Forms-based Service Composition for Domain Experts

Ingo Weber   Hye-Young Paik   Boualem Benatallah

Technical Report
UNSW-CSE-TR-1111
July 2011

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

In many cases, it is not cost effective to automate business processes which affect a small number of people and/or change frequently. We present a novel approach for enabling domain experts to model and deploy such processes from their respective domain as Web service compositions. The approach is based on user-editable service naming, a graphical composition language where Web services are represented as forms, a targeted restriction of control flow expressivity, automated process verification mechanisms, and code generation for executing orchestrations. A Web-based service composition prototype implements this approach, including a WS-BPEL code generator.

1 Introduction

A business process is "a set of logically related tasks performed to achieve a defined business outcome for a particular customer or market" [8]. Examples of business processes are hiring a new employee or ordering goods from a supplier. Business process management (BPM) refers to a management discipline and software suites that automate, improve, and optimize business processes to enhance productivity [29]. Despite the success of BPM, the reality is that today many processes are in fact *not* automated. First, among other reasons, BPM products are not suitably equipped to deal with processes that are ad-hoc [32]. Second, there are costs and high skills involved in implementing automated processes. This affects primarily the "long tail of processes" [24], i.e. processes that are less structured, that do not affect many people uniformly, or that are not critical. In addition, according to [36], among the organisations who use BPM software suites only 12% choose to use BPM automation components.

In recent reports and studies [26, 28, 25], the split between BPM technology and its value for end-users is acknowledged. The reports include recommendations on increasing the relevance of BPM for end-users, allowing process changes by non-technical personnel, and reducing the complexity of BPM technology. Another recent study found that the most important requirement for organizations world-wide looking at BPM tools is usability [25].

Motivated by the need for user-friendly BPM technology, the goal of this work is to devise an approach to support domain experts in their long-tail process automation needs. We focus on processes that can be implemented as Web service compositions. As a user group, we target business domain experts, i.e., non-IT professionals. We believe that these often have a good understanding of the processes they participate in; and that they are able to abstract from single process instances to the bigger picture of the process model containing alternatives and exceptions. An example is hiring a new employee, where HR recruiters have a good understanding of the default process and under which circumstances they may deviate from it.

The traditional approaches to BPM for process automation have inherited from programming. We believe this causes difficulties for the targeted users. The following lists the problems and requirements that are relevant to our goal:

- Programming tasks require writing code as abstract (symbolic, textual, or graphical) artifacts [12]. It is hard for untrained users to match their tasks to the abstractions.
- The so-called *selection barrier* [16] refers to the fact that often the users do not know how to express what they want the computer to do.
- Immediate verification of the programs is required, to provide feedback to the programmer [12].
- The system needs to understand high-level instructions of the programmer and translate them to a formal representation [12].

Our goal is to create a language and system for forms-based service composition, to allow domain experts to address their idiosyncratic, long-tail process automation needs themselves. Since we want to include processes where parts are executed conditionally, we propose a scripting approach to design process models. We focus on a graphical representation based on forms for designing processes. While we create a new tool and language for the core of our approach, the process designer, we use Web service standards (WS*) for the execution. In the approach, services can be described using names that are meaningful to the user, and independent of the services' technical names. We use these names during service discovery and in the process design.

Also, we aim at keeping the complexity of process modeling low, in order to make the approach applicable to domain experts. As such, we include features to verify the correct combination of the modeled control and data flows through automated verification techniques. A code generator can automatically translate the models to an executable language. The complexity is further limited through a targeted restriction of control flow expressivity. However, to enable fast execution, we include an automatic parallelization technique in our code generation. We note that the approach outlined above is very different to other service composition or workflow tools (e.g., JOpera[1], Kepler[2], Taverna[3]) which tend to support highly complex modelling and programming capacity, and demand higher level of assumed knowledge from their users. We conduct a preliminary case study with use cases from the financial domain: data analysis processes such as finding a correlation between news and stock price changes. The use cases are taken from our industry partner, Sirca[4].

In summary, the contributions of this report are the following:

- A forms-based composition method, where (i) user-editable forms are linked to Web services; and (ii) compositions can be modeled in a restricted, yet powerful generic language.
- Immediate automated process verification for reducing the burden on the user to build a correct composition.
- Automatic code generation with parallelization, to generate executable orchestrations from the forms-based composition language.

We also present a prototype and show how the approach can be enacted.[5]

The rest of the paper is structured accordingly. Section 2 describes the conceptual solution and an overview of our approach. The data flow modeling and verification are discussed in Section 3, the code generation in Section 4. The architecture and implementation are in Section 5, followed by guiding principles and a discussion in Section 6. Related work is described in Section 7, followed by a conclusion in Section 8.

2 Forms-based Service Composition Approach

In this section, we explain how services are created and managed in a repository, how they are represented for domain experts, and how domain experts can then model processes graphically. We start by introducing a running example.

[1]`http://www.jopera.org/`

[2]`https://kepler-project.org/`

[3]`http://www.taverna.org.uk/`

[4]`http://www.sirca.org.au`

[5]A demonstration video of the prototype can be found at `http://www.cse.unsw.edu.au/~FormSys/FormSys/`.

2.1 Use Case: News and Financial Data Analysis Process

In the following, we present a running example process. Research and development within Sirca on the possible utilization of available datasets led to the implementation of numerous Web services [30]. The types of Web services range from query/search, data cleaning, to complex statistical analysis. Currently, each Web service is invoked by a simple user interface based on Web forms, and the services operate independently. Most services operate on so-called time series, i.e., comma-separated values (CSV) files where every line marks an event at some point in time. For instance, news data contains news messages, along with meta-data like the time of publication, the headline, etc. The services mentioned below can manipulate theses files automatically; e.g., the merge service can merge two time series, based on time stamps. Certain analysis processes are performed repeatedly, but by manually locating and executing appropriate services each time. One such example is described in Fig. 2.1, where each step represents a Web service.

1. Find news data: *e.g., news data on the company 'BHP'*
2. Find performance data: *e.g., hourly stock price summary for code 'BHP.AX'*
3. Merge datasets: *e.g., merge the result data sets from the first two steps*
4. Perform statistical analysis: *e.g., which news were possibly influential on the price*
5. Visualize dataset: *e.g., influential news and the prices*

Figure 2.1: Financial data analysis from Sirca, for an Australian mining company 'BHP'

Repeating the above process by operating the Web forms involves around 30 mouse clicks, as well as entering the same information repeatedly at multiple steps. Once the processing is complete, the exact parameters that resulted in a given graph are lost. The set of changing parameters and the details of which service to use with which parameters differs between analysts and their tasks at hand. Therefore, while being repetitive, the processes are required to be flexibly executable or adaptable by the analyst.

Automating such processes is of interest to (i) reduce the required amount of user interaction, and (ii) retain the parameters that led to some visualization. The latter is important, because comparable graphs are required in certain periods. With this motivating example in mind, we present our approach next.

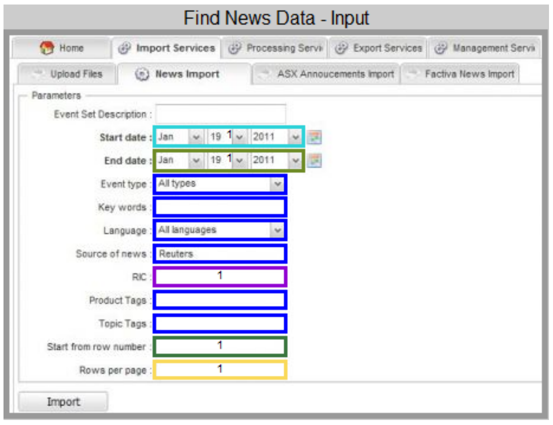
2.2 Forms as Service Interface Representations

In our repository, every service is collectively represented by a WSDL document, a user-editable name, an icon, and forms as graphical representations of input and output messages. In the following, we explain how and why we use these representations in addition to conventional WSDL documents. While WSDL needs to be present in the repository, and is used by our tool for generating an executable process, it is completely hidden from the domain expert designing processes.

Graphical Representations:

The service is a computational entity that performs some function, which is represented with an icon. For example, Figure 2.2 shows the icon for the "Find News Data" service.

Figure 2.2: Find News Data service as icon



Find News Data - Input

Parameters

Event Set Description :	
Start date :	Jan 19 1 2011
End date :	Jan 19 1 2011
Event type :	All types
Key words :	
Language :	All languages
Source of news :	Reuters
RIC :	1
Product Tags :	
Topic Tags :	
Start from row number :	1
Rows per page :	1

Figure 2.3: Graphical representation of Input Message for Find News Data

The technical information about a service is stored as standard WSDL. In fact, a service in our approach corresponds to a WSDL operation. An invocable WSDL operation has an input and an optional output message[6]. The input and output messages for a service are represented as forms – reflecting the service's running user interface. Fig. 2.3 shows an example of an input message as a form. The names of form fields, corresponding to the service's input/output parameters, as well as the names of services themselves are editable and can be tagged with names that are meaningful to the users, which is used during search and discovery of services. The form representation is also useful when the domain experts specify data mappings between messages.

Each data field from the message which is to used in process designs has a box associated to it, somewhere in the form. How the boxes correspond to data fields in the message has to be marked up manually for now, to enable automatic execution of designed processes. By default, the form could be rendered from the XML Schema type that belongs to the respective message. However, given our focus on domain experts, we believe that the form representation should be something the user is already familiar with. Hence, a screenshot of the UI through which the user commonly accesses this service makes a good representation.

[6]For simplicity, we currently neglect fault messages and exception handling, as well as certain XML Schema constructs and certain WSDL features.

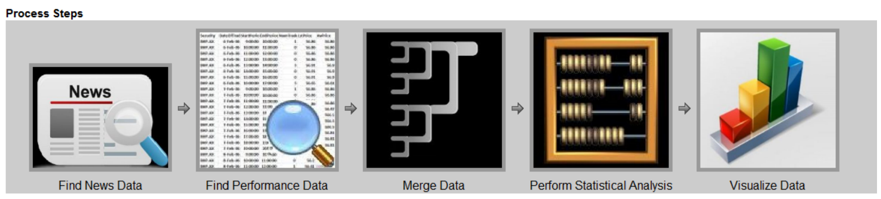


Figure 2.4: Graphical Process Modeling of the Running Example

Using Service Names Meaningful to Users:

Besides the technical information from WSDL and the graphical representations, the services in our repository are also given non-technical names. These names are created by users, at the time when entering a service into the repository. In the process modeling tool, the user can assign the service and its parameters names that they prefer to use for its process. For instance, while some user called a service “Find News Data”, another user may refer to it as “Import News Data”.

2.3 Forms-based Control Flow Modeling

Using the rich service descriptions outlined in the previous sections, modeling an executable process becomes a matter of drag-and-drop and clicking. We treat control flow and data flow as two separate, but not independent, layers. The control flow serves as an abstract process description: which services should be executed, under which conditions and in which order? The data flow adds more detail, by specifying how the input and output message fields of the various services interact.

In order to retain the focus on domain experts, the control flow modeling is restricted: services are arranged into a single sequence, and may be subject to some condition. If the condition evaluates to true in a process instance, the associated services are executed; if it evaluates to false, they are skipped in this instance. The conditions are free text, and are turned into questions to the user starting a process instance – see Section 4. The above restriction has significant impact on the expressiveness[7]. However, anecdotal evidence from experience with industry contacts suggests that forcing the occasional user of our system to understand the particularities of the semantics of an expressive language alienates most targeted domain experts. Therefore, we try to keep the control flow modeling as simple as possible – see Fig. 2.4 for a screenshot from our tool.

While the control flow modeling is limited, if there actually are no constraints which require a sequential execution of services, the execution can in fact be parallelized automatically. This is included in our approach and discussed formally in see Section 4.2.

[7]In Section 6.2, we discuss the expressiveness issue of our approach in detail with regards to workflow patterns.

3 Forms-based Data Flow Modeling

This section outlines which classes of data we encounter in our approach, the modeling of data flow, and how data flow relates to control flow. Then a few verification problems and their solutions are discussed (semi-)formally. We start with a short overview.

The data flow modeling works roughly as follows in our approach. Each service has an input and possibly an output message. A message consists of a set of fields, and has a form as graphical representation – c.f. Fig. 2.3. Data fields from one message can be mapped to data fields of another message in our approach. For instance, in our running example, the outputs of the two import services can be mapped to the merge service's input; the date range (i.e., from/to dates) in one data import service can be mapped to the date range of the other import service in our example process; etc. Besides mapping fields from one message to another, static values can be assigned to fields.

3.1 Input Data Field Classification and Handling

Data fields of input messages fall into one of three categories, with respect to processes: user-static, process-static, and process-instance-specific data.

- *user-static:* this is the type of information that rarely changes between process instances for the same person. For example, in a travel approval and reimbursement process, the name, title and address of 'John Smith', the traveler, are not likely to change from one trip (i.e., an instance) to the next. In the example process, this type of data plays no role.
  In general, in our approach, we want to support this kind of information by providing a *transparent data store*. That is, when the information is entered the first time, it is stored in a database. For subsequent process executions, the data is retrieved from the database and reviewed by the user. If any changes are made, the data store is updated. If applied correctly, this transparent data store achieves that the user does not have to enter the same data more than once.

- *process-static:* this class of data rarely changes between instances of the process, regardless of person executing them. In the example process, this includes parameters like the requested statistical measures. This is supported by allowing the process designer to assign static values to data fields.

- *process-instance-specific:* this is the type of information that changes from one instance of a process to another, even for the same user. In the example process, this includes the parameters for querying the data sources (e.g., from when to when). In our approach, this information is usually entered at the beginning of the process by the user. From the data flow and static assignments, our system determines which fields do need to be filled in, and those are presented to the user when starting a process instance. A specific subtype of information is *process-step-specific*, i.e., information which is only required in a single step of a process. However, we currently do not treat this sub-type differently.

3.2 Data Flow Model

In our method, the user can define mappings between fields of messages of different Web services. Depending on the kind of messages and the user's actions, the following can be the case:

- specifying that an output field of one service corresponds to the input field of another (**output-input mapping**);
- specifying that two (or more) input fields of separate services will get the same value from the process-specific user input (**input-input mapping**); or
- specifying a static value for an input field, which can be `null` (**static assignment**).

Output-output mappings are currently undefined and disallowed. By creating mappings, a user specifies the data flow explicitly, albeit not as a data flow graph. The reason we decided on this approach is that data flow graphs can get much more complicated than what we want the user to create in the first place.

Implicitly, a data flow graph is created as follows, where a directed edge represents the transfer of data from the edge's source activity to its target activity. An input activity (not shown) for the process user denotes the start of a process. It has a data flow edge to every regular activity in the process. An output-input mapping creates a new edge; a static assignment or an input-input mapping does not. An edge resembles a strict dependency: without the completion of the source node of the edge, the target node cannot execute. Therefore we require that the graph is acyclic for the obvious reason: circular dependencies are unsatisfiable. While edges stem from individual fields, nodes in the graph represent services; at most one edge is added from a given source to a given target.

There can be contradictions between the data flow and control flow of a process, as follows. Any execution of the process has to respect every edge in the data flow graph, i.e., the target can only be executed once the source's execution completed. A data flow graph constructed as explained above has potentially very little edges between the nodes, other than the edges from the start node. For such a graph, a set of possible valid *execution paths* exists, i.e., possible orders in which the graph's nodes can be visited without violating the dependencies. The control flow of the process is, however, primarily sequential as outlined in the previous section. Similarly, different execution paths for the control flow may exist, i.e., possible orders in which the nodes of the sequence can be visited, and where each condition may be true or false. In particular, the control flow may allow execution paths which are not allowed in the data flow graph. One of two different viewpoints of the relation between the control flow and data flow and their execution paths can be taken.[8]

1. The user specified the control flow, and the data flow cannot violate it. In other words, every execution path of the control flow has to be an

[8]A third alternative would be to abandon the control flow altogether, and rather maintain a "bag of services" relevant to the process, with their execution conditions. For scenarios where this interpretation is the best fit, the user interface should be designed to reflect this fact.

execution path of the data flow as well; the control flow is the dominant element.

2. Alternatively, the control flow is seen as more of a guideline, but if the data flow requires another ordering, this is taken to be more important; the data flow is dominant. Where there is a contradiction between the data flow and the control flow, the control flow is changed so as to fit the data flow.

We believe the second alternative is the most intuitive option for the proposed system: the user can design a more coarse-grained control flow and then add the more fine-grained data flow. If the user changes either of them in a way that creates a contradiction between the two, the system can ask the user if the control flow should be corrected to fit the data flow; if not, the problematic change is rolled back.

3.3 Process Verification

Above we described where some problems in the control and data flow may arise. In order to adequately deal with these, we give a short formalization of process graphs in our approach and the associated problems below, followed by their solutions. Generally we keep the formal discussion herein minimal, and focus on solving the problems encountered with standard graph algorithms for which textbook solutions exist – see, e.g., chapters 22-26 in [5].

Definition 1. *A* Process Graph (PG) *is a tuple $PG := (N, E, F, C)$, where*

- *N is a finite set of nodes $n_0, n_1, n_2, \ldots$ representing the start node (n_0) and Web services ($n_1, n_2, \ldots$).*
- *E is a finite set of directed data flow edges $e_1, e_2, \ldots$ where $e_i = (n_j, n_k)$ leading from n_j to n_k ($n_j \neq n_k$) is used to express data dependencies. The edge $e_i = (n_j, n_k)$ reads as "n_j provides data for n_k".*
- *F is a finite set of directed control flow edges $f_1, f_2, \ldots$ where $f_i = (n_j, n_k)$ leading from n_j to n_k ($n_j \neq n_k$) is a control dependency. The edge $f_i = (n_j, n_k)$ reads as "n_j should be executed before n_k". Each node can only be the source / target of at most one control flow edge: $\forall f = (n_i, n_j) \in F : \forall f' = (n_k, n_l) \in F \setminus \{f\} : n_k \neq n_i$ and $n_l \neq n_j$.*
- *C is a finite set of conditions $c_1, c_2, \ldots$ with $c_i =$ (<description>, N_{c_i}) being associated to a set of nodes $N_{c_i} \subseteq N$, and having some textual description.*

A data dependency path, *denoted as $\rightarrow_d$, from $n_i \rightarrow_d n_j$ exists iff $\exists e \in E : e = (n_i, n_j)$ or $\exists n_k \in N, e \in E : e = (n_i, n_k)$ and $n_k \rightarrow_d n_j$.*

An execution path *over the nodes in the process is an ordered permutation of a subset of the nodes in the graph and starts with the start node n_0: $[n_0, n_{e1}, n_{e2}, ...]$ where $\{n_0, n_{e1}, n_{e2}, ..\} := N_e \subseteq N$. n_{ei} is said to come before n_{ej} in some execution path $[n_0, n_{e1}, n_{e2}, ...]$, denoted as $n_{ei} \rightarrow_e n_{ej}$, iff $i < j$. An execution path $[n_0, n_{e1}, n_{e2}, ...]$ is said to be* compliant with the data flow *iff $\forall e = (n_i, n_j) \in E : n_i \rightarrow_e n_j$ or $n_i \notin N_e$ or $n_j \notin N_e$; and analogously the execution path is* compliant with the control flow *iff $\forall f = (n_i, n_j) \in F : n_i \rightarrow_e n_j$ or $n_i \notin N_e$ or $n_j \notin N_e$.*

Furthermore, we restrict data dependency paths such that n_i cannot depend on n_j (i.e., $n_j \rightarrow_d n_i$) with $n_j \in N_{c_k}$ for some condition c_k, unless $n_i \in N_{c_k}$ – we refer to this property the conditional execution rule. *In addition, each node can have at most one condition, i.e., the N_{c_i} are disjunct: $\forall c_i, c_j \in C : N_{c_i} \cap N_{c_j} \equiv \emptyset$.*

A *PG* consists essentially of two *directed acyclic graphs (DAGs)*, one for the control and one for the data flow, extended with the notion of conditions. The two graphs share the set of nodes and conditions, but have separate sets of edges. There are a few problems associated with PGs that we encounter in our approach:

1. Would adding a new data dependency introduce a violation of the conditional execution rule?
2. Would adding a new data dependency introduce a data flow cycle?
3. Would adding a new data dependency make the data flow graph contradictory to the control flow?
4. How to re-order the control flow to resolve a contradiction with the data flow graph?
5. Does an existing process violate the conditional execution rule?
6. Does an existing process contain a data flow cycle?

Problems 1 - 4 arise during editing, Problems 5 and 6 when loading or saving a process. Since all problems arise during interactions with the modeler, their respective solutions should be efficient. We now discuss each problem in more detail, and give a solution with polynomial runtime (over the *PG* size). While vast literature on process verification exists, due to the restrictions in our approach we can solve the problems with standard graph algorithms.

Problem 1 arises when the user wants to add a new data dependency, say from n_i to n_j. The question is, will adding this dependency violate the conditional execution rule, assuming it has not been violated as yet? This question can be answered easily by checking if $\exists c_k \in C : n_i \in N_{c_k}$ and $n_j \notin N_{c_k}$. This is a simple look-up, which can be done in $O(1)$.

Problem 2 arises also when the user wants to add a new data dependency, say from n_i to n_j. The question is, will adding this dependency introduce a cycle, assuming none exists as yet? This can be answered by checking if a dependency path exists from $n_j \rightarrow_d n_i$. If so, adding this new dependency will close a loop. Path existence can be checked with breadth-first search, running in $O(|N| + |E|)$. Alternatively, where this turns out to be too slow in practice, the transitive closure of the adjacency-matrix can be built and maintained, ideally while the user is not waiting for a response; a look-up in the matrix can be done in $O(1)$.

Problem 3 is the question if every control flow-compliant execution path is also data flow-compliant. If there are no conditions, the control flow has exactly one execution path. If, however, there are conditions, and assuming the conditional execution rule has not been violated, for every c_i any $n \in N_{c_i}$ is either a leaf node in the data flow graph in *PG* (has no outgoing data edges) or only has outgoing data edges to other $n' \in N_{c_i}$. This follows directly from the conditional execution rule: a node n' cannot depend on another node n that is subject to a condition, unless n' is also subject to that condition. Therefore, the nodes that are subject to one condition have no influence on any other nodes in the data flow graph. Contradictions between control and data flow can hence only arise between nodes that have the same condition, or nodes that have no condition, or nodes with a condition which depend on nodes with no condition; they cannot arise from interactions of nodes with different conditions. Therefore, all contradictions that may arise in some execution path, will also arise when *all* conditions are true. Hence it is sufficient to check whether the single execution path given by the control flow with all conditions being true, is a data flow-compliant execution path in PG. This can be done as follows: initially all nodes are unmarked, except for the start node; then we iterate through the nodes, in the order given by the execution path from the control flow with all conditions true, where we check if all incoming edges of the current node are satisfied, i.e., if the source of each incoming edge of the current node is marked. If not, we have a contradiction and end the procedure; if yes, we mark the current node. If no contradiction is found when reaching the end of the execution path, then none exists. This check can be done in $O(|N| + |E|)$, as every node and edge is visited exactly once.

Problem 4 concerns the question how to re-order the control flow in the case of a contradiction with the data flow, and can be solved using topological sort over the graph. See section 22.4 in [5] for an algorithm that runs in $O(|N|+|E|)$.

Problem 5 poses the question if any edge in a given process violates the conditional execution rule. This can be answered by checking if $\exists e = (n_i, n_j) \in E\ \exists c_k \in C : n_i \in N_{c_k}$ and $n_j \notin N_{c_k}$, in $O(|E|)$.

Problem 6 can be analyzed by computing the transitive closure of the graph, e.g., with an adjacency-matrix. A loop exists if any edge can be reached from itself. The runtime is around $O(|N|^3)$; see chapter 25 in [5] for details.

4 Code Generation for Process Execution

The domain expert having to do a set of tasks repetitively over and over again can create a process for this set of tasks, given there is a Web service for all of the required functionality.[9] The process can then be executed instead of triggering the tasks individually. The graphical model and data flow mapping from the user can be turned into an executable process by automatically generating WS-BPEL[10] code. Next, we describe code generation, input/output forms for the process and parallelization.

4.1 Process Input and Output

The goal of modeling a process in our approach is to automate the execution of repetitive tasks. All process-static data can be statically assigned in the process model. All process-instance-specific data should be entered only once per instance, even if used by multiple services. In our example process, instead

[9]Where the required functionality is not available as a Web service but a Web site, a service wrapper can be built [15] – this is not within the scope of our approach, and may require involving programmers.

[10]Web Services Business Process Execution Language (WS-BPEL), http://www.oasis-open.org/committees/wsbpel/

of manually triggering all steps individually, once all instance-specific data is entered, the process can complete without further user interaction.

In order to determine the necessary input for the process, our solution combines all inputs for all services, and removes any field which is the target of a mapping or static assignment. The result forms a message with the consolidated input data format to start the process. For this message, we generate a Web form, where the user can enter the information and trigger an instance of the process. Analogously, the outputs of all services are consolidated to one output message of the process, for which again a Web form is created. In the running example, the input to the analysis process will e.g., only have fields for the date range once, due to the input-input mapping; and no field for parameters of the statistical analysis, as they are assigned statically. The output Web form in the example contains a link to a Web page with the visualization.

Conditions from the control flow are in free text, and are included in the input message and input form of the process. For instance, the condition "If index data should be compared" is included as the question "Index data should be compared?" in the form, and an according Boolean data field in the message. The value of this field decides if the respective Web services are called or skipped.

4.2 Data Flow-based Parallelization

When desired by the user, our approach can parallelize steps in the process based on the data flow, by ignoring additional constraints from the control flow. In terms of the graph formalization from Section 3, the problem is how to generate a non-redundant data flow graph – i.e. without direct dependency links where transitive links exist. Formally, the aim is to remove any redundant edge in the graph, i.e., create $PG' := (N, E', F, C)$ from $PG = (N, E, F, C)$, such that (i) for any pair $n_i, n_j \in N$ we have $(n_i \rightarrow_d n_j) \Leftrightarrow (n_i \rightarrow_{d'} n_j)$, where $\rightarrow_{d'}$ refers to a data dependency path in PG'; and (ii) E' is minimal, i.e., removing any edge from E' would violate (i). A solution to this problem is to compute the so-called *transitive reduction* of PG [1], which can be done in $O(|N|^3)$[1, 5].

4.3 WS-BPEL Generation

While compositions from our approach could be translated to any execution language, we here chose to use the mature standard WS-BPEL, due to publicly available tooling such as Intalio[11]. Our tool can generate both the WS-BPEL code and the Web forms that are displayed to the process user – the latter in a format proprietary to the Intalio environment.

The parallelized data flow graph is translated to WS-BPEL as follows. Each node of the graph translates to a `sequence` element, containing two `assign` activities for the input and output data mapping, and in between, one `invoke` activity that calls the respective Web service. A `flow` in WS-BPEL enables parallel execution of all its contents, which can be partially ordered by `link` elements. Thus, all the sequences from the nodes are added to one `flow` element, and each data flow edge from the respective graph creates a `link` within the flow from and to the sequences corresponding to the respective nodes of the graph. Since a `link` is only introduced for each data flow dependency, and

[11] `http://www.intalio.com`

because the graph is non-redundant with respect to them, the resulting WS-BPEL process allows for parallel execution wherever no dependencies exist. If multiple links have the same target, the execution engine will wait for all of them to complete before executing the target element, which is the implicit join behavior in WS-BPEL.

As a root of the process structure, there is a `sequence` element with a `receive` as the initial activity, where a message with the consolidated input data is expected (from the generated Web form). After the `receive`, an `assign` activity initializes all message variables. This is followed by the above-explained `flow` element. The `flow` is followed by a `response` activity, which sends the consolidated output to the respective Web form in Intalio.

5 Architecture and Implementation

Our implementation builds on *FormSys Process Designer* [35], which focused on automating PDF form-based processes. The current version extends it heavily, so that processes can be built using arbitrary services.

5.1 Architecture

In order to allow domain experts to use the system without any upfront installation, it is implemented entirely as a Web application. The architecture, depicted in Fig. 5.1, comprises the following components:

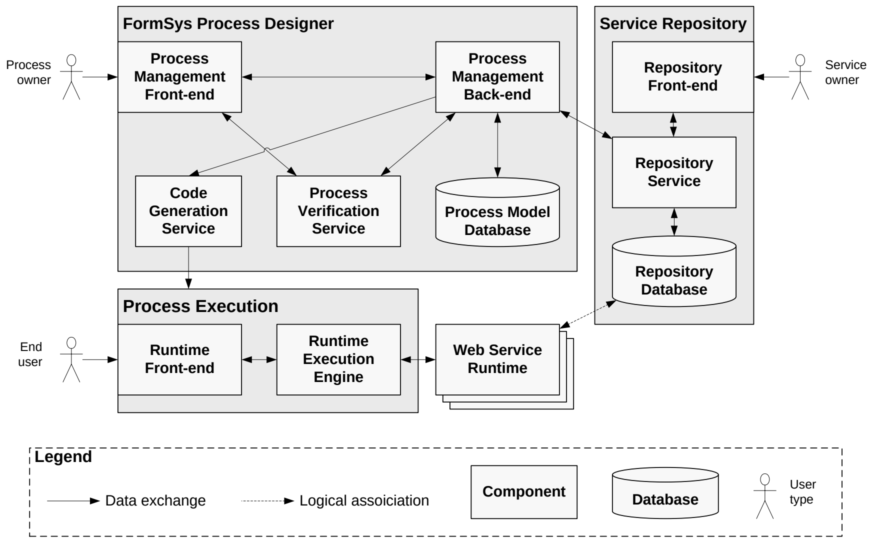


Figure 5.1: Architecture of FormSys Process Designer and related components.

- The process design environment for domain experts (the focus of this report):
  - The front-end, through which the user controls the tool.
  - A back-end, which handles most interactions with other components.

  - A database for persisting the process models.
  - Services for process verification and code generation. Note that the front-end interacts with the verification service directly.

- A repository containing metadata about services:
  - A front-end through which users can register new services and search for, edit, and delete existing ones.
  - A repository service, which offers the repository's functionalities to other components. In particular, the process management back-end uses the service for listing, searching, and retrieving information about available services.
  - A database for storing the service metadata, including the additional data required by our approach (icons, forms, etc.).

- A runtime process execution environment with a front-end through which process users can start new instances and retrieve notifications about completed instances; and an execution engine for enacting the service orchestrations.[12]

- A number of external Web services, which are represented in the repository and invoked from the runtime.

In the following we give an overview of the application program interfaces (APIs) of the services in our approach. We summarize these in Tables 5.1 - 5.3, informally listing the operation names, input parameters, and expected output or exceptions.

Operation Name	**Input**	**Output**
checkEdgeCanBeAdded	ProcessGraph, Source, Target	Ok or ExplanationText
checkControlVsDataFlow	ProcessGraph	Ok or ExplanationText
generateControlFromDataFlow	ProcessGraph	ControlFlowSequence
checkGraphIsCorrect	ProcessGraph	Ok or ExplanationText

Table 5.1: Operations of the verification service

Operation Name	**Input**	**Output**
generateBpelCode	ProcessControlFlow, Mappings, Assignments, DefaultValues, ServerURL, Parallelize?	ZipPackage or ErrorText
deployBpelCode	ProcessControlFlow, Mappings, Assignments, DefaultValues, ServerURL, Parallelize?	Ok or ErrorText

Table 5.2: Operations of the code generation service

[12]Currently, the Intalio suite serves this purpose, including the front-end. This choice requires using a proprietary format of Web forms for process input and output.

Operation Name	Input	Output
listAllServices		ListOfServices
searchForServices	Query	ListOfServices
getServiceDetails	ServiceID	ServiceDetails

Table 5.3: Operations of the repository service

5.2 Implementation

The implementation builds on our previous tool, FormSys Process Designer [35], but is a significant extension of it. In fact, the lines of code more than doubled from the previous version, and so has the number of database tables. The system is coded in PHP, using the Symfony framework[13] and JavaScript with various libraries. A screencast video of the tool in action is available[5].

The first proof-of-concept evaluation was done with some processes similar to the example process introduced in Section 2.1. In these processes, we use several data processing services and user interfaces developed in another project [30]. When a financial analyst uses these UIs and the services behind them, it takes quite a while to "click through" from entering the parameters to getting the desired visualization. Often, this process is repeated quite a few times with most parameters static, and only few ones changed.

Our tool has several UI screens, i.e., interactive Web pages, the most important of which is the process modeling part. Its top half is shown in Fig. 5.2. The first couple of fields carry standard header information of a process – name, owner, and free-text description – followed by buttons for saving and deploying the process. Below that is a *content flow area*, inspired by popular music player programs, which shows the icons and names for all services in the repository. Above the content flow, there is a search button for the respective dialog, where services can be searched by name. When a service is selected in the search, the content flow jumps to the respective icon.

The lower part of the modeling page contains areas for modeling control flow, labeled "Process Steps" and shown in Fig. 5.3, and data flow, labeled "Data Mappings" and shown in Fig. 5.4. Services are added by drag-n-dropping icons from the content flow into the control flow area. The icons can be re-arranged once placed in the control flow and conditions can be added.

When dragging a service icon into the "Data Mapping" area, the modeler is asked whether she wants to map the input or output message of the respective service – given both exist. The form of the chosen message is then loaded and displayed, where the fields in the message are highlighted as an overlay of HTML boxes. In Fig. 5.4 this has been done for the input messages of "Find news data" and "Find performance data" – i.e., here input-input mappings are modeled.

Mappings and static assignments are represented by colors; the list of declared mappings and assignments is shown as rectangles on the top of the mapping area, where rectangles with values inside are static assignments. Data field boxes belonging to a mapping/assignment are colored in the same way – e.g., the value "100" is assigned to the field "Rows per page". The list of declared mappings remains stable when replacing one or both of the forms in the mapping workspace. Hence, the user can specify mappings across fields of more than

[13]`http://www.symfony-project.org`

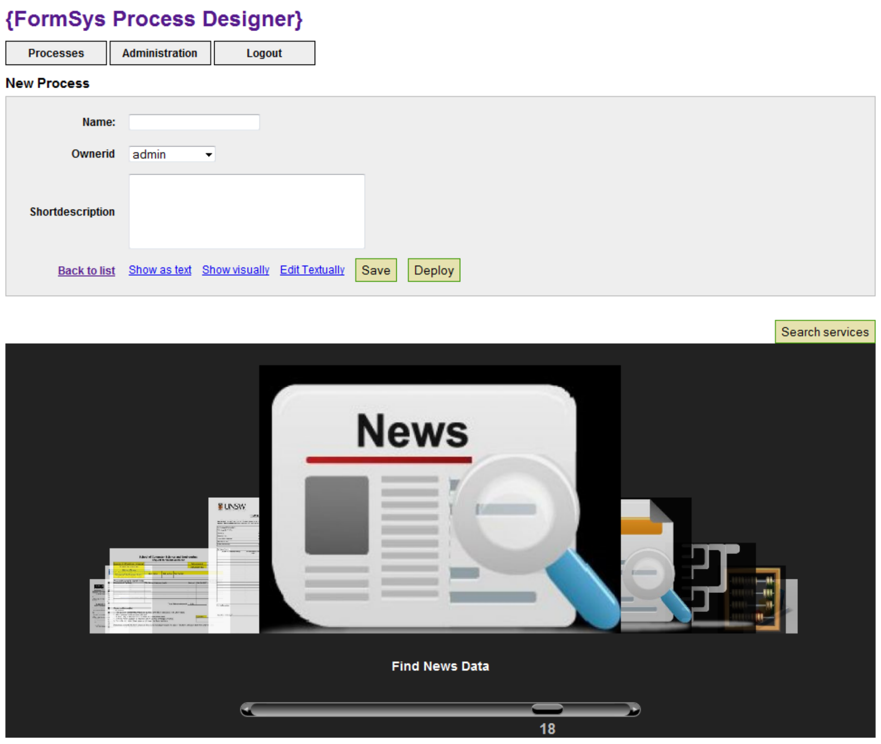


Figure 5.2: The Process Designer, upper part: header fields and content flow

two messages.

The verification described in Section 3.3 is enforced in the tool. To achieve this for Problems 1 - 4, JavaScript code transforms the mappings and sequence into data and control flow graph structures, respectively, and runs the algorithms, e.g., for detecting when a user would model a circular dependency.

We implemented a number of compositions similar to the running example process, with various of the Sirca data analysis services. While the feedback from our industry partner shows interest in the solution, we also learnt where we need to improve the approach, as discussed next. Firstly, the conditional execution rule from Section 3.3 is restrictive: some variations of the example process would benefit from conditional data flow, as the output of conditionally executed services could then be fed into other services. Secondly, the automatic

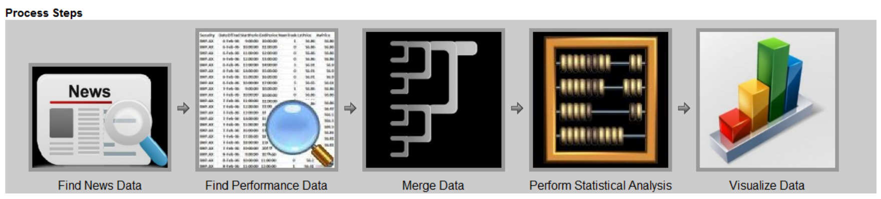
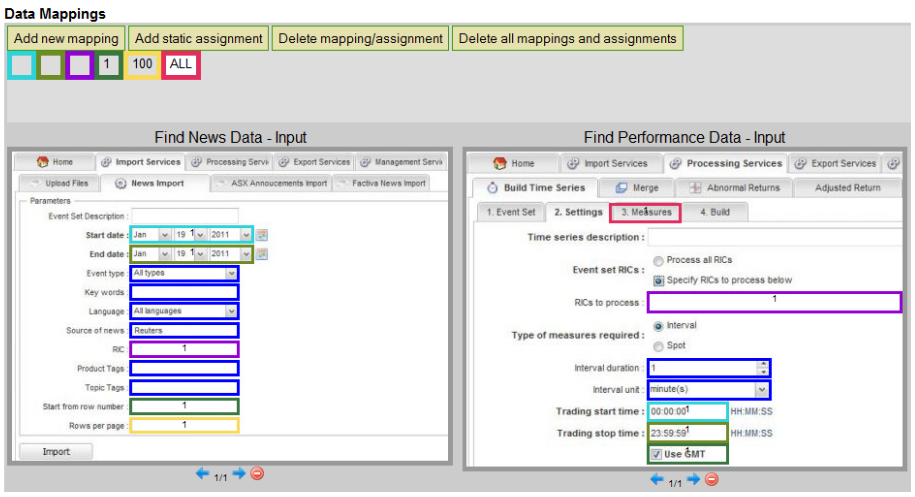


Figure 5.3: Graphical Process Modeling of the Running Example, repeated from Fig. 2.4.



Find News Data - Input

Parameters

Event Set Description :	
Start date :	Jan 19 2011
End date :	Jan 19 2011
Event type :	All types
Key words :	
Language :	All languages
Source of news :	Reuters
RIC :	1
Product Tags :	
Topic Tags :	
Start from row number :	1
Rows per page :	1

Find Performance Data - Input

Time series description :	
Event set RICs :	Process all RICs [✓ Specify RICs to process below]
RICs to process :	1
Type of measures required :	[✓ Interval] Spot
Interval duration :	1
Interval unit :	minute(s)
Trading start time :	00:00:00 HH:MM:SS
Trading stop time :	23:59:59 HH:MM:SS
	✓ Use GMT

Figure 5.4: The Process Designer's data flow part

generation of UIs is based on field names from XML Schema elements, which may be understandable names, but may also be very technical. In our previous tool [35] the user could rename the fields, which allowed more control over the labels on the UIs. Similarly, the tool should enable the user to specify which output she wants from the process. At the moment, this is the union of the output fields of all services. When the services have such fields, the automatically generated output UI of the process may end up flooded with irrelevant information. Once this option is implemented, specifying output-output mappings would make sense as well, e.g., to concatenate the results from two services into a single field.

6 Discussion

We first discuss some principles guiding the design of our approach. Then, the expressiveness of our modeling methodology is discussed.

6.1 Guiding Principles

Our approach aims to be usable for domain experts (not "end-user programmers", in contrast to many end-user programming approaches). As a user group, we target business domain experts as a class of end-users, i.e., non-IT-professionals. We believe that these often have (or can obtain through reflection) a good understanding of the processes they participate in, at least from their own point of view; and that they are able to abstract from single process instances to the bigger picture of the process model containing the alternatives and exceptions as well as how to handle them. An example is hiring a new employee, where HR recruiters will often have a very good understanding of the default process and under which circumstances they need to deviate from it: e.g., an ethics review for anyone hired in procurement because of the higher risk of exposure to favorism, bribing, and fraud; or higher management approval for salaries exceeding usual pay levels for a certain position.

Catching all these alternatives with programming-by-demonstration (PbD) would mean all alternatives have to be demonstrated to the system. Hence we believe a scripting approach is the best fit for this category of users and these kinds of processes; a thorough evaluation of this point remains interesting future work, but exceeds the scope of this report. Another point we plan to address in the future is how to achieve PbD for Web services: domain experts can hardly be expected to type SOAP messages themselves; only if there was a more user-friendly UI for every service PbD could be used.

As part of our approach, we proposed a language for service composition by domain experts, along with two representations as follows. The two representations of processes are (i) purely textual (listing the service names, similar to Fig. 2.1) and (ii) in a storyboard fashion, with graphical icons representing the individual Web services in addition to a textual label. A recent study [27] suggests that storyboards may be a more appropriate (i.e., intuitive) representation for users that are unfamiliar with formal modeling methods. Hence, we provide the user with a way to model in a (graphical) storyboard fashion, as shown in the body of the work. This representation is based on a scripting methodology, and interspersed with textual scripting features (typing commands in the search, etc.); a scripting-like edit mode will be added in the future. Generally, we believe different user types might prefer different representations – e.g., business users in finance might prefer the graphical representation; scientists might prefer a textual representation for scientific workflows.

6.2 Discussion of the Control Flow Expressiveness

The expressive power of process modeling approaches is often investigated by analyzing to which degree the *workflow patterns* are supported in a given approach [34]. Generally, stronger expressive power is regarded as a positive feature of a modeling approach. However, already in their seminal paper [34], van der Aalst et al. discuss the issue of suitability, i.e., that the modeling approach needs to be well suited for the problem to which it is applied. The trade-off chosen here is the attempt to achieve this suitability: achieving the simplicity required for domain experts modeling processes, without sacrificing necessary expressive power. In the following we discuss which patterns are supported by our approach, and which patterns are less relevant given the scope and target of our approach. The following assumes the reader is familiar with the set of workflow patterns presented in [34].

Out of the basic workflow patterns we obviously support sequence (pattern 1 in [34]. Parallel split and synchronization are supported indirectly, through the automated parallelization (Section 4.2). Exclusive split is supported to a large degree, in that process branches can be executed or skipped. That means that executing either A or B needs to be handled with two separate conditional branches. However, this construct is in our opinion more naturally suited to the use cases, and easier to understand. Since the conditions are by default interpreted as independent, this implements the multi-choice pattern. The more complicated split/join patterns are even hard to understand at first for IT professionals, and are clearly out of scope for our purposes.

Our execution semantics assume implicit termination (pattern 12 in [34], i.e., once all activities are completed, the process instance terminates. Loops or multiple instantiation are rarely needed in the use case scenarios we focus on: data processing services inherently compute results by iterating over entries, often in the hundreds of thousands or even millions – doing so via repeated Web service invocation with current technology would be prohibitively inefficient. Backtracking within a process instance to correct something and then execute a number of steps is interesting, but left for future work. Interleaved parallel routing (pattern 19), i.e., the execution of a set of activities in an arbitrary order, is essentially the result of the parallelization in our approach, together with WS-BPEL's `flow` semantics. In contrast, interleaving sequences (pattern 17) are not fully supported: while a partial order over parallel activities is indeed created by the parallelization, the requirement to only execute one activity at any time is not supported. This would be hard to achieve with WS-BPEL's `flow` semantics. Deferred choice (pattern 18) requires events, which we see as too advanced for many domain experts in the first iteration of our approach.

Cancellation of a process instance can be handled by the underlying workflow system. In our implementation (cf. Section 5) we rely on Intalio's process server, which supports cancellation through an administration console.

Inter-workflow synchronization (patterns 23-26) is not required in our approach, as all processes are assumed to have only stateless activities.

In [39], zur Muehlen and Recker investigated the usage of the BPMN modeling elements in 120 BPMN models. Among their analyses is the occurrence frequency of BPMN constructs in these models. In the ranking of occurrence frequency, the first construct that could be supported[14] by our approach, but is not, are subprocesses on rank 13; they are used in less than 30% of the models. In comparison, data-based exclusive split/join is used in up to 80% of the models, depending on the source of the diagram. Although BPMN is used for many purposes, we see the fact that most of the commonly used constructs in general-purpose process modeling are supported in our approach as an indication that the expressive power is likely to be sufficient for our purposes.

Finally, we note that the PICTURE approach [4, 3], discussed in Section 7, has been successful in practice, in a related setting with an expressive power slightly below ours.

7 Related Work

Section 1 discussed our work relative to works on traditional business processes, workflows, and Web service composition. Here we present other related work. **Mashups** have similar goals to our system. Topics such as data harvesting and visualization, composition of existing data and UI, and custom views or UIs for existing services are common in mashups [23, 37]. While this may facilitate certain processes, the predominant composition paradigms in mashups are event-based synchronization [38] and data flow between components, not control flow over process activities [10]. While there is some overlap between mashup approaches and ours, we see them as largely complementary.

Chapter 20 in [6] is a study on mashup developers: the vast majority of experienced mashup developers today has significant prior programming experience. "When the [user's] knowledge state [about a new technology] is very

[14]Pools and lanes are of limited use in the process models considered here, and message flow is implicit in our approach.

low, perceptions of benefits or payoff may have greater influence on the user's willingness to invest attention more than perceptions of cost [of learning the new skills]." In our tool, we believe the trade-off is quite favorable for the user: not too much is expected in terms of skills, while the benefits in terms of productivity gains can be significant. The same source finds that working examples are important for documentation in Web APIs. In future versions of our tool, it might be useful to provide example processes for novice users to experiment with, to see how the tool works, and to understand which services do what.

Collaborative BPM Tools: Recently, *collaborative business process modeling* has gained attention, e.g., [2, 31]. Tools in this area aim to facilitate collaborative discovery and process re-engineering. These are related to our work in that the target audiences include non-IT professionals or non-BPM experts. For example, *ARISalign*[15] [31] is a social network-based BPM tool whose focus is to involve stakeholders from within and outside an organization in designing executable processes. *Gravity* [2] is a real-time collaborative process modeling environment, embedded in Google Wave and SAP 12sprint. There is a high-level modeling perspective (BPMN) and an executable view which uses proprietary notations similar to Yahoo Pipes.

Blueprint[16] allows users to collaboratively discover process models. The user, or group of users, starts by creating a "process map", which resembles Porter's value chains. The map consists of a sequence of high-level steps, where more detailed process steps can be added below each high-level step. The precise control flow can be designed in BPMN, and exported for further refinement to execution in Lombardi's other BPM tool, Teamworks 7. A strong feature of Blueprint is Web 2.0 principles: users can share, comment, update, and be notified about changes in process models. The focus is, however, on BPM professionals discovering or improving critical processes in an organization.

A recent startup, *Signavio*[17], offers a browser-based BPMN and value chain editing tool. The basis of this was the tool Oryx [9]. Again, a focus is on collaboration, commenting, and sharing process models with others. Another feature is a user-defined dictionary, to encourage similar naming of activities, documents, and other labels. The open-source project Activiti[18] makes use of Signavio's modelling tool, so that BPMN models can be executed in the Activiti environment. In relation to our solution, the Signavio Process Editor is quite similar to Lombardi's Blueprint.

End-User Process Modelling: Todor Stoitsev [33] investigates using Task Management (Outlook plugin) for "process modelling by example": the system tracks how people split up larger tasks into subtasks, and delegate some subtasks to others; this can be used as input to a workflow design tool. [21] describes a technique for constructing process models with formal execution semantics from informal models (e.g., Powerpoint drawings). The technique stops at producing BPMN. It may be extended to generate executable models. However, the missing aspects to enable that (e.g., service selection, data flow) are not explored.

[7] describes BPEL4UI / MarcoFlow: a language and tool for enabling BPEL designers to incorporate distributed UI composition in BPEL processes.

[15]Software AG, `http://www.arisalign.com/`
[16]Lombardi, `http://www.lombardisoftware.com/`
[17]`http://www.signavio.com/`
[18]`http://www.activiti.org`

Mashup-like UI components are synchronized between each other (for a single user), with UI components at other users, and the process. Microsoft InfoPath[19] essentially is a code-free software engineering tool. However, from the mind set and knowledge required it is still for users familiar with programming, e.g., who know how databases work or what Web service are.

[14] uses a process language based on statecharts, for design professionals to draft screen processes. The language offers sequences and alternative branching. When reporting on their studies with real users, no problems were found regarding the process language – which is encouraging for our approach. However, the level of IT-literacy of the study's participants remains unclear.

PICTURE is a domain-specific modelling method and notation for public administration [3]. The approach is based on a fixed, domain-specific set of modeling constructs, the *building blocks*, and on using natural terms to the users, e.g., "receiving a document", "consult with other party". Building blocks are arranged in sequences to form local subprocesses, and different subprocesses can be linked via *anchors*. Interestingly, the argument for pure sequences in subprocesses is similar in our work. In fact, the evaluation of the system indicates that the fixed level of abstractions and familiar terms increased efficiency in process discovery. We take similar approaches to PICTURE in that we use simple abstractions and natural terms. The key differences are: PICTURE targets capturing the processes, and does not support creating executable processes; and PICTURE is domain-specific, whereas our tool is generic.

End-User Programming: A field of research and recently rising in popularity that has a similar goal to ours, although in different domain, is *end-user programming (EUP)*. EUP is the umbrella term for approaches that "make limited forms of programming sufficiently understandable and pleasant that end users will be willing and able to program" [6]. Good overviews of EUP are, e.g., [22, 6]. We have considered many ideas from EUP in this work. For instance, the two dominant paradigms in EUP [6] are *scripting languages* – usually simplified programming languages – and *programming by demonstration* (PbD) – the user demonstrates to the programming tool what she wants to be done, and the tool interprets and generalizes the intention to formulate a program.

Among the dangers in EUP are the lack of security and the lack of testing [13]. Security is (mostly) built into our system; the limitation on service composition means that users cannot create uncontrolled code. However, security here is based on the assumption that the underlying components (orchestration engine, messaging mechanisms, services, etc.) have not been compromised. The lack of testing is relevant for our approach, and motivates current work on enabling process simulation in the design environment.

A PbD approach in EUP that we consider closely related work is *CoScripter* (formerly called Koala) [18, 20], which primarily focuses on personal processes in the scope of browsing and using Web applications. The user can record, play and publish/share such browser processes on a public Wiki. The processes are stored in a simple end-user understandable language, using natural language keywords such as "go to <URL>" and "click on <link>". Personal user information is stored in a *personal database* on the user's machine. In the scenarios covered by this approach, the usefulness has been demonstrated by real users in their day-to-day work lives [18]. In contrast to our approach aiming at Web services,

[19] `http://office.microsoft.com/en-us/infopath/`

the scope of CoScripter is limited to the browser: all steps in a script need to be standard operations in a browser window. In CoScripter, all steps in a script need to be standard operations in a browser. While closely related to our work in terms of the understandability of process steps and the user focus, it does not support Web service invocation or conditional execution.

[19] combines CoScripter [18] with a table to store intermediate results, and a mashup-by-demonstration approach. The focus is on ad-hoc mashups, not on robust and reliable mashups that can be reused by millions, but rather on one-off data collection tasks and similar. While not directly related to our approach, it shows an aspect that can motivate end-users: instant gratification, as an effect of direct manipulation. Our work aims to leverage this effect as well.

Chapter 7 in [6] presents *Atomate*: a system where users can specify rules in Controlled Natural Language (CNL) for alerting them of certain situations in their Personal Information Managemant, such as: “SMS me whenever I get home on Tuesdays to take out the trash.” Through their constrained-input CNL UI they have some similarity with our textual process representation. The work also shows how real-time context and events can be included in user-focused tools.

Chapter 10 in [6] covers *d-mix*, a system for sampling information from Web APIs through their Web pages. The first step of their method is that professional developers write scripts that map between APIs (Web services) and corresponding Web sites – e.g., Google Maps, Amazon, Flickr etc. This has some similarity to entering a Web service into our repository. Other than that, d-mix pursues a quite different goal.

Chapter 11 in [6] describes *ITL*, a tool that enables demonstrating procedures for cross-application desktop workflows. The recorded workflows are displayed as textual processes, which can be edited in a constrained environment. The workflows are also executable, based on an automatically learned data flow. While the workflows look somewhat similar to our textual representation, there are two big difference to our approach: we target service composition, not desktop automation, and pursue a scripting approach, not programming by demonstration.

[17] reports on a clever extension of CoScripter: a natural-language command interface, where simple commands like forward phone line to home phone are understood and executed as CoScripter scripts. If the command needs further clarification, like input parameters (e.g., which phone line: work or mobile?), the tool asks the user about this. To realize this, the approach uses existing scripts and a browsing history, as well as a mechanism that extracts relevant sub-scripts from the history. The system offers various transport mechanisms, such as Twitter and SMS.

[11] presents a novel approach to scientific workflow modeling: the user types in a command, and a graphical workflow model is extended accordingly. One remarkable feature is that objects as well as commands can be accessed in this way. It is related to our search, where suggestions (for single steps at the moment) are presented to the user. This is currently not a core contribution of our work; future extensions may be inspired by [11].

8 Conclusion and Future Work

We have presented a forms-based service composition approach which allows domain experts with little technical knowledge to encode idiosyncratic, repetitive business processes themselves from design to execution. To realize this, we present a novel service composition method, where services have meaningful names and their input/output messages are represented by user-editable forms. We also offer automatic process verification. and support code generation to translate the process models to WS-BPEL. While we support limited modeling concepts for simplicity, executable processes can be optimized by automatic parallelization. The approach is implemented in a Web-based tool; a demonstration video is available[5]. Our preliminary evaluation revealed the following as desirable: user-editable input/output forms of the process; conditional data mappings; and process simulation in the design environment. For our use cases, the number of data mappings stayed reasonably small and therefore using colors to represent them was sufficient. However, if too many colors are required, they can eventually become confusing. These findings guide part of our immediate future work; in particular, a richer language for data mapping is under investigation, to enable more complex mappings and conditions over data fields. With these improvements the tool will be further tested by our industry partner, and user studies will be conducted.

Acknowledgements

This work has been supported by a grant from the Smart Services CRC[20]. We thank Maurice Peat, Fethi Rabhi, Kader Lattab, and Angel Lagares Lemos for their valuable feedback.

Bibliography

[1] A. Aho, M. Garey, and J. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[2] S. Balko, A. Dreiling, K. Fleischmann, and T. Hettel. Gravity – collaborative business process modelling and application development. SAP Community Network, `http://tinyurl.com/y8mn9g6`, accessed 2/6/2011, 23 Feb. 2010.

[3] J. Becker, L. Algermissen, D. Pfeiffer, and M. Räckers. Baustein-basierte Modellierung von Prozesslandschaften mit der PICTURE-Methode am Beispiel der Universitätsverwaltung Münster. *Wirtschaftsinformatik*, 49:267–279, 2007.

[4] J. Becker, D. Pfeiffer, and M. Räckers. PICTURE - a new approach for domain-specific process modelling. In *CAiSE Forum*, 2007.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, 2001.

[20]`http://www.smartservicescrc.com.au`

[6] A. Cypher, M. Dontcheva, T. Lau, and J. Nichols, editors. *No Code Required - Giving Users Tools to Transform the Web.* Morgan Kaufmann, 2010.

[7] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. From People to Services to UI: Distributed Orchestration of User Interfaces. In *BPM'10*, pages 310–326, 2010.

[8] T. H. Davenport and J. E. Short. The New Industrial Engineering: Information Technology and Business Process Redesign. *MIT Sloan Management Review*, 31(4):11–27, 1990.

[9] G. Decker, H. Overdick, and M. Weske. Oryx – an open modeling platform for the BPM community. In *Demonstrations at BPM'08: 6th Int'l Conf. on Business Process Management*, 2008.

[10] G. Di Lorenzo, H. Hacid, H.-Y. Paik, and B. Benatallah. Data Iintegration in Mashups. *SIGMOD Rec.*, 38(1):59–66, 2009.

[11] P. Groth and Y. Gil. A scientific workflow construction command line. In *Proceedings of the 14th international conference on Intelligent user interfaces*, IUI '09, pages 445–450, New York, NY, USA, 2009. ACM.

[12] D. Harel. Can Programming Be Liberated, Period? *Computer*, 41:28–37, 2008.

[13] W. Harrison. From the editor: The dangers of end-user programming. *IEEE Software*, 21:5–7, 2004.

[14] B. Hartmann, S. Follmer, A. Ricciardi, T. Cardenas, and S. R. Klemmer. d.note: revising user interfaces through change tracking, annotations, and alternatives. In *CHI '10: 28th Int'l Conf. on Human factors in computing systems*, pages 493–502, New York, NY, USA, 2010. ACM.

[15] H. P. Huy, T. Kawamura, and T. Hasegawa. How to make web sites talk together: web service solution. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, WWW '05, pages 850–855, New York, NY, USA, 2005. ACM.

[16] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *VLHCC '04*, pages 199–206, 2004.

[17] T. Lau, J. Cerruti, G. Manzato, M. Bengualid, J. P. Bigham, and J. Nichols. A conversational interface to web automation. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 229–238, New York, NY, USA, 2010. ACM.

[18] G. Leshed, E. Haber, T. Matthews, and T. Lau. CoScripter: Automating & Sharing How-To Knowledge in the Enterprise. *CHI Letters: Human Factors in Computing Systems*, 10(1):1719–1728, 2008.

[19] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *IUI '09: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 97–106, New York, NY, USA, 2009. ACM.

[20] G. Little, T. Lau, A. Cypher, J. Lin, E. Haber, and E. Kandogan. Koala: Capture, share, automate, personalize business processes on the web. *CHI Letters: Human Factors in Computing Systems*, 9(1):943–946, 2007.

[21] D. Mukherjee, P. Dhoolia, S. Sinha, A. Rembert, and M. Nanda. From Informal Process Diagrams to Formal Process Models. In *BPM'10*, pages 145–161, 2010.

[22] B. A. Myers, A. J. Ko, and M. M. Burnett. Invited Research Overview: End-User Programming. In *CHI '06*, pages 75–80, 2006.

[23] M. Ogrinz. *Mashup Patterns: Designs and Examples for the Modern Enterprise.* Addison-Wesley Professional, Mar. 2009.

[24] Oracle White Paper. State of the Business Process Management Market 2008. `http://tinyurl.com/3c4u436`, accessed 20/11/2009, Aug. 2008.

[25] S. Patig, V. Casanova-Brito, and B. Vögeli. IT Requirements of Business Process Management in Practice. In *BPM'10*, pages 13–28, 2010.

[26] C. Pettey and L. Goasduff. Gartner Reveals Five Business Process Management Predictions for 2010 and Beyond. Gartner Press Release, `http://www.gartner.com/it/page.jsp?id=1278415`, accessed 2/9/2010, January 13 2010.

[27] J. Recker, N. Safrudin, and M. Rosemann. How novices model business processes. In *BPM'10*, pages 29–44, 2010.

[28] H. Reijers, S. van Wijk, B. Mutschler, and M. Leurs. BPM in Practice: Who Is Doing What? In *BPM'10*, pages 45–60, 2010.

[29] C. Richardson, K. Vollmer, C. L. Clair, C. Moore, and R. Vitti. Business Process Management Suites, Q3 2009 – The Need For Increased Business Agility Drives BPM Adoption. Forrester TechRadar For BP&A Pros, 13 Aug. 2009.

[30] C. Robertson, F. Rabhi, and M. Peat. *A Service-Oriented Approach towards Real Time Financial News Analysis*, chapter in Consumer Information Systems and Relationship Management: Design, Implementation and Use. IGI Global, 2011.

[31] P. Sayer. Software AG Opens BPM Social Networking Beta Test. PCWorld Business Center, `http://tinyurl.com/yecoz2m`, accessed 02/06/2011, 2 Mar. 2010.

[32] T. Schurter. BPM State of the Nation 2009. bpm.com, `http://www.bpm.com/bpm-state-of-the-nation-2009.html`, accessed 25/11/2009, 2009.

[33] T. Stoitsev. *End-User Driven Business Process Composition.* PhD thesis, TU Darmstadt, Fachbereich Informatik, Telekooperation, 2009.

[34] W. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[35] I. Weber, H. Paik, B. Benatallah, C. Vorwerk, Z. Gong, L. Zheng, and S. Kim. Managing Long-tail Processes Using FormSys. In *ICSOC'10*, pages 702–703, 2010.

[36] C. Wolf and P. Harmon. The State of Business Process Management. Technical report, BPTrends, June 2006.

[37] J. Wong and J. Hong. What Do We "Mashup" When We Make Mashups? In *WEUSE'08*, pages 35–39, May 2008.

[38] J. Yu, B. Benatallah, F. Casati, and F. Daniel. Understanding Mashup Development. *IEEE Internet Computing*, 12(5):44–52, 2008.

[39] M. zur Muehlen and J. Recker. How Much Language is Enough? Theoretical and Practical Use of the Business Process Modeling Notation. In *CAiSE'08: 20th Int'l Conf. on Advanced Information Systems Engineering*, Montpellier, France, June 2008.