Pattern-Based Specification of Crowdsourcing Applications

Alessandro Bozzon[1], Marco Brambilla[2], Stefano Ceri[2],
Andrea Mauri[2], and Riccardo Volonterio[2]

[1] Software and Computer Technologies Department, Delft University of Technology,
Postbus 5 2600 AA, Delft, The Netherlands
a.bozzon@tudelft.nl
[2] Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB)
Politecnico di Milano, Piazza Leonardo da Vinci, 32, 20133 Milano, Italy
{name.surmame}@polimi.it

Abstract. In many crowd-based applications, the interaction with performers is decomposed in several tasks that, collectively, produce the desired results. Tasks interactions give rise to arbitrarily complex workflows. In this paper we propose methods and tools for designing crowd-based workflows as interacting tasks. We describe the modelling concepts that are useful in such framework, including typical workflow patterns, whose function is to decompose a cognitively complex task into simple interacting tasks so that the complex task is co-operatively solved.

We then discuss how workflows and patterns are managed by Crowd-Searcher, a system for designing, deploying and monitoring applications on top of crowd-based systems, including social networks and crowdsourcing platforms. Tasks performed by humans consist of simple operations which apply to homogeneous objects; the complexity of aggregating and interpreting task results is embodied within the framework. We show our approach at work on a validation scenario and we report quantitative findings, which highlight the effect of workflow design on the final results.

1 Introduction

Crowd-based applications are becoming more and more widespread; their common aspect is that they deal with solving a problem by involving a vast set of performers, who are typically extracted from a wide population (the "crowd"). In many cases, the problem is expressed in the form of simple questions, and the performers provide a set of answers; a software system is in charge of organising a crowd-based computation – typically by distributing questions, collecting responses and feedbacks, and organising them as a well-structured result of the original problem.

Crowdsourcing systems, such as Amazon Mechanical Turk (AMT), are natural environments for deploying such applications, since they support the assignment to humans of simple and repeated tasks, such as translation, proofing,

S. Casteleyn, G. Rossi, and M. Winckler (Eds.): ICWE 2014, LNCS 8541, pp. 218–235, 2014.

content tagging and items classification, by combining human contribution and automatic analysis of results [1]. But a recent trend (emerging, e.g., during the CrowdDB Workshop[1]), is to use many other kinds of platforms for engaging crowds, such as proprietary community-building systems (e.g., FourSquare or Yelp) or general-purpose social networks (e.g., Facebook or Twitter). In the various platforms, crowds take part to social computations both for monetary rewards and for non-monetary motivations, such as public recognition, fun, or genuine will of sharing knowledge.

In previous work, we presented CrowdSearcher [2, 3], offering a conceptual framework, a specification paradigm and a reactive execution control environment for designing, deploying, and monitoring applications on top of crowd-based systems, including social networks and crowdsourcing platforms. In Crowdsearcher, we advocate a top-down approach to application design which is independent on the particular crowd-based system. We adopt an abstract model of crowdsourcing activities in terms of elementary task types (such as: labelling, liking, sorting, classifying, grouping) performed upon a data set, and then we define a crowdsourcing task as an arbitrary composition of these task types; this model does not introduce limitations, as arbitrary crowdsourcing tasks can always be defined by aggregating several operation types or by decomposing the tasks into smaller granularity tasks, each one of the suitable elementary type. In general, an application cannot be submitted to the crowd in its initial formulation; transformations are required to organise and simplify the initial problem, by structuring it into a *workflow of crowd-based tasks* that can be effectively performed by individuals, and can be submitted and executed, possibly in parallel. Several works [4, 5] have analysed typical *crowdsourcing patterns*, i.e. typical cooperative schemes used for organising crowd-based applications.

The goal of this paper is to present a systematic approach to the design and deployment of crowd-based applications as arbitrarily complex workflows of elementary tasks, which emphasises the use of crowdsourcing patterns. While our previous work was addressing the design and deployment of a single task, in this paper we model and deploy applications consisting of arbitrarily complex task interactions, organised as a workflow; we use either *data streams* or *data batches* for data exchange between tasks, and illustrate that tasks can be controlled through *tight coupling* or *loose coupling*. We also show that our model supports the known crowd management patterns, and in particular we use our model as a unifying framework for a systematic classification of patterns.

The paper is structured as follows. Section 2 presents related work; Section 3 introduces the task and workflow models and design processes. Section 4 details a set of relevant crowdsourcing patterns. Section 5 illustrates how workflow specifications are embodied within the execution control structures of Crowdsearcher, and finally Section 6.3 discusses several experiments, showing how differences in workflow design lead to different application results.

[1] http://dbweb.enst.fr/events/dbcrowd2013/

2 Related Work

Many crowdsourcing startups[2] and systems [6] have been proposed in the last years. Crowd programming approaches rely on imperative programming models to specify the interaction with crowdsourcing services (e.g., see *Turkit* [7], *RABJ* [8], *Jabberwocky* [9]). Several programmatic methods for human computation have been proposed [7][8][9][10], but they do not support yet the complexity required by real-world, enterprise–scale applications, especially in terms of designing and controlling complex flows of crowd activities.

Due to its flexibility and extensibility, our approach covers the expressive power exhibited by any of the cited systems, and provides fine grained targeting to desired application behaviour, performer profiles, and adaptive control over the executions.

Several works studied how to involve humans in the creation and execution of workflows, and how to codify common into modular and reusable patterns. Process-centric workflow languages [11] define business artefacts, their transformations, and interdependencies trough tasks and their dependencies. Scientists and practitioners put a lot of effort in defining a rich set of control-driven workflow patterns.[3] However, this class of process specification languages: focus mainly on control flow, often abstracting away data almost entirely; disregard the functional and non-functional properties of the involved resources; do not specify intra- and inter-task execution and performer controls; and provide no explicit modelling primitives for data processing operations.

In contrast, data- driven workflows have recently become very popular, typically in domains where database are central to processes [12][13], and data consistency and soundness is a strong requirement. Data-driven workflows treat data as first-class citizens, emphasise the role of control intra- and inter-task control, and ultimately served as an inspiration for our work.

Very few works studied workflow-driven approaches for crowd work. CrowdLang [5] is notable exception, which supports process-driven workflow design and execution of tasks involving human activities, and provides an executable model-based programming language for crowd and machines. The language, however, focuses on the modelling of coordination mechanisms and group decision processes, and it is oblivious to the design and specification of task-specific aspects.

Several works tried to codify patterns for crowdsourcing. At task level, a great wealth of approaches has been proposed for the problems of output agreement [14], and performer control [15]. At workflow level, less variety can be witnessed, but a set of very consolidated patterns emerge [7][16][4][17][18]. In Section 4 we will provide an extensive review of the most adopted pattern in crowdsourcing, classifying them in the light of the workflow model of Section 3.

[2] E.g., CrowdFlower, Microtask, uTest.

[3] `http://workflowpatterns.com/`

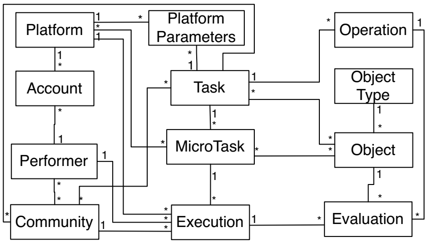


Fig. 1. Metamodel of task properties

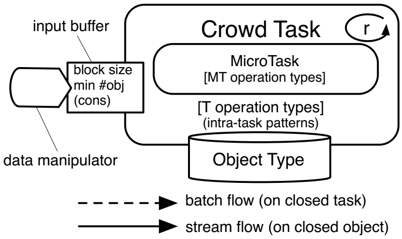


Fig. 2. Task notation

3 Models and Design of Crowd-Based Workflows

Although humans are capable of solving complex tasks by using their full cognitive capacity, the approaches used in crowd-based computations prefer to decompose complex tasks into simpler tasks and then elaborate their results [16]. Following this approach, we restrict crowdsourcing tasks be to simple operations which apply to homogeneous objects; operations are simple actions (e.g. labelling, liking, sorting, classifying, grouping, adding), while objects have an arbitrary schema and are assumed to be either available to the application or to be produced as effect of application execution.

3.1 Task Model

Tasks of a crowd-based application are described in terms of an abstract model, that was initially presented in [2], and represented in Fig. 1. We assume that each **task** receives as input a list of **objects** (e.g., photos, texts, but also arbitrarily complex objects, all conforming to the same **object type**) and asks performers to do one or more **operations** upon them, which belong to a predefined set of abstract **operation types**. Examples of operation types are *Like*, for assigning a preference to an item; or *Classify*, for assigning each item to one or more classes. The full list of currently supported operation types is reported in [2]. Task management requires specific sets of objects to be assembled into a unit of execution, called **micro-task**, that is associated with a given **performer**. Each micro-task can be invited or executed on different **platforms** and/or **communities**. The relation with platform is specified through a series of **platform parameters**, specific for each platform, that are needed in order retrieve the answers of the performers (e.g., the HIT identifier on AMT). A **performer** may be registered on several platforms (with different accounts) and can be part of several communities. Micro-task **execution** contains some statistics (e.g., start and end timestamps). The **evaluation** contains the answer of the performer for each object, whose schema depends on the operation type.

For example, a *like* evaluation is a counter that registers how many performers like the object, while a *classify* evaluation contains the category selected by the performers for that object.

3.2 Workflow Model

A **crowdsourcing workflow** is defined as a control structure involving two or more interacting tasks performed by humans. Tasks have an input buffer that collects incoming data objects, described by two parameters: 1) The **task size**, i.e. the minimum number of objects (m) that allow starting a task execution; 2) The **block size**, i.e. the number of objects (n) consumed by each executions.

Clearly, $n \leq m$, but in certain cases at least m objects must be present in the buffer before starting an execution; in fact n can vary between 1 and the whole buffer, when a task execution consumes all the items currently in the buffer. Task execution can cause **object removal**, when objects are removed from the buffer, or **object conservation**, when objects are left in the buffer, and in such case the term **new items** denotes those items loaded in the buffer since the last execution.

Tasks communicate with each other with **data flows**, produced by extracting objects from existing data sources or by other tasks, as streams or batches. **Data streams** occur when objects are communicated between tasks one by one, typically in response to events which identify the completion of object's computations. **Data batches** occur when all the objects are communicated together from one task to another, typically in response to events related to the closing of task's computations.

Flows can be constrained based on a condition associated with the arrow representing the flow. The condition applies to properties of the produced objects and allows transferring only the instances that satisfy the condition. Prior to task execution, a **data manipulator** may be used to compose the objects in input to a task, possibly by merging or joining incoming data flows.

We can represent tasks within workflows as described in Fig. 2, where each task is equipped with an input buffer and an optional data manipulator, and may receive data streams or data batches from other tasks. Each task consists of micro-tasks which perform given operations upon objects of a given object type; the parameter r indicates the number of executions that are performed for each micro-tasks, when statically defined (default value is 1). Execution of tasks can be performed according to intra-task patterns, as described in Section 4.

3.3 Workflow Design

Workflow design consists of designing tasks interaction; specifically, it consists of defining the workflow schema as a directed graph whose nodes are tasks and whose edges describe dataflows between tasks, distinguishing streams and batches. In addition, the coupling between tasks working on the same object type can be defined as loose or tight.

Loose coupling is recommended when two tasks act independently upon the objects (e.g. in sequence); although it is possible that the result of one task may have side effects on the other task, such side effects normally occur as an exception and affect only a subset of the objects. Loosely coupled tasks have independent control marts and monitoring rules (as described in Section 3.4).

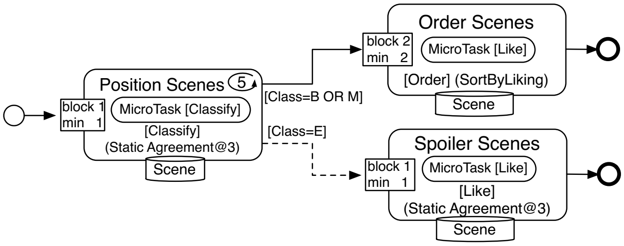


Fig. 3. Example of crowd flow

Tight coupling is recommended when the tasks intertwine operations upon the same objects, whose evolution occurs as combined effect of the tasks' evolution; tightly coupled tasks share the same control mart and monitoring rules.

Figure 3 shows a simple workflow example in the domain of movie scenes annotation. The *Position Scenes* tasks asks performers to say whether a scene appears at the beginning, middle or end of the film; it is a classification task, one scene at a time, with 5 repetitions and acceptance of results based on an agreement threshold of 3. Scenes in the ending part of the movies are transmitted to the *Spoiler Scenes* task, which asks performers whether the scene is a spoiler or not;[4] scenes at the beginning or in the middle of the movie are transmitted to the *Order Scenes* task, which asks performers to order them according to the movie script; each micro-task orders just two scenes, by asking the performer to select the one that comes first. The global order is then reconstructed. Given that all scenes are communicated within the three tasks, they are considered as tightly coupled.

3.4 Task Design

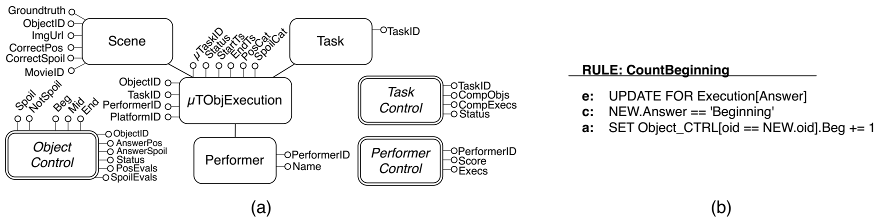
Crowdsourcing tasks are targeted to a single object type and are used in order to perform simple operations which either apply to a single object (such as **like**, **tag**, or **classify**) or require comparison between objects (such **choice** or **score**); more complex tasks perform operations inspired by database languages, such as **select**, **join**, **sort**, **rank**, or **group by**.

Task design consists of the following phases: 1) **Operations design** – deciding how a task is assembled as a set of operation types; 2) **Object and performer design** – defining the set of objects and performers for the task; 3) **Strategy design** – Defining how a task is split into micro-tasks, and how micro-tasks are assigned to subsets of objects and performers; 4) **Control Design** – Defining the rules that enable the run-time control of objects, tasks, and performers.

For monitoring task execution, a data structure called **control mart** was introduced in [3]; Control consists of four aspects:

- **Object control** is concerned with deciding when and how responses should be generated for each object.

[4] A *spoiler* is a scene that gives information about the movie's plot and as such should not be used in its advertisement.



(a)

```
RULE: CountBeginning
e:  UPDATE FOR Execution[Answer]
c:  NEW.Answer == 'Beginning'
a:  SET Object_CTRL[oid == NEW.oid].Beg += 1
```

(b)

Fig. 4. (a) Example of control mart for the tasks of Fig. 3; (b) Example of control rule that updates the number of responses in the *Position of Scenes* task

- **Performer control** is concerned with deciding how performers should be dynamically selected or rejected, on the basis of their performance.
- **Task control** is concerned with completing a task or re-planning task execution.

The control of objects, performers and tasks is performed by **active rules**, expressed according to the *event-condition-action* (ECA) paradigm. Each rule is triggered by **events** (e) generated upon changes in the control mart or periodically; the rule's **condition** (c) is a predicate that must be satisfied on order for the action to be executed; the rule's **actions** (a) change the content of the control mart. Rules properties (e.g., termination) can be been proven in the context of a well-organised computational framework [3].

Figure 4(a) shows a sample control mart for the three tasks in the example scenario, which we assume to be tightly connected, thus using the same data mart. The control mart stores all the required information for controlling the task's evolution and is automatically defined from the task specifications. Figure 4(b) reports a simple control rule that updates the number of responses with value "Beginning" after receiving an answer.
This rule has the following behaviour: every time a performer perform a new evaluation on a specific object (UPDATE event on μTObjExecution), if the selected answer is "Beginning" (the condition part of the rule), then it increases the counter of the "Beginning" category for that object (Object_CTRL[oid == New.oid] selected the correct object, then the correct property can be acessed with the dot notation). For a deeper description of the rule grammar and structure see our previous work [3].

4 Crowdsourcing Patterns

Several patterns for crowd-based operations are defined in the literature. We review them in light of the workflow model of Section 3. We distinguish them in three classes and we implement them in Crowdsearcher (see Section 5):

- **Intra-Task Patterns.** They are typically used for executing a complex task by means of a collection of operations which are cognitively simpler than the original task. Although these patterns do not appear explicitly in the workflow, they are an essential ingredient of crowd-based computations.
- **Workflow Patterns.** They are used for solving a problem by involving different tasks, which require a different cognitive approach; results of the different tasks, once collected and elaborated, solve the original problem.
- **Auxiliary Patterns.** They are typically performed before or after both intra-task and workflow patterns in order either to simplify their operations or to improve theirs results.

4.1 Intra-Task Patterns

Intra-task patterns apply to complex operations, whose result is obtained by composing the results of simpler operations. They focus on problems related to the planning, assignment, and aggregation of micro tasks; they also include quality and performer control aspects. Figure 5 describes the typical set of design dimensions involved in the specification of a task. When the operation applies to a large number of objects and as such cannot be mapped to a single pattern instantiation, it is customary to put in place a *splitting strategy*, in order to distribute the work, followed by an *aggregation strategy*, to put together results. This is the case in many data-driven tasks stemming from traditional relational data processing which are next reviewed.

Consensus Patterns. The most commonly used intra-task patterns aim at producing responses by replicating the operations which apply to each object, collecting multiple assessments from human workers, and then returning the answer which is more likely to be correct. These patterns are referred to as *consensus* or *agreement patterns*. Typical consensus patterns are: *a*) **StaticAgreement** [3]: accepts a response when it is supported by a given number of performers. For instance, in a tag operation we consider as valid responses all the tags that have been added by at least 5 performers. *b*) **MajorityVoting** [19]: accepts a response only if a given number of performers produce the same response, given a fixed number of total executions. *c*) **ExpectationMaximisation** [20]: adaptively alternates between estimating correct answers from task parameters (e.g. complexity), and estimating task parameters from the estimated answers, eventually converging to maximum-likelihood answer values.

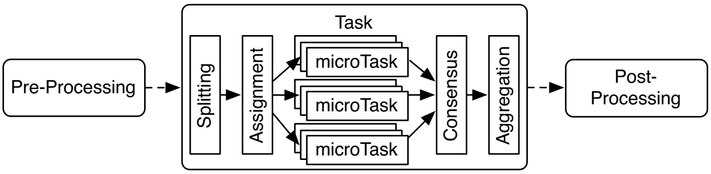


Fig. 5. Building blocks of an Intra-Task Pattern

Join Patterns. Crowd join patterns, studied in [14], are used to build an equality relationship between matching objects in the context of crowdsourcing tasks. We identify: *a*) **SimpleJoin** consists in defining microtasks performing a simple classification operation, where each execution contains a single pair of items to be joined, together with the join predicate question, and two buttons (Yes, No) for responding whether the predicate evaluates to true or false; *b*) **OneToManyJoin** is a simple variant that includes in the same microtask one left object and several right candidates to be joined; *c*) **ManyToManyJoin** includes in the same microtask several candidate pairs to be joined;

Sort Patterns. Sort patterns determine the total ordering of a set of input objects. The list includes: *a*) **SortByGrouping** [14] orders a large set of objects by aggregating the results of the ordering of several small subsets of them. *b*) **SortByScoring** [14] asks performers to rate each item in the dataset according to a numerical scale. *c*) **SortByLiking** [3] is a variant that simply asks the performer to select/like the items they prefer. The mean (or sum) of the scores achieved by each image is used to order the dataset. *d*) **SortByPairElection** [3] asks workers to perform a pairwise comparison of two items and indicate which one they like most. Then ranking algorithms calculate their ordering. *e*) **SortByTournament** [18], presents to performers a tournament-like structure of sort tasks; each tournament elect its champions that progress to the next level, eventually converging to a final order.

Grouping Patterns. Grouping patterns are used in order to classify or clustered several objects according to their properties. We distinguish:
a) **GroupingByPredefinedClasses**[21] occurs when workers are provided with a set of known classes. *b*) **GroupingByPreference** [22] occurs when groups are formed by performers, for instance by asking workers to select the items they prefer the most, and then clustering inputs according to ranges of preferences.

Performer Control Patterns. Quality control of performers consists in deciding how to engage qualified workers for a given task and how to detect malicious or poorly performing workers. The most established patterns for performer control include: *a*) **QualificationQuestion** [23], at the beginning of a microtask, for assessing the workers expertise and deciding whether to accept his contribution or not. *b*) **GoldStandard**, [3] for both training and assessing worker's quality through a initial subtask whose answers are known (they belong to the so-called *gold truth*. *c*) **MajorityComparison**, [3] for assessing performers' quality against responses of the majority of other performers, when no gold truth is available.

4.2 Auxiliary Intra-Task Patterns

The above tasks can be assisted by auxiliary operations, performed before or after their executions, as shown in Figure 5. *Pre-processing steps* are in charge of assembling, re-shaping, or filtering the input data so to ease or optimise the main task. *Post-processing steps* is typically devoted to the refinement or transformation of the task outputs into their final form.

Examples of auxiliary patterns are: *a*) **PruningPattern** [14], consisting of applying simple preconditions on input data in order to reduce the number of evaluations to be performed. For instance, in a join task between sets of actors (where we want to identify the same person in two sets), classifying items by gender, so as to compared only pairs of the same gender. *b*) **TieBreakPattern** [14], used when a sorting task produces uncertain rankings (e.g. because of ties in the evaluated item scores); the post-processing includes an additional step that asks for an explicit comparison of the uncertainly ordered items.

4.3 Workflow Patterns

Very often, a single type of task does not suffice to attain the desired crowd business logic. For instance, with open-ended multimedia content creation and/or modification, it is difficult to assess the quality of a given answer, or to aggregate the output of several executions. A **Workflow Pattern** is a workflow of heterogeneous crowdsourcing tasks with co-ordinated goals. Several workflow patterns defined in the literature are next reviewed; they are comparatively shown in Figure 6:

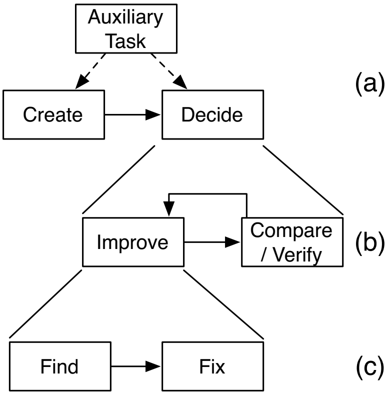


Fig. 6. Template for complex task patterns

a) **Create/Decide** [16], shown in Figure 6(a), is a two-staged pattern where first workers create various options for new content, then a second group of workers vote for the best option. Note that the *create* step can include any type of basic task. This pattern can have several variants: for instance, with a stream data flow, the vote is typically restricted to the solutions which are produced faster, while with a batch data flow the second task operates on all the generated content, in order to pick the best option overall. *b*) **Improve/Compare** [7], shown in Figure 6(b), iterates on the decide step to progressively improve the result. In this pattern, a first pool of workers creates a first version of a content; upon this version, a second pool of workers creates an improved version, which is then compared, in a third task, to decide which one is the best (the original or the improved one). The improvement/compare cycle can be repeated until the improved solution is deemed as final. *c*) **Find/Fix/Verify** [4], shown in Figure 6(c), further decomposes the *improve* step, by splitting the task of finding potential improvements from the task of actually implementing them.

4.4 Auxiliary Workflow Patterns

Auxiliary tasks can be designed to support the creation and/or the decision tasks. They include: *a*) **AnswerBySuggestion** [17]: given a create operations as input, the provided solution can be achieved by asking suggestions from the crowd as follows. During each execution, a worker can choose one of two actions: it can either stop and submit the most likely answer, or it can create another job and receive another response to the task from another performer. The auxiliary suggestion task produces content that can be used by the original worker to complete or improve her answer. *b*) **ReviewSpotcheck** strengthens the decision step by means of a two-staged review process: an additional quality check is performed after the corrections and suggestions provided by the performers of the decision step. The revision step can be performed by the same performer of the decision step or by a different performer.

5 Workflow Execution

Starting from the declarative specification described in Sections 3 and 4, an automatic process generates task descriptors and their relations. Single tasks and their internal strategies and patterns are transformed into executable specification; we support all the intra-task patterns described in Section 4, through model transformations that generate the control marts and control rules for each task [3]. Task interactions are implemented differently depending on whether interacting tasks are tightly coupled or loosely coupled.

Tightly coupled tasks share the control mart structure (and the respective data instances), thus coordination is implemented directly on data. Each task posts its own results and control values in the mart. Dependencies between tasks are transformed into rules that trigger the creation of new micro-tasks and their executions, upon production of new results by events of object or task closure.

Loosely coupled tasks have independent control marts, hence their interaction is more complex. Each task produces in output events such as `ClosedTask`, `ClosedObject`, `ClosedMicrotask`, `ClosedExecution`. We rely on an event based, publish-subscribe mechanism, which allows tasks to be notified by other tasks about some happening. Loosely coupled tasks do not rely on a shared data space, therefore events carry with them all the relevant associated pieces of information (e.g., a `ClosedObject` event carries the information about that object; a `ClosedTask` event carries the information about the closed objects of the task).

The workflow structure dictates how tasks subscribe to events of other tasks. Once a task is notified by an incoming event, the corresponding data is incorporated in its control mart by a-priori application of the data manipulation program, specified in the data manipulator stage of the task. Then, reactive processing takes place within the control mart of the task.

Modularity allows executability through model transformations which are separately applied to each task specification. Automatically generated rules and mart structures can be manually refined or enriched when non-standard behaviour is needed.

This approach is supported by CrowdSearcher, a platform for crowd management written in JavaScript. CrowdSearcher runs on *Node.js*, a full-fledged event-based system, which fits the need of our rule-based approach. Each control rule is translated into scripts; triggering is modelled through internal platform

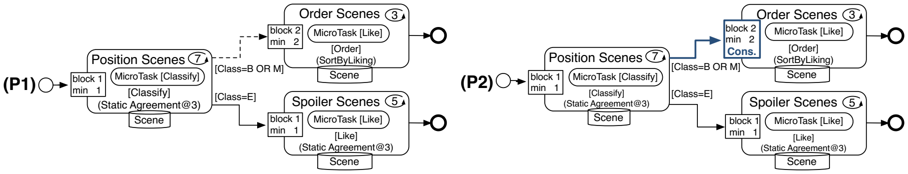


Fig. 7. Flow variants for the Positioning scenario

events. Precedence between rules is implicitly obtained by defining the scripts in the proper order. CrowdSearcher offers a cloud-based environment to transparently interface with social networks and crowdsourcing platforms, according to the task model described in Section 3.1. It features an online configuration interface where designers build complex crowdsourcing applications through a wizard–driven, step by step approach. A built-in *Task Execution Framework* (TEF) provides support for the creation of custom task user interfaces, to be deployed as stand-alone application, or embedded within third-party platforms such as Amazon Mechanical Turk. Specific modules are devoted to the invitation, identification, and management of performers, thus offering support for a broad range of expert selection paradigms, from pure pull approaches of open marketplaces, to pre-assigned execution to selected performers. Alternatives for the implementation of operations on crowd-based systems are discussed in [2].

6 Experiments

We demonstrate various pattern-based workflow scenarios, defined using our model and method and deployed by using Crowdsearcher as design framework and Amazon Mechanical Turk as execution platform. We consider several scenes taken from popular movies, and we enrich them with crowd-sourced information regarding their position in the movie, whether the scene is a spoiler, and the presence of given actors in each scene. In the experiments reported here we considered the movie "The Lord of the Rings: the Fellowship of the Ring". We extracted 20 scenes and we created a groundtruth dataset regarding temporal positioning and actors playing in the scenes. We compare cost and quality of executions for different workflow configurations.

Table 1. *Scenario 1 (Positioning)*: number of evaluated objects, microtask executions, elapsed execution time, performers, and executions per performer (for each task and for each scenario configuration)

	Position Scenes (payed $0.01)					**Order Scene (payed $0.01)**					**TOTAL**		
	#Obj	***#Exe***	***Time***	***#Perf***	***#Exe/Perf***	***#Obj***	***#Exe***	***Time***	***#Perf***	***#Exe/Perf***	**Time**	**Cost**	**#Perf**
P1	20	147	123	16	9.19	17	252	157	14	18.00	342	3.99$	26
P2	20	152	182	12	12.67	17	230	318	17	13.53	349	3.82$	26

Table 2. *Scenario 2 (Actor)*: number of evaluated objects, microtask executions, elapsed execution time, performers, and executions per performer (for each task and for each scenario configuration)

	Find Actors (payed $0.03)					**Validate Actors (payed $0.02)**					**TOTAL**		
	#Obj	*#Exe*	*Time*	*#Perf*	*#Exe/Perf*	*#Obj*	*#Exe*	*Time*	*#Perf*	*#Exe/Perf*	**Time**	**Cost**	**#Perf**
A1	20	100	120	18	5.56	–	–	–	–	–	120	3.00$	18
A2	20	100	128	10	10.00	–	–	–	–	–	128	3.00$	10
A3	20	100	123	14	7.15	20	21	154	10	2.10	159	3.42$	20
A4	20	100	132	10	10.00	41	19	157	9	2.10	164	3.38$	16
A5	20	100	126	13	7.69	69	60	242	17	3.53	257	4.20$	24
A6	66	336	778	56	6.00	311	201	821	50	4.02	855	14.10$	84

6.1 Scenario 1: Scene Positioning

The first scenario deals with extracting information about the temporal position of scenes in the movie and whether they can be considered as as spoilers. Two variants of the scenario have been tested, as shown in Figure 7: the task *Position Scenes* classifies each scene as belonging to the beginning, middle or ending part of the movie. If the scene belongs to the final part, we ask the crowd if it is a spoiler (*Spoile Scenes* task); otherwise, we ask the crowd to order it with respect to the other scenes in the same class (*Order Scenes* task).

Tasks have been configured according to the following patterns:

- *Position Scene*: task and microtask types are both set as *Classify*, using a *StaticAgreement* pattern with threshold 3. Having 3 classes, a maximum number of 7 executions grants that one class will get at least 3 selections. Each microtask evaluates 1 scene.
- *Order Scene*: task type is *Order*, while microtask type is set as *Like*. Each microtask comprises two scenes of the same class. Using a *SortByLiking* pattern, we ask performers to select (Like) which scene comes first in the movie script. A rank aggregation pattern calculates the resulting total order upon task completion.
- *Spoiler Scene*: Task and microtask type both set as *Like*. A *StaticAgreement* pattern with threshold 3 (2 classes, maximum 5 executions) defines the consensus requirements. Each microtask evaluates 1 scene.

We experiment with two workflow configurations. The first (**P1**) defines a *batch* data flow between the *Position Scene* and *Order Scene* tasks, while the second configuration (**P2**) defines the same flow as *stream*. In both variants, the data flow between *Position Scene* and *Spoiler Scenes* is defined as *stream*.

The **P2** configuration features a dynamical task planning strategy for the the *Order Scenes* task, where the construction of the scene pairs to be compared in is performed every time a new object is made available by the *Position Scenes* task. A conservation policy in the *Order Scenes* data manipulator ensures that all the new scenes are combined with the one previously received.

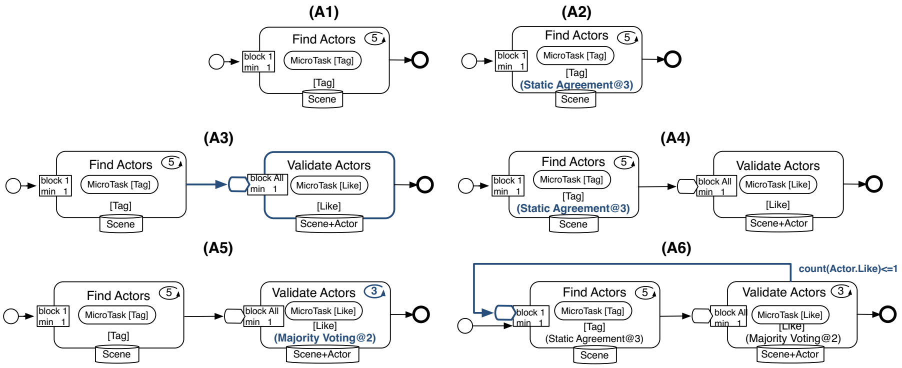


Fig. 8. Flow variants for the Actor scenario

6.2 Scenario 2: Actors

In the second scenario, we model a **create/decide** workflow pattern by asking the crowd to identify the actors that take part in the movie scenes; in *Find Actors*, performers indicate actors, in *Validate Actor* they confirm them. Tasks are designed as follows:

- *Find Actors*: Task and microtask types are set as *Tag*. Each microtask evaluates one scene; each scene is evaluated five times. Depending on the configuration, either no consensus pattern (**A1**, **A3**, **A5**) or a *StaticAgreement* pattern with threshold three (**A2**, **A4**, **A6**) is employed.
- *Validate Actors*: the task is preceded by a data manipulator function that transform the input Scene object and associated tags into a set of tuples $(Scene, Actor)$, which compose an object list subject to evaluation. In all configurations, microtasks are triggered if at least one object is available in the buffer. Note that each generated microtask features a different number of objects, according to the number of actors tagged in the corresponding scene. Configurations **A5** and **A6** features an additional *MajorityVoting* pattern to establish the final actor validation.

We tested this scenario with five workflow configurations, shown in Figure 8, and designed as follows:

- Configuration **A1** performs 5 executions and for each scene collects all the actors tagged at least once;
- Configuration **A2** performs 5 executions and for each scene collects all the actors tagged at least three times (StaticAgreement@3);
- Configuration **A3** adds the validation task to **A1**; the validation asks one performer to accept or reject the list of actors selected in the previous step;
- Configuration **A4** adds a validation task to **A3**, performed as in **A3**;
- Configuration **A5** is similar to **A3**, but the validation task is performed 3 times and a MajorityVoting@2 is applied for deciding whether to accept or not the object;

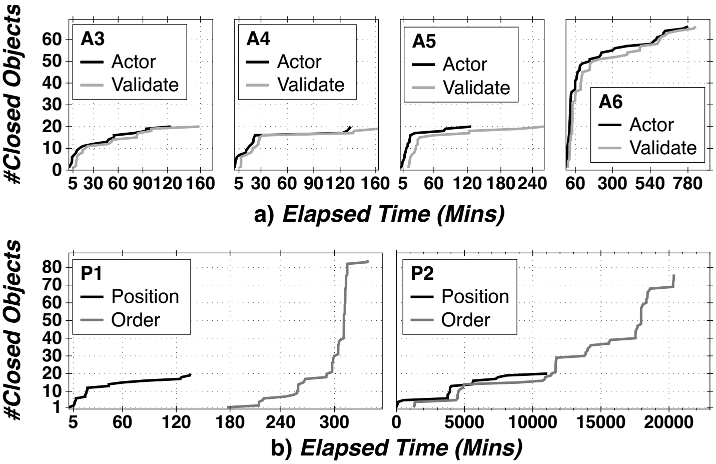


Fig. 9. Temporal distributions of closed objects

- Configuration **A6** extends **A5** by adding a StaticAgreement@3 on FindActors a feedback stream flow, originating from the *Validate Actors* task and directed to the *Find Actors* task, which notifies the latter about actors that were wrongly tagged in a scene (i.e., for which agreement on acceptance was not reached). Misjudged scenes are then re-planned for evaluation; for each scene, the whole process is configured to repeat until validation succeeds, or at most 4 re-evaluations are performed.

6.3 Results

We tested the performance of the described scenarios in a set of experiments performed on Amazon Mechanical Turk during the last week of September 2013. Table 1 and Table 2 summarise the experiment statistics for the two scenarios, 1700 HITS for a total cost of 39$.

Streaming Vs. Batch (Scenario 1: Positioning). In the first scenario we tested the impact on the application performance of the adoption of a stream data flow in a crowd workflow.

Time. Figure 9(b) shows the temporal distribution of closed objects for the **P1** and **P2** configurations. As expected, a stream flow (**P2**) allows for almost synchronous activation of the subsequent task in the flow, while batch scenario (**P1**) shows a strict sequential triggering of the second task. However, the overall duration of the workflow is not significantly affected by the change. While the first task of the flow behaves similarly in the two configurations, the second task runs significantly quicker in the batch flow, thus recovering the delay due to the sequential execution.

Quality. Table 3a shows the precision of the classification results of task *Position Scenes* (note that for this first part the two configurations are exactly the same, it makes no sense to compare the two results). Table 3b shows a measure of the quality of the obtained orders of scenes, i.e., Spearman's rank correlation coefficient of the resulting ranks from the *Order Scenes* task against the real order of scenes. Both tables show that the attained quality was not significantly influenced by the different task activation modes.

In summary, we didn't notice a different behaviour due to streaming. One possible reason is that in the batch configuration the entire set of assignments is posted at once on AMT, thus becoming more prominent in terms of number of available executions (and thus being preferred by performers, as widely studied [1]), while in a stream execution a small number of assignments is posted on AMT at every closing event of objects from the previous tasks.

Intra-Task Consensus Vs. Workflow Decision (Scenario 2: Actors). The second scenario aimed at verifying the impact that different intra-task and workflow patterns produced on the quality, execution time, and cost. We focused in particular on different validation techniques.

Time. Figure 9(a) and (c) shows the temporal distribution of closed object for configurations **A3-A6**. Configurations **A1** and **A2** are not reported because they are composed of one single task and thus their temporal distribution is not comparable. The temporal behaviour of the first and second tasks in the flow are rather similar (in the sense that the second one immediately follows the other). Validation is more delayed in **A5** due to the MajorityVoting pattern that postpones object close events. Configuration **A6** (Figure 9(c)) is significantly slower due to the feedback loop, which also generates a much higher cost of the campaign, as reported in Table 1. Indeed, due to the feedback, many tasks are executed several times before converging to validated results.

Quality. Table 4 reports the precision, recall and F-Score figures of the six configurations. The adoption of increasingly refined validation-based solutions (configurations **A3-A4-A5**) provides better results with respect to the baseline configuration **A1**, and also to the intra-task agreement based solution **A2**; validations do not have a negative impact in terms of execution times and costs. On the other hand, the complexity of of case **A6**, with the introduction of feedback, proved counter-productive, because the validation logic harmed the performance, both in monetary (much higher cost) and qualitative (lower results quality) senses, bringing as well overhead in terms of execution time. Notice

Table 3. Scenario 1 (Positioning), configuration P1 and P2: a) Precision of the *Position Scenes* classification task; b) Spearman's rank correlation coefficient of the resulting ranks from the *Order Scenes* task against the real order of scenes

(a)

Config.	**P Beg.**	**P Mid.**	**P End**
P1	0.50	1	0.11
P2	0.50	0.80	0.33

(b)

	Spearman Beg.	**Spearman Mid.**
P1	0.500	0.543
P2	0.900	0.517

Table 4. Scenario 2 (Actor): Precision, Recall, and F-score of the 6 configurations

	A1	**A2**	**A3**	**A4**	**A5**	**A6**
Precision	0.79	1	0.92	0.99	0.95	0.89
Recall	0.98	0.87	0.97	0.90	1	0.96
F-Score	0.85	0.91	0.93	0.93	0.97	0.90

that the configuration **A3** reaches the highest precision score. That's because the StaticAgreement strategy ensures that all the selected actors really appear in the image, while using the crowd for the validation part can add some errors (for instance some actors recognized in the *Find Actor* can be discarded in the *Validate Actors*). However note that the other configurations (**A3 - A5**) reach an higher recall and F-score value, meaning an overall better quality of the final result.

In summary, the above tests show an advantage of concentrating design efforts in defining better workflows, instead of just optimising intra-task validation mechanisms (based e.g. on majority or agreement), although overly complex configurations should be avoided.

7 Conclusions

We present a comprehensive approach to the modeling, design, and pattern-based specification of crowd-based workflows. We discuss how crowd-based tasks communicate by means of stream-based or batch data flows, and we define the option between loose and tight coupling. We also discuss known patterns that are used to create crowd-based computations either within a task or between tasks and we show how the workflow model is translated into executable specifications which are based upon control data, reactive rules, and event-based notifications.

A set of experiments demonstrate the viability of the approach and show how the different choices in workflow design may impact on the cost, time and quality of crowd-based activities.

References

[1] Law, E., von Ahn, L.: Human Computation. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers (2011)

[2] Bozzon, A., Brambilla, M., Ceri, S.: Answering search queries with crowdsearcher. In: 21st Int.l Conf. on World Wide Web, WWW 2012, pp. 1009–1018. ACM (2012)

[3] Bozzon, A., Brambilla, M., Ceri, S., Mauri, A.: Reactive crowdsourcing. In: 22nd World Wide Web Conf., WWW 2013, pp. 153–164 (2013)

[4] Bernstein, M.S., Little, G., Miller, R.C., Hartmann, B., Ackerman, M.S., Karger, D.R., Crowell, D., Panovich, K.: Soylent: a word processor with a crowd inside. In: Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology, UIST 2010, pp. 313–322. ACM, New York (2010)

[5] Minder, P., Bernstein, A.: How to translate a book within an hour: towards general purpose programmable human computers with crowdlang. In: WebScience 2012, Evanston, IL, USA, pp. 209–212. ACM (June 2012)

[6] Doan, A., Ramakrishnan, R., Halevy, A.Y.: Crowdsourcing systems on the world-wide web. Commun. ACM 54(4), 86–96 (2011)

[7] Little, G., Chilton, L.B., Goldman, M., Miller, R.C.: Turkit: tools for iterative tasks on mechanical turk. In: HCOMP 2009, pp. 29–30. ACM (2009)

[8] Kochhar, S., Mazzocchi, S., Paritosh, P.: The anatomy of a large-scale human computation engine. In: HCOMP 2010, pp. 10–17. ACM (2010)

[9] Ahmad, S., Battle, A., Malkani, Z., Kamvar, S.: The jabberwocky programming environment for structured social computing. In: UIST 2011, pp. 53–64. ACM (2011)

[10] Marcus, A., Wu, E., Madden, S., Miller, R.C.: Crowdsourced databases: Query processing with people. In: CIDR 2011, pp. 211–214 (January 2011), www.cidrdb.org

[11] (OMG), O.M.G.: Business process model and notation (bpmn) version 2.0. Technical report (January 2011)

[12] Wang, J., Kumar, A.: A framework for document-driven workflow systems. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 285–301. Springer, Heidelberg (2005)

[13] Nigam, A., Caswell, N.: Business artifacts: An approach to operational specification. IBM Systems Journal 42(3), 428–445 (2003)

[14] Marcus, A., Wu, E., Karger, D., Madden, S., Miller, R.: Human-powered sorts and joins. Proc. VLDB Endow. 5(1), 13–24 (2011)

[15] Kazai, G., Kamps, J., Milic-Frayling, N.: An analysis of human factors and label accuracy in crowdsourcing relevance judgments. Inf. Retr. 16(2), 138–178 (2013)

[16] Little, G., Chilton, L.B., Goldman, M., Miller, R.C.: Exploring iterative and parallel human computation processes. In: Proceedings of the ACM SIGKDD Workshop on Human Computation, HCOMP 2010, pp. 68–76. ACM, New York (2010)

[17] Lin, C.H., Mausam, Weld, D.S.: Crowdsourcing control: Moving beyond multiple choice. In: UAI, pp. 491–500 (2012)

[18] Venetis, P., Garcia-Molina, H., Huang, K., Polyzotis, N.: Max algorithms in crowdsourcing environments. In: WWW 2012, pp. 989–998. ACM, New York (2012)

[19] Nowak, S., Rüger, S.: How reliable are annotations via crowdsourcing: a study about inter-annotator agreement for multi-label image annotation. In: Proceedings of the International Conference on Multimedia Information Retrieval, MIR 2010, pp. 557–566. ACM, New York (2010)

[20] Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the em algorithm. Journal of the Royal Statistical Society 39(1), 1–38 (1977)

[21] Davidson, S.B., Khanna, S., Milo, T., Roy, S.: Using the crowd for top-k and group-by queries. In: Proceedings of the 16th International Conference on Database Theory, ICDT 2013, pp. 225–236. ACM, New York (2013)

[22] Adomavicius, G., Tuzhilin, A.: Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. IEEE Transactions on Knowledge and Data Engineering 17(6), 734–749 (2005)

[23] Alonso, O., Rose, D.E., Stewart, B.: Crowdsourcing for relevance evaluation. SIGIR Forum 42(2), 9–15 (2008)