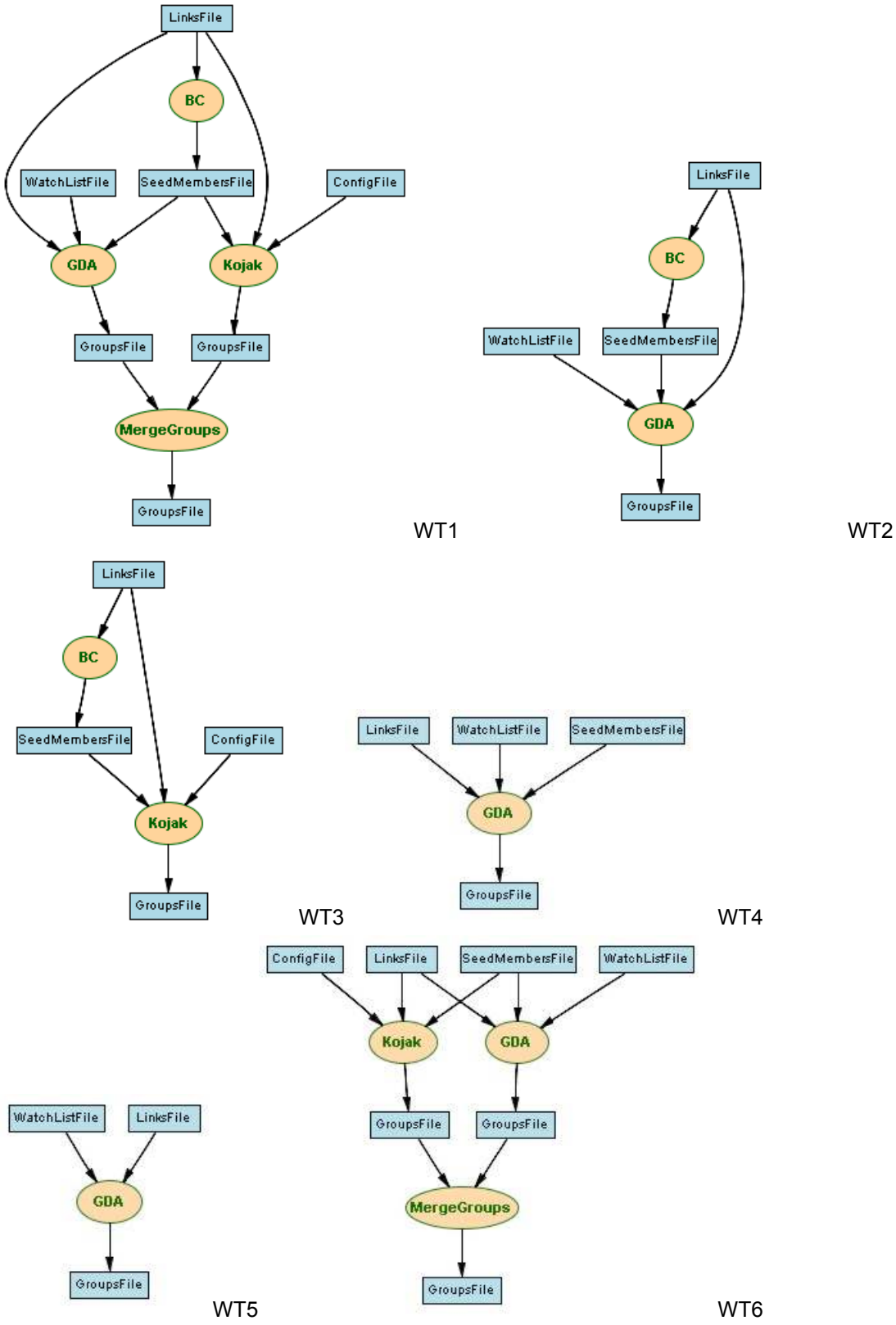


Figure 6: An example workflow template library for relational learning algorithms

```
[WT6TransferBackwd:
    Area(GroupsFile_output_1, ?a)  -> Area(LinksFile_input, ?a)
    GroupType(GroupsFile_output_1, ?g) ->
    GroupType(WatchListFile_input, ?g) ^
    GroupType(SeedMembers_input, ?g)
    StartYear(GroupsFile_output_1, ?s) ->
    StartYear(LinksFile_input, ?s) ^ StartYear(SeedMembers_input, ?s)
    EndYear(GroupsFile_output_1, ?s) ->
    EndYear(LinksFile_input, ?s) ^ EndYear(SeedMembers_input, ?s)]
```

Table 28: Examples of rules for workflow template WT6 used in the workflow services to answer queries during the backward sweep.

```
[WT3TransferFwd:
    wfIns:WorkflowTemplate(?w) ^
    aflns:hasNumLinks(LinksFile_flib, ?numl)  ^
    aflns:hasLinkWeight(?w, ?lw) ^
    aflns:hasLinkWeightError(?w, ?lwe) ^
    swrlb:divide(?p1, ?numl, ?lw)  ^
    swrlb:divide(?p2, ?numl, ?lwe)  ^
    swrlb:add(?p3, ?p1, ?p2)  ^
    swrlb:subtract(?p4, ?p1, ?p2)
     ->
    hasMinEntities(GroupsFile_flib, ?p4)  ^
    hasMaxEntities(GroupsFile_flib, ?p3)]
```

Table 29: Example of a rule for workflow template WT3 used in the workflow services to answer queries during the forward sweep.

6.2. Workflow Services for Workflow Generation

We have also used Wings with an implementation of the forward sweep and backward sweep using workflow-level rules. The workflow templates in this case combine pattern matching and other forms of relational learning algorithms such as the Kojak pattern matcher [Adibi et al 04], the GDA group detection algorithm [Kubica et al 03], and Betweenness Centrality to detect densely connected groups [Newman and Girvan 04]. For these algorithms, it is hard to characterize their individually differentiating factors especially when combined with other algorithms into a more complex analysis (a workflow). The choice of algorithms results in crucial tradeoffs between the quality of the result, the cost (in terms of false positives and false negatives), and the execution time. Therefore, rather than representing detailed models of each algorithm in a component service, we represent the information needed to support workflow generation in the workflow service.

Figure 6 shows a few templates built from the algorithms mentioned. Table 28 shows a rule used in the workflow service for the template WT6. This rule is used to compute the predicted minimum and maximum number of resulting groups given the number of links in the input file. The number of entities (groups) is obtained as a percentage (the "weight") of the number of links in the input with an associated error interval. Those numbers (hasLinkWeight and hasLinkWeightError) can be a default value or be empirically obtained

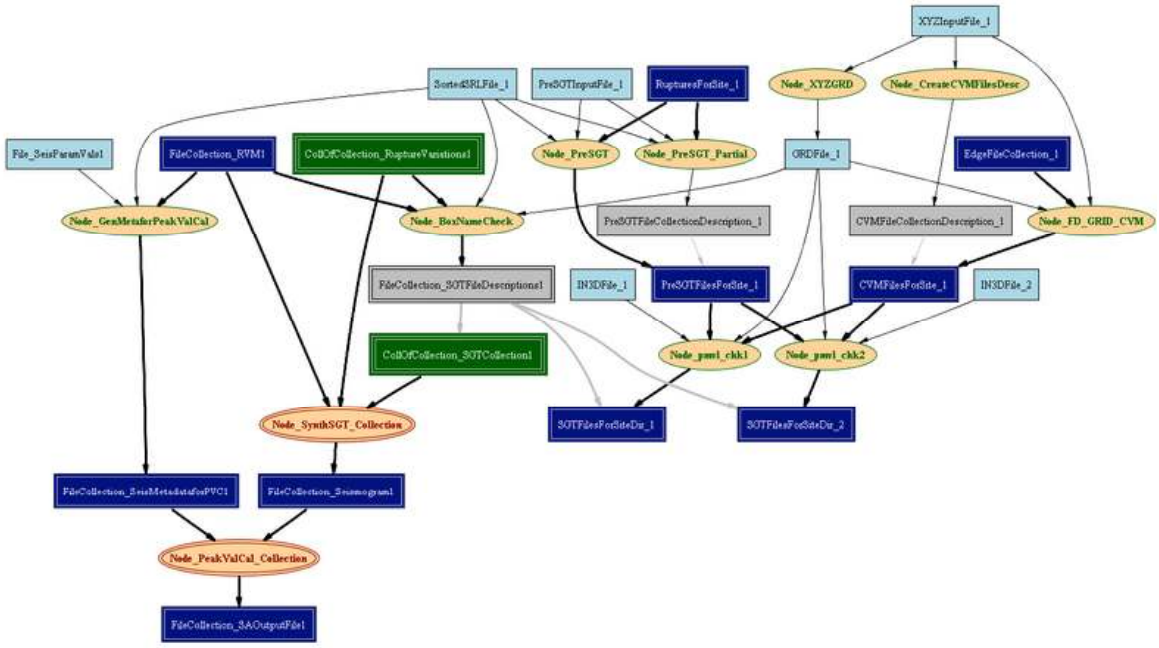


Figure 7. A Workflow Template for Seismic Hazard Analysis.

from data about previous executions of the templates. In comparing estimates for the number of groups, intervals that overlap are considered equivalent. The rule is used to answer queries during the backward sweep. Table 29 shows a rule for workflow WT3 used to answer queries during the forward sweep.

We also have used workflow-level constraints in workflows for seismic hazard analysis. Figure 7 shows a workflow template that estimates spectral acceleration based on physics-based simulations of potential fault ruptures. These workflows require an extended language that what we presented in this paper, since the template needs to capture how data collections of many items are processed by many jobs. The processing of data collections is abbreviated in the workflow template with nodes that represent collections of jobs that would execute the same component over each item of the data collection. The workflow-level constraints were represented and propagated as OWL assertions. The details of these representations are described in [Kim et al 06; Gil et al 07a]. In this application the initial request consisted of a bound workflow, that is, the initial data to be processed was given. Workflow-level constraints were then used to elaborate the original workflow template and specialize, configure, and instantiate it. Finally a ground workflow (also a Pegasus dax) was submitted for mapping and execution.

7. Solving Workflow Generation Requests

This section discusses in detail eight workflow requests and summarizes the workflow generation process for each of them. At the end of the section, we summarize the number of workflow candidates generated, the number of calls to the data and component services, and the time to run the workflow generation algorithm.

For these runs, we used a component catalog with the components shown in Table 1. We used a data catalog of similar properties to the datasets in Table 1, but we had 4 datasets of weather data and 4 datasets of soybean data. The first four requests use four different workflow templates, and illustrate how different numbers of datasets are matched

for each request. The fifth and sixth requests illustrates that candidate bound workflows can be eliminated during workflow generation. The seventh request shows that a request can use a workflow template that is partially bound. Finally, the eighth request is one where no matching datasets are found for any workflow candidates, so no solutions are generated.

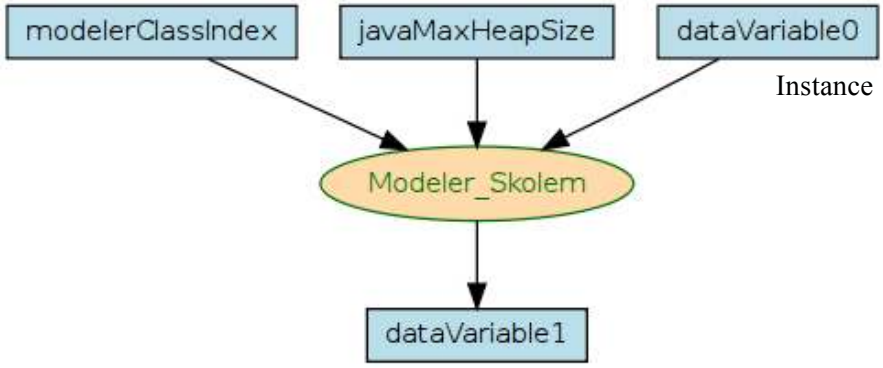
For each request, we show the diagram of the workflow template and the templte constraints if any. We also show the seed constraints. For brevity, we show the constraints on types of workflow data variables as annotations in the graph.

Request R1: ModelWeather

Request description:

Use the Modeler workflow template to create a model of weather instances where the maximum java heap size for running the modeler is 500M, and the instances dataset contains class ids in column 5.

Workflow template:



Seed:

```
dataVariable1 dcdm:hasDomain dcdm:weather
javaMaxHeapSize wflow:hasParameterValue 500M
modelerClassIndex wflow:hasParameterValue 5
```

Workflow Generation Run:

Since there is only one component in the template, during backward sweep, the system makes only one call for `c:find-DODs-given-output-requirements`. There are 6 modelers in the component catalog, and 6 binding-ready workflows are generated. There is only one data variable for each, so one `d:find-data-objects` call is created for each workflow. Out of the binding-ready workflows, 2 have 4 matching weather datasets for dataVariable0:

```
J48Modeler: [(dataVariable0 weather-2007-07-31-155754)], [(dataVariable0
weather-2007-07-31-101503)], [(dataVariable0 weather-2007-07-31-101656)],
[(dataVariable0 weather-2007-07-31-101501)]]
```

and

```
LmtModeler: [[(dataVariable0 weather-2007-07-31-101656)], [(dataVariable0
weather-2007-07-31-155754)], [(dataVariable0 weather-2007-07-31-101501)],
[(dataVariable0 weather-2007-07-31-101503)]]
```

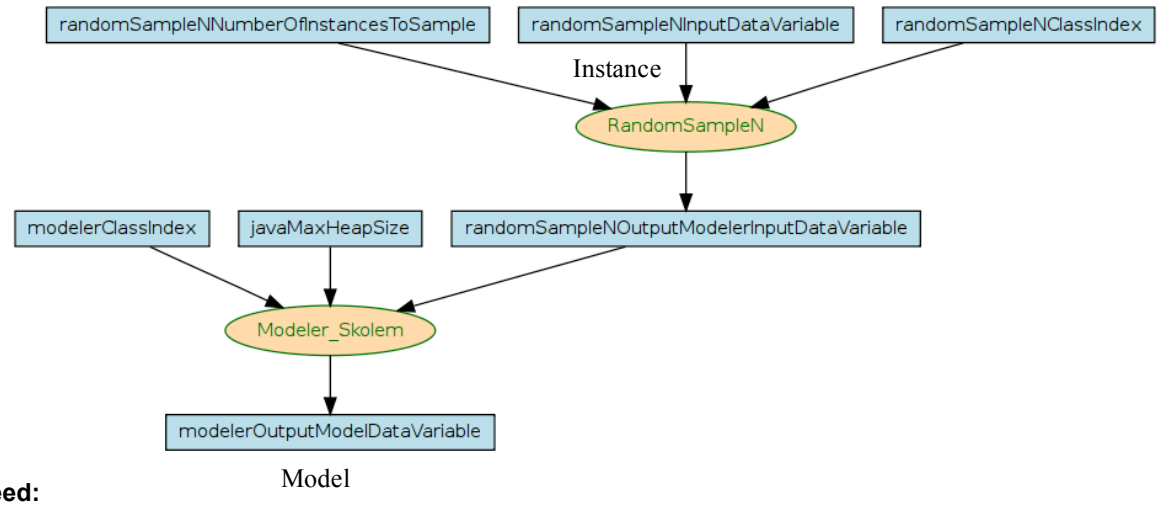
The other workflow candidates take DiscreteInstances only. With a data catalog with four weather domain datasets (weather-2007-31-101501, weather-2007-31-101503, weather-2007-31-101656, and weather-2007-31-155754) that are all ContinuousInstances, the system will not find matching datasets for the workflows that need DiscreteInstances. That is, 8 bound workflows are generated. During the forward sweep, one `c:predict-DODs-given-input-requirements` call is made for each workflow. All the bound workflows are configured after the forward sweep, and the system generates 8 workflow instances.

Request R2: SampleWeatherThenModel

Request description:

Use SampleThenModel workflow to generate a model of weather data. The instances dataset contains class ids in column 5.

Workflow template:



Seed:

```
randomSampleNClassIndex wflow:hasParameterValue 5
modelerOutputModelDataVariable dcdm:hasDomain dcdm:weather
modelerClassIndex wflow:hasParameterValue 5
```

Workflow Generation Run:

Since there are 6 modelers and 1 sampler in the component catalog, during the backward sweep, one `c:find-DODs-given-output-requirements` call for the Modeler component and six calls for the RandomSampleN component are made, generating 6 binding-ready workflows. Each of the workflow makes one `d:find-data-objects` call. Out of the six workflows, only 2 binding-ready workflows match the available weather data, and each have 4 matching datasets. As a result, the system generates 8 bound workflows:

```
RandomSampleN-J48Modeler: [(dataVariable0 weather-2007-07-31-155754)],
[(dataVariable0 weather-2007-07-31-101503)], [(dataVariable0 weather-
2007-07-31-101656)], [(dataVariable0 weather-2007-07-31-101501)]]
```

and

```
RandomSampleN-LmtModeler: [[(dataVariable0 weather-2007-07-31-101656)],
[(dataVariable0 weather-2007-07-31-155754)], [(dataVariable0 weather-
2007-07-31-101501)], [(dataVariable0 weather-2007-07-31-101503)]]
```

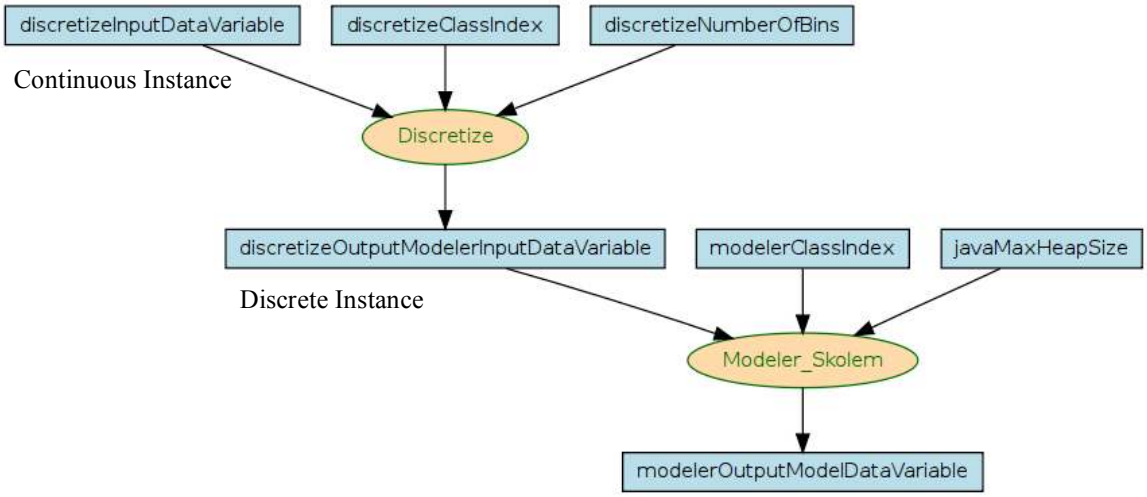
During the forward sweep, two `c:predict-DODs-given-input-requirements` calls are performed for each workflow since there are two components per each. All the bound workflows are configured after the forward sweep. That is, 8 configured workflows are generated.

Request R3: DiscretizeAndModelWeather

Request description:

Use DiscretizeAndModel workflow template to generate a model of weather data. The instances datset contains class ids in column 5.

Workflow template:



Seed: Model

```
modelerOutputModelDataVariable dcdm:hasDomain dcdm:weather
discretizeClassIndex wflow:hasParameterValue 5
modelerClassIndex wflow:hasParameterValue 5
```

Workflow Generation Run:

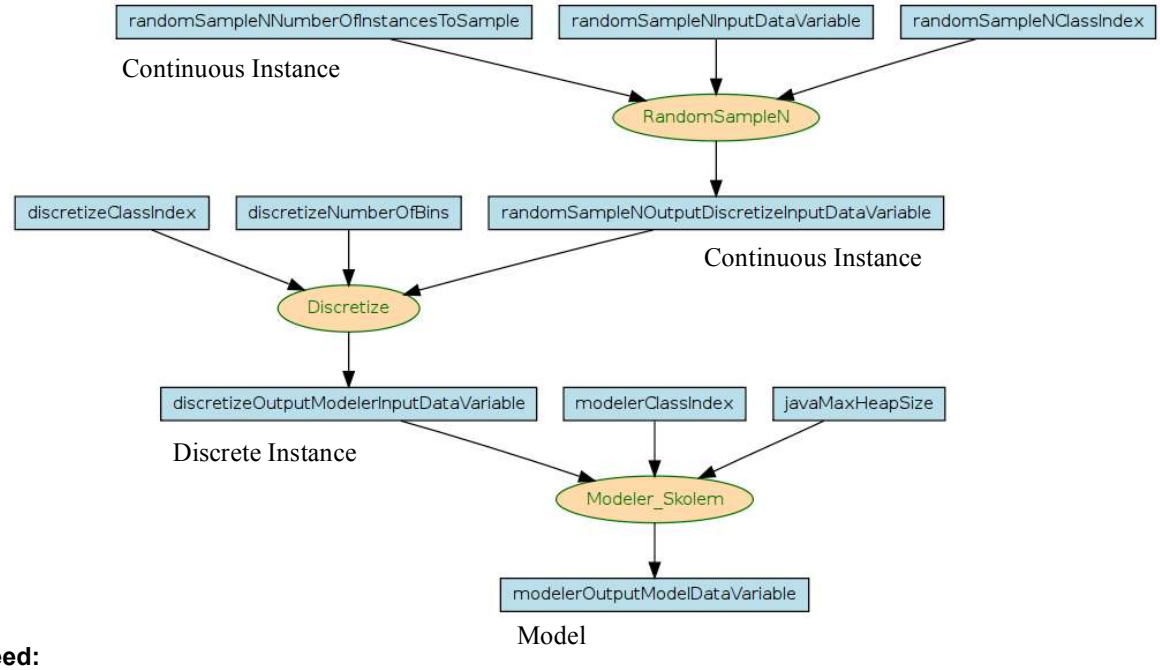
Since there are 6 modelers and 1 discretizer in the component catalog, 6 binding-ready workflows are generated. The number of `c:find-DODs-given-output-requirements` and `d:find-data-objects` calls are the same as the Request R2. The discretizer accepts ContinuousInstances, so any modeler can be used in this case. That is, for each binding-ready workflow, the system will match 4 weather datasets, and generate a total of 24 bound workflows. During the forward sweep, each workflow makes two calls for `c:predict-DODs-given-input-requirements`. All the bound workflows can be elaborated, resulting in 24 workflow instances.

Request R4: SampleDiscretizeThenModel

Request description:

Use SampleDiscretizeThenModel workflow to generate a model of weather data. The instances dataset contains class ids in column 5.

Workflow template:



Seed:

```
modelerClassIndex wflow:hasParameterValue 5
modelerOutputModelDataVariable dcdm:hasDomain dcdm:weather
randomSampleNClassIndex wflow:hasParameterValue 5
discretizeClassIndex wflow:hasParameterValue 5
```

Workflow Generation Run:

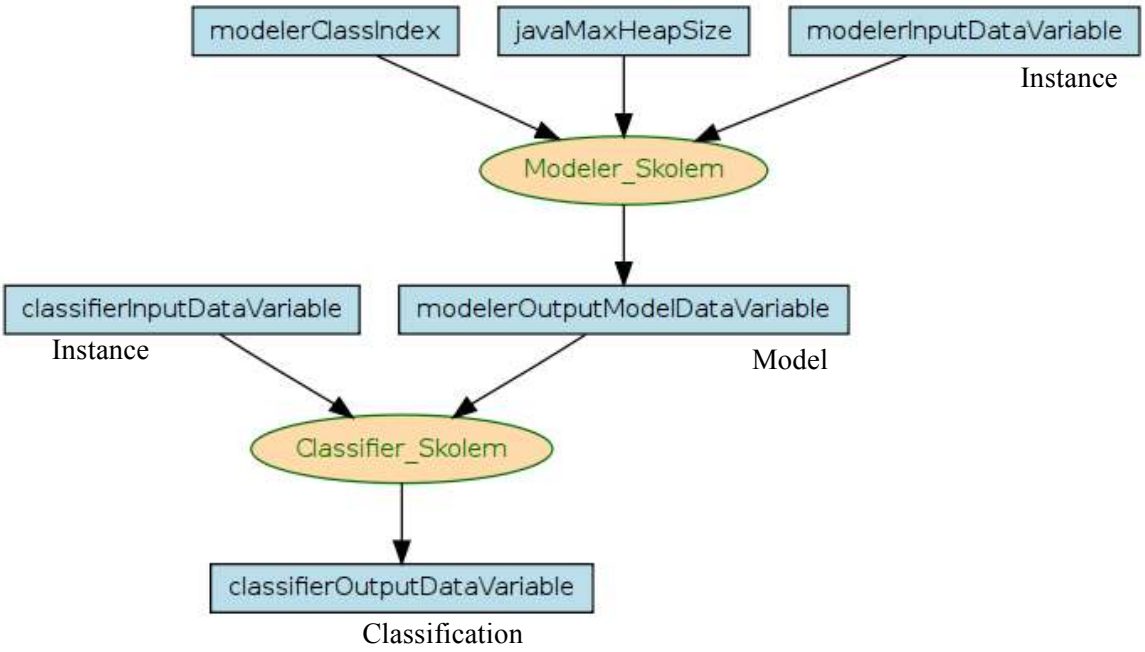
Since the discretizer turns the data into discrete data, all modelers can be used in this workflow. During the backward sweep, one `c:find-DODs-given-output-requirements` call for the Modeler component, six calls for the Discretize component and six more calls for the RandomSampleN are made. For each workflow candidate, one `d:find-data-objects` call is made, and there will be 4 matching datasets. That is, 24 bound workflows are generated. During the forward sweep, all the bound workflows can be elaborated, resulting in 24 configured workflows. A total of 72 calls for `c:predict-DODs-given-input-requirements` (3 calls for each bound workflow) are made.

Request R5: ModelerThenClassifier

Request description:

Use ModelThenClassify workflow to generate a classification of weather data. The modeler input dataset contains class ids in column 5.

Workflow template:



Template Constraints:

```
[instanceInequalityCheck:
     (:classifierInputDataVariable wflow:hasDataBinding ?classifierDS)
     (:modelerInputDataVariable wflow:hasDataBinding ?modelDS)
     equal(?classifierDS, ?modelDS)
     (?t rdf:type wflow:WorkflowTemplate)
      -> (?t wflow:isInvalid "true"^^xsd:boolean)]
```

Seed:

```
classifierOutputDataVariable dcdm:hasDomain dcdm:weather
classifierOutputDataVariable rdf:type dcdm:Classification
modelerClassIndex wflow:hasParameterValue 5
```

Workflow Generation Run:

During the backward sweep, one `c:find-DODs-given-output-requirements` call for the Classifier component and six calls for the Modeler component are made. The system finds that the 3 decision tree classifiers can only take decision tree models, so they can only be combined with one of the 3 decision tree modelers. Similarly, the 3 Bayes classifiers can only take as input Bayes models. That is, 9 workflow candidates are created with decision tree modelers and classifiers and 9 with Bayes models. So in total the system generates 18 binding-ready workflows. For each workflow candidate, one d:find-data-objects call is made for the two data variables.

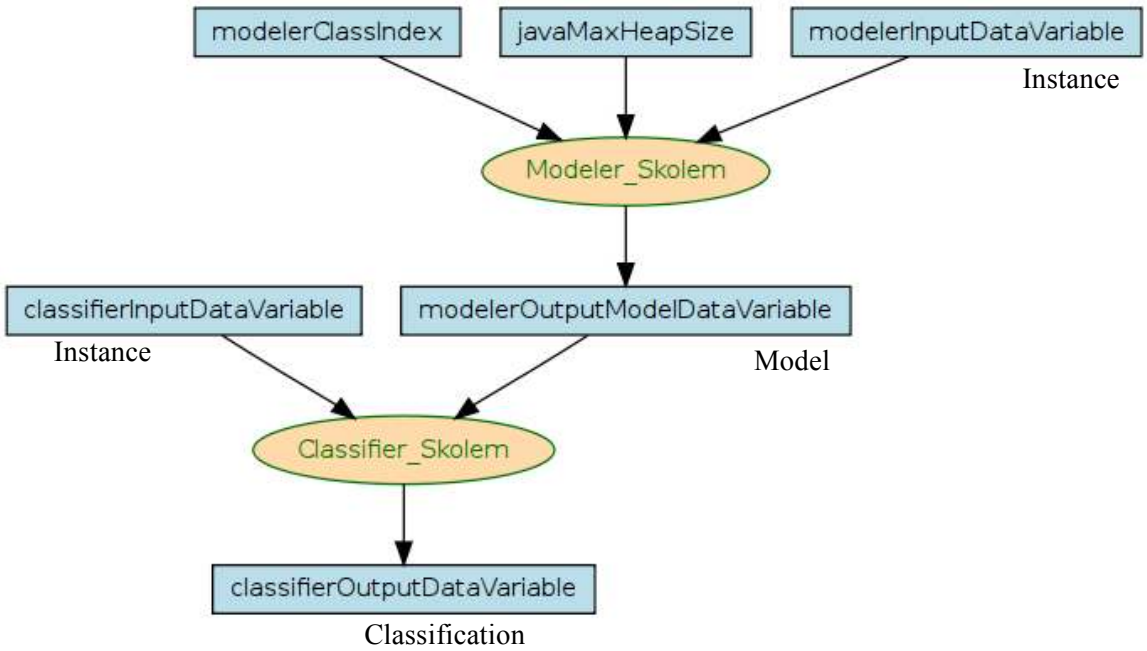
The system will not find matching weather datasets for the candidate workflows that need DiscreteInstances. In this case, only 4 of the binding-ready workflows with Lmt and J48 combinations (LmtModelerThenJ48Classifier, LmtModelerThenLmtClassifer, J48ModelerThen LmtClassifier, J48ModelerThen J48Classifier) will get matching datasets. Since there are four matching weather dataset for each of the two input variables, a total of 64 candidate bound workflows generated in our running example. During the forward sweep, total 128 calls for `c:predict-DODs-given-input-requirements` (2 calls for each candidate bound workflow) are made. However, 16 candidates are eliminated by the workflow template constraint, expressed in the rule above, that indicates that the bound datasets for classifierInputDataVariable and modelerInputDataVariable cannot be the same. As a result, a total of 48 configured workflows are generated.

Request R6: Soybean-ModelerThenClassifier

Request description:

Use ModelThenClassify workflow to generate a classification of soybean data. The modeler input dataset contains class ids in column 5.

Workflow template:



Template Constraints:

```
[instanceInequalityCheck:
     (:classifierInputDataVariable wflow:hasDataBinding ?classifierDS)
     (:modelerInputDataVariable wflow:hasDataBinding ?modelDS)
     equal(?classifierDS, ?modelDS)
     (?t rdf:type wflow:WorkflowTemplate)
      -> (?t wflow:isInvalid "true"^^xsd:boolean)]
```

Seed:

```
classifierOutputDataVariable dcdm:hasDomain dcdm:soybean
modelerClassIndex wflow:hasParameterValue 5
```

Workflow Generation Run:

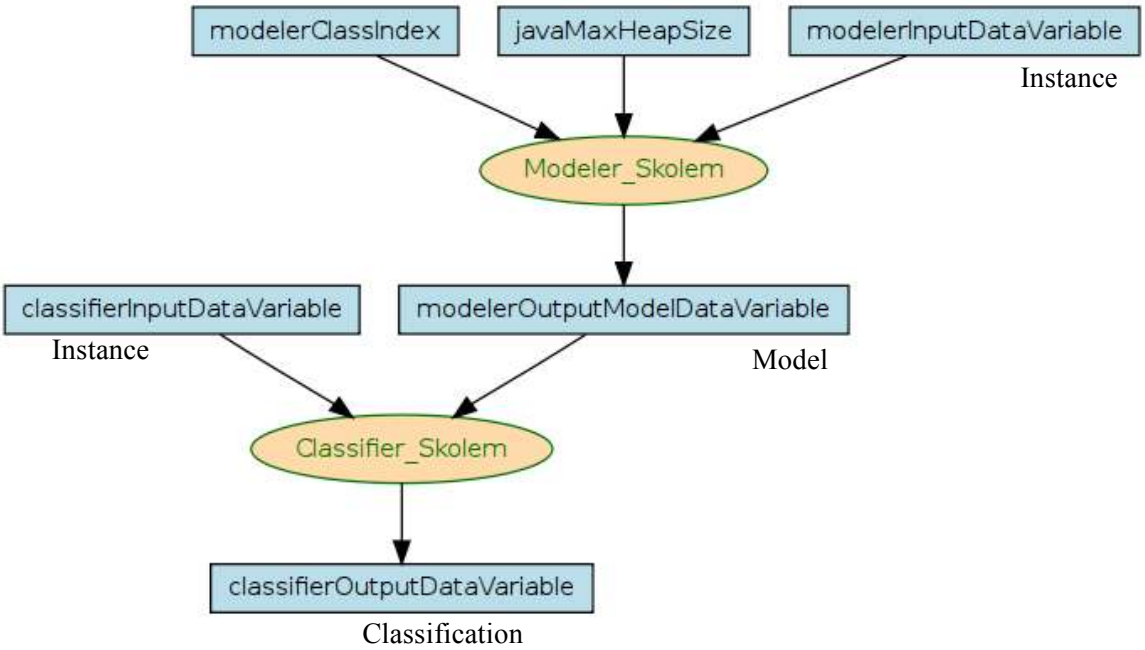
As in R5, the system generates 18 binding-ready workflows. The number of c:find-DODs-given-output-requirements and d:find-data-objects calls are the same as the Request R5. Since there are 4 discrete soybean datasets, the system generates 16 bindings for each of the 18 binding-ready workflows, which results in 288 bound workflows. During the forward sweep, a total of 576 calls for c:predict-DODs-given-input-requirements (2 calls for each candidate bound workflow) are made.

The template constraint above that the training and test datasets cannot be the same invalidates 72 (18x4) workflows, and as a result the system generates 216 configured workflows.

Request R7: TDataBound-ModelerThenClassifier

Request description:

Use ModelThenClassify workflow to generate a classification of a given weather data `weather-2007-07-31-101503`. The instances dataset contains class ids in column 5.



Template Constraints:

```
[instanceInequalityCheck:
     (:classifierInputDataVariable wflow:hasDataBinding ?classifierDS)
     (:modelerInputDataVariable wflow:hasDataBinding ?modelDS)
     equal(?classifierDS, ?modelDS)
     (?t rdf:type wflow:WorkflowTemplate)
      -> (?t wflow:isInvalid "true"^^xsd:boolean)]
```

Seed:

```
modelerInputDataVariable  wflow:hasDataBinding  dclib:weather-2007-07-
31-101503
classifierOutputDataVariable dcdm:hasDomain dcdm:weather
modelerClassIndex wflow:hasParameterValue 5
```

Workflow Generation Run:

This request, unlike the others above, specifies a dataset to use for the modeler component (i.e. the training data).

As in R5, the backward sweep generates 18 candidate binding-ready workflows. The same number of calls to the component catalog and the data catalog are made. Also as in R5, only 4 of the binding-ready workflows with Lmt and J48 combinations (LmtModelerThenJ48Classifier, LmtModelerThenLmtClassifer, J48ModelerThen LmtClassifier, J48ModelerThen J48Classifier) will get a matching dataset for the classifier input data variable. For each of the four workflows, the system will return 4 bindings for the input dataset to the classifier. That creates a total of 16 bound workflows. During the forward sweep, 32 calls for `c:predict-DODs-given-input-requirements` (2 calls for each candidate bound workflow) are made.

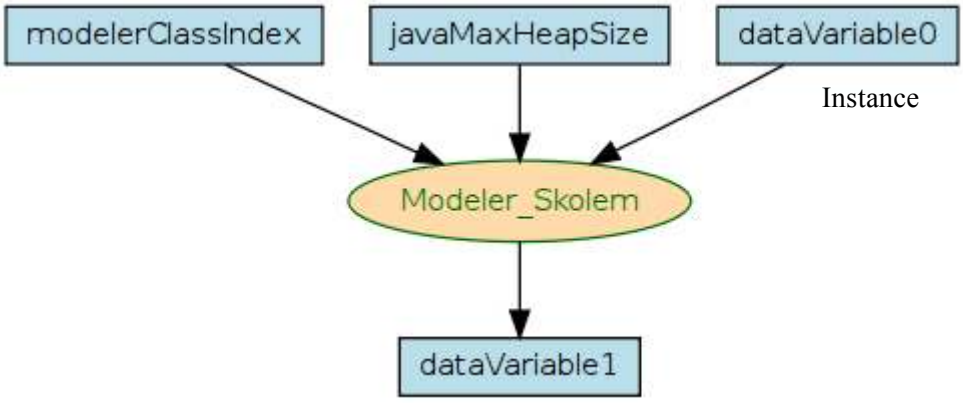
The workflow constraint above that the training and test datasets must not be equal causes the elimination of 4 of the candidate bound workflows. As a result, only 12 configured workflows are created.

Request R8: ModelWeather-NoBindings

Request description:

Use the Modeler workflow to create a model of a soybean-nominal domain where the maximum java heap size for running the modeler is 500M, and the instances dataset contains class ids in column 5.

Workflow Template:



Model

Seed:

```
dataVariable1 dcdm:hasDomain dcdm:soybean-nominal
javaMaxHeapSize wflow:hasParameterValue 500M
modelerClassIndex wflow:hasParameterValue 5
```

Workflow Generation Run:

This template results in 6 binding-ready workflows, one for each modeler. The number of `c:find-DODs-given-output-requirements` and d:find-data-objects calls are the same as the Request R2. However, there are no matching datasets. There are no bound or configured workflows generated for this request.

7.1. Summary

Table 30 summarizes the above requests and the results from running the workflow generation algorithm. The three columns show the number of binding-ready workflow candidates, bound workflow candidates, and configured workflow candidates. These represent the number of workflow candidates generated after step backward sweep, input data object selection and forward sweep respectively. The next three columns show the number of queries to data catalog and component catalog for individual candidate workflows. The current implementation handles individual workflow separately, and different candidate workflows do not share the calls.

Request ID	# Binding-Ready Workflow Candidates	# Bound Workflow Candidates	# Configured Workflow Candidates	# Calls to c:find-DODs-given-output-requirements	# Calls to d:find-data-objects	# Calls to c:predict-DODs-given-input-requirements	Workflow Generation Time
R1	6	8	8	1	6	8	5 s
R2	6	8	8	7	6	16	4 s
R3	6	24	24	7	6	48	7 s
R4	6	24	24	13	6	72	8 s
R5	18	64	48	7	18	128	22 s
R6	18	288	216	7	18	576	81 s
R7	18	16	12	7	18	32	10 s
R8	6	0	0	1	6	0	1 s

Table 30: Generation of workflows for a variety of workflow requests.

8. Related Work

We discuss related work grouped into four topics: plan generation, workflow systems, workflow representations, and software composition.

The emphasis of our work is not on workflows for data mining or machine learning, rather we use this as a domain to illustrate our approach. But it is worth noting that workflow planning has been proposed as an approach to automatically compose data mining applications [Bernstein et al 05]. There are also several projects that suggest the use of grid execution infrastructure for data mining, and in particular for the Weka libraries [Cannataro et al 04; Talia et al 05].

8.1. Related Work in Plan and Workflow Generation

Workflow generation can be seen as a planning problem, where the workflow request specifies features of the initial or goal status. We have used AI planning as a framework in our prior work to assist users to create workflows [Kim et al 04] and to assemble workflows from individual components [Blythe et al 03]. Earlier work that proposed the use of AI planning to compose software applications incluyes [Chien and Mortensen 96].

A major novel aspect of our work is that our algorithm is the first that truly distributes the functions of component reasoning and data reasoning to external services. In the plan and workflow generation approaches described below all reasoning about components and data is done internally in the workflow system rather than by invoking services provided by external component and data catalogs.

MOLGEN used skeletal planning that propagated constraints through a skeletal plan in order to generate a concrete ground plan [Friedland and Iwasaki 85; Stefik 81]. In skeletal planning, the same steps that appear in the skeletal plan are the same as those apprearing in the concrete plan. Our algorithm follows this principle, it does not allow the component catalog to return additional arguments when specializing a workflow component. It would be useful to extend the algorithm so that additional steps could be added to the workflow, perhaps allowing for non-critical components to be added. For example, components that convert data into other formats, or enrich the data in some useful manner. This would be a form of abstract planning, where our algorithm would use a high-level template to build the plan at an abstract level and then a less abstract level where additional workflow components are added.

Planning by analogy and case-based planning reuse an existing plan and adapt it to satisfy new goals [Veloso 94; Kambhampati and Hendler 92]. Existing plans are modified according to the aspects of the new state and goals that are not present in past cases. The adaptations may include new steps or remove steps from the previous plan. In constrast, our algorithm does not add or remove steps from the workflow template.

The backward sweep and the forward sweep have some commonalities with techniques used in planning to propagate constraints in initial and goal states throughout the steps of a plan. The backward sweep is essentially a form of goal regression, while the forward sweep is in essence a forward projection [McDermott 91; Fikes and Nilsson 71]. There is an analogy between the workflow requests that include requirements on data products and goal statements in planning. The requiremenst in workflow requests that specify the bindings for initial data sources can be seen as similar to initial states in planning. However, initial states in planning contain ground predicates while workflow requests can include requirements that are not ground predicates. That is, workflow requests specifying the initial data sets to be used turns into ground predicates, but specifying properties of those datasets does not.

AI planning and other techniques have been used to generate workflows both automatically and interactively [McIlraith and Son 01; McDermott 02; Narayanan and McIlraith 02; Medjahed et al 03; Blythe et al 03; Kim et al 04; Ambite and Kapoor 07]. However, these approaches require representations of components that can support composition from first principles. In many domains, it is impractical or impossible to obtain the detailed information required to support this kind of workflow generation from first principles. In contrast, by starting from a library of known-to-work workflow templates, the models required for the components can be much lighter, since they are only required to specialize workflows but not to compose them. Another issue shared by these approaches is the incorporation of user requirements and preferences. Scientists have very extensive requirements about the kind of analysis that they want performed on a dataset. Often these requirements amount to following a method (workflow) that is published in the literature or is proven to work well [Gil 06; Gil et al 07b; Goderis et al 05a; Goderis et al 05b; Goderis et al 06; Wroe et al 07]. Automatic generation of workflows from libraries of well-designed templates is a practical alternative approach for many application domains.

An alternative to generating workflows from first principles is the use of hierarchical task network planning [Sirin et al 04]. As we discussed in section 2.5, the reasoning about data and components is done within the planner rather than distributed to the data and component catalogs as in our approach. Another difference is that our approach allows users to state rich constraints on the kinds of solutions sought. More recent work [Lin et al 08] supports user preferences on the types of results or initial data or components to be used. Our approach allows users to also specify precisely relationships between datasets and components (e.g., that the output of a specific modeler within the workflow be of a

certain type, not just any modeler in the workflow), as well as to express requirements on intermediate datasets. In this sense, our approach is closer to planning from sketches in hierarchical task network planning, where users provide some high-level features of a plan and the system elaborates them into a complete executable plan [Myers et al 03]. Based on the sketch, the system retrieves relevant plan fragments, and uses the sketch to guide the completion of the plan. Plan sketches are not as complete as workflow templates, as they can omit many steps needed in the plan. Generating workflows from such high-level sketches has not been investigated to date, but might provide more flexibility than allowing for user advice only. It might also support more flexibility than template-based workflow generation.

8.2. Related Work in Workflow Systems

Most research on workflow systems is concerned with workflows of web services, and in that sense their execution is done in a distributed environment [Oinn et al 06; Ludaescher et al 06]. In contrast, our work addresses computational workflows, where each workflow component is an executable code that can be submitted to execution on a remote host through grid services.

The myGrid project (www.mygrid.org.uk) is perhaps the most closely related as it focuses on the discovery, reuse and repurposing of bioinformatics workflows [Wroe et al 07]. They distinguish between direct "re-use" of an existing workflow as is and "re-purposing" an existing workflow by modifying some aspect of it. [Goderis et al 05a] defines requirements and bottlenecks for workflow re-use based on many user interviews and practical experiences. They identify seven bottlenecks to workflow reuse, including the need for more appropriate and interoperable workflow description and discovery models, intellectual property rights on workflows that hinder sharing, service accessibility, relative workflow rankings, and knowledge engineering needed to appropriately annotate workflows. Our work presents additional requirements stemming from a different set of application areas, as well as a different perspective since we consider workflows of computations rather than of web services. One of the requirements addressed in myGrid is the retrieval of workflows based on requests specifying structural properties of the workflows, which is a requirement not addressed in our solution described in this paper. A first solution using role specialization in OWL Lite allowed only to consider sequential composition of components in the workflow is shown in [Goderis et al 05a], an improved technique for graph sub-isomorphism matching is presented in [Goderis et al 06]. As in our work, functional properties of workflows are used to support workflow retrieval with expressive requirements, addressing interactive refinement of queries that is also complementary to the focus of this paper [Goderis et al 05b]. Other than automating this kind of workflow retrieval, there is little automation for actually elaborating the workflows into executable ones. An important novel aspect of our work is the automatic selection of input data objects and the automatic ranking and selection of workflows, which are strong requirements in the domains that we have worked with.

Some workflow systems automate many aspects of the execution process. Prior work on Pegasus [Deelman et al 05; Deelman et al 03] automated the selection of execution sites and data replicas of computational workflows so that the user did not need to be concerned with the specifics of the execution environment. Although that research automates an important aspect of the overall workflow creation process and relieves a significant burden from end users, it is only concerned with automating decisions regarding resource selection and performance optimization. It does not address the selection of what software or data sources are needed in order to accomplish some data analysis goals that the user may have.

8.3. Related Work in Workflow Representations

The formalization and algorithms presented in this paper does not commit to specific languages or representations. This was intentional, as we believe that our approach can be implemented in any choice of language. More expressive languages can support more complex reasoning capabilities, but our approach would work with less expressive languages as well.

Our workflows are represented using structure (nodes and links) and constraints (metadata properties of workflow data variables). Our algorithms could be used with any workflow representations that can represent both. Popular workflow languages for web services, such as BPEL (www.oasis-open.org/committees/wsbpel), use complex control structures such as conditionals and iterations that we do not include. We assume a simple DAG control structure that offers several advantages and has been sufficient to support a variety of applications that we have addressed so far in our work [Gil et al 07a; Kim et al 06; Hall et al 08; Deelman et al 05]. It is easier to recover from execution failures if there are simple dependencies across components. In addition, the workflow components are likely to be more composable if they hide internally any complex control structures and are likely to be easier to compose for non-programmers.

Some languages that include semantic constraints expressed in OWL have been proposed [Ankolekar et al 01; Roman et al 05]. Our workflow generation algorithm could be used with any of those languages, though only a small subset of the constructs allowed in those languages are used in our framework.

8.4. Related Work in Software Composition

Computational workflows are software artifacts. From that point of view, our approach has some commonalities as well as distinct requirements that are worth considering here.

The Model Driven Architecture (MDA) (www.omg.org/mda) as defined by the OMG consortium shares with the approach presented here the interest on separating the problem domain from the execution environment in a software system. MDA proposes the use of different models for a software system along with a number of transformations going from more abstract models into more specific ones. In MDA terms, its goal is to separate business and application logic, which tends to be more stable, from the underlying platform technology, which may evolve more quickly due to technological evolution. MDA is intended mainly for large-scale distributed web-based business applications. The main differences with our approach come from the type of models used to describe a software system. MDA models are built with UML and do not support complex semantic constraints other than is-a and part-of hierarchies. On the other hand, UML models allow expressing not only data flow aspects of an application but also module relations and operation sequencing. Finally, while MDA tries to model the whole software system from a very high level down to the function level, a workflow system considers course-grained component level descriptions of data and execution requirements without concern for how those components are implemented internally.

Workflow-based approaches promote reuse at different levels: code reuse, since algorithms are componentized they can be plugged into different workflows; design reuse, since every workflow template provides an abstract design that can be specialized with different components; requirements reuse, since the formal description of the problem solved by a given workflow documents interesting combinations of problem characteristics; and test reuse, since workflow instantiation and execution are also record for future use. However, as it is well documented in the research on software reuse [Frakes and Kang 05], any methodology that promotes reuse, and somehow prioritizes a long-term view of

software development that penalizes short-term results, must take into account not only technical but also human and organizational factors in order to be successfully applied. We advocate what in the software product lines literature is known as "minimally invasive transitions" [Krueger 06], trying to minimize disruption of ongoing development efforts and to take advantage of existing software assets to make possible an incremental adoption of the workflow-based approach.

9. Discussion

A variety of search strategies could be used to implement the workflow generation algorithm. The algorithm as described in this paper follows breadth-first search strategy, and generates all possible workflow candidates. Our implementation supports beam search, which is more efficient but does not guarantee completeness. The same algorithm could be applied with a depth-first search strategy. In that case, the algorithm could be configured to stop after generating k solutions, rather than generating an exhaustive list of candidates. This would not guarantee that the top-k ranked candidates would be generated and selected, but would expedite the search in cases where the space is complex. An alternative search strategy could be to conduct a heuristic search over the space of candidates. This would require heuristic functions at each level to determine which candidates are superior to others based on the information available at that point about each of the candidates. Other search strategies could combine these heuristic estimates and expand only the k-best candidates at each level.

The generation of workflows need not be a completely stratified process divided into independent stages as presented in this paper. In some contexts, it may be desirable to make the process less stratified. For example, several iterations of the backward or forward sweep may be needed so that incrementally more detailed descriptions of the data are constructed. In addition, the algorithm considers one candidate at a time but it could be implemented to take advantage of what it has learned for a candidate in order to prune others. This would be very useful when the search space is very complex and the number of candidates explodes. Candidates may have significant shared workflow strands or data properties that could be reused to avoid redundant expansions and queries, particularly to eliminate candidates more efficiently.

An interesting area of research is the extension of the algorithm to enable the incorporation of new components in the workflow template as workflow generation proceeds. The algorithm described in this paper assumes that the initial workflow template contains all the steps that will appear in any bound workflow resulting from the workflow generation process. This requires that abstract component classes are defined in the component catalog to have the same amount of input and output arguments as their component subclasses. A simple extension to the current algorithm is to allow component classes to not contain all the parameters that will appear in the specialized component classes. The algorithm would add links and data variables to the workflow candidate as needed to represent the additional parameters once the template is being specialized. A different kind of extension may be to add components that do not create new kinds of data but simply reformat the same dataset. That is, a component may output a dataset in its own format, and the next component may need the dataset in a different format. The dataset properties as far as the workflow generation algorithm would be the same, but the actual realization of the dataset would be in different formats. Format conversion components could be automatically inserted where needed in the workflow. A more complex extension would be to extend the algorithm to be able to add new nodes (components) to the workflow template. Specifically, this could be triggered when no data sources are found to

match the requirements of the existing input data variables for a workflow. In those cases, the algorithm could recursively turn the input data requirements into a request that it then tries to find, instantiate, and execute workflows to satisfy it.

The development, refinement, and validation of component models, workflow models, and data models needs to occur in the context of running the algorithm. Throughout the workflow generation process each step generates or eliminates workflow candidates based on knowledge from the catalogs. It is possible that the models in the component catalog and workflow catalog do not contain enough constraints and therefore do not eliminate candidates that are inappropriate. In that case, candidate workflows will be generated that either fail to execute or that generate invalid results. If the models in the catalogs are overly constraining, some workflow candidates may not be generated and solutions may be missed. By analyzing the pool of candidates, a developer can refine the catalogs and improve their fidelity over time.

Similarly, it is not possible to determine whether a workflow request issued to the system is consistent with the component, workflow, and data catalogs untils it is processed by the workflow generation algorithm. By propagating the constraints through the workflow template and by incorporating constraints of the individual components, the algorithm would encounter inconsistencies and the pool of candidates would be empty. For example, consider a request that specifies bindings for a subset of the input variables. Peforming the backward sweep would validate the request and ensure that the bindings provided are consistent with the components and workflow template descriptions.

Automation requires knowledge about components or workflows to support the workflow generation algorithm. The queries to the catalogs performed during workflow generation require that the appropriate knowledge to answer them exists in the catalogs. Therefore, automation comes at a cost incurred in developing and debugging those catalogs. However, this cost depends on the kind of information needed and the amount of automation required. First, there is a choice between adding knowledge about components or about workflows. Adding knowledge about components is done in the component catalog, and can support the component-level backward and forward sweep. Component-level knowledge has the advantage of being applicable to all workflows that contain a component, however for this very reason it needs to be specified with care so that the knowledge generalizes across components. An alternative choice is to add knowledge about workflows in the workflow catalog to support the workflow-level backward and forward sweep instead. Workflow-level knowledge is specific to a workflow, and is not transferable across workflows. On the other hand, it is easier to specify as it is only concerned with the workflow at hand. A hybrid approach may be best, where knowledge is specified for components where generalizations are possible, but otherwise the knowledge is specified for workflows. The algorithm would perform the backward and forward sweep for both the component-level queries and the workflow-level queries, combining the constraints obtained from both. The second choice regarding cost is the degree of automation desired as allowed in the workflow requests. That is, depending on the amount of information included in the original workflow request, some of the steps of the algorithm may not be needed. If the workflow request is seedless, then there is no need to seed the workflow template. If the template in the request is concrete, then there is no need for the component catalog to be able to specialize abstract classes. If the workflow request is already bound, then there is no need for the backward sweep or to find data objects. If the workflow request is configured, then there is no need for the configuration query knowledge in the forward sweep. A third issue regarding cost arises from the generation of properties of new workflow data products. That is, the algorithm can generate ground workflows without generating any metadata properties of the new data products.

Generating descriptions of new data products during the forward sweep requires knowledge to generate such descriptions. The advantage is that data reuse can be supported. That is, before the workflow is executed, the workflow system can notice that a data product already exists based on its anticipated properties. In that case, the workflow system can remove unnecessary computations and simply reuse that dataset. If data reuse is not desirable, then there is no need to specify the knowledge that generates descriptions of new data products. In domains where a selective characterization of the individual algorithmic components is not possible, a different approach to workflow generation may be needed. Anticipating which algorithms are more appropriate for a given dataset may not be possible, and therefore pruning workflow candidates would not occur as needed. An alternative path to workflow generation would be a portfolio approach for algorithm selection [Leyton-Brown et al 03], where all the available algorithms for the request would be executed and their results would be evaluated and compared in order to select the algorithms that performed best.

Learning the knowledge needed for component selection and characterization is an intriguing possibility. Performance data could be obtained by executing all possible workflow candidates formed with all combinations of available algorithms present in a workflow template. The performance data could be compared with a gold standard for known datasets, and applicability rules could be learned that correlate the characteristics of those datasets with the performance obtained. Similarly, running all possible workflow templates relevant to a request would enable the collection of performance data that would enable learning knowledge needed to discriminate among workflow templates in the initial library.

Designing a shared workflow library also poses difficult challenges. An expert in the domain may design a workflow template by combining available components applied on certain types of data, but different experts may come with different solutions which would be, in principle, equally good. Designing workflow templates is also challenging if they must be reusable, retrieved in the appropriate contexts, and contain enough information to be instantiated for new workflow requests. Learning abstract workflow templates from many example executions of workflows may be a desirable approach.

10. Conclusions

We have presented a novel algorithm to generate workflows automatically from high-level specifications of a user request. The algorithm conforms to a formalism proposed in the paper to characterize datasets, workflow components, and workflows that frames the process of workflow generation from a user request. Workflow generation starts from a generic workflow template and incrementally adds detail to the workflow by selecting components and data sources, while it generates or eliminates workflows from a pool of candidates. The major novel features of this workflow generation algorithm are:

- It provides users with flexibility both in the kinds of information that they can provide and in the amount of information provided. The algorithm can start from a user request specified at different levels of abstraction and completeness.
- It either uses a workflow template provided in the request or retrieves relevant workflow templates from a catalog, and generates possible candidates that are consistent with the user request. This is an important requirement that allows users to reuse typical workflow structures that reflect common approaches to a problem or proven methodology shared by a community.
- It explicitly calls out to external semantic data and component catalogs for specific reasoning tasks concerning data and components, therefore explicitly declaring

what reasoning is required from third-party catalogs to support workflow generation. It assumes an architecture where the workflow system is separate from data and component services, which are distributed and provided by others. This is important in many application areas, where components and data are made available by third parties that are separate from the workflow libraries and the workflow system itself. This is the case in the design of e-Science "collaboratories" where data and software are made available in physically distributed environments, and the producers and the users are different sets of people. The workflow generation algorithm invokes external services that reason about components and data, and obtains properties and constraints that are then propagated to the rest of the workflow.

There is a cost to automating workflow generation in terms of the amount of engineering of the knowledge needed to answer queries about components and about workflow templates. We discussed in the paper the tradeoffs between automating only some aspects of workflow generation instead of the whole process. The workflow generation algorithm presented has two variants: one that requires deeper knowledge in the component catalog, and one that requires deeper knowledge in workflow templates. The algorithm also includes a workflow ranking to select a subset of all the workflow candidates generated. This approach degrades gracefully: more investment in knowledge engineering leads to more pruning in the space explored and therefore less wofklow candidates considered during workflow generation. But less investment in knowledge engineering is still acceptable, as the system will just generate many more workflows to be ranked.

Scientific data processing has reached a new era of complexity and scale. Instruments are now available that can collect terabytes of data in a day. Shared data repositories make vast amounts of data available to entire communities for analysis and experimentation. Tremendous amounts of computing power are available to scientists. Scientific exploration and experimentation can no longer be conducted in the realm of handcrafted processes carried out by individual investigators. Automated workflow generation processes can explore a larger space of hypotheses and execute experimental tests more effectively than manual data analysis processes. Knowledge representation, planning, and other artificial intelligence techniques can contribute to the automation of important aspects of the scientific discovery process. Scientists can then be empowered by intelligent workflow systems that can be tasked at a high level to conduct specific experiments and to analyze the results. Automated workflow generation systems as a form of intelligent assistance have tremendous potential to significantly accelerate scientific progress.

Acknowledgements

We would like to thank Ewa Deelman, Gaurang Mehta, and Karan Vahi for their feedback on this work and the integration of our workflow generation algorithm with their Pegasus workflow mapping and execution system. We would also like to thank Christian Fritz and Paul Groth for many useful comments. This work was supported in part by the National Science Foundation under grant CCF-0725332.

References

Adibi, J., Chalupsky, H., Melz E., and Valente, A. "The KOJAK Group Finder: Connecting the Dots via Integrated Knowledge-Based and Statistical Reasoning". Proceedings of the Sixteenth Innovative Applications of Artificial Intelligence Conference (IAAI), San Jose, CA, July 2004.

Al-Masri, E. and Q. H. Mahmoud. "Investigating Web Services on the World Wide Web." Proceedings of the World Wide Web Conference, Beijing, China, 2008.

Ambite, J.L. and Kapoor, D. "Automatically Composing Data Workflows with Relational Descriptions and Shim Services". Proceedings of the 6th International Semantic Web Conference (ISWC-2007), Busan, Korea, November 2007.

Ankolekar, A., M. Burstein, J.R. Hobbs, O. Lassila, D.L. Martin, S.A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. "DAML-S: Semantic Markup For Web Services". Proceedings of the International Semantic Web Workshop, 2001.

Ashley, K. D. and Aleven, V, 1997, Reasoning symbolically about partially matched cases. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence. San Francisco, CA: Morgan Kaufmann, pp. 335–341.

Asuncion, A. & Newman, D.J. (2007). UCI Machine Learning Repository [http://www.ics.uci.edu/~mlearn/MLRepository.html]. Irvine, CA: University of California, School of Information and Computer Science.

Atkins, D. E., K. K. Droegemeier, S. I. Feldman, H. Garcia-Molina, M. L. Klein, D. G. Messerschmitt, P. Messina, J. P. Ostriker, M. H. Wright. "Revolutionizing Science and Engineering Through Cyberinfrastructure," National Science Foundation Blue-Ribbon Advisory Panel on Cyberinfrastructure January 2003. http://www.nsf.gov/publications/pub_summ.jsp?ods_key=cise051203

Baader, F. and P. Narendran, "Unification of Concept Terms in Description Logics", Journal of Symbolic Computation, 2001.

Baader, F. C. Lutz, M. Milicic, U. Sattler, and F. Wolter. "Integrating Description Logics and Action Formalisms: First Results", Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05), Pittsburgh, PA, USA, 2005.

Belhajjame, K., Embury, S.M., Paton, N. W., Stevens, R., and C. A., Goble. "Automatic annotation of Web services based on workflow definitions." ACM Transactions on the Web, 2(2), 2008.

Bergmann R and Stahl, A, 1998, Similarity measures for object-oriented case representations. In Proceedings of the Fourth European Workshop on Case-Based Reasoning. Berlin: Springer, pp. 25–36.

Bernstein, A., F. Provost and S. Hill. "An Intelligent Assistant for the Knowledge Discovery Process: An Ontology-based Approach" IEEE Transactions on Knowledge and Data Engineering 17(4), pp. 503-518, 2005.

Blythe, J., Deelman, E., Gil, Y., Kesselman, C., Agarwal, A., Mehta, G., Vahi, K.. "The Role of Planning in Grid Computing." Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS), June 9-13, 2003, Trento, Italy.

Bussler, C., Fensel, D., and A. Maedche, A. " A conceptual architecture for semantic Web-enabled Web services." SIGMOD Record 31(4) 2002.

Cannataro, M., Comito, C., Schiavo, F.L. and Veltri, P. 2004: Proteus, a Grid based Problem Solving Environment for Bioinformatics: Architecture and Experiments. IEEE Computational Intelligence Bulletin, 3(1) (2004) 7-18.

Champin, PA and Solnon, C, 2003, Measuring the similarity of labeled graphs. In Proceedings of the Fifth International Conference on Case-Based Reasoning. Berlin: Springer, pp. 80–95.

Chien, S. A., and H. B. Mortensen, "Automating Image Processing for Scientific Data Analysis of a Large Image Database," IEEE Transactions on Pattern Analysis and Machine Intelligence 18 (8): pp. 854-859, August 1996.

Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Blackburn, K., Lazzarini, A., Arbree, A., Cavanaugh, R., and Koranda, S. "Mapping Abstract Workflows onto Grid Environments." Journal of Grid Computing, Vol. 1, No. 1, 2003.

Deelman, E., Singh, G., Su, M., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C., and D. S. Katz. "Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems". Scientific Programming Journal, Vol 13(3), 2005.

Deelman, E., and Gil, Y. (Eds). "Final Report of the NSF Workshop on Challenges of Scientific Workflows", National Science Foundation, Arlington, VA, May 1-2, 2006. http://www.isi.edu/nsf-workflows06.

Fan, J. and S. Kambhampati. "A Snapshot of Public Web Services." ACM SIGMOD Record, March 2005.

Fikes, R. E., and Nilsson, N. J. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", Artificial Inelligence, 2(3/4):189-208, 1971.

Forbus, K, Gentner, D and Law, K, 1994, MAC/FAC: a model of similarity-based retrieval. Cognitive Science 19(2), 141–205.

Frakes, W., and K. Kang, "Software Reuse Research: Status and Future," IEEE Trans. on SW Eng., vol.31, no. 7, pp. 529-536, July 2005.

Friedland, P. and Iwasaki, Y. "The Concept and Implementation of Skeletal Plans." Journal of Automated Reasoning, 1(2): 161-208, 1985.

Gil, Y., Ratnakar, V., and Deelman, E. Virtual Metadata Catalogs: Augmenting Metadata Catalogs with Semantic Representations. Fourth International Semantic Web Conference (ISWC-05), Galway, Ireland, November 7-10, 2005.

Gil, Y. "Workflow Composition". In Workflows for e-Science, D. Gannon, E. Deelman, M. Shields, I. Taylor (Eds), Springer Verlag, 2006.

Gil, Y., Ratnakar, V., Deelman, E., Mehta, G. and J. Kim. "Wings for Pegasus: Creating Large-Scale Scientific Applications Using Semantic Representations of Computational Workflows." Proceedings of the 19th Annual Conference on Innovative Applications of Artificial Intelligence (IAAI), Vancouver, British Columbia, Canada, July 22-26, 2007.

Gil, Yolanda, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers. "Examining the Challenges of Scientific Workflows," IEEE Computer, vol. 40, no. 12, pp. 24-32, December, 2007.

Gil, Y. "From Data to Knowledge to Discoveries: Scientific Workflows and Artificial Intelligence." Scientific Programming, Volume 17, Number 3, 2009.

Gil, Yolanda, Jihie Kim, Gonzalo Florez, Varun Ratnakar, and Pedro A. Gonzalez Calero. "Workflow Matching Using Semantic Metadata." Proceedings of the Fifth International Conference on Knowledge Capture (K-CAP), Redondo Beach, CA, September 1-4, 2009.

Gil, Yolanda, Paul Groth, Varun Ratnakar, and Christian Fritz. Expressive Reusable Workflow Templates. Proceedings of the Fifth IEEE International Conference on e-Science, Oxford, UK, December 9-11, 2009.

Gil, Yolanda, Varun Ratnakar, Jihie Kim, Pedro Antonio Gonzalez-Calero, Paul Groth, Joshua Moody, and Ewa Deelman. "Wings: Intelligent Workflow-Based Design of Computational Experiments." To appear in IEEE Intelligent Systems, 2010.

Goderis, A., Ulrike Sattler, Phillip Lord and Carole Goble. Seven bottlenecks to workflow reuse and repurposing. Proc. of the 4th Int. Semantic Web Conference, Galway, Ireland, 2005.

Goderis, A., Sattler, U., Goble, C.A.: Applying description logics for workflow reuse and repurposing. In Horrocks, I., Sattler, U., Wolter, F., eds.: Description Logics. Volume 147 of CEUR Workshop Proceedings., CEUR-WS.org (2005)

Goderis, A., Li, P., Goble, C.A.: Workflow discovery: the problem, a case study from e-science and a graph-based solution. In: ICWS, IEEE Computer Society (2006) 312–319

Hall, M., Gil, Y. and Lucas, R. "Self-Configuring Applications for Heterogeneous Systems: Program Composition and Optimization Using Cognitive Techniques". Proceedings of the IEEE, Special Issue on Edge Computing, February 2008.

Hull, D., Zolin, E., Bovykin, A., Horrocks, I., Sattler, U., and Stevens, R. "Deciding Semantic Matching of Stateless Services." Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI), 2006.

Kambhampati, S. and J. A. Hendler. "A Validation Structure-Based Theory of Plan Modification and Reuse", Artificial Intelligence Journal. Vol 55. pp. 193-258. 1992.

Kim, Jihie, Marc Spraragen, and Yolanda Gil. An Intelligent Assistant for Interactive Workflow Composition, In proceedings of the 2004 International Conference on Intelligent User Interfaces (IUI), Madeira Islands, Portugal, January 2004.

Kim, Jihie, Yolanda Gil, and Varun Ratnakar. "Semantic Metadata Generation for Large Scientific Workflows." Proceedings of the Fifth International Semantic Web Conference (ISWC-06), Athens, GA, November 5-9, 2006.

Kim, J., Deelman, E., Gil, Y., Mehta, G., and V. Ratnakar. "Provenance Trails in Wings/Pegasus", Journal of Computation and Concurrency: Practice and Experience, Special issue on the First Provenance Challenge, L. Moreau and B. Ludaescher (Eds), 2008.

Krueger, C. W. "New methods in software product line practice," Communications of the ACM, vol. 49, no. 12 pp. 37-40, December 2006.

Kubica, J., Moore, A., Schneider, J. Tractable Group Detection on Large Link Data Sets. The Third IEEE International Conference on Data Mining, 2003.

Langley, P., Simon, H.A., Bradshaw, G.L., Zytkow, J.M. "Scientific Discovery: Computational Explorations of the Creative Processes." Cambridge, MA: The MIT Press, 1987.

Leyton-Brown, K., Nudelman, E., Galen A., McFadden, J. Shoham, Y. "A Portfolio Approach to Algorithm Selection." Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) 2003.

Li, L., and Horrocks, I., "A software framework for matchmaking based on semantic web technology", Proceedings of the World Wide Web Conference (WWW), 2003.

Lin, N., Kuter, U., and E. Sirin "Web Service Composition with User Preferences." Proceedings of the Fifth European Semantic Web Conference, 2008.

Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger-Frank, E., Jones, M., Lee, E., et al. “Scientific Workflow Management and the Kepler System”. Concurrency and Computation: Practice and Experience, Special Issue on Workflow in Grid Systems, 18(10), 2006.

McDermott, D. “Regression Planning”. International Journal of Intelligent Systems, Volume 6, Issue 4, 1991.

McDermott, D. "Estimated-Regression Planning for Interactions with Web Services." Proceedings of the AI Planning Systems Conference, 2002.

McIlraith, S. and Son, T. "Adapting Golog for Composition of Semantic Web Services," Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning, 2002.

Medjahed, B., A. Bouguettaya, and A. K. Elmagarmid. "Composing Web services on the Semantic Web." The International Journal on Very Large Databases, 12(4), November 2003.

Mitchell, T. M. “The Discipline of Machine Learning.” Machine Learning Department technical report CMU-ML-06-108, Carnegie Mellon University, July 2006.

Moore R. W., Boisvert, R., and Tang, P, Data Management Systems for Scientific Applications. "The Architecture of Scientific Software," pp. 273-284, Kluwer Academic Publishers, 2001.

Moreau, L. and B. Ludaescher (Eds). Special issue on the First Provenance Challenge, Journal of Computation and Concurrency: Practice and Experience, 2008.

Morik, K. and Scholz, M. "The MiningMart Approach to Knowledge Discovery in Databases". In: Ning Zhong and Jiming Liu (editors), Intelligent Technologies for Information Analysis, Springer, pages 47 -- 65, 2004.

Muggleton, Stephen H. “2020 Computing: Exceeding human limits,” *Nature*, Special Issue on 2020 Computing, Vol. 440, pp. 413-414, 2006.

Myers, K. L., P. A. Jarvis, W. M. Tyson, and M. J. Wolverton. "A Mixed-initiative Framework for Robust Plan Sketching", In Proceedings of the 13thInternational Conferences on AI Planning and Scheduling , Trento, Italy, June, 2003.

Narayanan, S., and S. McIlraith. " Simulation, verification and automated composition of Web services." Proceedings of the 11th International World Wide Web Conference, 2002.

Nature Editorial: “2020 computing: Milestones in scientific computing”, Special Issue on 2020 Computing, Nature, Volume 440, Number 7083, March 23 2006.

Newman, M. E. J. and M. Girvan. Finding and evaluating community structure in networks. Physical Review, 2004.

Oinn, T., Greenwood, M., Addis, M., Nedim Alpdemir, M., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., et al. “Taverna: Lessons in creating a workflow environment for the life sciences.” Concurrency and Computation: Practice and Experience, Special Issue on Workflow in Grid Systems, Volume 18, Issue 10, August 2006.

Opitz, D. and Maclin, R. “Popular Ensemble Methods: An Empirical Study.” Journal of Artificial Intelligence Research, Vol 11, 1999.

Roman, Dumitru, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel: Web Service Modeling Ontology, Applied Ontology, 1(1): 77 - 106, 2005.

Singh, G., S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A Metadata Catalog Service for Data Intensive Applications. Proceedings of the Supercomputing Conference, November 2003.

Sirin, E., B. Parsia, D. Wu, J. Hendler, and D. Nau. "HTN planning for web service composition using SHOP2." Journal of Web Semantics, 1(4):377-396, 2004.

Sonnenburg, Sören, Mikio L. Braun, Cheng Soon Ong, Samy Bengio, Leon Bottou, Geoffrey Holmes, Yann LeCun, Klaus-Robert Müller, Fernando Pereira, Carl Edward Rasmussen, Gunnar Rätsch, Bernhard Schölkopf, Alexander Smola, Pascal Vincent, Jason Weston, Robert Williamson; "The Need for Open Source Software in Machine Learning", Journal of Machine Learning Research, Vol 8(Oct):2443--2466, 2007.

St Amant, R. and Cohen, P. "Intelligent Support for Exploratory Data Analysis." Journal of Computational and Graphical Statistics, 7(4), 1998.

Stefik, M. "Planning with Constraints", *Artificial Intelligence*, 16:111-140, 1981.

Szalay A. and J. Gray, "2020 Computing: Science in an exponential world," *Nature*, Special Issue on 2020 Computing, Vol. 440, pp. 413-414, 2006.

Talia, D., P. Trunfio, and O. Verta, "Weka4WS: a WSRF-enabled Weka Toolkit for Distributed Data Mining on Grids," in *9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2005)*. Porto, Portugal, 2005.

Taylor, I., Deelman, E., Gannon, D., Shields, M., (Eds). "Workflows for e-Science", Springer Verlag, 2006.

Thain, D., Tannenbaum, T., and Livny, M. "Distributed Computing in Practice: The Condor Experience" Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4, pages 323-356, February-April, 2005.

Tuchinda, R., Thakkar, S., Gil, Y., and Deelman, E. Artemis: Integrating Scientific Data on the Grid. Proceedings of the 16th Annual Conference on Innovative Applications of Artificial Intelligence (IAAI), San Jose, CA, July 25-29, 2004.

Veloso, M. M., Planning and Learning by Analogical Reasoning: Springer Verlag, December 1994.

Von Laszewski, G., Hategan, M., and D. Kodeboyina. "Java CoG Kit Workflow." In Workflows for e-Science, D. Gannon, E. Deelman, M. Shields, I. Taylor (Eds), Springer Verlag, 2006.

Washington, W. M. et al. "National Science Board 2020 Vision for the NSF," National Science Board Report December 2005.www.nsf.gov/publications/pub_summ.jsp?ods_key=nsb05142

Wieczorek, M., R. Prodan, and T. Fahringer, "Scheduling of Scientific Workflows in the ASKALON Grid Environment," *SIGMOD Record*, vol. 34, 2005.

Witten, I. H. and Frank E. "Data Mining: Practical Machine Learning Tools and Techniques." Morgan Kaufmann, San Francisco, 2 edition, 2005.

Wroe, C., Goble, C., Goderis, A., Lord, P., Miles, S., Papay, J., Alper, P., Moreau, L.: "Recycling workflows and services through discovery and reuse." Concurrency and Computation: Practice and Experience 19(2) (2007) 181–194.

Zhao, J., Goble, C., Stevens, R., Turi, D. "Mining Taverna's semantic web of provenance." Journal of Computation and Concurrency: Practice and Experience, Special issue on the First Provenance Challenge, L. Moreau and B. Ludaescher (Eds), 2008.