A Tag-Based Approach for the Design and Composition of Information Processing Applications

Eric Bouillet, Mark Feblowitz, Zhen Liu, Anand Ranganathan, Anton Riabov
IBM Research
{ericbou,mfeb,zhenl,arangana,riabov}@us.ibm.com

Abstract

In the realm of component-based software systems, pursuers of the holy grail of automated application composition face many significant challenges. In this paper we argue that, while the general problem of automated composition in response to high-level goal statements is indeed very difficult to solve, we can realize composition in a restricted context, supporting varying degrees of manual to automated assembly for specific types of applications. We propose a novel paradigm for composition in flow-based information processing systems, where application design and component development are facilitated by the pervasive use of faceted, tag-based descriptions – of processing goals, of component capabilities, and of structural patterns of families of application. The facets and tags represent different dimensions of both data and processing, where each facet is modeled as a finite set of tags that are defined in a controlled folksonomy. All data flowing through the system, as well as the functional capabilities of components are described using tags. A customized AI planner is used to automatically build an application, in the form of a flow of components, given a high-level goal specification in the form of a set of tags. End-users use an automatically populated faceted search and navigation mechanism to construct these high-level goals. We also propose a novel software engineering methodology to design and develop a set of reusable, well-described components that can be assembled into a variety of applications. With examples from a case study in the Financial Services domain, we demonstrate that composition using a faceted, tag-based application design is not only possible, but also extremely useful in helping end-users create situational applications from a wide variety of available components.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
Copyright © 2008 ACM 978-1-60558-215-3/08/10...$5.00

Categories and Subject Descriptors H4.0 [*INFORMATION SYSTEMS APPLICATIONS*]: General

General Terms Algorithms, Design, Experimentation

Keywords Composition, Programmable Web, Tag Cloud, Automated Application Assembly, Tag-based Component Description, Component Reuse, Faceted Navigation

1. Introduction

Composing software applications from components has long been a topic of interest and research. Notions of operator composition were established well before the introduction of pipes into UNIX in 1972 [26]. This theme has resurfaced recently, in the form of web service composition techniques [14] and in hosted data mashup systems such as Yahoo Pipes™[1] and IBM®DAMIA [3]. Each applies recent advances in service descriptions, graphical editing, etc., to the problem of composing applications from components, with the goals of enabling automated composition or simplifying manual composition.

Yet many of the familiar challenges in architecting and engineering component-based applications remain: the need to partition application code into scalable and/or reusable components, the need to provide sufficient descriptive content such that some agent - human or automated - can compose applications from these components, etc. Advances in object-oriented computing [2] support the definition of abstract component interfaces. But composing applications from software components requires more descriptive content than can typically be gleaned from components' interface declarations. For component selection [29] and for application composition [12], additional knowledge is needed - of the components' functional capabilities, of constraints on each component's applicability for a particular task, of the rules of assembly with other components, etc. The key challenge remains: finding a practical means of expressing, formally or informally, just the right amount of information at the right level of detail and/or abstraction to convey to the *application assembler* the information that can assist in composition of the best component assemblies for a given task.

Even with the needed descriptive content, simply describing components does not ensure their broad applicability across more than a few component assemblies, or their appropriate use in a given assembly. The process of architecting and composing component-based applications requires attention not just to individual components, but requires a broader perspective on the evolving collection of components, which range from domain-specific to widely reusable components, and to the various applications that will incorporate these components. In other words, there must be a careful design and engineering process to come up with a set of reusable components that can be combined together into a large number of different flows.

In this paper we present a new methodology that we developed for facilitating the design and composition of *flow-based* information processing applications. Flow-based applications [16] are component assemblies arranged in a directed acyclic graph (*flow*) of black-box components connected by data flow links (Figure 1). Systems like Yahoo Pipes and IBM DAMIA support the creation of data mashups as flow-based applications. Stream processing systems (like System S [9]) also support flow-based applications that continuously process streaming data. Acyclic workflows in service-oriented systems can also be viewed as flow-based applications, where services exchange messages amongst one another either directly or through a coordinator service (like a BPEL workflow).

At the core of our approach is a novel tag model where domain-specific tags, organized into facets, are used to describe 1) end-user information processing goals, 2) component functional processing and data semantics, and 3) application requirements and structural constraints. The prominent role that tags and tag-based descriptions play in all three areas establishes a strong visible thread from users' information needs to dynamically assembled flow-based applications. Our application design methodology makes use of the tag-based model to capture application requirements and subsequently identify requirements for individual components. Our composition approach makes use of the tags to decide whether certain components can be composed together into a flow.

To support flow composition, we have developed tools that can process component descriptions and can aid in the process of composing the applications. The composition process can be manual, completely automated or assisted. In manual composition, an end-user (or developer) constructs the flow by connecting compatible components using an interactive editor. In completely automated composition, the end-user can specify a processing goal, which gets compiled into a flow. Assisted composition allows users to come up with a flow by interweaving automatic flow generation and manual editing. While our previous work, embodied in a tool called MARIO (Mashup Automation with Runtime Orchestration and Invocation) [25], focuses on automating mashup composition and user interface generation, our methodology is suitable for any kind of composition approach.

The key contribution of this paper is a tag-based application design methodology that facilitates the composition of customized flows that satisfy end-user goals. The aim of the application design methodology is to come up with a set of reusable, well-described components that can be combined together into a wide variety of flows. We describe the need for specifying composition constraints (Section 2), introduce the terminology (Section 3), describe the software engineering process (Section 4), formally describe the models (Sections 5 and 6), and finally, compare our approach with related work (Section 7).

2. Motivation - The Role of Composition Constraints in Situational Applications

The emergence of data mashups offers renewed promise in the oft-revisited field of component-based application assembly. Yahoo Pipes, IBM DAMIA, etc., offer graphical tools to compose applications from web-based data sources and from a fixed set of components. These systems aim to address the need for *situational applications*, end-user driven computing that Cherbakov [6] describes as "applications built to address a particular situation, problem, or challenge." Situational applications are "usually built by casual programmers using short, iterative development life cycles," are updated frequently as needs evolve, and are often abandoned in favor of new solutions.

Assembling situational applications from components becomes easier with tools that assist in the assembly of compatible components into functioning compositions. In practice, though, even with tool support, the large number of available components and the high complexity of composition constraints make manual assembly a daunting task.

In existing systems, component composition constraints are commonly specified by publishing component interfaces, and by strict typing of input and output parameters. In flow-based applications this approach translates to strict typing of the data that is ingested into and produced by each component. However, practical implementations of the strict typing approach can suffer from either under- or over-specification. Under-specification occurs when the data type does not sufficiently capture the semantics, and therefore the constraint is not restrictive enough. For example, a component that inputs an integer can be connected to a variety of inputs that are integers, such as temperature, humidity, etc., while the component may expect the integer to represent the age of a person. Over-specification occurs, e.g., when one defines a type that is too narrowly scoped, inadvertently precluding legitimate uses of the component. These problems clarify the need for another mechanism of expressing composition constraints in addition to strict typing. Designing such a mechanism in a way that is flexible and easy to use is a significant

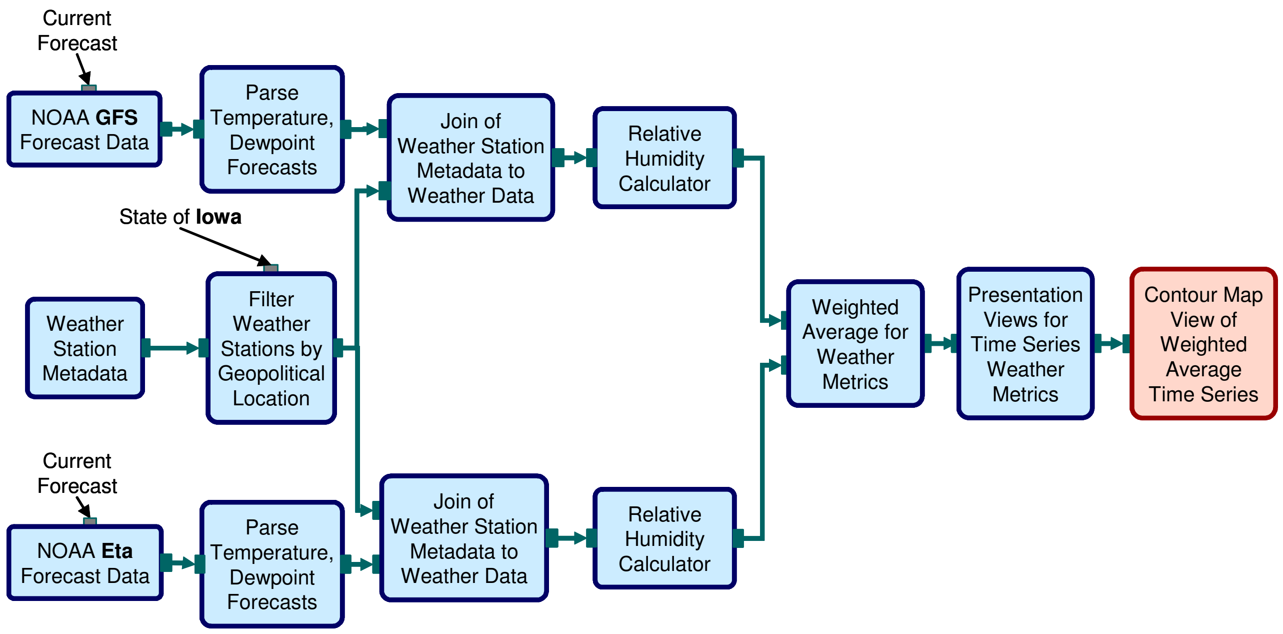

Current Forecast
NOAA GFS Forecast Data
Parse Temperature, Dewpoint Forecasts
Join of Weather Station Metadata to Weather Data
Relative Humidity Calculator
State of Iowa
Weather Station Metadata
Filter Weather Stations by Geopolitical Location
Weighted Average for Weather Metrics
Presentation Views for Time Series Weather Metrics
Contour Map View of Weighted Average Time Series
Current Forecast
NOAA Eta Forecast Data
Parse Temperature, Dewpoint Forecasts
Join of Weather Station Metadata to Weather Data
Relative Humidity Calculator

Figure 1. Flow example for the "RelativeHumidity IA WeightedAverage GFS Eta" goal

challenge. We address this by using tags, as we explain in following sections.

When composition constraints are well specified, it can be possible to compose applications automatically for user-specified composition goals, using constraints to guarantee application correctness. MARIO, an example of such a system, is capable of practical, real-time automatic composition for flow-based applications. In MARIO, users specify composition goals using tags. The set of tags available for user selection depends on the set of components and composition constraints. Only the tags that correspond to supported goals are presented to the end user, limiting the choices to those applications satisfying all constraints. MARIO also ranks alternative compositions that satisfy the specified goals, presents the best match to the user, and provides a clustered view of alternatives.

Fully automatic composition has many benefits, including automatic adaptation to changes in the set of available components and significant simplification for the end user. In some cases, however, the composition constraints may not be specified precisely enough to achieve full automation. In these cases, an approach that combines manual and automatic decisions can still take advantage of the constraints to assist the end user.

3. Overview: The Pervasive Use of Tags

To respond to users urgent needs for an expanding variety of situational applications, our work focuses on ways to support the composition of components into flow-based applications, as depicted in Figure 1.

Our approach to composing these applications relies on the use of faceted, tag-based descriptions. We use tags, associated with organizing facets, to describe

- the processing goals for an application
- the application's data
- the application's components
- the structure of families of related applications

Processing in flow-based applications is performed by a graph of interconnected components, each transforming input data to output data. At a syntactic (code) level, the compatibility of each component is typically defined and constrained using types declared in component interfaces and used in, e.g., method invocations. But combining these components into functioning assemblies requires more than syntactic compatibility - it also requires knowledge of the component's semantics, not typically available at the type level. For the latter, we rely on tag-based component descriptions. Each component's inputs and outputs are described by (likely different) sets of tags.

The flow depicted in Figure 1 might have been assembled for a commodities broker, who is concerned with risks associated with commodity prices for, say, corn. She identifies *weather* as a natural hazard, potentially posing an investment risk (or opportunity). Figure 2 shows some examples of facets relevant to this domain (Forecast Metrics, Geopolitical Regions, etc.) as well as some examples of tags belonging to these facets (RelativeHumidity, BostonMetro, etc).

These facets and tags play a central role in the specification of users' processing goals, in the description of components, and in the assembly of components into flow-based applications. In addition, they can be used in pattern templates that guide in the process of architecting, engineering, and testing flow-based applications.

3.1 Tag-Based Goals

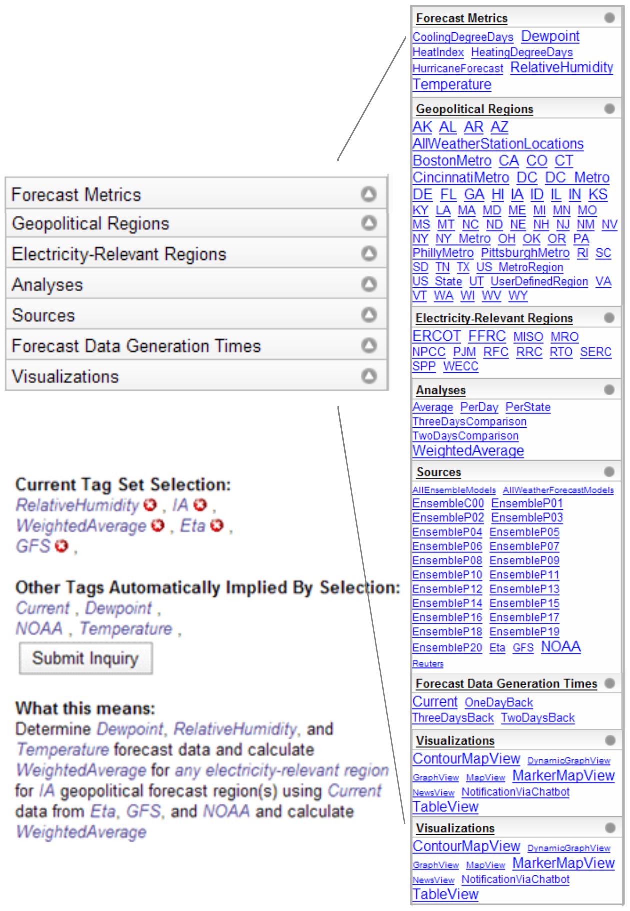
Any given flow-based application can be thought of as achieving some processing goal, typically the production of some kind of information needed by the user. In typical information systems, users express their information goals as queries against some data store. Some query processor takes that query, interprets embedded operations – possibly optimizing to improve access – and then executes the query. Our approach is centered around the users' expression of a *goal* to be achieved by some composed flow. Goals require neither the identification of the information source(s) nor the description of the means of retrieving and processing the data. Thus, if the user wants an estimate of the size and speed of a hurricane, she would express that and only that. How the result is determined – whether using a flow that draws the answer from some web service or using a flow with components that examine satellite imagery – would depend on which components were available and tagged as delivering this information, and which flow is determined "best" (by some measure of "best").

While most users find it challenging to craft some formal description of their information needs, many are quite comfortable with faceted navigation. Look at any web-based shopping experience and you'll see some form of faceted navigation of tags that quickly leads a potential customer from a vague notion of *"shoes to browse"* to the style, size, and color of *"the shoes I gotta have."* The same mechanism can be used by an information analyst who seeks information to inform some important decision. Thus, users formulate their processing goals as simple collections of tags, possibly accumulated with the help of a faceted browsing interface.

For example, a commodities broker might want to watch for projected extremes in relative humidity that might indicate a drought, indicating an opportunity to trade corn futures. She would express this as the goal RelativeHumidity, IA, WeightedAverage, GFS, Eta, ContourMapView, which represents a request for a flow that delivers the relative humidity forecast for the state of Iowa, based on the weighted average of the forecasts as projected by two weather models, NOAA's GFS and Eta forecast models, respectively [17], presented on a contour map. She has requested the weighted average of two different predictions, based on two forecast models, perhaps to arrive at a more precise prediction.

This goal forms the basis for composing the flow in Figure 1 (any flow satisfying this goal must contain these tags, according to the composition model described in section 5). This particular flow takes in raw data values from different sources and processes them to produce the desired result data.

In the case of automated composition, a goal-driven user interaction paradigm shields the end-user from the complexities of manually composing appropriate flows. Using



Current Tag Set Selection:
RelativeHumidity, IA, WeightedAverage, Eta, GFS,

Other Tags Automatically Implied By Selection:
Current, Dewpoint, NOAA, Temperature,

What this means:
Determine Dewpoint, RelativeHumidity, and Temperature forecast data and calculate WeightedAverage for any electricity-relevant region for IA geopolitical forecast region(s) using Current data from Eta, GFS, and NOAA and calculate WeightedAverage

Forecast Metrics
CoolingDegreeDays Dewpoint HeatIndex HeatingDegreeDays HurricaneForecast RelativeHumidity Temperature

Geopolitical Regions
AK AL AR AZ AllWeatherStationLocations BostonMetro CA CO CT CincinnatiMetro DC DC_Metro DE FL GA HI IA ID IL IN KS KY LA MA MD ME MI MN MO MS MT NC ND NE NH NJ NM NV NY NY_Metro OH OK OR PA PhillyMetro PittsburghMetro RI SC SD TN TX US_MetroRegion US_State UT UserDefinedRegion VA VT WA WI WV WY

Electricity-Relevant Regions
ERCOT FFRC MISO MRO NPCC PJM RFC RRC RTO SERC SPP WECC

Analyses
Average PerDay PerState ThreeDaysComparison TwoDaysComparison WeightedAverage

Sources
AllEnsembleModels AllWeatherForecastModels EnsembleC00 EnsembleP01 EnsembleP02 EnsembleP03 EnsembleP04 EnsembleP05 EnsembleP06 EnsembleP07 EnsembleP08 EnsembleP09 EnsembleP10 EnsembleP11 EnsembleP12 EnsembleP13 EnsembleP14 EnsembleP15 EnsembleP16 EnsembleP17 EnsembleP18 EnsembleP19 EnsembleP20 Eta GFS NOAA Reuters

Forecast Data Generation Times
Current OneDayBack ThreeDaysBack TwoDaysBack

Visualizations
ContourMapView DynamicGraphView GraphView MapView MarkerMapView NewsView NotificationViaChatbot TableView

Visualizations
ContourMapView DynamicGraphView GraphView MapView MarkerMapView NewsView NotificationViaChatbot TableView

Figure 2. Example of faceted navigation menu and user-selected, tag-based goal

tag-based goals is appealing in its simplicity; even non-programmers can easily construct goal by selecting a set of tags.

One effect of using tag based goals is the potential ambiguity in interpreting a set of tags. The same set of tags can be interpreted in different ways by different users. We tackle this problem, in part, through a user-interface that shows (and speaks) a natural language interpretation of the goal from the set of tags, so as to provide feedback to the end-user on how the system interprets the goal. In addition, the interface generates a natural language description of the composed flow so that the user can get an idea of the processing performed to generate the results, as well as the specific properties of the results.

Figure 2 shows one of our faceted navigation interfaces that guides the user in formulating a goal. Navigating through a faceted collection of tags, our commodities broker clicks on any of the unexpanded facets (depicted on the left) to reveal a cloud of tags organized under that facet (depicted on the right).[1]

[1] Note that some tags are larger, indicating that they are relevant to a larger number of user-specifiable goals.

These tags appear in the faceted tag cloud because they are associated with information that can be produced by at least one flow built from a library of tag-described components. That is, some flow is capable of producing RelativeHumidity, another is capable of producing a WeightedAverage, etc. The fact that all of the tags in the aforementioned goal are jointly selectable is because at least one flow can be assembled producing information satisfying this collection of tags. In this case, the flow depicted in Figure 1 was selected by MARIO as the best assembly of known components to satisfy the user-specified goal.

As the comodities broker selects the tags – in any order – the selected tags are added to the goal. As with faceted catalog navigation, selection of these tags successively refines the goal. Asking merely for RelativeHumidity results in the assembly of a large number of deployable flows, each of which would produce relative humidity values, using various models, for various forecast timeframes, etc., in the same way asking for running shoes would display pages upon pages of shoes in all available sizes, colors,.... The key for the user is to navigate to the desired refined goal, preferably with a minimum of clicks; the key for application architects and component designers is to craft the components, their descriptive tags, and the organizing facets, such that users can quickly pick the tags needed for their specific processing needs, resulting in one or more processing graphs that fulfills those needs.

The result of deploying and running this flow might look like the marker map in upper right panel of Figure 3.

3.2 Tags for Component Description

Briefly, component descriptions are black-box functional descriptions, using tags to identify each component's applicability to some task(s). We use taxonomically arranged tags to describe a component's input data requirements, output data capabilities, and, in some cases, configuration parameters. Using this information, flows can be assembled to achieve some tag-described processing goals, connecting a collection of components by associating the tags describing some components' outputs to tags describing other components' inputs. Matching can be either a direct match (Temperature = Temperature) or a taxonomic match (Temperature satisfies WeatherMetric), since the latter is a "parent tag" of the former - see figure 4(c).

Both domain-specific analytic components (e.g., weather or financial models) and generic components (averages or comparisons of time series data), can be assembled into multiple flows, each representing alternative means of addressing a particular processing goal. MARIO performs this assembly automatically, using a planner, presenting multiple alternative component assemblies for a given processing goal.

Figure 4 shows the tagging and assembly of a subset of the components in the Figure 1 flow, with a simplified subset of the the components' tags: 4(a) shows the components and their tagging, 4(b) shows the assembly of the components into a subflow, and 4(c) shows a fragment of the tag taxonomy.

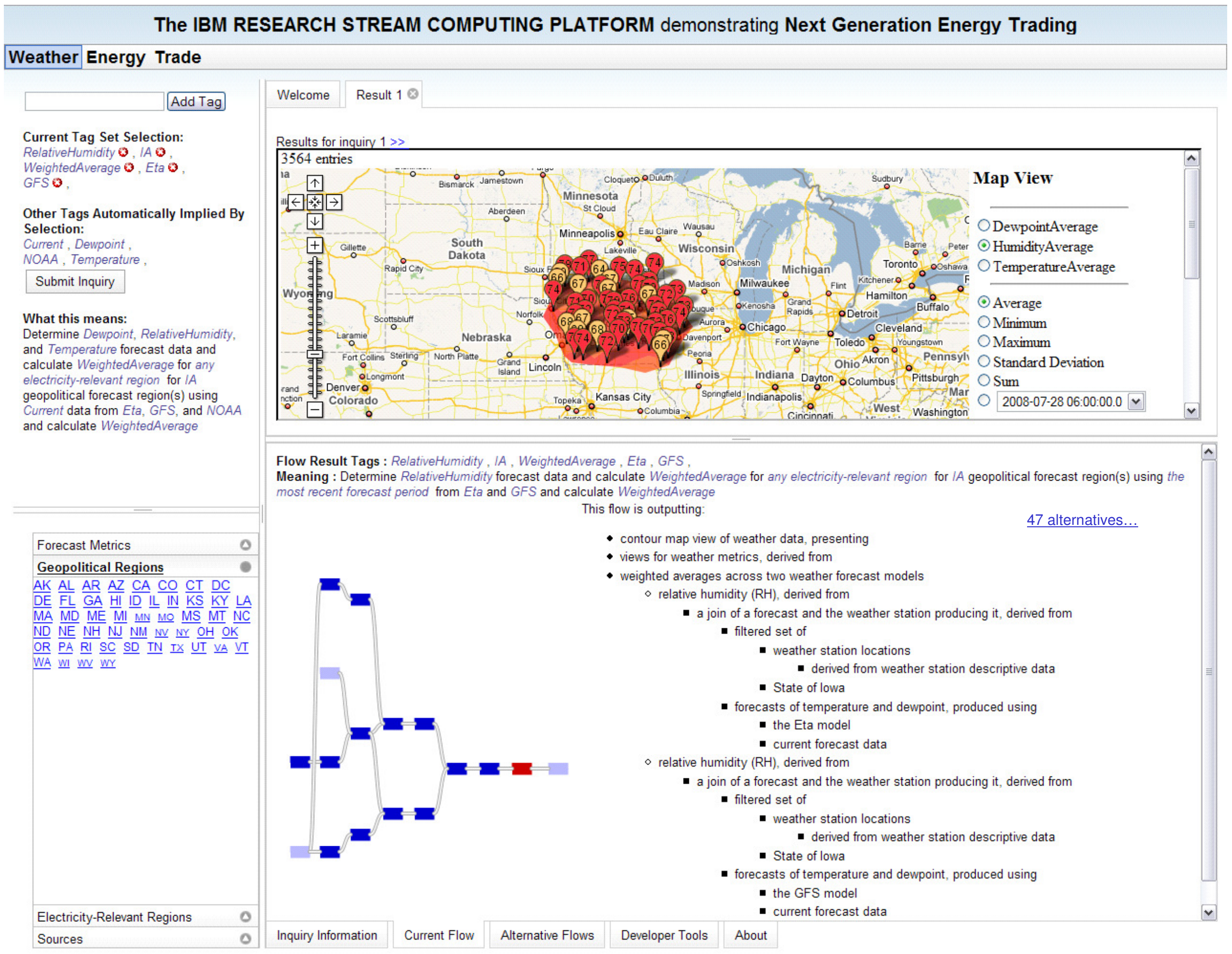
The input to the *Relative Humidity Calculator* component is tagged with Temperature and Dewpoint, indicating that, in order to calculate a relative humidity value, both metrics are needed. The output is tagged with RelativeHumidity, indicating that the component produces a RelativeHumidity output, for consumption by some "down-flow" component. So, the *Relative Humidity Calculator* component can be assembled with some up-flow component if that up-flow component has at least one output tagged as producing at least Temperature and Dewpoint.

The *Weighted Average for Two Weather Metrics* component has two inputs, each described with a WeatherModel tag and a ?WeatherMetric *variable*. It also has a single output described with a WeightedAverage tag and a ?WeatherMetric variable. As depicted in 4(c), the ?WeatherMetric variable is of type WeatherMetric and thus can be replaced by the tag Temperature, Dewpoint, or RelativeHumidity. Note that the variable also establishes the constraint that the ?WeatherMetric variable on the two inputs and the one output must be replaced with the same tag – in this case, the RelativeHumidity tag. (For a more formal treatment, see section 5.)

In assembling the flow in Figure 1 these two components can be assembled into the three-component sub-flow in 4(b), to produce the weighted average of relative humidity predictions across the GFS and Eta models 1) because the RelativeHumidity output of each *Relative Humidity Calculator* instance satisfies the ?WeatherMetric constraint on each of the inputs of the *Weighted Average for Two Weather Metrics* component, and 2) because the GFS and Eta tags, propagated from somewhere up-flow, satisfy the WeatherModel constraint on each input.

The propagation occurs because some up-flow components (specifically, the *NOAA GFS Forecast Data* and the *NOAA Eta Forecast Data* components) introduced those tags and declared them as *sticky tags* (of type _StickyTag). By doing this, these WeatherModel tags are automatically propagated from the point of introduction down the flow to the right-most components' outputs. Similarly, the Temperature and Dewpoint tags are propagated from each of the "Parse Temperature, Dewpoint Forecasts" components.

This is one of a family of propagation mechanisms that are key to allowing general-purpose components, such as the generic *Relative Humidity Calculator*, to be inserted into a specific flow, without requiring any modification to the generic component's description (in spite of the fact that the tag is needed down-flow). So, for example, we do not need to create a special-purpose *Relative Humidity Calculator* that includes a WeatherModel tag on its input and output descrip-


The IBM RESEARCH STREAM COMPUTING PLATFORM demonstrating Next Generation Energy Trading

Weather Energy Trade

Add Tag

Current Tag Set Selection:
RelativeHumidity , *IA* , *WeightedAverage* , *Eta* , *GFS* ,

Other Tags Automatically Implied By Selection:
Current , *Dewpoint* , *NOAA* , *Temperature* ,

Submit Inquiry

What this means:
Determine *Dewpoint*, *RelativeHumidity*, and *Temperature* forecast data and calculate *WeightedAverage* for *any electricity-relevant region* for *IA* geopolitical forecast region(s) using *Current* data from *Eta*, *GFS*, and *NOAA* and calculate *WeightedAverage*

Forecast Metrics

Geopolitical Regions

AK AL AR AZ CA CO CT DC
DE FL GA HI ID IL IN KS KY LA
MA MD ME MI MN MO MS MT NC
ND NE NH NJ NM NV NY OH OK
OR PA RI SC SD TN TX UT VA VT
WA WI WV WY

Electricity-Relevant Regions

Sources

Welcome | Result 1

Results for inquiry 1 >>

3564 entries

Map View

( ) DewpointAverage
(•) HumidityAverage
( ) TemperatureAverage

(•) Average
( ) Minimum
( ) Maximum
( ) Standard Deviation
( ) Sum
( ) 2008-07-28 06:00:00.0

Flow Result Tags : *RelativeHumidity* , *IA* , *WeightedAverage* , *Eta* , *GFS* ,
Meaning : Determine *RelativeHumidity* forecast data and calculate *WeightedAverage* for *any electricity-relevant region* for *IA* geopolitical forecast region(s) using *the most recent forecast period* from *Eta* and *GFS* and calculate *WeightedAverage*

This flow is outputting:

47 alternatives...

- contour map view of weather data, presenting
- views for weather metrics, derived from
- weighted averages across two weather forecast models
  - relative humidity (RH), derived from
    - a join of a forecast and the weather station producing it, derived from
      - filtered set of
        - weather station locations
          - derived from weather station descriptive data
        - State of Iowa
      - forecasts of temperature and dewpoint, produced using
        - the Eta model
        - current forecast data
  - relative humidity (RH), derived from
    - a join of a forecast and the weather station producing it, derived from
      - filtered set of
        - weather station locations
          - derived from weather station descriptive data
        - State of Iowa
      - forecasts of temperature and dewpoint, produced using
        - the GFS model
        - current forecast data

Inquiry Information | Current Flow | Alternative Flows | Developer Tools | About

Figure 3. MARIO UI: Goal Composition (ul), Facets & Tags (ll), Assembled Flow (lr), and Processing Results (ur)

tions, just because the *Weighted Average* component requires a WeatherModel input. Sticky tag propagation handles that. [2]

The tags depicted here are descriptive of the information flowing between components. Tags can also indicate the state of processing of some piece of data. The tags can mean anything the description author desires, but must be applied with caution, since each tag influences flow assembly. So, a carelessly placed tag can lead to potentially many incorrectly assembled flows, or to the non-assembly of otherwise correct flows.

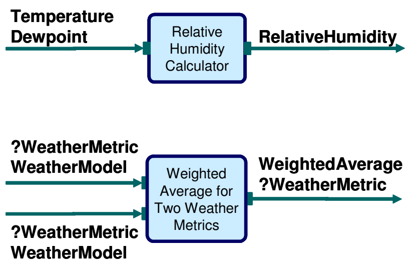
There is a direct link here between the users' collective processing needs and the architects' design of the associated application. The same tags are navigated by the user in formulating a processing goal and also to describe the components' functional capabilities. Thus, there is a necessary and productive interaction between the users' application requirements and the application architects' designs. Simply put, this tag-centric approach forces the components' design to be aligned with users' information requirements.

3.3 Tooling for Application Composition

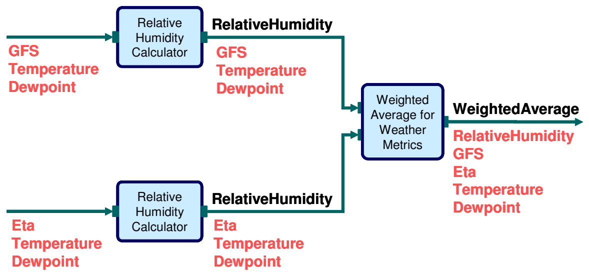
So far, we have shown a glimpse of faceted navigation of tags (Figure 2), a hand-drawn example of an automatically generated flow (Figure 1) and, in the introduction, a brief mention of "mash-up visualizations of the flows' outputs." Each of these elements has been implemented in MARIO, one example of a system that supports the composition and deployment of flow-based applications.

This tooling provides support for a spectrum of composition modes covering manual, partially automated (mixed initiative) and fully automated composition. The formally specified composition constraints, expressed in tag-based descriptions enable automated assembly, can guide partially automated assembly, and can help eliminate mistakes in manual and partially automated composition, depending on the quality of component descriptions and the level of end user's understanding of individual component capabilities.

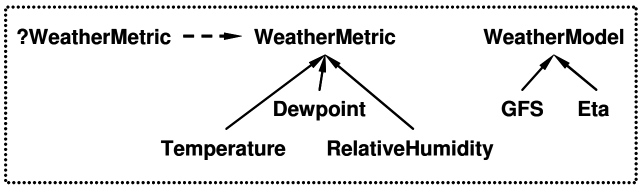
[2] However, the programming model – or the deployment infrastructure – must support the pass-through of such data.



(a) Tagged individual components



(b) Assembled flow



(c) Tag taxonomy fragment, with variable type declaration

Figure 4. Assembling a flow using tagged components

Figure 5 depicts an architecture that supports a combination of automated flow generation (left side) and manual and mixed initiative flow composition (right side).

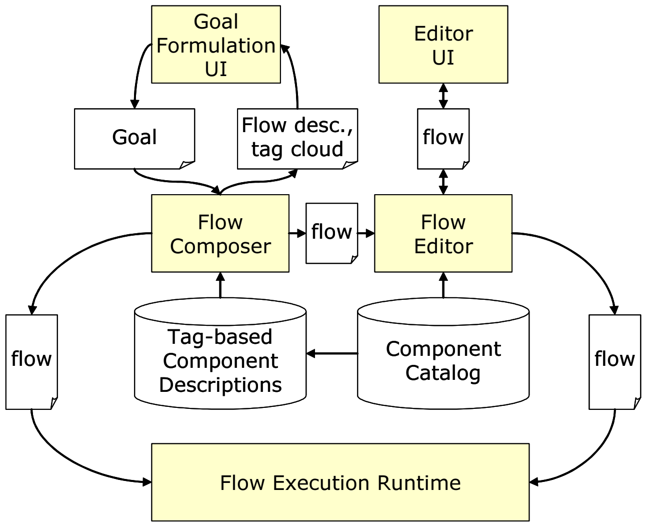


Figure 5. Flow composition and deployment architecture

3.3.1 Automated Composition

In the fully automated mode, flows are created and deployed automatically. The end user of the generated flow does not need to understand the functionality of individual components. His central focus is on expressing processing goals, using tags drawn from familiar vocabularies.

For automated composition, the *Flow Composer* consults a repository of tag-based component descriptions and generates a set of all possible flows. The Flow Composer also generates a faceted tag cloud containing any tag that describes any component in the set of generated flows. The tag cloud is sent to the *Goal Formulation UI*, which presents to the user a means of navigating the facets and tags needed to specify a processing goal (figures 2 and 3 present two views of one such UI). With each subsequent tag selection, the goal is expanded. The Flow Composer generates a new, smaller set of flows narrowing the scope to those flows that produce the tags in the expanded goal and regenerates the tag cloud accordingly.

In a partially automated composition scenario, the application flow can be partially assembled automatically based on specified goals, and later manually changed by the user before deployment and execution. This approach requires end users to have a better understanding of component functionality, but can tolerate less precise component descriptions from developers.

The Goal Formulation UI also displays the the "best" flow for the given goal (Figure 3) as determined by the Flow Composer, selected based on a combination of result quality and resource consumption metrics. When the user has selected the tags expressing her processing goal and is satisfied with the generated flow, she can push a button and the generated flow is deployed to, and run in, the Flow Execution Runtime.

Note, in Figure 3, that the Flow Composer has produced 47 alternative flows. Including the flow displayed, there are 48 possible flows for the goal "IA, RelativeHumidity, WeightedAverage, Eta, GFS." This is because the goal is general enough that there are many possible ways of achieving it. The user can explore these possible flows by selecting additional tags from the displayed tag cloud. For example, the user can select additional Geopolitical Region tags to indicate she wants the results for multiple US States. The user

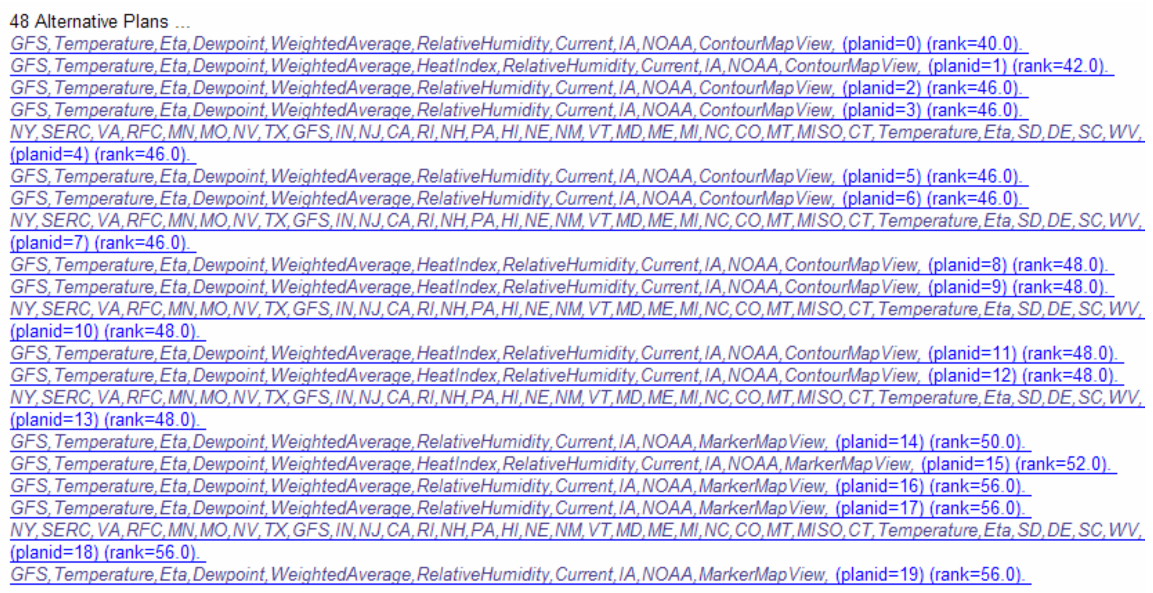

48 Alternative Plans ...
GFS,Temperature,Eta,Dewpoint,WeightedAverage,RelativeHumidity,Current,IA,NOAA,ContourMapView, (planid=0) (rank=40.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,HeatIndex,RelativeHumidity,Current,IA,NOAA,ContourMapView, (planid=1) (rank=42.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,RelativeHumidity,Current,IA,NOAA,ContourMapView, (planid=2) (rank=46.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,RelativeHumidity,Current,IA,NOAA,ContourMapView, (planid=3) (rank=46.0).
NY,SERC,VA,RFC,MN,MO,NV,TX,GFS,IN,NJ,CA,RI,NH,PA,HI,NE,NM,VT,MD,ME,MI,NC,CO,MT,MISO,CT,Temperature,Eta,SD,DE,SC,WV, (planid=4) (rank=46.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,RelativeHumidity,Current,IA,NOAA,ContourMapView, (planid=5) (rank=46.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,RelativeHumidity,Current,IA,NOAA,ContourMapView, (planid=6) (rank=46.0).
NY,SERC,VA,RFC,MN,MO,NV,TX,GFS,IN,NJ,CA,RI,NH,PA,HI,NE,NM,VT,MD,ME,MI,NC,CO,MT,MISO,CT,Temperature,Eta,SD,DE,SC,WV, (planid=7) (rank=46.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,HeatIndex,RelativeHumidity,Current,IA,NOAA,ContourMapView, (planid=8) (rank=48.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,HeatIndex,RelativeHumidity,Current,IA,NOAA,ContourMapView, (planid=9) (rank=48.0).
NY,SERC,VA,RFC,MN,MO,NV,TX,GFS,IN,NJ,CA,RI,NH,PA,HI,NE,NM,VT,MD,ME,MI,NC,CO,MT,MISO,CT,Temperature,Eta,SD,DE,SC,WV, (planid=10) (rank=48.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,HeatIndex,RelativeHumidity,Current,IA,NOAA,ContourMapView, (planid=11) (rank=48.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,HeatIndex,RelativeHumidity,Current,IA,NOAA,ContourMapView, (planid=12) (rank=48.0).
NY,SERC,VA,RFC,MN,MO,NV,TX,GFS,IN,NJ,CA,RI,NH,PA,HI,NE,NM,VT,MD,ME,MI,NC,CO,MT,MISO,CT,Temperature,Eta,SD,DE,SC,WV, (planid=13) (rank=48.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,RelativeHumidity,Current,IA,NOAA,MarkerMapView, (planid=14) (rank=50.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,HeatIndex,RelativeHumidity,Current,IA,NOAA,MarkerMapView, (planid=15) (rank=52.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,RelativeHumidity,Current,IA,NOAA,MarkerMapView, (planid=16) (rank=56.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,RelativeHumidity,Current,IA,NOAA,MarkerMapView, (planid=17) (rank=56.0).
NY,SERC,VA,RFC,MN,MO,NV,TX,GFS,IN,NJ,CA,RI,NH,PA,HI,NE,NM,VT,MD,ME,MI,NC,CO,MT,MISO,CT,Temperature,Eta,SD,DE,SC,WV, (planid=18) (rank=56.0).
GFS,Temperature,Eta,Dewpoint,WeightedAverage,RelativeHumidity,Current,IA,NOAA,MarkerMapView, (planid=19) (rank=56.0).

Figure 6. Alternative plans for selected goal tags

can also select additional forecast metrics, like HeatIndex, for calculation and display. [3] The goals and the ranks for some of the alternative flows are shown in Figure 6. In this case, the ranks are obtained based on a cost metric, where every component is associated with a cost, and the cost of an application is the sum of the costs of the individual components. Our composer can support other definitions of the cost metric as well [24].

Also note that the composer has included Current (meaning current as opposed to historical data), NOAA, Dewpoint, and Temperature in the "Current request and its interpretation." These are the remaining tags that describe the automatically selected first-choice flow but that were not explicitly requested by the user. In this case, the goal is just what the user wanted, and a few clicks less than might have been needed. If, however, the user instead wanted to results based on historical data, she could explicitly select, e.g., TwoDaysBack (overriding the composer-selected value of "Current"), and the generated flow would reflect that request.

3.3.2 Manual and Assisted Composition

Manual and mixed initiative composition modes come into play when a user wants or needs to manually compose or alter a flow. This might arise for the application architect, when she is experimenting with new components or flows or needs to alter existing flows to reflect new application needs, or when a known composition is required but somehow is not supported in the tags.

In such cases the flow composition is defined using the Editor UI, either from scratch or as a derivation of a generated flow. In the partially automated composition scenario, the application's components can be partially assembled automatically, based on specified goals, and manually changed by the user before deployment and execution. This approach can tolerate less precise component descriptions, but requires the end user to understand component functionality, something not required in the fully automated case.

4. Overview of Software Engineering Methodology

The main purpose of our software engineering methodology is to build a library of reusable components that can be assembled into different applications. As introduced by Prieto-Díaz [19], faceted descriptions can be used to aid in the classification and retrieval of components. For purposes of automated assembly, retrieved components must be designed to be assembled in a variety of contexts and assembled only with those other components that are syntactically and semantically compatible. This places burden on the application architect to both design reusable (sub)assemblies and to appropriately constrain the component descriptions to prevent undesirable assemblies. In the absence of a comprehensive formal model and provably correct assemblies, testing is a necessity, as are any other techniques that improve the quality and reliability of component (sub)assemblies.

We propose an overall software engineering process that is specifically tailored to the needs of flow-based application composition. Figure 7 shows the stages in our software engineering process. Our process can be roughly described as spiral-like refinement, starting with requirements and iterating over the development and testing of application assemblies composed from both reusable components and newly

[3] This might sound complicated, but the faceted navigation mechanism simplifies the user experience to something much like the shoe shopper experiences when looking at various sizes or styles of shoes.

developed components. Finally, the set of all components is made available to end-users for composition into diverse flows. In addition, user interfaces (such as the the one shown in figures 2 and 3) that facilitate the composition of flows by end-users are generated.

The development and composition processes include a combination of top-down and bottom-up elements. The top-down elements provide structure to the design and development processes, and guarantee that certain compositions can be achieved from the set of developed components. The bottom-up elements enable reuse of both individual and composite components in different contexts, and also allow for the serendipitous assembly of new flows in response to new end-user requests or other runtime requirements.

The interplay between the top-down development lifecycle and the bottom-up composition approach is crucial in our methodology. As a result of the top-down development, certain components are developed and associated with tag-based descriptions. These components are guaranteed to be composable into certain flows that meet the end-user requirements. However, when an end-user submits a goal to the system, it constructs satisfying flows, in a bottom-up manner, using the tag-based descriptions of the components. These flows include those that were designed by the application architects during the top-down lifecyle. In addition, it may also come up with new flows that were not explicitly designed. This dynamic bottom-up composition also potentially allows the assembly of flows for goals that were not part of the initial end-user requirements.

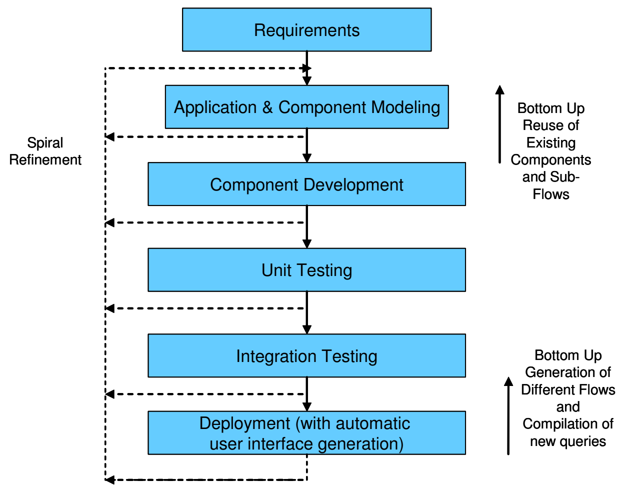

Requirements
Application & Component Modeling
Component Development
Unit Testing
Integration Testing
Deployment (with automatic user interface generation)
Spiral Refinement
Bottom Up Reuse of Existing Components and Sub-Flows
Bottom Up Generation of Different Flows and Compilation of new queries

Figure 7. Software Engineering Process for Automated Composition

We shall now provide an overview of the different stages in our software engineering process. Our process starts with the description of functional requirements from end-users. In information processing systems, the functional requirements describe the kinds of data the end-user desires. In our approach, these functional requirements are expressed as patterns of goals that the user would like to submit. Note that this paper focuses on functional requirements and not non-functional requirements like security, performance and cost.

The functional requirements are taken by an application architect who comes up with a high-level design of the overall application(s) and of individual components. The architect first constructs one or more application templates that satisfy the requirements. An application template is a high-level description of the flow structure and is modeled as a graph of abstract sub-flows, where each abstract sub-flow performs a certain segment of the overall required information processing. Each abstract sub-flow in turn consists of a graph of component classes, where a component class is an equivalence class of components that share similar properties and are substitutable in certain contexts. The modular and substitutable nature of components are critical in making composition possible. In addition, the decomposition of the application into abstract sub-flows allows reuse of not just components, but also entire sub-flows.

The application architect can reuse existing components (and component classes) in designing the application. In some cases, however, new components may need to be developed, or existing components modified, to satisfy new end-user requirements. The new components may either be part of an existing component class, or may belong to entirely new component classes. The architect defines the semantic requirements of the component in terms of tags describing the input and output data. In addition, the architect defines the syntactic interfaces of the component so as to enable its interaction with other components in the application. These semantic and syntactic component requirements get passed to a developer, who develops the component. Also, as shown in the figure, the process of application template construction and refinement, component requirements and component developed is typically iterative involving both the application architect and the developer.

Once the component is developed, it goes through unit tests as defined by the application architect and/or the developer. Integration testing is a crucial part of the process, especially for automated composition. There are two levels at which integration tests are performed - at the goal compilation level, where the tag-based semantic composability of different components is checked; and at the deployment level, where the runtime interactions of different components is checked. At both levels, an integration test is performed by selecting sample goals that belong to the goal pattern. Each sample goal is tested by submitting it to the flow composer, checking to see if the set of flows generated includes at least one that is prescribed by the application template, deploying a sample of the flows generated on the underlying platform, and verifying the end-results generated by the flow.

The final step is deployment of any new or modified components in the backend platform(s). The deployment results in two changes to the system. First, the end-user interface is modified, automatically, to enable users to express the new kinds of goals (as described in the requirements). This may result in new tags or combinations of tags that can be selected from the tag cloud in Figure 2. Second, the new components can be included in flows. These flows are not just the ones defined in the application templates, but potentially new flows that may be created based on the semantic definitions of the new set of components.

Although the methodology as presented has a top down emphasis, it does support the bottom-up construction of flows. First, in the application template construction stage, it is possible to reuse existing components or sub-flows in defining the template. Second, after deployment, our composition approach is not constrained by the pre-defined application templates. Instead, our flow composer can construct new flows to satisfy user goals using the available components. The flow composer uses a planning approach; for each goal, it composes a plan by using the input and output descriptions of individual components. The flow composer is not aware of the application templates; instead it creates flows anew from the goal specification. This allows for the spontaneous generation of new flows from existing components that were not necessarily designed by the application architect.

One of the key features of our system is the end-user interface, which presents with a tag cloud from which they can select one or more tags to express their processing goals. The tag cloud is automatically generated, based on an analysis of the valid flows that can be generated for the given set of components. In other words, the tag cloud includes all those tags that appear in at least one valid flow, assembled from previously developed and tagged components, thus presenting users with the ability to dynamically assemble flows responsive to their current processing requirements. This tag-based end-user interface, together with the automatic goal compilation, make it relatively easy for any users (in particular, non-programmers) to construct queries, deploy situational applications, and obtain the desired information.

The roles involved in the creation and use of our flow-based assemblies are the usual: Requirements Engineers, End-Users, Application Architects, Developers, Unit and Integration Testers. A few tasks differ, though. Users, Requirements Engineers, and Application Architects are concerned with modeling the various facets, tags, and taxonomies that convey the users' perspective on relevant vocabularies and processing results. Application Architects are concerned with partitioning application functionality based on various blends of reusability and scalability, taking into consideration the various semantic contexts in which the components and sub-assemblies must function.

5. Formal Model of Tags, Goals and Components

The basic unit of describing requirements, components and goals is a tag (or a keyword). The notion of a "tag" is similar to various collaborative tagging applications (or folksonomies) that have arisen in Web 2.0 (such as del.icio.us and Flickr) where users annotate various kinds of resources (like bookmarks and images) with tags. These tags are then used to aid search and retrieval of resources. A key aspect of the tagging model is that it is relatively simple, in comparison to more expressive models such as those based on Semantic Web ontologies and other formal logics. Hence, the use of tags offers a lower barrier to entry, for both end-users and developers, to use tags to describe resources. In our case, the resources are various types of data artifacts, like data streams, files, messages, etc.

There is, however, an important difference between our tagging model and other collaborative tagging applications in Web 2.0. For purposes of composition, we need greater control on the evolution of the set of available tags and on the way tags are assigned to various kinds of data and to components. While most other folksonomies allow free and arbitrary tagging of resources by users, tagging in our system requires more careful consideration since the assignment of tags has ramifications on the composability of components.

5.1 Tag Hierarchies

Let $T = \{t_1, t_2, \ldots, t_k\}$ be the set of tags in our system. In most social tagging applications, the set of tags, T, is completely unstructured, i.e. there is no relation between individual tags. Introducing a hierarchy structure in T, however, enhances the expressivity by allowing additional tags to be inferred for resources, and also aids end-users in navigating a large number of tags.

A tag hierarchy (or taxonomy), H, is a directed acyclic graph (DAG) where the vertices are the tags, and the edges represent "sub-tag" relationships. It is defined as $H = (T, S)$, where T is the set of tags and $S \subseteq T \times T$ is the set of sub-tag relationships.

A sub-tag has an associative semantics, where if x is a sub-tag of y, then x is commonly associated with y by the community of users. In other words, if a resource is tagged with x, then it would be semantically appropriate for it to be tagged with y. In this paper, we use the symbol $\prec$ to represent sub-tag relationships. For example, $\mathsf{NewYorkTimes} \prec \mathsf{Newspaper}$, which implies that any resource tagged with NewYorkTimes may also be tagged with Newspaper. In our methodology, we assume that the tag hierarchy is defined by application architects and component developers.

Formally, a tag $t_1 \in T$ is a sub-tag of $t_2 \in T$, denoted $t_1 \prec t_2$, if all resources annotated by t_1 can also be annotated by t_2. The sub-tag relation is transitive, i.e. if $t_1 \prec t_2$ and $t_2 \prec t_3$ implies $t_1 \prec t_3$ for $\forall t_1, t_2, t_3 \in T$. For notational convenience, we will further assume that each tag is a sub-tag of itself, i.e. $\forall t \in T,\ t \prec t$.

5.2 Data Model

In our approach, tags are used to describe data artifacts. A data artifact may be a data stream, a table in a database, a file, an RSS feed, a web page, a message exchanged between services, etc. The tags may describe both the syntax and the semantics of the data artifacts.

Each data artifact, a is characterized by a set of tags $d(a) \subseteq T$. For example, consider a component that pulls in weather forecast data from an FTP server maintained by NOAA (National Oceanographic and Atmospheric Administration). The current weather forecast obtained from one such model called Eta is available as an ASCII file on the FTP server. This data artifact, a_1, may be described by the tags Eta, NOAA, Text, 3DayForecast, ASCII, Current, Temperature} . These tags provide syntactic and semantic information about the data.

Data artifacts need not only be the raw data from various sources, but may also be processed data. For example, if the ASCII file was processed by a flow of components to calculate the average temperature of New York City over the three day period, then the resulting data artifact, a_2, may be described by the tags Eta, NOAA, Average, 3DayPeriod, NewYorkCity, Temperature}. Depending on the kind of flow and the runtime environment, the result data may be made available in a variety of forms, e.g. as a SOAP message in a web service environment, an RSS feed in a mashup environment, or a single streamed tuple in a stream processing environment.

5.3 End-User Goal Model

End-user goals describe the semantics of the desired data artifacts. A goal, $q \subseteq T$, selects a subset $R.q$ of a resource set $R = \{r\}$ such that each resource in the selected subset has all the tags in q or subtags thereof. Formally,

$$R.q = \{r \in R | \forall t \in q\ \exists t' \in d(r) \text{ such that } t' \prec t\}.$$

For example, the goal {MOS, Average, NewYorkCity} will match the data artifact, a_2 described in Section 5.2, making use of the sub-tag relationship Eta $\prec$ MOS (Figure 4). MOS, which stands for Model Output Statistics, refers to a general class of weather forecast models, which includes Eta.

The explicit matches of a goal are data artifacts that satisfy the goal requirements. Implicitly, however, the results of a goal are not just the data artifacts but the flows that produce the desired artifacts. Goal satisfaction, thus, can be viewed as a search in the space of all possible flows that can be constructed from a given set of data sources and processing components.

Note that while the goal, by definition, selects all matching data artifacts (and flows), the end-user does not have to be presented with all matching resources, because there may be too many. Also, any algorithm that searches for satisfying flows need not actually come up with all possible flows because there may be too many of them, and it may be inefficient to search for all of them. It may instead try to come up with a ranked list of top flows based on some ranking function.

5.4 Facets

Facets represent dimensions for characterizing resources (data artifacts). Let $F = \{f_i\}$ be the set of facets in our system. Each facet is modeled as a set of tags, i.e. $f_i \subseteq T$. In some facet classification systems, facets are mutually exclusive. That is, two facets share no common elements. In our approach, though, tags may be shared across facets.

Facets play an important role in identifying sets of similar tags that can be grouped together and used to describe abstract flows or components. They also help in organizing the potentially large number of tags in the user interface and thus aid the user in locating desired tags.

5.5 Component Model

Our model uses the tags from the taxonomy to associate semantic information with the input and output data artifacts of components. This semantic information is complementary to the syntactic (or structural) information that may be provided in the interface description of the component (for example, in a WSDL document). Both the semantic and syntactic information may be used together for purposes of checking composability of components.

One of the key features of our model if that it captures the notion of *semantic propagation*, i.e. the semantic description of the output data artifacts of a component depend on the semantics of the input data artifacts. Our model includes two mechanisms to describe how semantic properties are propagated from the input to the output data artifacts. The first is through the use of *common variables* in the inputs and the outputs. Whatever value (tag) the variable is bound to in the input is the same value it is bound to in the output. The second is through the use of *sticky* tags. These are specially designated tags that are directly copied to the output from the input description; even if they do not explicitly appear in the output description. In our model, "stickiness" is a universal property of the tag, i.e. it is not specific to a certain component. Typically sticky tags are those that refer to semantics of the sources (like NOAA) and hence can logically be associated with all raw and derived data produced from this source. Figure 4 shows examples of components as well as the propagation of semantic information from inputs to outputs in a flow of components.

A *variable*, v, is a member of the set V where V is infinite and disjoint from T. A variable is represented with a preceding "?" . Each variable is associated with one or more types (which are also tags). Let $\tau : V \rightarrow T$ be a function that maps a variable in a component description to a set of types. A variable, v can be bound to a tag, t if the tag is a

sub-tag of all the types of the variable, i.e.
$canbind(v, t)$ iff $\forall x \in \tau(v), t \prec x$

An *input data constraint* of an component (or operator), o, denoted by I_o describes the required properties of input data artifacts to a component. It is in the form of a set of tags and variables, i.e. $I_o \subseteq (T \cup V)$.

An *output data description* of a component, o, denoted by O_o describes the properties of an output data artifact produced by a component. It is in the form of a set of tags and variables, i.e. $O_o \subseteq (T \cup V)$.

Let C be the set of all components in the system. A component, $o \in C$, is defined as the pair $(\{I_o\}, \{O_o\})$ where

1. $\{I_o\}$ is a set of zero or more input message constraints
2. $\{O_o\}$ is a set of one or more output message constraints
3. The set of variables in the output set, $\{O_o\}$, is a subset of the set of variables in the input set, $\{I_o\}$. This constraint ensures that no free variables exist in the output description.

Our model also includes other information such as binding (i.e. how exactly to instantiate or invoke a component) and other documentation on the component. The actual binding model depends on the underlying deployment platform. We shall not cover this aspect in this paper.

We note that the folksonomy-based model is less expressive than Semantic Web Service models like OWL-S and WSMO, which can capture richer semantic information about the input and output data, using assertions in description logic, horn logic or other kinds of logics. Our model, for instance, cannot capture relationships between tags. However, based on our experiences, our model can still capture sufficient constraints for use in automated composition. In addition, our model has a lower barrier for entry since it is easier to annotate components using just a set of keywords rather than using assertions in a logical formalism. This will hopefully prompt more developers and end-users to annotate components and reap the immediate benefits of discovery and composition. Once these developers and end-users see the value in annotating components, they can then move on to more expressive models if the need arises.

As a general design principle and in the interest of modularity, it is preferable to design components where the tags on each component's inputs and outputs are associated with a small number of facets. The presence of tags from a larger number of facets is a likely indicator that the component is performing multiple operations crossing multiple domains, and is thus a candidate for further partitioning.

5.5.1 Modeling Component Parameters

Many components can be instantiated or configured with parameter values that influence the way they behave. For example, a parameter to a contour map view visualization component used for visualizing the temperature ranges is the number of contours, which is a measure of the granularity of the result visualization. We model component parameters as input constraints. The value of the parameter can be obtained in two ways - from an end user or from another component that provides a single value to this component.

5.5.2 Composition Constraints

An important part of composing flows in information processing systems is determining whether a data artifact, produced by some component, can be given as input to another component. There are two main kinds of constraints we consider: semantic and syntactic. The semantic constraints can be expressed using the tag-based model. The syntactic constraints depend on the actual deployment platform, and are related to the actual datatypes required and produced by various components. These syntactic constraints can also, in some cases, be described using tags. For example, we can use tags to represent an XML schema or a Java interface name.

The semantics of a data artifact, a, can be described by the set of tags, $d(a)$. We define that $d(a)$ matches an input constraint, I_o (denoted by $d(a) \sqsubseteq I_o$), with a variable substitution function, $\theta : V \rightarrow T$, iff

1. For each tag in I_o, there exists a sub-tag that appears in $d(a)$. Formally, $\forall y \in (I_o \cap T), (\exists x \in d(a), x \prec y)$.
2. For each variable in I_o, there exists a tag in $d(a)$ to which the variable can be bound. Formally, $\forall y \in (I_o \cap V), (\exists x \in d(a), canbind(y, x))$. Also, that mapping $\theta(y) = x$ is created.

If a component has n inputs, then a set of data artifacts $\{d(a_1,), \ldots d(a_n)\}$ match the input constraints $\{I_o^1, \ldots I_o^n\}$ if $d(a_i) \sqsubseteq I_o^i, i = 1 \ldots n$ with a common variable substitution function.

5.5.3 Output Generation

When a component's input requirements are satisfied, the component generates new output data artifacts. The semantics of an output data artifact can be described based on the semantics of the corresponding output data description, O_o, after substituting any variables that appear in the description based on the substitution function, θ and after propagating any sticky tags from any of the inputs to the output.

Let $S \subseteq T$ be the set of all sticky tags. Let $d_i(a), i = 1 \ldots n$, be the n input data artifacts to the component. Let θ be the common variable substitution function used for matching the input data artifacts to the input requirements of the component. The semantics of an output data artifact, b, corresponding to the output description, O_o of a component, can be described by the set of tags, $d(b)$, where

$d(b) = \{t : t \in O_o \cap T\} \cup \{\theta(v) : v \in O_o \cap V\}$
$\cup \{t : t \in (\cup_{i=1}^{n} d_i(a)) \cap S\}$

This means that the output data artifact can be described by all tags that appear in the output description of the component, all tags that are obtained after substituting the variables (based on the matches at the input side), and all sticky tags that appear in the input data artifacts.

5.6 Overall Application Model

Applications are modeled as flows of components that jointly process information to produce desired end-results. A flow is a graph $G(V, E)$ where G is a DAG (Directed Acyclic Graph). Each vertex $v \in V$ is a component instance. Each edge (u, v) represents a logical flow of data artifacts from u to v. The data artifact corresponding to each edge, (u, v), can be described by a set of tags, $d((u, v))$. This model of applications translates easily to various flow models, including BPEL, flows in stream processing systems and in data mashups.

5.6.1 Goal Driven Composition

The problem of goal-driven composition can now be simply defined as the problem of constructing a flow that produces a data artifact satisfying the goal. Given a composition problem $\mathcal{P}(T, C, g)$, where:

- T is a tag hierarchy,
- C is a set of components,
- g is a composition goal, $g \subseteq T$,

the solution set is defined as follows:

The set of solutions $\mathcal{S}(T, C, g)$ to the goal-driven composition problem $\mathcal{P}(T, C, g)$ is the set of all valid applications, $\mathcal{G}$, such that $\forall G(V, E) \in \mathcal{G}$,

- The data artifact corresponding to at least one edge in E must satisfy the goal, i.e. $E.g \neq \emptyset$
- for all component instances in V, at least one data artifact produced by this instance serves as input to another component instance, or satisfies the goal.

The second condition in the definition above helps eliminate from consideration inefficient compositions that have dead-end component instances producing unused objects.

5.6.2 Composition Ranking

Before the set of compositions $\mathcal{S}(T, C, g)$ can be presented to the user, the compositions must be ranked, with those most likely to satisfy user's intent appearing first in the list. The ranking is based on a heuristic metric reflecting composition quality. Each component $c \in \mathcal{C}$ is assigned a fixed cost $cost(c)$. Cost of a component instance in a composition is equal to the cost of the corresponding component.

Rank, $rank(G)$, of a composition, $G(V, E) \in \mathcal{G}$, is the sum of the costs of components instances, i.e.

$$rank(G) = \sum_{c_i \in V} cost(c_i).$$

By default, for all components, $cost(c) = 1$. Hence, the best compositions are the smallest ones. During configuration of the system, the number can be left equal to the default, or configured for some components to reflect component quality.

The composition problem can be solved using manual, automated or assisted approaches. We shall briefly describe our planning-based automated composition approach in this paper, in Section 6.3.

6. Tag-based Software Engineering and Composition

The pervasive use of tags and facets to describe the artifacts produced during various parts of the software engineering lifecycle is key in establishing the connection from end-user requirements to developed components and composable flows. We now describe, formally, how tags and facets are used in the specification of requirements, of application templates and component classes. Finally, the tag-based descriptions of the components are used by a planner to support automatic composition of flows in response to tag-based end-user goals.

6.1 Multi-faceted Requirements Specification

In our software engineering methodology, end-users can specify requirements for the kinds of information they would like to obtain. These requirements are used by the application architects in designing the components and the overall applications.

In any large scale information processing system, many kinds of information must be processed in a variety of ways. Hence, requirements are not specified in terms of actual goals but as whole classes of goals, that are described by goal patterns.

A goal pattern is described as a set of tags and facets. Each facet is associated with a cardinality constraint. The cardinality constraint specifies how many tags in the facet should be part of the goal.

We first define the set of cardinality constraints, CC, as the set of all ranges of positive integers. Then a goal pattern, $QP = \{(x, c)|x \in F, c \in CC\} \cup \{t|t \in T\}$. A goal pattern requirement means that end-users are interested in all data artifacts that can be described by a combination of tags that are drawn from the facets in the goal pattern, according to the cardinality constraints.

An example of a goal pattern is Source[≥ 1], WeatherForecastModel[≥ 2], MultipleModelAnalysis[1], BasicWeatherMetric[≥ 1], Visualization[1].

This represents the class of all data artifacts that can be described by one or more tags that belong to the Source facet, 2 or more tags that belong to the WeatherForecastModel facet, one tag belonging to the MultipleModelAnalysis facet, and one tag belonging to the Visualization facet.

An important point to note is that the goal pattern can refer to a large number of possible goals. So, for example, if there are 5 tags in the Source facet, 50 tags in the Model facet, 5 in MultipleModelAnalysis and 10 in Visualization, there are up to $2^5 \times 2^{50} \times 5 \times 10$ possible kinds of data that may be producible by the information processing system. Hence, the goal pattern helps in succinctly expressing the combinatorial number of possible goals that can be submitted to the system.

6.1.1 Application Templates

An application architect takes a requirement, in the form of a goal pattern, and constructs one or more application templates that can satisfy all the goals in the goal pattern. An application template is a high-level description of the application structure. Each goal instance belonging to the goal pattern can be satisfied by an application instance that belongs to the defined application templates. Figure 8 has an example of an application template. Figure 1 shows one such instantiation of the subgraph.

The application templates are intended to guide the goal answering process. It is important to note that the templates do not capture all solutions. It is possible to assemble a different flow that uses potentially different components to satisfy the same goal.

An application template is a directed acyclic graph, where the vertices are abstract sub-flows and edges represent transfer of data artifacts between components in different sub-flows. Each abstract sub-flow, itself can be described by a directed acyclic graph, where the vertices are component classes and edges represent the transfer of data artifacts between component classes.

Formally, an application template is defined as a directed acyclic graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, p, \lambda)$, where each vertex, $v \in \mathcal{V}$ is an abstract sub-flow and each edge $e \in \mathcal{E}$ represents data flow between abstract sub-flows. Next, $p : g \rightarrow \mathcal{CC}$, where g is a subgraph of $\mathcal{G}$. In other words, p associates each sub-graph with a parallelism constraint. In the example above, one of the subgraphs is associated with a constraint that at least 2 instances of the subgraph run in parallel. By default, a subgraph is associated with a cardinality of 1.

Finally, $\lambda : \mathcal{V} \rightarrow \mathcal{GP}$, where $\mathcal{GP}$ is the set of all possible goal patterns. Each abstract sub-flow in the flow is associated with a goal pattern that describes the kinds of goals that the sub-flow formed by this sub-flow plus all preceding sub-flows in the flow can answer. λ is the function that associates a sub-flow with the goal pattern it produces as output.

6.1.2 Abstract Sub-Flows

An abstract sub-flow is a directed acyclic graph $S(V_S, E_S)$. Each vertex $v \in V_S$ is a component class (defined later). Each edge $(u, v) \in E_S$ represents a logical flow of data artifacts from a component in the class u to a component in the class v. Each stage can be viewed as a high-level component with input requirements and output capabilities.

An example of a stage is shown in Figure 9. It consists of two component classes, the first fetches a file given a URL, and the second parses a weather forecast file.

Figure 10 shows a concrete instance of the abstract sub-flow, where the component classes have been instantiated with specific component instances. In the case of the first component class, the instantiation occurs through the specification of a parameter (a specific URL). The second component class is instantiated with a specific component, called MOSParser, that parses MOS forecasts from NOAA to extract temperature and dew point predictions for various weather stations in the US.

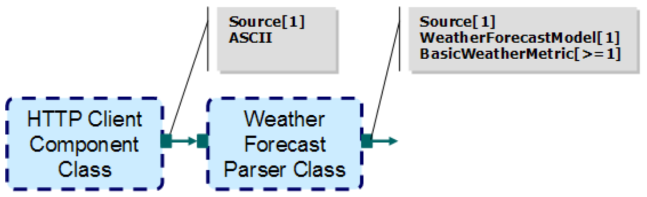


Figure 9. Example Abstract Sub-Flow for Weather Forecast Extraction

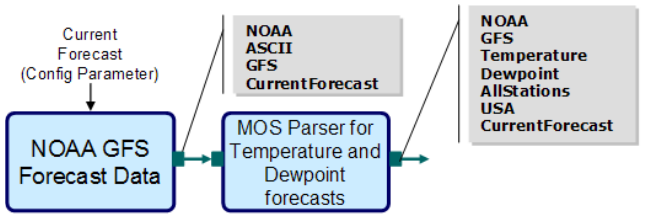


Figure 10. Example Instantiation of Weather Forecast Extraction Sub-Flow

6.2 Component Class and Component Requirements

Components that perform similar tasks and have similar input constraints can be grouped together into a class. For example, all components that take a set of weather forecasts from various sources and aggregate them in some fashion (e.g. performing an average, or coming up with a probability distribution, or finding the minimum or maximum or clustering or detecting outliers) may be grouped together into a class.

The key intuition behind a component class is that all the members of a component class are substitutable in a certain context. That is, in any given flow, a component can be replaced by another component in the same class without any syntactic or semantic mismatch. Hence, the definition of a component class is specific to a certain flow (or a certain class of flows).

This notion of substitutability of components is critical in our approach to automated composition. Our composition approach starts with a high-level application template definition that is made up of a flow of substitutable components. Different substitutions of components result in different instances of the templates that can satisfy specific goals.

Let $C = \{c\}$ be the set of all components in the system. Then the set of all component classes is $\mathcal{C} \subseteq 2^C$. In addition, a component class, $X \in \mathcal{C}$, is specific to a certain position in a flow, or set of flows. If $a \in X$ appears in this position, then it can be substituted by any $b \in X$.

The inputs and outputs of a component class, X, can be described by goal patterns that include variables. We define

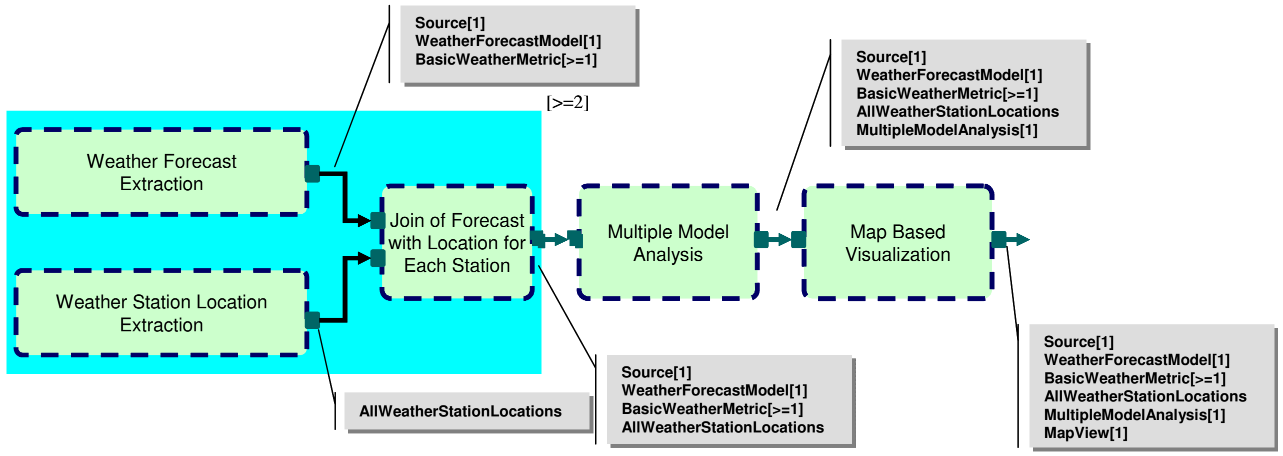


Figure 8. Example of Application Template showing various abstract sub-flows. Each abstract sub-flow is associated with a parallelism constraint (default is 1), and a goal pattern that it can satisfy.

the set of all *variable goal patterns* as $VQP = \{(x, c) | x \in F \cup V, c \in CC\} \cup \{t | t \in T \cup V\}$. Then a component class, X, can be defined as the pair $(\{I_X\}, \{O_X\})$

1. $\{I_X\}$ is a set of variable goal patterns that describe a class of input message constraints
2. $\{O_X\}$ is a set of goal patterns that describe a class of output messages
3. The set of variables in the output set, $\{O_X\}$, is a subset of the set of variables in the input set, $\{I_X\}$. This constraint ensures that no free variables exist in the output description.

We assume that each component belongs to a trivial component class, which is a singleton set. Figure 11 shows an example component class on the left. The input and output descriptions include the variable ?source whose type is WthrSource. This means that both the input and the output include the same tag, which is a sub-tag of WthrSource, such as NOAA.

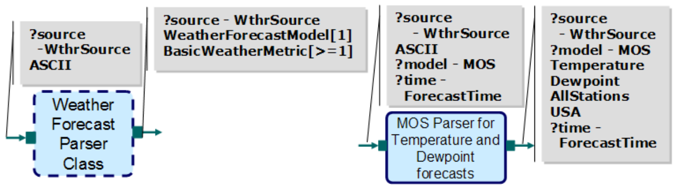


Figure 11. Example Component Class and Example Component

A component class can also act as a requirement specification for a new component, or a set of components. A component requirement is specified according to the model described in Section 5.5, i.e. in terms of tag-based descriptions of the inputs and outputs. A developer takes this requirement, along with any other requirement provided by the application architect, and develops the component. After unit and integration testing, this new component can be used in new application flows.

6.3 Planning-based, Bottom-Up, Automatic Composition

Once new components are developed and tested, they can be used in new flows. Our system includes an AI planner that composes flows from the available components given the goal. The planner plays a crucial role in the serendipitous assembly of new flows. It is not aware of the flow templates; hence, it can compose flows that follow the templates and also possibly new flows, which don't fall into any of the explicitly designed templates.

As an example, assume that there is a component developed in a different context that took weather data and stored it as tables in a database. Then this component can potentially replace any of the visualization components developed as part of the flow template in Figure 8. Hence, a dynamic user goal such as RelativeHumidity, IA, WeightedAverage, GFS, Eta, *DatabaseStorage* may be satisfiable even though it was not part of the original user requirements.

We have described the planning algorithm for solving the composition problem $\mathcal{P}(T, C, g)$, in earlier work [25]. The algorithm finds a solution with the best possible rank and generates a weighted list of tags for the tag cloud by analyzing descriptions of output data artifacts produced by alternative solutions. It can also provide a summary description for a specified range of alternative solutions.

The algorithm operates on an abstract description of the composition problem represented in SPPL (Stream Processing Planning Language) that was introduced in [24]. An SPPL planner performs much better in typical flow composition problems than the best general AI planners. By making objects a part of the domain model, SPPL planner avoids unnecessary grounding and resolves symmetries, reducing the search space by an exponential factor. The planning algorithm includes a problem analysis and presolve stage, and a forward search stage. During the presolve stage the search space is reduced by several techniques that include intelligent grounding of variables, elimination of irrelevant components based on fast backward search in a relaxed formulation, and grouping symmetrical components where appropriate. The forward search stage composes and ranks candidate flows starting from sources.

Finding optimal plans is a theoretically hard problem. In general, there do not exist optimal or constant-factor-approximation SPPL planners that can guarantee termination in polynomial time on all tasks. STRIPS planning, which is a special case of general SPPL planning, is known to be PSPACE-complete [4].

In practice, however, such composition problems can be solved automatically much faster than humans can solve them. In our experiments with an example based on a set of up to 273 Yahoo Pipes components and feeds, the time needed for solving the composition problem and generating the tag cloud for goal refinement was consistently under 5 seconds, and typically under 1 second or less [25].

7. Related Work

Faceted classification, introduced in 1933 by S. R. Ranganathan [22], was applied to software component reuse in 1991 [19], becoming the “industrial state of the art” by 1994 [15].

Subsequent approaches incorporated semantic models to improve search precision, via goal-based search [32]. Quality models were added, to enhance goal-driven retrieval with non-functional applicability criteria [11]. Still, the predominant focus was on specification and retrieval but not composition.

Component composition at the code level is addressed in [23]. More recent work proposes a dynamic hierarchical component composition [10], which also emphasizes type-oriented composition models.

Faceted classification, as applied to software requirements, was introduced by Opdahl [18]. Goal-directed Requirements Engineering was introduced in the KAOS system [33].

Web services composition is described in [28], and composition using UML is described in [30]. Rule-based service composition is addressed in [20] and [34]. Semantic service composition in [5]. A planner for service compositions described in [27]. An approach to DL planning for service composition is described in [21]. Knowledge engineering for workflow composition, [31], and ontology modeling for web service composition in [7], also focus on the use of rich semantic descriptions, which pose an extra burden on those describing components and on those establishing composition goals.

Other automated software composition work includes the work by Margaria and Steffen [13], who were able to synthesize sequential orchestrations, expressed in BPEL, given process constraints defined in linear temporal logic. The key difference in our approach is that we take a goal-directed planning approach to the task of composition, where the goals are expressed using sets of tags. The goal-directed, tag-based approach makes it easy for end-users to come up with customized compositions just by selecting one or more tags.

8. Conclusion

We have described a tag-based application design methodology that facilitates the composition of customized flows to satisfy end-user goals. At the core of our approach is a novel tag model where domain-specific tags, organized into various facets, are used to describe 1) end-user information processing goals, 2) component functional processing and data semantics, and 3) application requirements and structural constraints. The prominent role that tags and tag-based descriptions play in all three areas establishes a strong visible thread from users' information needs to dynamically assembled flow-based applications.

We have also described our application design and composition methodology that incorporates both top-down and bottom-up elements in order to come up with a set of components that can be composed into a large number of applications. Some of the advantages of our methodology are :

1. The top-down development lifecyle guarantees that the components developed can be composed to create applications that meet the initial end-user requirements.
2. The tag-based descriptions of all components facilitates their recombination in new ways to create new applications that satisfy new end-user goals, which may may not have been part of the initial requirements.
3. The common, yet extensible, facets and tag hierarchies establish a simple, shared vocabulary that is used architects, developers and end users
4. End-user requirements are captured in a formal manner. This enables us to verify that the requirements are actually satisfied by a set of composable services.

We have undertaken this application design process for a deployment in the financial services domain. This deployment included a total of 135 components. The development and annotation of the services was undertaken by a team of 5 people, with one person serving as a requirements engineer and application architect, 3 component developers and one supporting the basic architecture. Some of the components ran on the Project Zero platform [8], which allows the deployment of REST-based services and data mashups. Others were components in IBM's System S [9], which is a stream processing system. The flow size ranged from 5 to 150 component instances. Our preliminary experiences in this deployment have convinced us of the usefulness of our approach for developing rapidly composable flows from modular components. Our future work involves studying the properties and the evolution of the components and flows in this deployment, as well as deploying in different domains, and on different platforms.

References

[1] Yahoo pipes.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[3] Mehmet Altinel, Paul Brown, Susan Cline, Rajesh Kartha, Eric Louie, Volker Markl, Louis Mau, Yip-Hing Ng, David Simmen, and Ashutosh Singh. Damia: a data mashup fabric for intranet applications. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1370–1373. VLDB Endowment, 2007.

[4] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[5] L Chen, N.R. Shadbolt, C. Goble, F. Tao, S.J. Cox, C. puleston, and P. Smart. Towards a knowledge-based approach to semantic service composition. In *The Second International Semantic Web Conference (ISWC2003)*, 2003.

[6] Luba Cherbakov, Andy J. F. Bravery, and Aroop Pandya. SOA meets situational applications, 2007.

[7] Juntao Cui, Jiamao Liu, Yujin Wu, and Ning Gu. An ontology modeling method in semantic composition of web services. In *CEC-EAST '04: Proceedings of the E-Commerce Technology for Dynamic E-Business, IEEE International Conference*, pages 270–273, Washington, DC, USA, 2004. IEEE Computer Society.

[8] IBM. Project zero. *http://www.projectzero.org/*.

[9] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 431–442, New York, NY, USA, 2006. ACM.

[10] In-Gyu Kim, Doo-Hwan Bae, and Jang-Eui Hong. A component composition model providing dynamic, flexible, and hierarchical composition of components for supporting software evolution. *J. Syst. Softw.*, 80(11):1797–1816, 2007.

[11] Julio Leite, Yijun Yu, Lin Liu, Eric Yu, and John Mylopoulos. Quality-based software reuse. *Lecture Notes in Computer Science*, 3520:535–550, January 2005.

[12] Zhen Liu, Anand Ranganathan, and Anton Riabov. A planning approach for message-oriented semantic web service composition. In *AAAI*, pages 1389–1394. AAAI Press, 2007.

[13] Tiziana Margaria and Bernhard Steffen. Ltl guided planning: Revisiting automatic tool composition in eti. In *SEW '07: Proceedings of the 31st IEEE Software Engineering Workshop*, pages 214–226, Washington, DC, USA, 2007. IEEE Computer Society.

[14] D. Mennie and B. Pagurek. An architecture to support dynamic composition of service components. 2000.

[15] Rym Mili, Ali Mili, and Roland T. Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Trans. Softw. Eng.*, 23(7):445–460, 1997.

[16] J. Paul Morrison. Data responsive modular, interleaved task programming system. *IBM Technical Disclosure Bulletin*, 13(8):2425–2426, January 1971.

[17] NOAA. Acronyms and abbreviations used by the statistical modeling branch, 2008.

[18] Andreas L. Opdahl and Guttorm Sindre. Facet models for problem analysis. In *CAiSe '95: Proceedings of the 7th International Conference on Advanced Information Systems Engineering*, pages 54–67, London, UK, 1995. Springer-Verlag.

[19] Rubén Prieto-Díaz. Implementing faceted classification for software reuse. *Commun. ACM*, 34(5):88–97, 1991.

[20] Ken Pu, Vagelis Hristidis, and Nick Koudas. Syntactic rule based approach to web service composition. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 31, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Lirong Qiu, Fen Lin, Changlin Wan, and Zhongzhi Shi. Semantic web services composition using ai planning of description logics. In *APSCC '06: Proceedings of the 2006 IEEE Asia-Pacific Conference on Services Computing*, pages 340–347, Washington, DC, USA, 2006. IEEE Computer Society.

[22] S R Ranganathan. *Colon Classification*. Asia Publishing House, Bombay, India, 1933.

[23] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, 2000.

[24] A. Riabov and Z. Liu. Planning for stream processing systems. In *AAAI*, 2005.

[25] Anton V. Riabov, Eric Bouillet, Mark D. Feblowitz, Zhen Liu, and Anand Ranganathan. Wishful search: Interactive composition of data mashups. In *WWW*, April 2008.

[26] Dennis Ritchie. The evolution of the unix time-sharing system. In *Proceedings of a Symposium on Language Design and Programming Methodology*, pages 25–36, London, UK, 1980. Springer-Verlag.

[27] Mithun Sheshagiri, Marie desJardins, and Tim Finin. A Planner for Composing Services Described in DAML-S. In *Proceedings of the AAMAS Workshop on Web Services and Agent-based Engineering,*, June 2003.

[28] Kaarthik Sivashanmugam, John A. Miller, Amit P. Sheth, and Kunal Verma. Framework for semantic web process composition. *Int. J. Electron. Commerce*, 9(2):71–106, 2003.

[29] Maxym Sjachyn and Ljerka Beus-Dukic. Semantic component selection – SemaCS. In *ICCBSS '06: Proceedings of the Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, page 83, Washington, DC, USA, 2006. IEEE Computer Society.

[30] David Skogan, Roy Gronmo, and Ida Solheim. Web service

composition in uml. In *EDOC '04: Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International*, pages 47–57, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Renata Slota, Joanna Zieba, Bartosz Kryza, and Jacek Kitowski. Knowledge evolution supporting automatic workflow composition. In *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 37, Washington, DC, USA, 2006. IEEE Computer Society.

[32] Vijayan Sugumaran and Veda C. Storey. A semantic-based approach to component retrieval. *SIGMIS Database*, 34(3):8–24, 2003.

[33] Axel van Lamsweerde, Anne Dardenne, B. Delcourt, and F. Dubisy. The kaos project: Knowledge acquisition in automated specification of software. In *Proceedings of the AAAI Spring Symposium Series*, pages 59–62, Stanford University, Stanford, CA, 1991. American Association for Artificial Intelligence'.

[34] Jian Yang, Mike P. Papazoglou, Bart Orriëns, and Willem-Jan van Heuvel. A rule based approach to the service composition life-cycle. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, page 295, Washington, DC, USA, 2003. IEEE Computer Society.