Crowd-Based Mining of Reusable Process Model Patterns

Carlos Rodríguez, Florian Daniel, and Fabio Casati

University of Trento,
Via Sommarive 9, I-38123, Povo (TN), Italy
{crodriguez,daniel,casati}@disi.unitn.it

Abstract. Process mining is a domain where computers undoubtedly outperform humans. It is a *mathematically complex* and *computationally demanding* problem, and event logs are at *too low a level of abstraction* to be intelligible in large scale to humans. We demonstrate that if instead the data to mine from are *models* (not logs), datasets are *small* (in the order of dozens rather than thousands or millions), and the knowledge to be discovered is *complex* (reusable model patterns), humans outperform computers. We design, implement, run, and test a *crowd-based pattern mining* approach and demonstrate its viability compared to automated mining. We specifically mine *mashup* model patterns (we use them to provide interactive recommendations inside a mashup tool) and explain the analogies with mining business process models. The problem is relevant in that reusable model patterns encode valuable modeling and domain knowledge, such as best practices or organizational conventions, from which modelers can learn and benefit when designing own models.

Keywords: Model patterns, Pattern mining, Crowdsourcing, Mashups

1 Introduction

Designing good business processes, i.e., *modeling* processes, is a non-trivial task. It typically requires not only fluency in the chosen modeling language, but also intimate knowledge of the target domain and of the common practices, conventions and procedures followed by the various actors operating in the given domain. These requirements do not apply to business processes only. We find them over and over again in all those contexts that leverage on model-driven formalisms for the implementation of process-oriented systems. This is, for instance, the case of data mashups, which are commonly based on graphical data flow paradigms, such as the one proposed by Yahoo! Pipes (`http://pipes.yahoo.com`).

In order to ease the modeling of this kind of data mashups (so-called *pipes*), in a previous work, we developed an extension of Pipes that interactively recommends mashup model patterns while the developer is modeling a pipe. A

[1] The final publication is available at `link.springer.com`

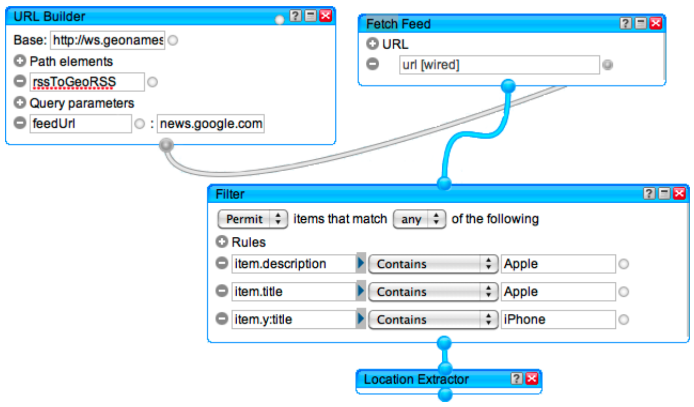


Fig. 1. A Yahoo! Pipes data mashup model pattern for plotting news on a map. The mashup logic is expressed as data flow diagram.

click on a recommended pattern automatically weaves the pattern into the partial pipe in the modeling canvas. Patterns are mined from a repository of freely accessible pipes models [11]. We mined patterns from a dataset of 997 pipes taken from the "most popular" category, assuming that popular pipes are more likely to be functioning and useful. Before their use, patterns were checked by a mashup expert assuring their meaningfulness and reusability (e.g., see Figure 1 for an example of a good pattern). The extension is called Baya [13], and our user studies demonstrate that recommending model patterns has the potential to significantly lower development times in model-driven environments [12].

The approach however suffers from problems that are common to pattern mining algorithms: identifying support threshold values, managing large numbers of produced patterns, coping with noise (useless patterns), giving meaning to patterns, and the cold start problem. Inspired by the recent advent of *crowdsourcing* [6], the intuition emerged that it might be possible to attack these problems with the help of the *crowd*, that is, by involving *humans* in the mining process. The intuition stems from the observation that pure statistical support does not always imply interestingness [2], and that human experts are anyway the ultimate responsibles for deciding about the suitability or not of patterns.

In this paper, we report on the results of this investigation and demonstrate that crowd-based pattern mining can indeed be successfully used to identify meaningful model patterns. We describe our crowd-based mining algorithm, the adopted software/crowd stack, and demonstrate the effectiveness of the approach by comparing its performance with that of the algorithm adopted in Baya. We also show how our results and lessons learned are applicable to and impact the mining of model patterns from business process models.

2 Background and Problem Statement

2.1 Reference process models: data mashups

Mashups are composite web applications that are developed by integrating data, application logic, and pieces of user interfaces [1]. ***Data mashups*** are a special type of mashups that specifically focuses on the integration and processing of data sources available on the Web. Typical *data sources* are RSS or Atom feeds, plain XML- or JSON-encoded static resources, or more complex SOAP or RESTful web services. *Data mashup tools* are IDEs for data mashup development. They provide a set of data processing *operators*, e.g., filters or split and join operators, and the possibility to interactively configure data sources and operators (we collectively call them *components*).

In this paper, we specifically focus on the data mashup tool Yahoo! Pipes and our pattern recommender Baya [13]. The components and mashups supported by these tools can be modeled as follows: Let CL be a library of ***components*** of the form $c = \langle name, IP, IF, OP, emb \rangle$, where $name$ identifies the component (e.g., RSS feed or Filter), IP is the set of input ports for data flow connectors, IF is the set of input fields for the configuration of the component, OP is the set of output ports, and $emb \in \{yes, no\}$ tells whether the component allows for the embedding of other components or not (e.g., to model loops). We distinguish three classes of components: *Source* components fetch data from the Web or collect user inputs at runtime. They don't have input ports: $IP = \emptyset$. *Data processing* components consume data in input and produce processed data in output: IP, $OP \neq \emptyset$. A *sink* component (the Pipe Output component) indicates the end of the data processing logic and publishes the output of the mashup, e.g., using JSON. The sink has neither input fields nor output ports: $IF, OP = \emptyset$.

A ***data mashup*** (a pipe) can thus be modeled as $m = \langle name, C, E, DF, VA \rangle$, where $name$ uniquely identifies the mashup, C is the set of integrated components, $E \subseteq C \times C$ represents component embeddings, $DF \subseteq (\cup_i OP_i) \times (\cup_j IP_j)$ is the set of data flow connectors propagating data from output to input ports, and $VA \subseteq (\cup_k IF_k) \times STRING$ assigns character string values to input fields. Generic strings are interpreted as constants, strings starting with "item." are used to map input data attributes to input fields (see Figure 1).

A pipe is considered *correct*, if it (i) contains at least one source component, (ii) contains a set of data processing components (the set may be empty), (iii) contains exactly one sink component, (iv) is connected (in the sense of graph connectivity), and (v) has value assignments for each mandatory input field.

A ***mashup model pattern*** can thus be seen as a tuple $mp = \langle name, desc, tag, C, E, DF, VA \rangle$, with $name$, $desc$ and tag naming, describing and tagging the pattern, and C, E, DF, VA being as defined above, however with relaxed correctness criteria: a pattern is *correct* if it (i) contains at least two components, (ii) is connected, and (iii) has value assignments for each mandatory input field.

2.2 Crowdsourcing

Crowdsourcing (CS) is the outsourcing of a unit of work to a crowd of people via an open call for contributions [6]. A ***worker*** is a member of the crowd (a human) that performs work, and a ***crowdsourcer*** is the organization, company or individual that crowdsources work. The crowdsourced work typically comes in the form of a ***crowd task***, i.e., a unit of work that requires human intelligence and that a machine cannot solve in useful time or not solve at all. Examples of crowd tasks are annotating images with tags or descriptions, translating text from one language into another, or designing a logo.

A ***crowdsourcing platform*** is an online software infrastructure that provides access to a crowd of workers and can be used by crowdsourcers to crowdsource work. Multiple CS platforms exist, which all implement a specific ***CS model***: The *marketplace* model caters for crowd tasks with fixed rewards for workers and clear acceptance criteria by the crowdsourcer. The model particularly suits micro-tasks like annotating images and is, for example, adopted by Amazon Mechanical Turk (`https://www.mturk.com`) and CrowdFlower (`http://crowdflower.com`). The *contest* model caters for tasks with fixed rewards but unclear acceptance criteria; workers compete with their solutions for the reward, and the crowdsourcer decides who wins. The model suits creative tasks like designing a logo and is, e.g., adopted by 99designs (`http://99designs.com`). The *auction* model caters for tasks with rewards to be negotiated but clear acceptance criteria. The model suits creative tasks like programming software and is, e.g., adopted by Freelancer (`http://www.freelancer.com`).

For the purpose of this paper, we specifically leverage on micro-tasks in marketplace CS platforms. Crowdsourcing a task in this context involves the following steps: The crowdsourcer publishes a *description* of the task to be performed, which the crowd can inspect and possibly express interest for. In this step, the crowdsourcer also defines the reward workers will get for performing the task and how many answers he would like to collect from the crowd. Not everybody of the crowd may, however, be eligible to perform a given task, either because the task requires specific capabilities (e.g., language skills) or because the workers should satisfy given properties (e.g., only female workers). Deciding which workers are allowed to perform a task is commonly called *pre-selection*, and it may be done either by the crowdsourcer manually or by the platform automatically (e.g., via questionnaires). Once workers are enabled to perform a task, the platform creates as many *task instances* as necessary to collect the expected number of answers. Upon completion of a task instance (or a set thereof), the crowdsourcer may inspect the collected answers and *validate* the respective correctness or quality. Work that is not of sufficient quality is not useful, and the crowdsourcer *rewards* only work that passes the possible check. Finally, the crowdsourcer may need to *integrate* collected results into an aggregated outcome of the overall CS process.

2.3 Problem statement and hypotheses

This paper aims to understand whether it is possible to crowdsource the mining of mashup model patterns of type *mp* from a dataset of mashup models $M = \{m_l\}$ with $l \in \{1...|M|\}$ and $|M|$ being "small" in terms of dataset sizes required by conventional data mining algorithms (dozens rather than thousands or millions). Specifically, the work aims to check the following hypotheses:

Hypothesis 1 (Effectiveness) *It is possible to mine reusable mashup model patterns from mashup models by crowdsourcing the identification of patterns.*

Hypothesis 2 (Value) *Model patterns identified by the crowd contain more domain knowledge than automatically mined patterns.*

Hypothesis 3 (Applicability) *Crowd-based pattern mining outperforms machine-based pattern mining for small datasets.*

It is important to note that the above hypotheses and this paper as a whole use the term "mining" with its generic meaning of "discovering knowledge," which does not necessarily require machine learning.

3 Crowd-Based Pattern Mining

The core ***assumptions*** underlying this research are that (i) we have access to a *repository* of mashup models (the dataset) of limited size, like in the case of a cold start of a modeling environment; (ii) the identification of patterns can be crowdsourced as *micro-tasks* via maketplace-based CS platforms; and (iii) the interestingness of patterns as judged *subjectively* by workers has similar value as that expressed via minimum support thresholds of automated mining algorithms.

3.1 Requirements

Crowdsourcing the mining of mashup model patterns under these assumptions asks for the fulfillment of a set of requirements:

R1: Workers must pass a *qualification test*, so as to guarantee a minimum level of familiarity with the chosen mashup modeling formalism.
R2: Mashup models m must be *represented* in a form that is easily intelligible to workers and that allows them to conveniently express patterns mp.
R3: It must be possible to input a *name*, a *description* and a list of *tags* for an identified pattern, as well as other *qualitative* feedback.
R4: To prevent *cheating* (a common problem in CS) as much as possible, all inputs must be checked for formal correctness.
R5: The crowdsourced pattern mining algorithm should make use of *redundancy* to guarantee that each mashup model is adequately taken into account.
R6: Workers must be *rewarded* for their work.
R7: Collected patterns must be *integrated and homogenized*, and repeated patterns must be merged into a set of patterns MP.
R8: Collected patterns must undergo a *quality check*, so as to assure the reusability and meaningfulness of identified patterns.

Given a set of crowd-mined patterns MP, accepting or rejecting our hypotheses then further requires comparing the quality of MP with that of patterns that are mined automatically (we use for this purpose our algorithm described in [11]).

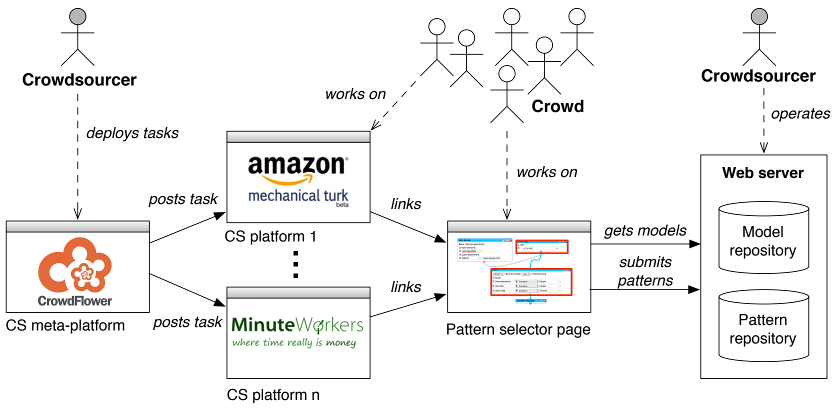


Fig. 2. Approach to crowd-based pattern mining with CrowdFower

3.2 Approach

Figure 2 provides an overview of our approach to crowdsource the mining of mashup model patterns using CrowdFlower (`http://www.crowdflower.com`) as the crowdsourcing platform. Starting from the left-hand side of the figure, the *crowdsourcer* deploys the task on CrowdFlower. Doing this requires the creation of the forms to collect data from the crowd, the uploading of the dataset that contains the units of work (i.e., the mashup models), the preparation of the qualification tests (***R1***), among other tasks that are specific to CrowdFlower. Once the tasks are deployed, CrowdFlower posts them to third-party platforms such as Amazon Mechanical Turk and MinuteWorkers where the *crowd* can actually perform the requested work. Each mashup model is configured to be shown to at least three workers, in order to guarantee that each model gets properly inspected (***R5***), and a monetary reward is set for each pattern provided by the crowd (***R6***). We will discuss more about this last aspect in Section 4.

Each task points to an external *Pattern Selector* page where the crowd can select patterns from the mashups in the dataset. The Pattern Selector page consists in a standard web application implemented in Java, HTML, CSS and Javascript, which displays the image of a pipe in its original representation (screen shot) and allows the worker to define patterns on top (***R2***). In addition, the worker can provide a name, a description and a list of tags that describe the pattern (***R3***). All inputs provided by the worker are validated, e.g., to check that the worker indeed selects a pattern within the mashup (***R4***).

The web application for the Pattern Selector page is hosted on a *web server* operated by the crowdsourcer. The web server hosts a *model repository* where the mashup models are stored and from where the Pattern Selector page gets the models. It also hosts a *pattern repository* where the patterns selected by the crowd are submitted and stored for further analysis, which includes the filtering, validation and integration of the collected patterns (***R7*** and ***R8***)

Algorithm 1: Crowd

```
Data: input dataset $IN$, pattern mining tasks $PMT$, data partitioning strategy $DPE$, data
      partition size $DPS$, answers per partition $APP$, per-task reward $rw$, time alloted $ta$
Result: set $MP$ of mashup model patterns $\langle name, desc, tag, C, E, DF, VA \rangle$
1 $\overline{IN}$ = initializeDataset($IN$);
2 $DP$ = partitionInputDataset($\overline{IN}, DPE, DPS$);
3 $T$ = mapDataPartitionsToTasks($DP$, $PMT$, $APP$);
4 distributeTasksToCrowd($T, rw, ta$);
5 $rawPatterns$ = collectPatternsFromCrowd();
6 $MP$ = filterResults($rawPatterns$);
7 return $MP$;
```

3.3 Algorithm

Algorithm 1 (we call it the *Crowd* algorithm) illustrates a *generic* algorithm that brings together human and machine computation for the mining of patterns from mashup models. The algorithm receives as input:

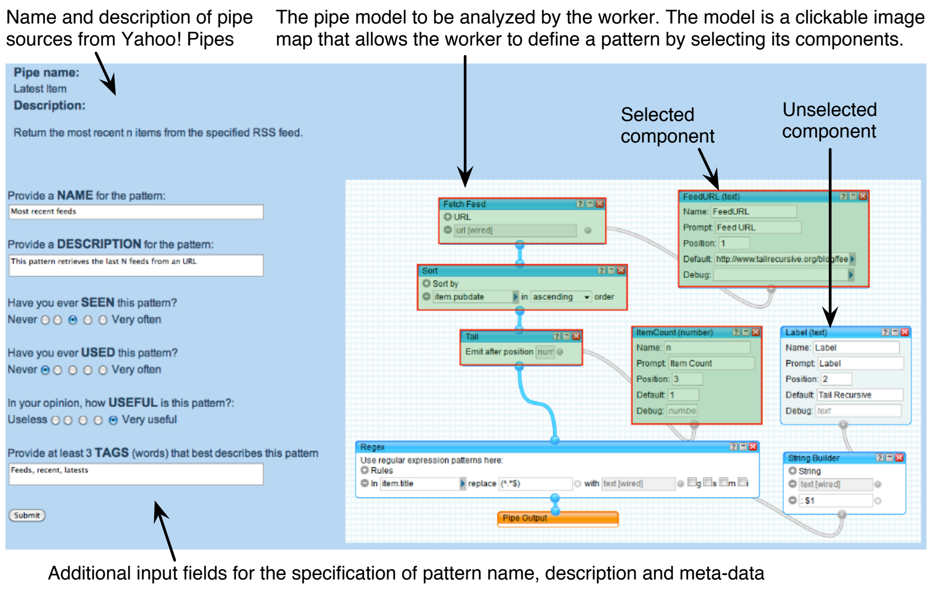
- The dataset IN of *mashup models* from which to identify patterns,
- The design of the parameterized *pattern mining task* PMT to be executed by the crowd (the parameters tell which mashup model(s) to work on),
- The *data partitioning strategy* DPE telling how to split IN into sub-sets to be fed as input to PMT,
- The *data partition size* DPS specifying the expected size of the input datasets,
- The *number of answers* APP to be collected per data partition,
- The *per-task reward* rw to be paid to workers, and
- The *time allotted* ta to execute a task.

The algorithm consists of seven main steps. First, it initializes the input dataset IN (line 1) and transforms it into a format that is suitable for the crowdsourcing of pattern mining tasks (we discuss this step in the next subsection). Then, it takes the initialized input dataset $\overline{IN}$ and partitions it according to parameters DPE and DPS (line 2). In our case, DPE uses a random selection of mashup models, $DPS = 1$ and $APP = 3$. Next, the algorithm maps the created partitions of mashup models to the tasks PMT (line 3) and distributes the tasks to the workers of the crowd (line 4). Once tasks are deployed on the crowdsourcing platform, it starts collecting results from the crowd until the expected number of answers per model are obtained or the allotted time of the task expires (line 5). Finally, it filters the patterns according to predefined quality criteria (line 6), so as to keep only patterns of sufficient quality (we provide more details of this last step in Section 4).

Note that the core of the approach, i.e., the identification of patterns, is not performed by the algorithm itself but delegated to the crowd as described next.

3.4 Task design

In order to make sure that only people that are also knowledgeable in Yahoo! Pipes perform our tasks, we include a set of five multiple choice, pre-selection


Name and description of pipe sources from Yahoo! Pipes
The pipe model to be analyzed by the worker. The model is a clickable image map that allows the worker to define a pattern by selecting its components.
Selected component
Unselected component
Additional input fields for the specification of pattern name, description and meta-data
Submit
Pipe Output

Pipe name:
Latest Item
Description:
Return the most recent n items from the specified RSS feed.

Provide a **NAME** for the pattern:
Most recent feeds

Provide a **DESCRIPTION** for the pattern:
This pattern retrieves the last N feeds from an URL

Have you ever **SEEN** this pattern?
Never ○ ○ ● ○ ○ Very often

Have you ever **USED** this pattern?
Never ● ○ ○ ○ ○ Very often

In your opinion, how **USEFUL** is this pattern?:
Useless ○ ○ ○ ○ ● Very useful

Provide at least 3 **TAGS** (words) that best describes this pattern
Feeds, recent, latests

Fetch Feed
URL
url [wired]

Sort
Sort by
item.pubdate in ascending order

Tail
Emit after position num

FeedURL (text)
Name: FeedURL
Prompt: Feed URL
Position: 1
Default: http://www.tailrecursive.org/blog/fee
Debug:

ItemCount (number)
Name: n
Prompt: Item Count
Position: 3
Default: 1
Debug: numbe

Label (text)
Name: Label
Prompt: Label
Position: 2
Default: Tail Recursive
Debug: text

Regex
Use regular expression patterns here:
Rules
In item.title replace (^.*$) with text [wired] g s m i

String Builder
String
text [wired]
: $1

Fig. 3. Task design for the selection, description and rating of mashup model patterns.

questions such as *"Which of the following components can be embedded into a loop?"* and *"What is the maximum number of Pipe Output modules permitted in each pipe?"* In order for a worker to be paid, he/she must correctly answer these questions, for which we already know the answers (so-called *gold data*).

Another core decision when crowdsourcing a task is the UI used to interact with workers. In general, all crowdsourcing platforms available today allow a crowdsourcer to design form-based user interfaces directly inside the crowdsourcing platform. For the crowdsourcing of simple tasks, such as the annotation of images or the translation of a piece of text, this is sufficient to collect useful feedback. In more complex crowdsourcing tasks, such as our problem of identifying patterns inside mashup models, textual, form-based UIs are not enough and a dedicated, purposefully designed graphical UI is needed. The task that workers can perform through this UI corresponds to the implementation of the $collectPatternsFromCrowd()$ function in Algorithm 1, i.e., the actual mining.

In order to make workers feel comfortable with the selection of patterns inside pipes models, we wanted the representation of the pipes to be as close as possible to what real pipes look like. In other words, we did not want to create an abstract or simplified representation of pipes models (e.g., a graph or textual description) and, instead, wanted to keep the full and realistic expressive power of the original representation. We therefore decided to work with screen shots of real pipes models, on top of which workers are able to select components of the pipe and to construct patterns by simply clicking on the respective components. Figure 3 shows a screen shot of the resulting GUI for selecting patterns, in which we show a pipe with a total of 9 components, of which 5 have been selected by the worker to form a pattern (see the green-shaded components).

As shown in the figure, next to selecting a pattern, the worker must also provide information about the pattern such as the *name*, *description* and list of *tags* (at least 3 tags). In addition, the worker may also rank the pattern regarding to how often he/she has already *seen* or *used* the pattern before, and to how *useful* he/she thinks the pattern is.

4 Evaluation and Comparison

To study the described *Crowd* algorithm, we performed a set of experiments with CrowdFlower and compared the results with those obtained by running our original automated pattern mining algorithm [11] with different minimum support levels and dataset sizes. We refer to this latter as to the *Machine* algorithm.

4.1 Evaluation metrics

While for automated mining it is clear by design how the output of an algorithm looks like, this is not as clear if the identification of patterns is delegated to the crowd. As described earlier, it is very common that workers cheat and, hence, do not provide meaningful data. To filter out those patterns that we can instead reasonably trust, we define a set of minimum criteria for crowd-mined patterns: a ***good mashup pattern*** is a pattern that consists of *at least two modules* and where the modules are *connected*, the name and description of the pattern are *not empty*, and the description and the actual pattern structure *match semantically.* The first three criteria we enforce automatically in the pattern identification UI illustrated in Figure 3. Whether the description and pattern structure match semantically, i.e., whether the description really tells what the pattern does, is assessed manually by experts (us). The result of this analysis is a Boolean: either a pattern is considered *good* (and it passes the filter) or it is considered *bad* (and it fails the filter). Note that with "good" we do not yet assert anything about the actual value of a pattern; this can only be assessed with modelers using the pattern in practice. The same expert-based filter is usually also applied to the outputs of automated mining algorithms and does not introduce an additional subjective bias compared to automated mining scenarios.

In order to compare the performance of the two algorithms and test our hypotheses, we use three metrics to compare the sets of patterns they produce in output: the ***number of patterns found*** gives an indication of the effectiveness of the algorithms in finding patterns; the ***average pattern size***, computed as the average number of components of the patterns in the respective output sets, serves as an indicator of how complex and informative identified patterns are; and the ***distribution of pattern sizes*** shows how diverse the identified patterns are in terms of complexity and information load.

4.2 Experiment design and dataset

The ***Crowd algorithm*** is implemented as outlined in Algorithm 1 using the popular CS platform CrowdFlower. Running the algorithm is a joint manual and automated effort: our Pattern Selector application takes care of initializing the dataset (the pipes models), partition it, and map partitions to tasks at runtime. The actual tasks are deployed manually on CrowdFlower and executed by the crowd. Filtering out good patterns is again done manually. For each pipe, we request at least 3 judgments, estimated a maximum of 300 sec. per task, and rewarded USD 0.10 per task.

The ***Machine algorithm*** is based on a frequent sub-graph mining algorithm described in [11] and implemented in Java. The core parameter used to fine-tune the algorithm is the *minimum support* that the mined sub-graphs must satisfy; we therefore use this variable to test and report on different test settings.

The ***dataset*** used to feed both algorithms consists of 997 pipes (with 11.1 components and 11.0 connectors on average) randomly selected from the "most popular" pipes category of Yahoo! Pipes' online repository. We opted for this category because, being popular, the pipes contained there are more likely to be functional and useful. The pipes are represented in JSON. The dataset used for the *Crowd* algorithm consists in a selection of 40 pipes out of the 997 (which represents a small dataset in conventional data mining). The selection of these pipes was performed manually, in order to assure that the selected pipes are indeed runnable and meaningful. In addition to the JSON representation of these 40 pipes, we also collected the screen shots of each pipe through the Yahoo! Pipes editor. The JSON representation is used in the automated input validators; the screen shots are used to collect patterns from the crowd as explained earlier.

For our comparison, we run $Machine$ with datasets of 997 (big dataset) and 40 pipes (small dataset). We use $Machine^{997}$ and $Machine^{40}$ to refer to the former and the latter setting, respectively. We run $Crowd$ only with 40 pipes and, for consistency, refer to this setting as to $Crowd^{40}$.

4.3 Results and interpretation

Figure 4 summarizes the task instances created and the patterns collected by running $Crowd^{40}$. The crowd started a total of 326 task instances in CrowdFlower, while it submitted only 174 patterns through our Pattern Selector application. This means that a total of 152 task instances were abandoned without completion. Out of the 174 patterns submitted, only 42 patterns satisfied our criteria for *good mashup patterns*. These data testify a significant level of noise produced by workers who, in the aim of finishing tasks as quickly as possible and getting paid, apparently selected random fragments of pipes and provided meaningless descriptions. The cost of this run was USD 17.56, including administrative costs.

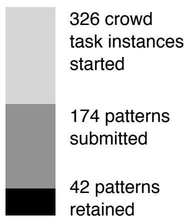
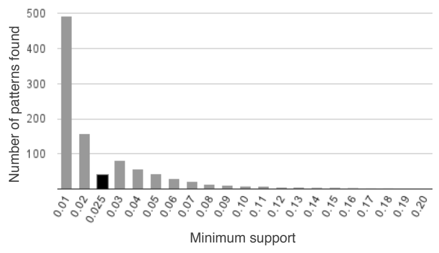
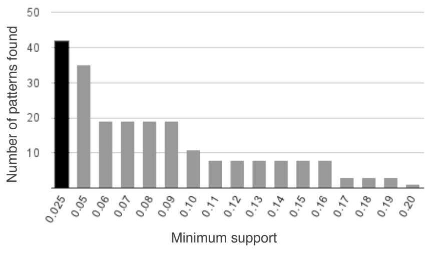


Fig. 4. Task instances and patterns in $Crowd^{40}$.

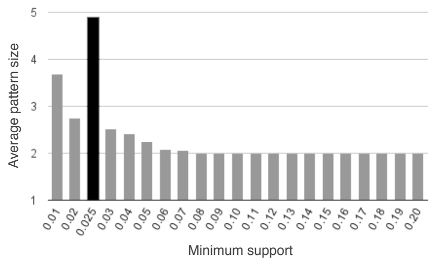


(a) *Machine*^{997}(gray) compared to *Crowd*^{40} *(black)*

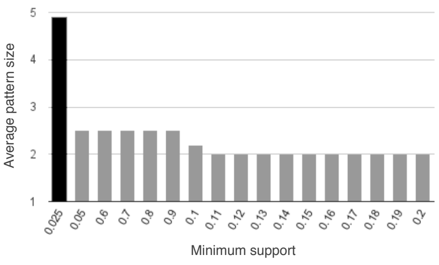


(b) *Machine*^{40} (gray) compared to *Crowd*^{40} *(black)*

Fig. 5. Number of patterns produced by the two automated mining algorithms under varying minimum support levels. For comparison, the charts also report the number of pattern produced by the crowd-based mining algorithm (in black).



(a) *Machine*^{997}(gray) compared to *Crowd*^{40} *(black)*



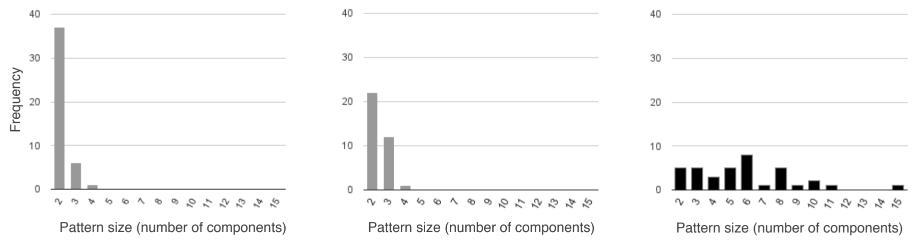
(b) *Machine*^{40} (gray) compared to *Crowd*^{40} *(black)*

Fig. 6. Average size of the patterns produced by the two automated mining algorithms under varying minimum support levels. For comparison, the charts also report the average size of patterns produced by the crowd-based mining algorithm (in black).

The charts in Figures 5–7 report on the numbers of patterns, average pattern sizes and the distribution of pattern sizes obtained by running $Machine^{997}$ and $Machine^{40}$ with different minimum relative support levels sup_{min}. The bars in gray are the results of the $Machine$ algorithm; the black bars represent the results of $Crowd^{40}$. For comparison, we placed $Crowd^{40}$ at a support level of $sup_{min} = 0.025$, which corresponds to $1/40 = 0.025$, in that we ask workers to identify patterns from a single pipe without the need for any additional support.

H1 (Effectiveness). Figure 5(a) illustrates the number of patterns found by $Machine^{997}$. The number quickly increases for $Machine^{997}$ as we go from high support values to low values, reaching almost 500 patterns with $sup_{min} = 0.01$. Figure 5(b) shows the results obtained with $Machine^{40}$. The lowest support value for $Machine^{40}$ is $sup_{min} = 0.05$, which corresponds to an absolute support of 2 in the dataset. It is important to note that only very low support values produce a useful number of patterns. In both figures, the black bar represents the 42 patterns identified by $Crowd^{40}$.

The two figures show the typical problem of automated pattern mining algorithms: only few patterns for high support levels (which are needed, as support is the only criterion expressing significance), too low support levels required to



(a) *Machine*^{997} with 44 patterns in output and support of 0.05
(b) *Machine*^{40} with 35 patterns in output and support of 0.05
(c) *Crowd*^{40} *with 42 patterns*

Fig. 7. Size distribution of the patterns by the three algorithms. To ease the comparison, the histograms of $Machine^{997}$ and $Machine^{40}$ refer to the run with the minimum support level that produced an output dataset close to the one produced by $Crowd^{40}$.

produce useful output sizes with small datasets (our goal), and an explosion of the output size with large datasets. As illustrated in Figure 4, $Crowd^{40}$ is instead able to produce a number of patterns in output that is similar to the size of the dataset in input. Notice also that, while Figure 5 reports on *all* the patterns found by $Machine$, the data for $Crowd^{40}$ include only *good patterns.* This means that not only $Crowd^{40}$ is able to find patterns, but it is also able to find practically meaningful patterns. We thus *accept* Hypothesis 1 and conclude that *with* $Crowd^{40}$ *it is possible to mine reusable mashup model patterns by crowdsourcing the identification of patterns.*

H2 (Value). Figure 6 shows the average pattern sizes of $Machine^{997}$ and $Machine^{40}$ compared to that of $Crowd^{40}$. In both settings, the average pattern size obtained with $Crowd^{40}$ clearly exceeds the one that can be achieved with $Machine$, even for very low support values (0.01). With Figure 7, we look more specifically into how these patterns look like by comparing those runs of $Machine^{997}$ and $Machine^{40}$ with $Crowd^{40}$ that produce a similar number of patterns in output. In both settings this happens for $sup_{min} = 0.05$ and produced 44 and 35 patterns, respectively. Figures 7(a) and (b) show that automatically mined patterns are generally small (sizes range from 2–4), with a strong prevalence of the most simple and naïve patterns (size 2).

Figure 7(c), instead, shows that the results obtained with $Crowd^{40}$ present a much higher diversity in the pattern sizes, with a more homogeneous distribution and even very complex patterns of sizes that go up to 11 and 15 components. $Crowd^{40}$ is thus able to collect patterns that contain more complex logics and that are more informative, and thus, possibly contain more domain knowledge. These patterns also come with a characterizing name, description and list of tags. These annotations not only enrich the value of a pattern with semantics but also augment the domain knowledge encoded in the pattern and its reusability. We thus *accept* Hypothesis 2 and conclude that *patterns mined with* $Crowd^{40}$ *contain more domain knowledge than the automatically mined patterns.*

H3 (Applicability). The above assessment of the *effectiveness* and *value* of $Crowd^{40}$ shows that *crowd-based pattern mining outperforms machine-based min-*

ing for small datasets, that is, we *accept* Hypothesis 3. For large datasets, automated mining still represents a viable solution, but for small datasets crowd-based ming is not only applicable but also more effective. With a cost per pattern of USD 0.42 and a running time of approximately 6 hours, crowd-based mining proves to be a very competitive alternative to hiring a domain expert, which would be the alternative to attack a cold start in our small dataset scenario.

5 Discussion and Analogy with BPM

Regarding the above results, we performed a set of additional experiments to analyze the ***robustness*** of $Crowd^{40}$ along two dimensions: reward and task design. We did not notice any reduction of the number of tasks instantiated by the crowd or the number of patterns collected if we lowered the *reward* from USD 0.10 down to USD 0.05. We essentially got the same response as described in Figure 4, which indicates that we could have gotten the same results also for less money without any loss of quality. Instead, we noticed that there is a very strong sensitivity regarding the *task design*, but only on the number of patterns that can be collected, not on the number of task instances. Concretely, we tried to introduce a minimum level of support (at least 2 times in 3, respectively, 10 pipes shown to the worker). The result was only a strong reduction of the number of patterns submitted. The lesson learned is thus to keep the task as simple as possible, that is, to apply the KISS (Keep It Simple, Stupid) principle, and to concentrate the effort instead on the validation of collected data.

There are two key aspects when designing a CS task: input validation and intuitiveness. We have seen that it is strongly advised to check all inputs for ***formal validity*** (e.g., no empty strings), otherwise workers may just skip inputs or input fake content (e.g., a white space). As for the ***intuitiveness***, we considered collecting patterns via textual input (e.g., the list of component names in the pattern) or via abstract data flow graphs (automatically constructed from the JSON representation of pipes), but in the end we opted for the screen shots. This has proven to be the representation workers are most familiar with; in fact, screen shots do not introduce any additional abstraction.

In order to filter out workers that had some minimum knowledge of Pipes, we performed a ***pre-selection*** in the form of *gold data*. Yet, our questions were too tough in our first tests, and we had to lower our expectations. Interestingly, however, this did not affect much the quality of the patterns (but workers that did not pass the test, did not get paid). We also noticed a *natural selection* phenomenon: the majority of patterns was submitted by only few workers. We assume these were workers with good knowledge in Pipes that simply liked this kind of modeling tasks and, therefore, submitted patterns not only for the sake of the reward but also for personal satisfaction. We believe that, with the right quality criteria in place, the pre-selection could be omitted, and the "experts" (and good patterns) emerge automatically, at the cost/speed of simple CS tasks.

As for the ***quality of patterns***, we carefully analyzed each of the 42 patterns identified by the crowd and conclude with confidence that *all* patterns that satisfy our criteria for *good* patterns are indeed meaningful. Particularly important in this respect are the additional annotations (name, description, tags) that equip the patterns with semantics. It is important to notice that assessing the quality of patterns is non-trivial in general and that the annotations do not only allow one to grasp better the meaning and purpose of patterns; they also allow one to tell serious workers and cheaters apart, which increases quality.

In this paper, we specifically focus on *mashup model patterns*, as we use them to provide interactive recommendations in Baya. Yet, the approach and findings are general enough to be applicable almost straightway also to ***business process models***. A *business process* (BP) is commonly modeled as $P = \langle N, E, type \rangle$, with N being the set of nodes (events, gateways, activities), E being the set of control flow connectors, and *type* assigning control flow constructs to gateway nodes. Our definition of *mashups* is not dissimilar: $m = \langle name, C, E, DF, VA \rangle$. The components C correspond to N, and the data flow connectors DF correspond to E. These are the constructs that most characterize a pattern. In fact, our task design requires workers only to mark components to identify a pattern (connectors, embeddings and value assignments are included automatically). If applied to BP models, this is equivalent to ask workers to mark tasks.

Our mashup model is further *data flow* based, while BP models are typically *control flow* based (e.g., BPMN or YAWL) and contain control flow constructs (gateways). If identifying patterns with the crowd, the question is whether gateways should be marked explicitly, or whether they are included automatically (as in the current task design). In our case, for a set of components to form a pattern it is enough that they are connected. In the case of BP patterns, this may no longer be enough. Commonly, BP fragments are considered most reusable if they are well structured, i.e., if they have a single entry and a single exit point (SESE). It could thus be sensible to allow workers to select only valid SESE fragments, although this is not a strict requirement.

As for the comparison of $Crowd^{40}$ with $Machine^{997}$ and $Machine^{40}$, it is important to note that the automated mining algorithm would very likely produce *worse* results if used with BP models. In fact, mashup models are particularly suited to automated mining: the components they use are selected from a predefined, limited set of component *types* (e.g., several dozens). Similarities can thus be identified relatively easily, which increases the support of patterns. BP models, instead, are more flexible in their "components" (the tasks): task labels are *free text*, and identifying "types" of tasks is a hard problem in itself [7]. For instance, the tasks "Pay bill", "Pay" and "Send money" can all be seen as instances of a common task type "Payment." This, in turn, means that the value of $Crowd^{40}$ could be even more evident when mining BP models.

6 Related Work

Crowdsourcing has been applied so far in a variety of related areas. In the specific context of *machine learning*, Sheng et al. [14] collect training labels for data items from the crowd to feed supervised induction algorithms. In the same context, von Ahn et al. [17] propose an interactive *game* that requires multiple players to agree on labels for images, enhancing the quality of labels. In [10], Sheng et al. propose *CrowdMine*, a game that leverages on the crowd to identify graphical patterns used to verify and debug software specifications.

In the context of BPM, the term "patterns" is commonly associated with the ***workflow patterns*** by van der Aalst et al. [16]. Initially, the focus of these patterns was on control flow structures, but then the idea evolved and included all the aspects (control flow, data flow, resources, exception handling) that characterize workflow languages (`http://www.workflowpatterns.com`). The proposed patterns are an analytical approach to assess the strengths and weaknesses of workflow languages, more than an instrument to assist developers while modeling, although Gschwind et al. [4] also explored this idea.

The *automated identification* of process models or fragments thereof is commonly approached via *process mining*, more specifically ***process discovery*** [15]. Process discovery aims to derive model patterns from *event logs*, differently from the problem we address in this paper, which aims to find patterns in a set of *process models*. The main assumptions of process discovery techniques are: (i) each process instance can be identified as pertaining to a process, (ii) each event in the log can be identified as pertaining to a process instance, (iii) each event in the log corresponds to an activity in the process, (iv) each event in the log contains the necessary information to determine precedence relationships. Derived process models thus represent patterns of the dynamics of a single process and generally do not have cross-process validity. Examples of process discovery algorithms are the α-algorithm [15], Heuristic miner [18], and Fuzzy mining [5].

Only few works focus on ***mining patterns*** from process models. Lau et al. [8] propose to use frequent sub-graph and association rules discovery algorithms to discover frequent sub-graphs (patterns) to provide modeling recommendations. Li et. al [9] mine process model variants created from a given reference process model, in order to identify a new, generic reference model that covers and represents all these variants better. The approach uses a heuristic search algorithm that minimizes the average distance (in terms of change operations on models) between the model and its variants. Greco et al. [3] mine workflow models (represented as state charts) using two graph mining algorithms, *c-find* and *w-find*, which specifically deal with the structure of workflow models.

We would have liked to compare the performance of our $Crowd$ algorithm also with that of the above algorithms, yet this would have required either adapting them to our mashup model or adapting $Crowd$ to process models. We were not able to do this in time. However, the works by Lau et al. [8] and Greco et al. [3] are very close to our $Machine$ algorithm: they share the same underlying frequent sub-graph mining technique. We therefore expect a very similar performance. The two algorithms also advocate the use of a support-based notion of patterns and thus present the same problems as the one studied in Section 4.

7 Conclusion

Mining model patterns from a dataset of mashup or process models is a hard task. In this paper, we presented a crowd-based pattern mining approach that advances the state of the art with three contributions: we demonstrate that *it is possible to crowdsource a task as complex as the mining of model patterns*, that *patterns identified by the crowd are rich of domain knowledge*, and that *crowd-based mining particularly excels with small datasets*. We further explained how the *Crowd* algorithm can be adapted to mine patterns from BP models. To the best of our knowledge, this is the first investigation in this direction.

In our future work, we would like to study how crowdsourcing can be leveraged on for big datasets, e.g., by using pattern similarity metrics and the notion of support, and how the quality of patterns on the reward given. We also intend to adapt the *Crowd* algorithm to BPMN, to compare it with other BPMN-oriented approaches in literature [8, 3], and to study if the crowd can also be used for quality assessment (to automate the complete pattern mining process).

References

1. F. Daniel and M. Matera. *Mashups: Concepts, Models and Architectures*. Springer, 2014.
2. L. Geng and H. Hamilton. Interestingness measures for data mining: A survey. *ACM Computing Surveys*, 38(3):9, 2006.
3. G. Greco, A. Guzzo, G. Manco, and D. Sacca. Mining and reasoning on workflows. *Knowledge and Data Engineering, IEEE Transactions on*, 17(4):519–534, 2005.
4. T. Gschwind, J. Koehler, and J. Wong. Applying Patterns During Business Process Modeling. In *BPM*, pages 4–19. Springer, 2008.
5. C. W. Günther and W. M. Van Der Aalst. Fuzzy mining–adaptive process simplification based on multi-perspective metrics. In *BPM*, pages 328–343. 2007.
6. J. Howe. *Crowdsourcing: Why the Power of the Crowd Is Driving the Future of Business*. Crown Publishing Group, New York, NY, USA, 1 edition, 2008.
7. C. Klinkmüller, I. Weber, J. Mendling, H. Leopold, and A. Ludwig. Increasing Recall of Process Model Matching by Improved Activity Label Matching. In *BPM*, pages 211–218, 2013.
8. J. M. Lau, C. Iochpe, L. Thom, and M. Reichert. Discovery and analysis of activity pattern cooccurrences in business process models. In *ICEIS*, 2009.
9. C. Li, M. Reichert, and A. Wombacher. Discovering reference models by mining process variants using a heuristic approach. In *BPM*, pages 344–362. 2009.
10. W. Li, S. A. Seshia, and S. Jha. CrowdMine: towards crowdsourced human-assisted verification. In *DAC*, pages 1250–1251. IEEE, 2012.
11. C. Rodríguez, S. R. Chowdhury, F. Daniel, H. R. M. Nezhad, and F. Casati. Assisted Mashup Development: On the Discovery and Recommendation of Mashup Composition Knowledge. In *Web Services Foundations*, pages 683–708. 2014.
12. S. Roy Chowdhury, F. Daniel, and F. Casati. Recommendation and Weaving of Reusable Mashup Model Patterns for Assisted Development. *ACM Trans. Internet Techn.*, 2014 (in print).
13. S. Roy Chowdhury, C. Rodríguez, F. Daniel, and F. Casati. Baya: assisted mashup development as a service. In *WWW Companion*, pages 409–412. ACM, 2012.

14. V. S. Sheng, F. Provost, and P. G. Ipeirotis. Get another label? improving data quality and data mining using multiple, noisy labelers. In *SIGKDD*, pages 614–622. ACM, 2008.
15. W. Van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *Knowledge and Data Engineering, IEEE Transactions on*, 16(9):1128–1142, 2004.
16. W. M. P. 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.
17. L. Von Ahn and L. Dabbish. Labeling images with a computer game. In *SIGCHI*, pages 319–326. ACM, 2004.
18. A. Weijters, W. M. van der Aalst, and A. A. De Medeiros. Process mining with the heuristics miner-algorithm. *TU Eindhoven, Tech. Rep. WP*, 166, 2006.