Knowledge Engineering for Planning-based Data-flow Composition

Mark D. Feblowitz, Anand Ranganathan, Anton V. Riabov, and **Octavian Udrea**
IBM T. J. Watson Research Center
PO Box 704, Yorktown Heights, NY 10598, USA

Abstract

In this paper, we discuss knowledge engineering within a software composition framework, MARIO, that uses AI Planning for enabling end-users to *compose* software components into data analysis flows in a goal-driven manner. A data flow is as a directed graph that describes how data is obtained from one or more sources, processed by one or more software components, and finally sent to one or more sinks. Such flows are often used by domain experts in different domains, on different platforms, to distill raw data into useful intelligence. In many organizations, there are two roles who are involved in developing and deploying analysis flows. *Developers* develop one or more flows and end-users pick among the available flows, and parameterize and deploy them to obtain relevant intelligence. MARIO uses a language called Cascade for describing the planning domain, including commonly used analysis flow patterns. The patterns cover different possible variations of the flows, including variations in the structure of the flow, the software components in the flow and the possible parameterizations of these components. Developers describe the planning domain using the Cascade language, a tag-based knowledge representation model, and Eclipse-based tools. For end-users, who specify composition goals, MARIO includes a web-based interface to the planner, where end-users can specify planning goals, view the automatically composed flows, parameterize them and deploy them on one or more platforms.

Introduction

A data processing flow obtains data from different sources, aggregates or integrates them in different manners, applies different kinds of analyses on the data and finally, visualizes or handles the end-results in different ways. We view a data processing flow as a directed graph of black-box software components (including data sources) connected by data flow links. A flow is an intuitive abstraction that helps decouple the low-level details of information processing from the high-level view of the application. Flows are used to describe information processing applications on different platforms including Service Oriented Systems, Event-Driven Systems, Data Mashups, Stream Processing Systems, Extract-Transform-Load systems and the Grid.

Such flows are often used by experts in different domains, on different platforms, to obtain customized information. However, a key challenge in the use of data processing flows, especially by domain experts and other end-users, who are not skilled programmers, is flow assembly. Assembly is complex since there may be a very large number of components available, and the users may not be aware of the syntactic and the semantic constraints as well as the best practices in assembling components into complex flows.

Several visual programming tools have been developed to help end-users construct flows in different domains. Tools like Yahoo Pipes (Yahoo, Inc.) and IBM Mashup Center (IBM Mashup Center) have become fairly popular among casual programmers, with hundreds of thousands of flows having been created on them already. In the enterprise domain, the IBM WebSphere Message Broker Toolkit (IBM WebSphere Message Broker) allows creating flows for the Enterprise Service Bus. LabView (National Instruments) is another popular graphical programming tool for creating dataflows.

However, in our experience, domain experts still find it very difficult to create flows using visual composition tools to meet their customized data processing needs. There are a few reasons for this. First, even though visual programming environments like Yahoo Pipes make mashup construction more approachable for non-programmers, it still requires careful manual assembly of the flows by the end user. These end-users may not be aware of composition constraints of different components, and may also not be aware of the best practices in creating flows for their problems. Second, visual programming environments become increasingly difficult to use as the number of available components increases and/or the size of the composed flows increases. For example, in practice end-users can compose in a goal-driven manner flows containing many tens, even hundreds of components, which are very difficult to compose quickly using visual tools. Finally, most visual programming environments just support a fixed set of components. They do not work very well when the set of components is extensible and evolves with time.

AI Planning is a good candidate for tackling the challenge of flow assembly. In previous works (Riabov & Liu 2006), we had described planning approaches for composing flows in stream processing, complex-event processing, web ser-

vices and other flow-based systems. The key idea is to model each component as a planning action, model any constraints on the inputs and outputs of the component as preconditions and effects of the planning action, and finally, model the properties of the information to be produced by the assembled flow as a planning goal. Over the last few years, we have experimented with several approaches for representing the input and output constraints, for representing the goals and for composing individual components into flows. Lately, though, we have settled on a tag-based knowledge-representation model and a pattern-based approach for describing the space of available flows. We believe that these offer a good trade-off of expressiveness versus simplicity.

In a number of domains (such as financial services, manufacturing, security, etc.), end-users are reliant on an IT development team to support them in different data analysis tasks. The IT team may create a set of flows for use by the experts (often using the enterprise-side assembly tools like WebSphere Message Broker and LabView). If the expert wants to change the structure of the flow or parameterize it differently, she may actually have to change the underlying flow script in the language supported by the system. This may not be practical for most domain experts. Hence, most often domain experts must refer back to the developer to change the flows to meet new needs. This lack of flexibility is a big problem when the experts need to respond rapidly to a certain situation and there is no pre-built flow that meets their current needs.

In this paper, we propose an approach to simplify the construction, parameterization and deployment of flows by end-users, especially domain experts. We make use of the observation that in many domains, the set of useful flows for end-users (domain experts) often follow certain patterns. Hence, in our approach, flow developers can specify not just independent flows, but *patterns of flows*. A flow pattern describes a space of possible flows that are structurally similar and perform similar tasks. Flow patterns are similar to regular expressions, in the sense that just as regular expressions define a set of satisfying strings, a pattern defines a set of satisfying flows. Combined with tag taxonomies, input constraints, and output descriptions, the patterns describe the planning domain for goals specified by end-users.

Different platforms have their own flow languages, e.g. BPEL (Alves, A. *et al* 2006) for service-oriented systems, SPADE (Gedik *et al.* 2008), used in IBM's System S Stream Processing Platform (IBM InfoSphere Streams), Pig Latin used in (Apache Pig), etc. In addition, we often see data processing being performed using a set of shell or batch scripts. We have previously introduced Cascade (Ranganathan, Riabov, & Udrea 2009), our language for specifying patterns of flows. Cascade is platform and domain independent, i.e. it can be used to describe flows on any platform. It allows components to be described recursively, where a component is either a primitive component or a composite component, which internally defines a flow of components. A primitive component can embed code snippets from any platform-specific flow-based language (like BPEL, SPADE, shell scripts, etc). Some of the other key features of Cascade are an inheritance model for components, a number of ways for parameterizing components and the ability to define different structural variations for flows.

End-users (domain experts) explore the set of flows encapsulated within a flow pattern and select one that meets their data processing needs. For this, we make use of a tool called MARIO (Bouillet *et al.* 2008), which provides a tag-based, faceted navigation user interface where users can specify their needs (or goals) as a set of tags. MARIO then returns all satisfying flows to the end-user, ranked according to a pre-defined measure of cost. The composed flows can then be translated into a flow-script in a target platform (such as BPEL or SPADE), using the code snippets embedded within the primitive components. MARIO uses an AI planner for composing the flows dynamically given end-user goals. In order to compose the flows using MARIO, we have developed a Pattern Compiler that generates the planning domain corresponding to Cascade patterns. The planning domain includes input and output constraints that can be used by the MARIO planner to compose flows.

Patterns provide several advantages over free-form, bottom-up composition that we have presented in previous work (Bouillet *et al.* 2008). They offer a top-down, structured approach to defining allowable flows rather than relying on independent descriptions of individual components. In this way, they help restrict the search space of the planner to a smaller set of useful flows. They also help capture reusable design patterns for information processing in a certain domain.

In this paper, we focus on the knowledge engineering challenges that the developer faces in describing planning domains for flow assembly in such a manner that an end-user can assemble the right flow for his need. We believe that tags offer a convenient and concise way for developers to describe what kind of information a certain flow produces. A taxonomy of tags offers a common, extensible and flexible lingua franca for a team of developers and end-users. The main advantage is its simplicity compared to other representation schemes like OWL. We also describe Cascade and tools available to the developer for annotating components and patterns of flows with tags and the typical development process involved in creating a Cascade pattern and tag annotations for a given problem domain. These annotations are used by the AI Planner in MARIO for composing flows in response to user goals. We also describe our experiences in a case study at a client's site, where the Cascade language and associated tools were used by developers to "deliver" flows for use by domain experts.

Composition Tools and Plan Execution

Before discussing our tools for planning domains in following sections, in this section we introduce the cross-platform application flows, into which the plans are translated in our architecture, and outline the required infrastructure for goal specification and plan execution.

Cross-Platform Flow-Based Applications

In developing our composition framework we were primarily focusing on enabling automated composition of stream

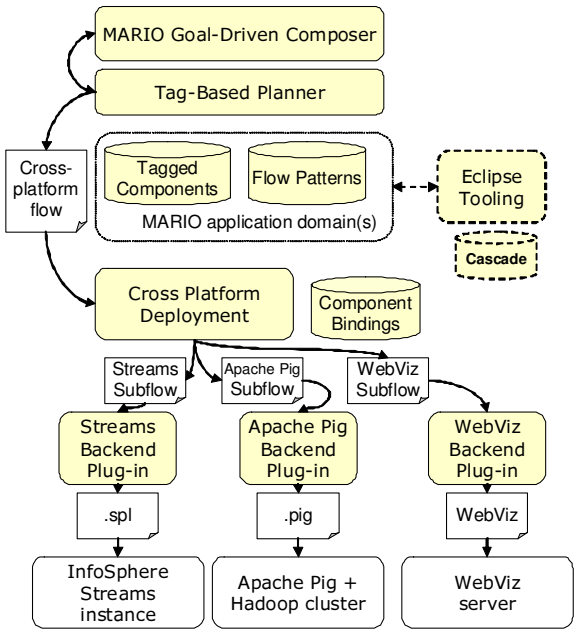


Figure 1: MARIO Architecture.

processing applications, such as those deployed in IBM's InfoSphere Streams middleware platform. However, as we have further observed, the techniques we developed for composition of stream processing applications easily extend to other platforms, and specifically the platforms where composable applications can be viewed, at some level of abstraction, as data flows. We have implemented composer prototypes that work with three different stream processing languages, certain types of Web Services domains, Enterprise Service Bus message flows, SQL queries, Apache Pig applications on Hadoop, IBM's DAMIA and WebSphere sMash middleware, and others.

This observation led us to develop MARIO, a *cross-platform* flow composer, which could be used to compose and deploy applications across multiple information processing platforms. MARIO generates high-level platform-independent flows, and then invokes platform-specific backend plug-ins to generate and deploy platform-specific implementations of these flows. This extensible approach lets MARIO, using a single planning domain, create and deploy flows that run on multiple platforms in separation, or have different components of one flow deployed in several different platforms.

MARIO represents information processing applications as flows, or, equivalently, flow graphs, comprised of components connected by communication links. In our current implementation the graphs assembled by MARIO are acyclic.

The cross-platform flow graph is a set of components connected by communication links sending data between the ports of components, where each component includes:

- zero or more input ports;
- zero or more parameters;
- one or more output ports;

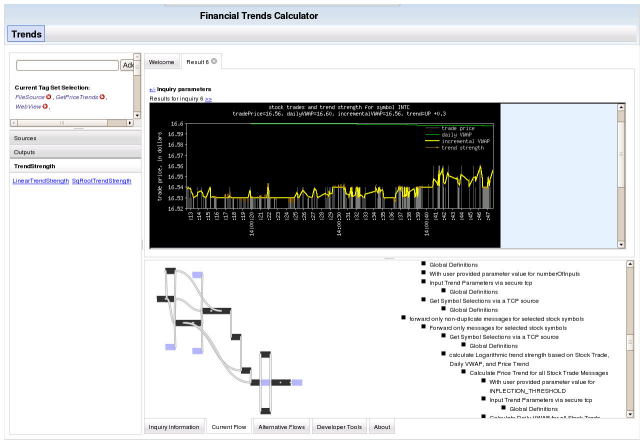


Figure 2: MARIO Composer for End-Users.

- platform type for this component;
- platform bindings information.

Platform bindings information, represented as a multi-line string, is required by the backend plug-in to generate the code corresponding to the component, and can have any format, as long as it can be processed by the backend.

Goal Specification and Plan Execution

Our composition and deployment architecture, shown graphically in Figure 1, is designed to support multiple platforms, for example (IBM InfoSphere Streams) and Hadoop with (Apache Pig), as well as a built-in web-based visualization server WebViz. New platforms are integrated by adding backend plug-ins. The plug-ins are responsible for translating subgraphs into generated platform-specific code, and adding bridges to generated code in order to establish data transfers between platforms. The plug-ins also control the lifecycle of the application within each platform by starting and stopping corresponding subgraphs of the generated applications.

End-users interact with MARIO via a web based interface shown on Figure 2. The interface allows end-users to specify goals by entering tags (e.g., *FilteredTrade, ByIndustry*), refine goals by adding new tags, and view composed flows, as well as the results of flow execution. Many elements of the interface, including result visualization, titles, prompts, and tag groups can be customized by the developers.

The Tag-Based Planner component of MARIO is responsible for finding the best flow for the specified goal. In MARIO, Cascade description of the planning domain, tag taxonomies, and the tags of the user-specified goal are translated into domain description in SPPL description language, and processed by a specialized planner (Riabov & Liu 2006). The SPPL descriptions are internal to the Tag-Based Planner, and are not visible to end-users or developers.

Cascade – Flow Pattern Description Language

In this section we discuss Cascade, the language we have developed for describing planning domains for data flow

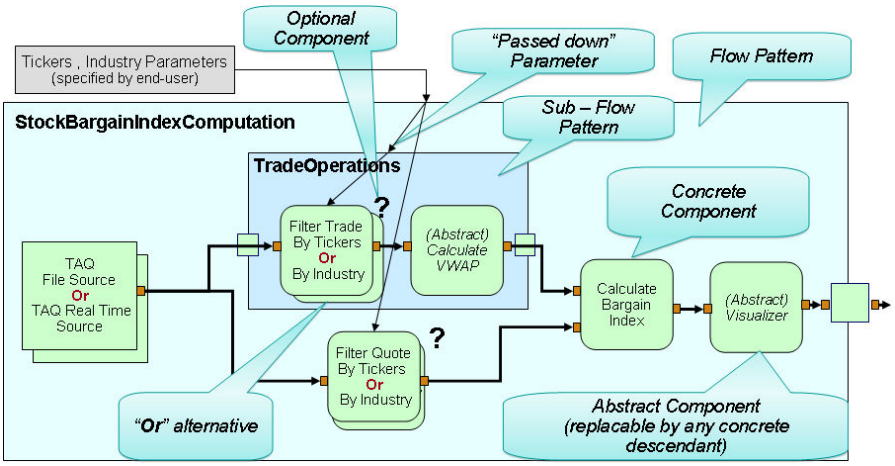


Figure 3: Cascade flow pattern example.

composition. A more complete overview of Cascade can be found in (Ranganathan, Riabov, & Udrea 2009).

Our language improves on existing planning domain description languages, when used in software composition applications, in three main areas:

- Cascade improves performance of the planning algorithm by allowing the algorithm to take advantage of explicitly stated composition constraints;
- Cascade uses hierarchical and modular composition patterns, allowing the developers to partition the domain description and work on different sub-patterns independently;
- Cascade includes semantic reasoning based on a tag taxonomy, allowing the end users to specify their composition requirements generally, if needed, by selecting tags from higher levels of tag hierarchies.

Cascade Graph Patterns

At the core of Cascade is the language for describing graph patterns. The patterns serve as composition constraints restricting the flow graph structure. To enforce this, we use generated preconditions and effects in the generated planning domain.

Graph pattern description consists of a few simple constructs shown graphically using the example in Figure 3:

- *Concrete components*, associated with a specific platform type and a binding.
- *Composites*, such as "TradeOperations", i.e., components whose implementation is a subgraph comprised of a graph other components, composites, or choice nodes.
- Choice nodes: *optional components*, *"Or" alternatives* and *abstract components*. The optional components may be removed from the flow graph during composition. The "Or" alternatives allow the planner to choose one of the listed concrete components or composites. The abstract components can be replaced by concrete components or composites implementing the abstract.
- And finally, *parameters*, which represent an external input variable requested from the end-user when plan execution is initiated.

An example of a Cascade pattern in text form is shown in the editor window in Figure 5. It includes the definition of a composite *BIComputationCore*, two concrete components *BIComp_Simple* and *BIComp_Complex*, and an abstract component *BIComp* that in our text example corresponds to the *Calculate Bargain Index* concrete component in the figure.

Semantic Annotations of Components and Flows

Semantic descriptions in MARIO are sets of tags, i.e., keywords representing business-relevant terms that end-users understand. In MARIO tags are assigned to flows in part by developers, and in part automatically: the planner uses tag annotations of components assigned by developers to automatically compute annotations of generated flows. For valid flows, the generated description of the output of the flow contains all tags of the user-specified composition goal.

Developers annotate Cascade components by associating a set of tags with each output port. For example, the output of *BIComp_Simple* is annotated with tags *Simple, BargainIndex, LinearIndex*. The planner uses these tags to compute the set of tags associated with the output of the composed flow, recursively, by taking the union of tags on input links of each component, and associating the result with each output, after adding tags specified by developers on that output. Developers can also specify tags that should be removed, and associate tags with inputs. Tags associated with inputs of a component describe composition constraints: in valid flows annotations of links connected to such inputs must include all tags from the input annotation.

Other Annotations Allowed in Cascade

Cascade allows developers to add hard and soft security constraints controlling the composition. MARIO can then make use of end user's credentials to tailor compositions to conform with user's access permissions.

To explore the tradeoffs between costs and benefits of alternative flows matching the same goal, MARIO planner can

analyze multiple metrics associated with flows, including cost and quality vectors. For this, the developers can associate cost and quality metrics with individual components.

Language Design Considerations

We have designed Cascade as an intermediate language that is automatically compiled into a planning domain represented in a planning domain description language, SPPL (Riabov & Liu 2006), which is an extension of PDDL. This has allowed our planning technology to be used by a broader community of developers. In addition to the language itself, we had to develop a set of tools, which we will discuss in the next section. However, the language itself is an important tool that makes planning usable for software composition.

We had to address a number of challenges specific to knowledge engineering for planners when designing Cascade. Cascade had to be expressive enough to create new non-trivial flows for the end users. It had to enable efficient planning algorithms that find optimal plans in seconds (Riabov & Liu 2006). Finally, Cascade had to support an effective development environment for domain descriptions, such that it could be easily adopted by software developers without requiring extensive training in logic, functional programming, or knowledge representation.

MARIO IDE

The MARIO IDE is a set of Eclipse (Eclipse Foundation) plugins that helps developers create planning domain descriptions by describing tag taxonomies and Cascade patterns, and manage and deploy them in MARIO. In this section, we describe the practical requirements for the design of the tools comprising the IDE, and their functionality. The MARIO IDE consists of a set of editors, views and wizards brought together in an Eclipse perspective that cover the lifecycle of developing, testing and "executing" a Cascade description of a planning domain. Broadly, the IDE tools can be categorized as:

- Editor for the Cascade description of the planning domain
- Editors for associated artifacts: (i) the tag taxonomy that models relations between tags used as component annotations and (ii) the application manifest that permits customization of the end-user interface.
- Testing of Cascade patterns.
- Cascade application compilation, deployment and management of MARIO servers.

We structure the rest of the section to follow the development steps outlined in Figure 4. Developers start with creating a new project, followed by the definition of the Cascade pattern and components. The development stages conclude with the engineering of a tag taxonomy and an application manifest. Developers typically proceed by testing Cascade patterns.Development concludes by preparing the Cascade description of the planning domain for a production deployment in MARIO.

Typically, the developer experience starts with the creation of a new MARIO project through a customized *New Project* wizard. Newly generated projects contain a few sample components and a sample Cascade pattern. The components contain very simple platform-specific code, from a backend platform chosen by the developer that is supported by the MARIO server (e.g., IBM InfoSphere Stream's Stream Processing Language (SPL) (IBM)). The typical project structure consists of:

- A set of Cascade pattern files. Although there are no strict requirements on how a set of patterns should be divided into multiple files, we will discuss guidelines for structuring pattern files that prove useful in practice.
- A project configuration file – this is a simple properties file (key-value format) where developers can specify server-side properties that override the default server configuration for this project.
- A tag taxonomy and an application file, both in XML format, for which there are special editors.
- Optionally, folders for optional resources to be used in the end-user interface (e.g., images) and for required toolkits that Cascade components depend on (e.g., InfoSphere Streams toolkits).

The developer(s) then begin crafting the Cascade patterns for their domain of interest. Cascade patterns are expressed in a domain-specific textual language inspired by IBM's Stream Processing Language (SPL). Although there are different ways in which notional Cascade patterns could be realized – for instance, drag-and-drop editors, description languages supporting graphs (GraphML, even RDF) –, the choice of syntax was motivated by the fact that many Cascade developers were already familiar with SPL, therefore making the transition to the Cascade pattern language easy. The MARIO IDE contains a syntax-highlighting editor for Cascade depicted in Figure 5, also featuring code completion, an outline and a framework for refactoring. The editor was implemented on top of the Xtext (Itemis) framework for domain-specific languages.

The syntax of a Cascade component consists of a set of annotations (contained between special markers /#* and *#/), the header and body. The special annotations are of the type *@annotation-name annotation-arguments*. These are typically the description (title) of the component, the platform on which the component's code is intended to run (e.g., *@type "spl"*, but only for concrete components, which have platform bindings) and the tag annotations on the input and output ports of the component (e.g., *@tag oport CSV*). The header consists of the name of the component and the input and output ports and, for certain components the designation *abstract* or inheritance statements. The structure of the body consists of a list of formal parameter names (if any), platform bindings enclosed between /$ and $/ for concrete components with specified *@type*, flow graph pattern descriptions for composite components, or empty body for abstract components.

In practice, we have seen two ways of developing Cascade patterns for an application domain. In the first development mode, the architect and developers start with already existing custom applications that solve a similar problem. They then uplift the elements of these applications (e.g., operators, subgraphs, etc.) into Cascade components. To aid the creation of such components we have provided import wiz-

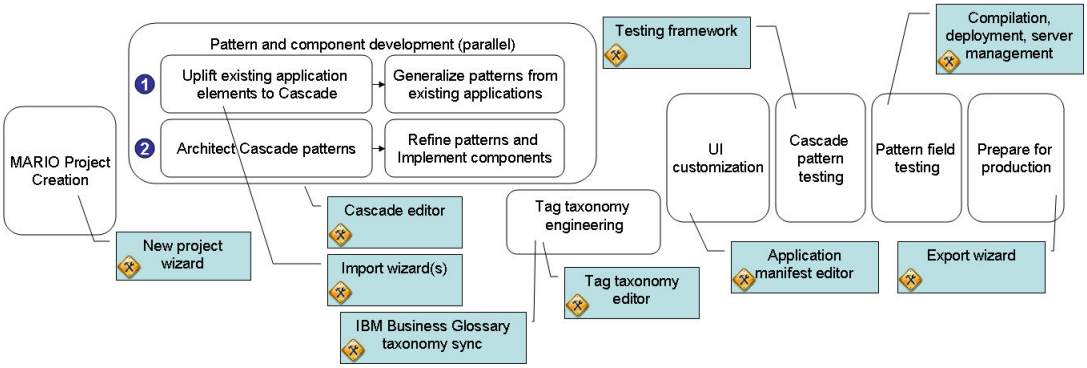


Figure 4: Notional development cycle with the MARIO IDE and associated tools

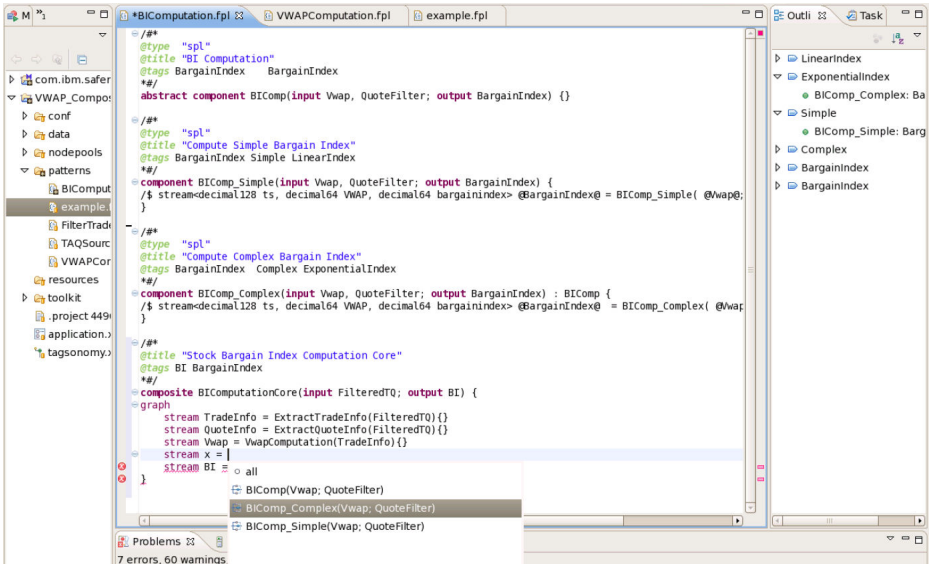


Figure 5: MARIO Cascade editor with syntax highlighting, outline, collapsible sections, code completion and refactoring

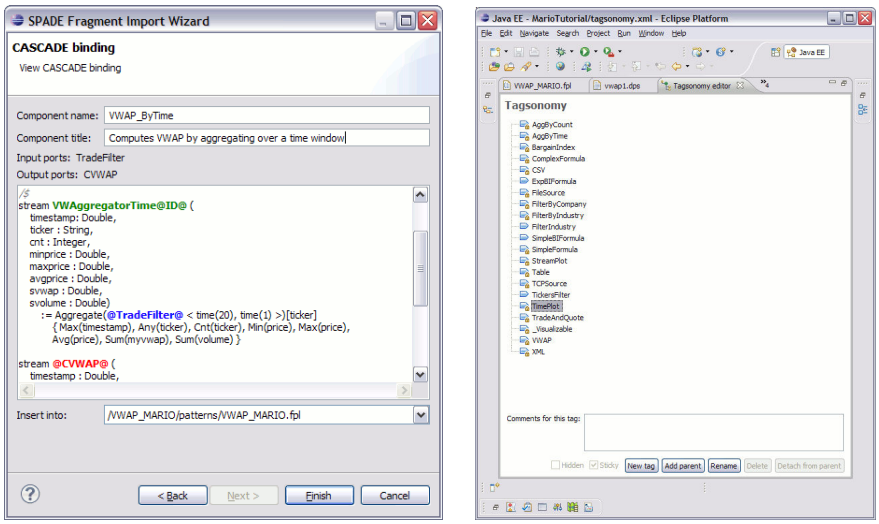
ards from platform languages typically used with MARIO. The import wizard for SPADE (the language for IBM InfoSphere Streams 1.2, a predecessor of SPL (Gedik *et al.* 2008)) is shown in Figure 6(a). Finally, in step 3, the architect and developers will generalize their custom applications into a design patterns describing the space of solutions (analytic flows) for the domain. This description, in the form of the Cascade pattern, is then compiled into a planning domain description and used by MARIO to enable end-users to create analytic flows on the fly for their situational needs.

In the second mode of development – which typically occurs very soon (sometimes even the second application developed) after the development team becomes proficient in Cascade and the use of MARIO –, the application architect designs the space of solutions for the domain in the form of a Cascade pattern, typically using abstract components to denote functionality which must be implemented or refined further. The use of abstract components serves multiple purposes during this mode of development:

- The simplest use of abstracts is as a placeholder for primitive components that must be implemented in the future.
- A more complex use is to abstract away entire subgraphs of the pattern that can be delegated or detailed in a later stage (i.e., have a composite defined that "implements" the abstract).
- A third widespread use is to divide functionality in hierarchies (for a very specific example, consider an abstract *Classifier* component, from which inherit children abstracts such as *ClusteringClassifier*, *SVMClassifier*, *DecisionTreeClassifier*; in turn, concrete clustering, SVM or decision tree implementations will inherit from the second level abstracts).

In this mode of development, we have observed that developers organize work in separate pattern files under the Cascade project structure. Each file contains either a refinement of a subgraph of the pattern or related concrete implementations of an abstract component.

During the process of creating primitive components and composites, part of the development team or the architect can begin creating the tag taxonomy using the editor de-



(a) (b)

Figure 6: (a) Import wizard for existing code; (b) Tag taxonomy editor (drag and drop tags into parent-child relationships)

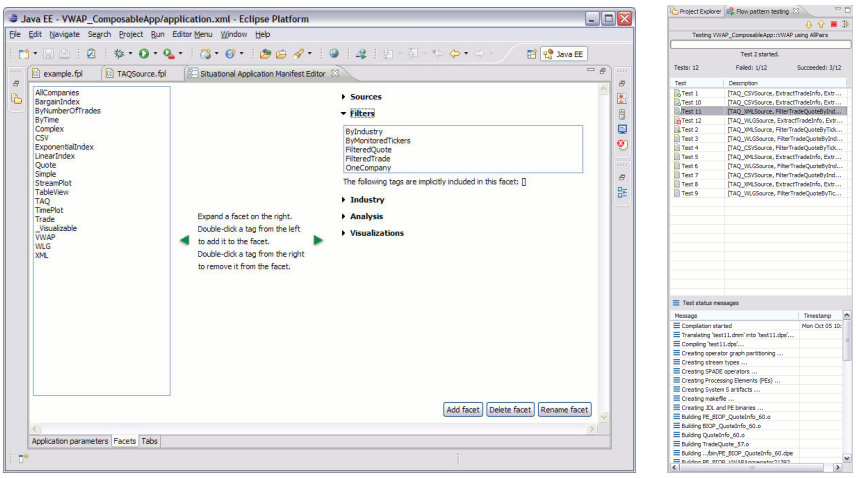
picted in Figure 6(b). The tag taxonomy consists of parent-child relationships between tags that annotate the input and output ports of Cascade components. The tag taxonomy enables reasoning over the space of constraints declared by components, as well as over the space of equivalent plans for under-specified goals (for which there is more than one plan). The tags used in the Cascade patterns are typically sets of business terms that end-users understand; with this use in mind, the MARIO IDE also features a wizard that allows tags to be imported and synchronized with an instance of the IBM Business Glossary (IBM Business Glossary), a tool that permits enterprise users to define and manage business terms and even associate them with data sources. When the tag tagsonomy is relatively stable, the application manifest editor (Figure 7(a)) allows the customization of the end-user interface, including the grouping and ordering of tags into facets and the customization of the application front page.

After the completion of the Cascade pattern and associated taxonomy and application manifest, the developers can use the Cascade testing framework to *test* their pattern. Since any Cascade pattern can result in a myriad of analytic flows being planed and composed based on end-users goal, there is no easy to way to test that all the compositions described by the pattern result in valid (i.e., compilable) applications. Using the method described in (Winbladh & Ranganathan 2011), developers can generate a small number of test plans and applications embodying these plans that have been empirically shown to catch the vast majority of common errors. The interface for the testing framework (Figure 7(b)) is inspired after the JUnit interface, with the important difference that developers choose a testing methodology and parameters. Tests are automatically generated and compiled (potentially on a remote machine) by the IDE. The developers can retrieve generated code; the IDE places markers where compilation errors have been reported.

Finally, after testing, the developers have the option of field-testing their Cascade patterns on a MARIO server. MARIO projects are by default compiled into a planning domain specification on every save (if enabled in Eclipse); errors are displayed in the usual way, using the *Problems* view, as well as markers in the Cascade editor. The MARIO Servers view (Figure 8(a)) allows developers to define the location of MARIO installations, potentially on remote machines, and also change some common configuration elements for the servers (e.g., the HTTP port numbers). The IDE communicates with the MARIO servers via *ssh* and *scp* using private–public key authentication. After a server location has been defined, developers will be prompted to choose the MARIO application they want to run when starting the server. The application will then be recompiled and the generated planning domain specification and associated taxonomy, manifest and other artifacts (configuration, libraries) uploaded to the server. Server log messages are displayed in a separate *Console* view. The developers can double-click the running server to bring up the MARIO user interface in a stand-alone browser window; they can also drill down and look at any the application source code generate by the planner as a result of end-user goals and bring that source code back to the MARIO IDE machine for inspection. Servers support one-click restarts when any element of the MARIO project is changed by the developers. After field-testing the application, the developers can export a compiled MARIO project (Figure 8(b)) to a zip file or directory on a local or remote machine. The exported file or directory can be used to start production MARIO servers for the application.

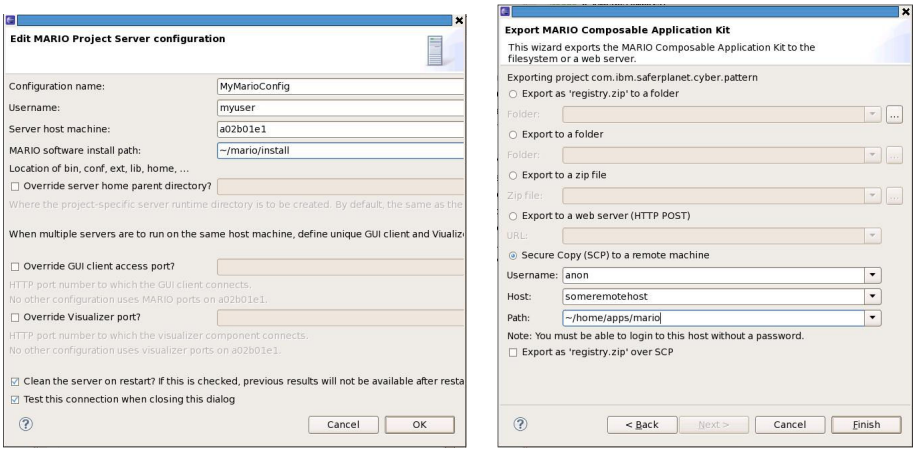
Discussion

We have implemented an automated composer, MARIO, and Cascade, a language for describing flow-based software composition domains, and a full set of development tools for Cascade, including an Eclipse-based IDE. These tools have



(a) (b)

Figure 7: (a) Application manifest editor; (b) Cascade testing framework



(a) (b)

Figure 8: (a) MARIO server management from IDE; (b) Export wizard

been used in pilot deployments since 2009, and we have made modifications and improvements based on experience and feedback from these deployments.

Installing the tools requires a target deployment environment supported by the tools. For the main branch of our tools we require Hadoop or IBM's InfoSphere Streams, although we had built experimental prototypes for other platforms.

The tools have reached the level of robustness required to be applied in real-world applications.

These tools have been applied in a data analysis domain, where a total of 5 developers (also referred to as technical analysts) developed flow patterns for use by a total of 20 domain experts (also referred to as business analysts). In this case study, all composed flows were converted to the SPADE language, and were deployed on IBM InfoSphere Streams.

In one case study 4 different applications were developed. Each application targeted a certain data processing problem and consisted of a set of patterns. Each application was developed by a small team of developers and was delivered for use by a team of domain experts, as follows:

- Application 1 had 2 Developers and 2 Business Analysts
- Application 2 had 2 Developers and 8 Business Analysts
- Application 3 had 3 Developers and 6 Business Analysts
- Application 4 had 3 Developers and 8 Business Analysts

Some key high-level statistics on the usage of patterns are:

- Number of Cascade patterns : 16
- Number of Cascade components : 186
- Total number of assemble-able flows : 1200+
- Number of applications imported from existing flows : 2
- Number of applications built from ground up : 2

We also received feedback from three of the developers about some properties of the patterns they created. Table 1 summarizes this feedback.

While the statistics we have collected so far are from a relatively small sample set of developers, they give us some

	Developer 1	Developer 2	Developer 3
Number of patterns developed	8	2	4
Number of components in patterns	Max: 20, Avg: 10	26 in one pattern, 20 in the other	Max: 20
Max depth of composite-containment hierarchy (i.e. composites containing other components)	4	4	3
Max depth of component-inheritance hierarchy	3	2	2
Max branching factor in enumerations	7	4	5
Max number of flows in a pattern	1084	36	25
How were the patterns created	Generalized Existing Flows	From scratch	From scratch

Table 1: Feedback from 3 Cascade developers

hints on the usage of patterns. Firstly, developers are comfortable in terms of partitioning a flow in a native flow language into different Cascade components and organizing these components hierarchically in high level composites. The also made use of different features of the language, including the component-inheritance and specifying structural variations. While one the developers always started by first creating a SPADE flow and then generalizing it into a pattern, the others started coding the flows and the patterns together (i.e. they had the space of possible flows in mind from the very beginning).

We also asked the developers about the reasons for creating the patterns. The main reason was to support different kinds of custom processing and alerting requested by domain experts. They said that with the delivery of the patterns, the number of individual requests from the domain experts for small modifications to the pattern had significantly decreased. The domain experts were more self-sufficient with the use of the patterns and the end-user interface. There was still some communication between developers and the domain experts, which mainly involved getting requirements and feedback; however the frequency of this interaction had decreased since they started using patterns.

The developer tools aided the development of patterns. For example, Developer 1 said that the first pattern took him 4 hours to develop (he was generalizing an existing SPADE flow). However, he was able to develop subsequent patterns much more rapidly.

Overall the feedback we received from developers was positive and encouraging. At the same time automated composition brings new challenges to developers. One notable example of the new class of development challenges are challenges originating from the use of code generation in MARIO. Debugging automatically generated code on various platforms and tracing back problems to Cascade source was not always easy. To address this, at deployment time, MARIO saves generated code to make external debugging possible, and includes comments in generated code referring to the original Cascade components. And at development time, the testing framework included with the IDE helps identify problems by automatically composing and testing multiple instantiations of the patterns.

Acknowledgements

The authors thank their former colleagues Eric Bouillet, Hanhua Feng and Zhen Liu for critical contributions to MARIO project. The authors also thank MARIO pilot participants for providing invaluable feedback.

References

Alves, A. *et al*. 2006. Web services business process execution language version 2.0. *OASIS Committee Draft*.

Apache Pig http://pig.apache.org/.

Bouillet, E.; Feblowitz, M.; Liu, Z.; Ranganathan, A.; and Riabov, A. 2008. A tag-based approach for the design and composition of information processing applications. In *OOPSLA*, 585–602.

Eclipse Foundation Eclipse Project. http://eclipse.org.

Gedik, B.; Andrade, H.; Wu, K.-L.; Yu, P. S.; and Doo, M. 2008. SPADE: the System S declarative stream processing engine. In *SIGMOD 2008*, 1123–1134.

IBM Business Glossary http://www.ibm.com /software/data/infosphere/business-glossary.

IBM InfoSphere Streams http://www.ibm.com /software/data/infosphere/streams/.

IBM Mashup Center http://www.ibm.com /software/info/mashup-center/.

IBM WebSphere Message Broker http://www.ibm.com /software/integration/wbimessagebroker/.

IBM IBM InfoSphere Streams SPL Specification. http://publib.boulder.ibm.com/infocenter/streams/v2r0.

Itemis Xtext Framework Documentation. http://xtext.itemis.com.

National Instruments Labview. http://www.ni.com/labview.

Ranganathan, A.; Riabov, A.; and Udrea, O. 2009. Mashup-based information retrieval for domain experts. In *CIKM*, 711–720.

Riabov, A., and Liu, Z. 2006. Scalable planning for distributed stream processing systems. In *ICAPS*.

Winbladh, K., and Ranganathan, A. 2011. Evaluating test selection strategies for end-user specified flow-based applications. In *ASE*, 400–403.

Yahoo, Inc. pipes.yahoo.com.