Wisdom-Aware Computing: On the Interactive Recommendation of Composition Knowledge

Soudip Roy Chowdhury, Carlos Rodríguez, Florian Daniel and Fabio Casati

University of Trento, Via Sommarive 14, 38123 Povo (TN), Italy

{rchowdhury,crodriguez,daniel,casati}@disi.unitn.it

Abstract. We propose to enable and facilitate the development of service-based development by exploiting *community composition knowledge*, i.e., knowledge that can be harvested from existing, successful mashups or service compositions defined by other and possibly more skilled developers (the *community* or *crowd*) in a same domain. Such knowledge can be used to assist less skilled developers in defining a composition they need, allowing them to go beyond their individual capabilities. The assistance comes in the form of *interactive advice*, as we aim at supporting developers while they are defining their composition logic, and it adjusts to the skill level of the developer. In this paper we specifically focus on the case of *process-oriented, mashup-like applications*, yet the proposed concepts and approach can be generalized and also applied to generic algorithms and procedures.

1 Introduction

Although each of us develops and executes various procedures in our daily life (examples range from cooking recipes to low-level programming code), today very little is done to support others, possibly *less skilled developers* (or, in the extreme case, even end users) in developing their own. Basically, there are two main approaches to **enable less skilled people** to "develop": either development is eased by *simplifying* it (e.g., by limiting the expressive power of a development language) or it is facilitated by *reusing knowledge* (e.g., by copying and pasting from existing algorithms).

Among the **simplification** approaches, the workflow and BPM community was one of the first to claim that the abstraction of business processes into tasks and control flows would allow also the less skilled users to define own processes, however with little success. Then, with the advent of web services and the service-oriented architecture (SOA), the web service community substituted tasks with services, yet it also didn't succeed in enabling less skilled developers to compose services. Recently, web mashups added user interfaces to the composition problem and again claimed to target also end users, but mashup development is still a challenge for skilled developers. While these attempts were aimed at simplifying technologies, the human computer interaction community has researched on end user development approaching the problem from the user interface perspective. The result is simple applications that are specific to a very limited domain, e.g., an interactive game for children, with typically little support for more complex applications.

As for what regards **capturing and reusing knowledge**, in IT reuse typically comes in the form of program libraries, services, or program templates (such as generics in Java or process templates in workflows). In essence, what is done today is either providing building blocks that can be composed to achieve a goal, or providing the entire composition (the algorithm – possibly made generic if templates are used), which may or may not suit a developer's needs. In the nineties and early 2000s, AI planning [1] and automated, goal-oriented compositions (e.g., as in [2]) became popular in research. A typical goal there is to derive a service composition from a given goal and a set of components and composition rules. Despite the large body of interesting research, this thread failed to produce widely applicable results, likely because the goal is very ambitious and because assumptions on the semantic richness and consistency of component descriptions are rarely met in practice. Other attempts to extract knowledge are, for example, oriented at identifying social networks of people [3] or at providing rankings and recommendations of objects, from web pages (Google's Pagerank) to goods (Amazon's recommendations). An alternative approach is followed by expert recommender systems [4], which, instead of identifying knowledge, aim at identifying knowledge holders (the experts), based on their code production and social involvement.

In this paper, we describe **WIRE**, a *WIsdom-awaRE development environment* we are currently developing in order to enable less skilled developers to perform also complex development tasks. We particularly target process-oriented, mashup-like applications, whose development and execution can be provided as a service via the Web and whose internals are characterized by relatively simple composition logic and relatively complex tasks or components. This class of programs seems to provide both the benefit of (relative) simplicity and a sufficient information base (thanks to the reuse of components) to learn and reuse programming/service composition knowledge. The idea is to *learn from existing compositions* (or, in general, *computations*) and to provide the learned knowledge in form of *interactive advice* to developers while they are composing their own application in a visual editor. The aim is both to allow developers to go beyond their own development capabilities and to speed up the overall development process, joining the benefits of both simplification *and* reuse.

Next, we discuss a state of the art composition scenario and we show that it is everything but trivial. In Section 3, we discuss the state of the art in assisted composition. In Section 4 and 5, we investigate the idea of composition advices and provide our first implementation ideas, respectively. Then we conclude the paper and outline our future work.

2 Example Scenario and Research Challenges

In order to better understand the problem we want to address, let's have a look at how a mashup is, for instance, composed in Yahoo! Pipes (*http://pipes.yahoo.com/pipes/*), one of the most well-known mashup platforms as of today. Let's assume we want to develop a simple pipe that sources a set of news from *Google News*, filters them according to a predefined filter condition (in our case, we want to search for news on products and services by a given vendor), and locates them on a *Yahoo! Map*.

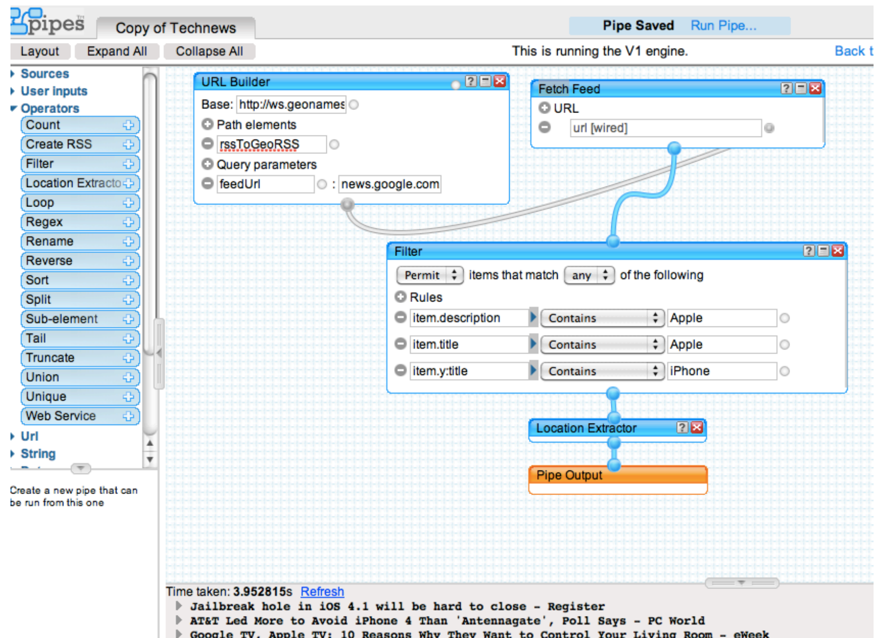

Figure 1 Implementation of the example scenario in Yahoo! Pipes

The pipe that implements the required feature is illustrated in Figure 1. It is composed of five components: The *URL Builder* is needed to set up the remote *Geo Names* service, which takes a news RSS feed as an input, analyzes its content, and inserts geo-coordinates, i.e., longitude and latitude, into each news item (where possible). Doing so requires setting some parameters: *Base=http://ws.geonames.org*, *Path elements=rssToGeoRSS*, and *Query parameters=FeedUrl:news.google.com/news?topic=t&output=rss&ned=us*. The so created URL is fed into the *Fetch Feed* component, which loads the geo-enriched news feed. In order to filter out the news items we are really interested in, we need to use the Filter component, which requires the setting of proper filter conditions via the *Rules* input field. Feeding the filtered feed into the *Location Extractor* component causes Pipes to plot the news items on a *Yahoo! Map*. Finally, the *Pipe Output* component specifies the end of the pipe.

If we analyze the development steps above, we can easily understand that developing even such a simple composition is out of the reach of people without programming knowledge. Understanding which components are needed and how they are used is neither trivial nor intuitive. The URL Builder, for example, requires the setting of some complex parameters. Then, components need to be suitably connected, in order to support the data flow from one component to another, and output parameters must be mapped to input parameters. But more importantly, plotting news onto a map requires knowing that this can be done by first enriching a feed with geo-coordinates, then fetching the actual feed, and only then the map is ready to plot the items.

Enabling non-expert developers to compose a pipe like the above requires telling (or teaching) them the necessary knowledge. In **WIRE**, we aim to do so by providing non-expert developers with interactive development advices for composition, inside

an assisted development environment. We want to obtain the knowledge to provide advices by extracting, abstracting, and reusing compositional knowledge from existing compositions (in the scenario above, pipes) that contain community knowledge, best practices, and proven patterns. That is, in *WIRE* we aim at bringing the *wisdom of the crowd* (possibly even a small crowd if we are reusing knowledge within a company) in defining compositions when they are both defined by an individual (where the crowd supports an individual) or by a community (where the crowd supports social computing, i.e., itself in defining its own algorithms). The final goal is to move towards a new frontier of knowledge reuse, i.e., *reuse of computational knowledge*.

Doing so requires approaching a set of **challenges** that are non-trivial:

1. First of all, *identifying the types of advices that can be given and the right times when they can be given*: depending on the complexity and expressive power of the composition language, there can be a huge variety of possible advices. Understanding which of them are useful is crucial to limit complexity.
2. *Discovering computational knowledge*: how do we harvest development knowledge from the crowd, that is, from a set of existing compositions? Knowledge may come in a variety of different forms: component or service compatibilities, data mappings, co-occurrence of components, design patterns, evolution operations, and so on.
3. *Representing and storing knowledge*: once identified, how do we represent and store knowledge in a way that allows easy querying and retrieval for reuse?
4. *Searching and retrieving knowledge*: given a partial program specification under development, how do we enable the querying of the knowledge space and the identification of the most suitable and useful advice to provide to the developer, in order to really assist him?
5. *Reusing knowledge*: given an advice for development, how do we (re)use the identified knowledge in the program under development? We need to be able to "weave" it into the partial specification in a way that is correct and executable, so as to provide concrete benefits to the developer.

In this paper, we specifically focus on the *first challenge* and we provide our first ideas on the second challenge and on the assisted development environment.

3 State of the Art

In **literature**, there are approaches that aim at similar goals as WIRE, yet they mainly focus on the *retrieval* and *reuse* of composition knowledge. In [6], for instance, mashlets (the elements to be composed) are represented via their inputs and outputs, and glue patterns are represented as graphs of connections among them; reuse comes in the form of auto-completion of missing components and connections, selected by the user from a ranked list of top-k recommendations obtained starting from the mashlets used in the mashup. In [8], light-weight semantic annotations for services, feeds, and data flows are used to support a text-based search for data mashups, which are actually generated in an automated, goal-oriented fashion using AI planning (the search tags are the goals); generated data processing pipes can be used as is or further edited. The approach in [9] semantically annotates portlets, web apps, widgets, or Java beans

and supports the search for functionally equivalent or matching components; reuse is supported by a semantics-aided, automated connection of components. Also the approach in [10] is based on a simple, semantic description of information sources (name, formal inputs [allowed ones], actual inputs [outputs consumed from other sources], outputs) and mashups (compositions of information sources), which can be queried with a partial mashup specification in order identify goals based on their likelihood to appear in the final mashup; goals are fed to a semantic matcher and an AI planner, which complete the partial mashup. This last approach is the only one that also automatically *discovers* some form of knowledge in terms of popularity of outputs in existing mashup specifications (used to compute the likelihoods of goals).

In the context of business process modeling, there are also some works with similar goals as ours. For instance, in [7], the authors more specifically focus on business processes represented as Petri nets with textual descriptions, which are processed (also leveraging WordNet) to derive a set of descriptive tags that can be used for search of processes or parts thereof; reuse is supported via copy and paste of results into the modeling canvas. The work presented in [13] proposes an approach for supporting process modeling through object-sensitive action patterns, where these patterns are derived from a repository of process models using techniques from association rule learning, taking into consideration not only actions (tasks), but also the business objects to which these actions are related. Finally, [14] presents a model for the reuse data mining processes by extending the CRISP-DM process [15]. The proposed model aims at including data mining process patterns into CRISP-DM and to guide the specialization and application of such patterns to concrete processes, rather than actually exploiting the community knowledge.

In general, the **discovery of community composition knowledge** is not approached by the works above (or they do it in a limited way, e.g., by deriving only behavioral patterns from process definitions). Typically, they start from an annotated representation of mashups and components and query them for functional compatibilities and data mappings, improving the quality of search results via semantics, which are explicit and predefined. WIRE, instead, specifically focuses on the elicitation and collection of *crowd wisdom*, i.e., composition knowledge that derives from the ways other people have solved similar composition problems in the past and that has a significant support in terms of number of times it has been adopted. This means that in order to create knowledge for WIRE, we do not need any expert developer or domain specialist that writes and maintains explicit composition rules or logics; knowledge is instead harvested from how people compose their very own applications, without requiring them to provide additional meta-data or descriptions (which typically doesn't work in practice).

4 Wisdom-Aware Development: Concepts and Principles

Identifying which advices can be provided and which advices do indeed have the potential to help less skilled developers to perform complex development tasks requires, first of all, understanding the *expressive power* of the composition language at hand. We approach this task next. Then we focus on the advices.

4.1 Expressiveness of the Composition Language

Let us consider again Yahoo! Pipes. The platform has a very advanced and pleasant user interface for drag-and-drop development of data mashups and supports the composition of also relatively complex processing logics. Yet, the strong point of Pipes is its *data flow based composition paradigm*, which is very effective and requires only a limited set of modeling constructs. As already explained in the introduction, constraining the expressive power of composition languages is one of the techniques to simplify development, and Pipes shares this characteristic with most of today's mashup platforms.

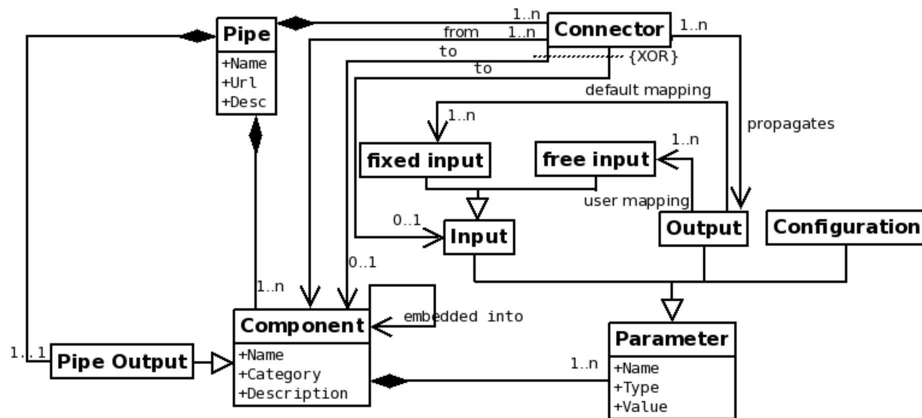

Figure 2 A meta-model for Yahoo! Pipes' composition language

In order to better understand the expressiveness of Yahoo! Pipes, in Figure 2 we derived a **meta-model** for its composition language. A *Pipe* is composed of components and connectors. *Components* have a name and a description and may be grouped into categories (e.g., source components, user input components, etc.). Each pipe contains always one *Pipe Output* component, i.e., a special component that denotes the end of data flow logic or the end of the application. A component may be *embedded* into another component; for example components (except user inputs and operators) can be embedded inside a *Loop Operator* component. Components may also have a set of parameters. A *Parameter* has a name, a type, and may have a value assigned to it. There are basically three types of parameters: *input* parameters (accept data flows attributes), *output* parameters (produce data flow attributes), and *configuration* parameters (are manually set by the developer). For instance, in our example in Section 2, the *URL* parameter of the *Fetch Feed* component is an input parameter; the *longitude* and *latitude* attributes of the RSS feed fetched by the *Fetch Feed* component are output parameters; and the *Base* parameter of the *URL Builder* component is an example of configuration parameter.

Data flows in Pipes are modeled via dedicated connectors. A *Connector* propagates output parameters of one component (indicated in Figure 2 by the *from* relationship) to either another component or to an individual input field of another component. If a connector is connected to a whole component (e.g., in the case of the connector from

the *Fetch Feed* component to the *Filter* component in Figure 1), all attributes of the RSS item flowing through the connector can be used to set the values of the target component's input parameters. If a connector is connected only to a single input parameter, the data flow's attributes are available only to set the value of the target input parameter. Input parameters are of two types: either they are *fixed inputs*, for which there are predefined *default mappings*, or they are *free inputs*, for which the user can provide a value or choose which flow attribute to use. That is, for free inputs it is possible to specify a simple attribute-parameter data mapping logic.

Figure 2 shows that Yahoo! Pipes' meta-model is indeed very **simple**: only 10 concepts suffice to model its composition features. Of course, the focus of Pipes is on data mashups, and there is no need for complex web services or user interfaces, two features that are instead present in our own mashup platform, i.e., mashArt [5]. Yet, despite these two additions, mashArt's meta-model only requires 13 concepts. If instead we look at the BPMN modeling notation for business processes [11], we already need more than 20 concepts to characterize its expressive power, and the meta-model of BPEL [12] has almost 60 concepts! Of course, the higher the complexity of the language, the more difficult it is to identify and reuse composition knowledge.

4.2 Advising Composition Knowledge

Given the meta-model of the composition language for which we want to provide composition advices, it is possible to identify which concrete **compositional knowledge** can be extracted from existing compositions (e.g., pipes). The gray boxes in the conceptual model in Figure 3 illustrate the result of our analysis. The figure identifies the key entities and relationships needed to provide composition advices.

An *Advice* provides composition knowledge in form of composition patterns. An advice can be to *complete* a given pattern (given it's partial implementation in the modeling canvas) or to *substitute* a pattern with a similar one, or the advice can *highlight* compatible elements in the modeling canvas or *filter and rank* advices.

Patterns represent the actual recommendation that we deliver to the user. They can be of five types (all these patterns can be identified in the model in Figure 3):

- *Parameter Value Patterns*: Possible values for a given parameter. For instance, in the *URL Builder* component the *Base* parameter value in a pattern can be set to "http://ws.geonames.org", while the *Path elements* parameter value can be "rssToGeoRSS", and feedUrl can be "news.google.com/news?topic=t&output=rss&ned=us", as shown in our example scenario. Alternatively, we can have the *URL Builder* component with the *Base* parameter set to "news.google.com/news" and the *Query parameters* set with different values.
- *Component Association Patterns*: Co-occurrence patterns for pairs of components. For instance, in our scenario, whenever a user drags and drops the *URL Builder* on the design canvas, a possible advice derived from a component association pattern can be to include in the composition the *Fetch Feed* component and connect it to the *URL Builder.*
- *Connector Patterns*: Component-component or component-input parameter patterns. This pattern captures the dataflow logic, i.e., how components are

connected via connector elements. For example, *URL Builder – connector-Fetch Feed* is a connector pattern in our example scenario.

- *Data Mapping Patterns*: Associations of outputs to inputs. In Figure 1, for instance, we map the *description*, *title*, and *y:title* attributes of the fetched feed to the first input field of the first, second, and third rule, respectively, telling the *Filter* component how we map the individual attributes in input to the individual, free input parameters of the component.
- *Complex Patterns:* Partial compositions consisting of multiple components, connectors, and parameter settings. In our example scenario, different combinations of components and connectors, having their parameter values set and with proper data mappings, as a part and as a whole represent complex patterns. For example, the configuration *URL Builder – Fetch Feed – Filter – Location Extractor*, along with their settings, represents a complex pattern.

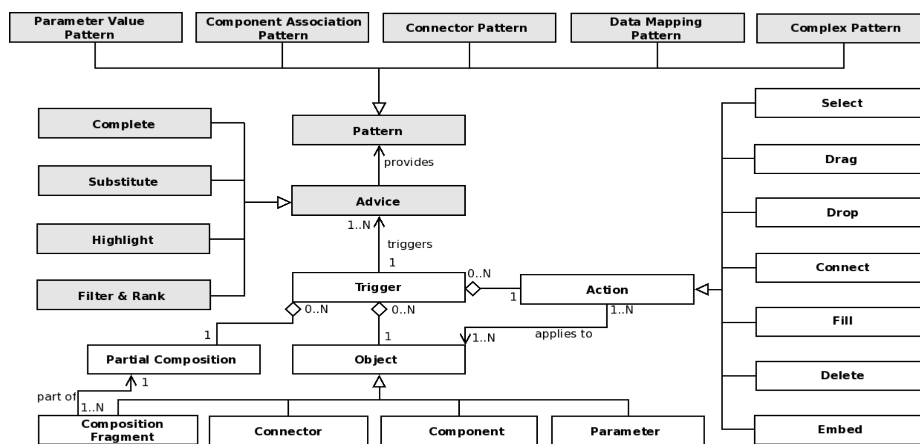

Figure 3 Conceptual model of WIRE's advice approach. Gray entities model the ingredients for advices; white boxes model the advice triggering logic inside the design environment.

An *Advice* provides composition knowledge in form of composition patterns. An advice can be to *complete* a given pattern (given it's partial implementation in the modeling canvas) or to *substitute* a pattern with a similar one, or the advice can *highlight* compatible elements in the modeling canvas or *filter and rank* advices.

Patterns represent the actual recommendation that we would like to deliver to the user. They can be of five different types: *Complex Patterns* (partial compositions possibly consisting of multiple components, connectors, and parameter settings), *Parameter Value Patterns* (possible values for a given parameter), *Component Association Patterns* (co-occurrence patterns for pairs of components), *Connector Patterns* (component-component or component-input parameter patterns), and *Data Mapping Patterns* (associations of outputs to inputs).

Now, let us discuss the "white part" of the model. This part represents the entities that jointly define the conditions under which advices can be triggered. A *Trigger* for an advice is defined by an object, an action of the user in the modeling canvas, and the state of the current composition, i.e., the partial composition in the modeling can-

vas. This association can be thought of as a triplet that defines the triggering condition. The *Objects* a user may operate are *Composition Fragments* (e.g., a selection of a subset of the pipe in the canvas), individual *Components*, *Connectors*, or *Parameters* (by interacting with the respective graphical input fields). The *Action* represents the action that the user may perform on an object during composition. We identify seven actions: *Select* (e.g., a composition fragment or a connector), *Drag* (e.g., a component or a connector endpoint), *Drop*, *Connect*, *Fill* (a parameter value), *Delete*, and *Embed* (one component into another). Finally, the *Partial Composition* represents the status of the current overall composition.

While the object therefore identifies *which* advice may be of interest to the user, the action decides *when* the advice can be given, and the state *filters* out advices that are not compatible with the current partial composition (e.g., if the *Location Extractor* component has already been used, recommending its use becomes useless).

Regarding the model in Figure 3, not all associations may be needed in practice. For instance, not all components are compatible with the *embed* action. Yet, the model identifies precisely which advices can be given and when.

5 The WIRE Platform

Figure 4 illustrates the high-level architecture of the assisted development environment with which we aim at supporting wisdom-aware development according to the model described in the previous section: developers can design their applications in a *wisdom-aware development environment*, which is composed of an *interactive recommender* (for development advice) and an *offline recommender* as well as the *wisdom-aware editor* implementing the interactive development paradigm. Compositions or mashups are stored in a *compositions repository* and can be executed in a dedicated *runtime environment*, which generates *execution data*. Compositions and execution data are the input for *the knowledge/advice extractor*, which finds the repeated and useful patterns in them and stores them as development and evolution advice in the *advice repository*. Then, the *recommenders* provide them as interactive advices through its query interface upon the current context and triggers of the user's development environment. Here, we specifically focused on development advices related to composition; we will approach evolution advices in our future work (evolution advices will, for instance, take into account performance criteria or evolutions applied by developers over time on their own mashups).

We realize that each domain will have suitable languages and execution engines, such as a mashup engine or a scientific workflow engine. Our goal is not to compete with these, but to define mechanism to "WIRE" these languages and tools with the ability to extract knowledge and provide advice. For this reason, in this paper we started with studying the case of Yahoo! Pipes, which is well known and allows us to easily explain our ideas. We however intend to apply the wisdom-aware development paradigm to our own mashup editor, mashArt [5], which features a *universal composition* paradigm user interface components, application logic, and data web services, a development paradigm that is similar in complexity to that of Pipes.

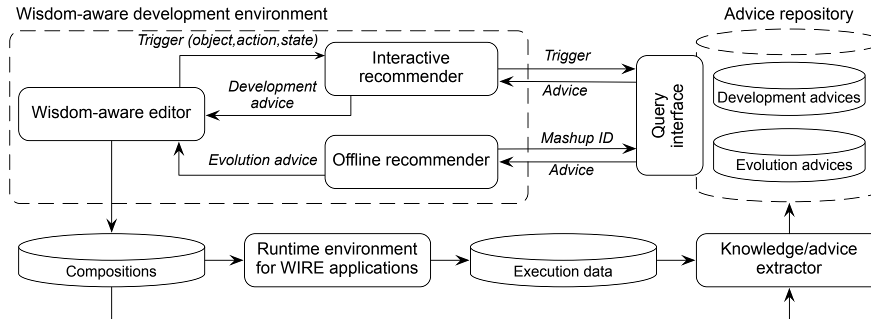

Figure 4 High-level architecture of the envisioned system for wisdom-aware development

As for the reuse of knowledge, the **WIRE approach** is not based on semantic annotations, matching, or AI planning techniques, nor do we aim at automated or goal-driven composition or at identifying semantic similarity among services. We also do not aim at having developers tag components or add metadata to let others better reuse services, processes, or fragments. In other words, we aim at collecting knowledge *implicitly*, as we believe that otherwise we would face an easier wisdom extraction problem but end up with a solution that in practice does not work because people do not bother to add the necessary metadata. WIRE will rather leverage on statistical data analysis techniques and data mining as means to extract knowledge from the available information space. To do so, we propose the following core steps:

1. *Cleaning, integration, and transformation:* We take as input previous compositions and execution data and prepare them for the analysis.
2. *Statistical data analysis and data mining:* On the resulting data, we apply statistical data analysis and data mining techniques, which may include mining of frequent patterns, association rules, correlations, classification and cluster analysis. The results of this step are used to create the composition patterns.
3. *Evaluation and ranking of advices (knowledge):* Once we have discovered the potential advices, we evaluate and rank them using standard interestingness measures (e.g., support and confidence) and ranking algorithms.
4. *Presentation of advices:* The advices are presented to the user through intuitive visual metaphors that are suitable to the context and purpose of the advice.
5. *Gathering of user feedback:* The popularity of advices is gathered and measured in order to better rank them.

Among the techniques we are applying for the discovery tasks, we are specifically leveraging on data mining approaches, such as *frequent itemset mining*, *association rules learning*, *sequential pattern mining*, *graph mining,* and *link mining*. Each of these techniques can be used to discover a different type of advice:

- *Frequent itemset mining:* The objective of this technique is to find the co-occurrence of items in a dataset of transactions. The co-occurrence is considered "frequent" whenever its support equals or exceeds a given threshold. This technique can be used as a support for discovering any of the advices introduced before. For instance, in the case of discovering *Component Association Patterns* we can this technique.

- *Association rules:* This technique aims at finding rules of the form *A*→*B*, where *A* and *B* are disjoint sets of items. This technique can be applied to help in the discovery of any of the proposed advices. For instance, in the case of the *Parameter Value Pattern*, given the value of two parameters of a component, we can find an association rule that suggests us the value for a third parameter.
- *Sequential pattern mining:* Given a dataset of sequences, the objective of sequential pattern mining is to find all sequences that have a support equal or greater than a given threshold. This technique can be applied to discover *Complex Patterns*, *Component Association Patterns,* and *Connector Patterns*. For instance, in the case of the *Connector Pattern*, we can use this technique to extract patterns that can be then used for suggesting connectors among components placed on the design canvas.
- *Graph mining:* given a set of graphs, the goal of graph mining is to find all subgraphs such that their support is equal or greater than a given threshold. For our purpose, we can use graph mining for discovering *Complex Patterns* and *Connector Patterns*. For instance, for *Complex Patterns* we can suggest a list of existing ready compositions based on the partial composition the user has in the canvas, whenever this partial composition is deemed as frequent.
- *Link mining:* rather than a technique, link mining refers to a set of techniques for mining data sets where objects are linked with rich structures. Link mining can be applied to support the discovery of any of the proposed advices. For example, in the case of *Data Mapping Patterns*, we can discover patterns for mapping the parameters of two components, based on the types these parameters.

Once community composition knowledge has been identified, we store the extracted knowledge in the advice repository in the form of directed graphs. In our advice repository, elements in the patterns, e.g., a component or a connector, are represented as nodes of the graph, and relationships among them, e.g., a component "has" a parameter, are represented as edges between those nodes. We also store a set of rules in our advice repository, which represent the trigger conditions under which a specific knowledge can be provided as an advice. Based upon this information, through our query interface we can match knowledge with the current composition context and retrieves relevant advices from the advice repository. Retrieved advices are filtered, ranked, and delivered based on user profile data (e.g., the programming expertise of the user or his/her preferences over advice types).

6 Conclusion

In this paper we propose the idea of *wisdom-aware computing*, a computing paradigm that aims at *reusing community composition knowledge* (the wisdom) to provide interactive development advice to less skilled developers. If successful, WIRE can e*xtend the "developer base"* in each domain where reuse of algorithmic knowledge is possible and it can *facilitate progressive learning and knowledge transfer*.

Unlike other approaches in literature, which typically focus on *structural and semantic similarities*, we specifically focus on the elicitation of composition knowledge that derives from the expertise of people and that is expressed in the compositions

they develop. If, for instance, two components have been used together successfully multiple times, very likely their joint use is both syntactically and semantically meaningful. There is no need to further model complex ontologies or composition rules.

In order to provide identified patterns with the necessary semantics, we advocate the application of the WIRE paradigm to composition environments that focus on *specific domains*. Inside a given domain, component names are self-explaining and patterns can easily be understood. In the Omelette (*http://www.ict-omelette.eu/*) and the LiquidPub (*http://liquidpub.org/)* projects, we are, for instance, working on two domain-specific mashup platforms for telco and research evaluation, respectively.

For illustration purposes, in this paper we used Yahoo! Pipes as reference mashup platform, as Pipes is very similar in complexity to our own *mashArt* platform [5] but better known. In order to have access to the compositions that actually hold the knowledge we want to harvest, we will of course apply WIRE to mashArt.

Acknowledgements: This work was supported by funds from the European Commission (project OMELETTE, contract no. 257635).

References

1. H. Geffner. Perspectives on artificial intelligence planning. *AAAI'02*, pp.1013-1023.
2. D. Roman, J. de Bruijn, A. Mocan, H. Lausen, J. Domingue, C. Bussler, D. Fensel. WWW: WSMO, WSML, and WSMX in a Nutshell, *ASWC'06*, pp. 516-522.
3. A. Koschmider, M. Song, H.A. Reijers. Social Software for Modeling Business Processes. *BPM'08 Workshops*, pp. 642-653.
4. T. Reichling, M. Veith, V. Wulf. Expert Recommender: Designing for a Network Organization. *Computer Supported Cooperative Work*, vol. 16, no. 4-5, pp. 431-465, Oct. 2007.
5. F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. *ER'09*, pp. 428-443.
6. O. Greenshpan, T. Milo, N. Polyzotis. Autocompletion for mashups. *VLDB'09*, pp.538-549.
7. T. Hornung, A. Koschmider, G. Lausen. Rommendation Based Process Modeling Support: Method and User Experience. *ER'08*, pp. 265-278.
8. A.V. Riabov, E. Bouillet, M.D. Feblowitz, Z. Liu, A. Ranganathan. Wishful Search: Interactive Composition of Data Mashups. *WWW'08*, pp. 775-784.
9. A.H.H. Ngu, M. P. Carlson, Q.Z. Sheng. Semantic-Based Mashup of Composite Applications. *IEEE Transactions on Services Computing*, vol. 3, no. 1, Jan-Mar 2010.
10. H. Elmeleegy, A. Ivan, R. Akkiraju, R. Goodwin. MashupAdvisor: A Recommendation Tool for Mashup Development. *ICWS'08*, pp. 337-344.
11. OMG. Business Process Model and Notation (BPMN) - Version 1.2, January 2009. [Online] *http://www.omg.org/spec/BPMN/1.2*
12. OASIS. Web Services Business Process Execution Language Version 2.0, April 2007. [Online]. *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*
13. S. Smirnov, M. Weidlich, J. Mendling, M. Weske. Object-Sensitive Action Patterns in Process Model Repositories. *BPM'10 Workshops*, NJ, USA, September 2010.
14. D. Wegener, S. Rueping. On Reusing Data Mining in Business Processes – A Pattern-based Approach. *BPM'10 Workshops*, NJ, USA, September 2010.
15. C. Shearer. The CRISP-DM model: the new blueprint for data mining. *Journal of Data Warehousing*, Vol. 5, Nr. 4, pp. 13–22, 2000.