Journal of Web Engineering, Vol. 14, No. 5&6 (2015) 443–473

DESIGNING COMPLEX CROWDSOURCING APPLICATIONS COVERING MULTIPLE PLATFORMS AND TASKS

ALESSANDRO BOZZON
Software and Computer Technologies Department. Delft University of Technology
Postbus 5 2600 AA, Delft, The Netherlands
a.bozzon@tudelft.nl

MARCO BRAMBILLA STEFANO CERI ANDREA MAURI RICCARDO VOLONTERIO
Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB)
Politecnico di Milano. Piazza Leonardo da Vinci, 32. 20133 Milano, Italy
{name.surname}@polimi.it

A number of emerging crowd-based applications cover very different scenarios, including opinion mining, multimedia data annotation, localised information gathering, marketing campaigns, expert response gathering, and so on. In most of these scenarios, applications can be decomposed into tasks that collectively produce their 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 this framework, including typical workflow patterns, whose function is to decompose a cognitively complex task into simple interacting tasks for cooperative solving.

We then discuss how workflows and patterns are managed by CrowdSearcher, 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.

Keywords: Crowdsourcing, Workflow, Social Network, Community, Control.

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, content tagging and item classification, by combining human contribution and automatic analysis of results [10]. But a recent trend

(emerging, e.g., during the CrowdDB Workshop[a]), 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 [4, 5], 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 of 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; we define a crowdsourcing task as an arbitrary composition of these task types. This model is not introducing 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. Such problem decomposition assigns different roles to performers, so that they can collectively and cooperatively solve the original problem, reproducing a sort of "society of minds" [12]. In particular, several works [1, 11] have analysed typical *crowdsourcing patterns*, i.e. typical cooperative schemes used for organising crowd-based applications.

The motivation of our work is backed by several real-world case studies, that describe the need of organizing the work of performers through several subsequent steps of activity, possibly spanning different crowdsourcing platforms and/or communities of workers. The study by Oosterman et al. [13] represents a typical motivation example. In their study they used the crowd to count and identify different types of flowers in old artworks. In the experiments the authors tried different configurations (e.g., asking the crowd to directly counting the flowers; drawing the flowers upon the original images; or counting the flower inside a bounding box) and analyzed the different levels of performance of the crowd. The experiments show that by adding a preliminary step, where the performer draws a box around the flowers, the quality of the final result improved. This suggests that a study on the alternative task pattern designs can result in optimized quality of the outcome of the tasks.

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] `http://dbweb.enst.fr/events/dbcrowd2013/`

a systematic classification of patterns.

The paper is structured as follows. After reviewing the related work in Section 2, Section 3 presents the task and workflow models and design processes. Then, Section 4 classifies the crowdsourcing patterns and in particular distinguishes between intra-task patterns and workflow patterns, where the former apply to a single task and the latter relate two or more tasks. Section 5 illustrates how workflow specifications are embodied within the execution control structures of CrowdSearcher, and finally Section 6 discusses several experiments, showing how differences in workflow design lead to different application results.

2 Related Work

Many crowdsourcing startups[b]and systems [**?**] 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* [**?**], *RABJ* [**?**], *Jabberwocky* [**?**]).

As highlighted by [**?**], several programmatic methods for human computation have been proposed [**?**, **?**, **?**, **?**], 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.

Recent works propose approaches for human computation which are based on high level abstractions, sometimes of declarative nature. In [**?**], authors describe a language that interleaves human-computable functions, standard relational operators and algorithmic computation in a declarative fashion. *Qurk* [**?**] is a query system for human computation workflows that exploits a relational data model and SQL. *CrowdDB* [**?**] also adopts a declarative approach by using CrowdSQL (an extension of SQL). *DeCo* [**?**] allows SQL queries to be executed on a crowd-enriched datasource, with human tasks defined as *fetch* and *resolution* rules programmed in a scripting language (Python). These works do not discuss how to specify the control associated with the execution of human tasks, leaving its management to opaque optimisation strategies; we instead believe that the performance of crowdsourcing tasks can hardly be estimated, hence some provision for dynamic control is essential. Moreover, current declarative crowdsourcing platforms do not include means for precise targeting of users based on expertise and for addressing multiple platforms and communities.

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 [**?**, **?**] 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[c]. 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.

[b] E.g., CrowdFlower, Microtask, uTest.
[c] `http://workflowpatterns.com/`

In contrast, data-driven workflows have recently become very popular, typically in domains where database are central to processes [**?**, **?**], and data consistency and soundness is a strong requirement. Data-driven workflows treat data as first-class citizens, emphasise the role of intra- and inter-task control, and ultimately served as an inspiration for our work. For instance [**?**] describes a simple model for collaborative data-driven workflows, with the goal of carrying out runtime reasoning about the global run of the system.

Very few works studied workflow-driven approaches for crowd work. CrowdLang [11] is a 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. *Datasift* [**?**] is a toolkit for configuring search queries so as to involve crowds in answering them, which allows users to decide the number of involved human workers, the query reformulation in steps, the number of items involved at each step and their cost; the system is structured as several plug-and-play components. For these features, Datasift is similar to CrowdSearcher, although it is dedicated to a single crowdsourcing engine (Amazon Mechanical Turk) and it lacks an abstract query model.

3 Models and Design of Crowd-based Workflows

Although humans are capable of solving complex tasks by using their full cognitive capacity, classical approaches used in crowd-based computations prefer to decompose complex tasks into simpler tasks and then elaborate their result. For instance, [**?**] proposes methods for producing a total ordering of elements of a set by ordering subsets and then composing the partial orders, while [**?**] proposes a coupling of two activities (e.g. classifying and verifying) so that the action performed by a given person is verified by a different person. Simple tasks can be easily embedded into games; by using them, the diversity which is typical of human behaviours is restricted to produce comparable answers, which can then be assembled and aggregated so as to determine the "crowd's opinion" through statistical approaches. Thus, task decomposition is an important ingredient of crowd-based applications.

Following such 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.

Thus, the first step in the design of a complex crowd-based application is the design of its high-level workflow schema, which coordinates the execution of simple tasks. Prior to dwelling into workflow design, we define tasks and their properties.

3.1 Task Model

Tasks of a crowd-based application are described in terms of an abstract model [4], crafted after a careful analysis of the systems for human task executions and of many applications and case studies, and represented in Fig. 1.

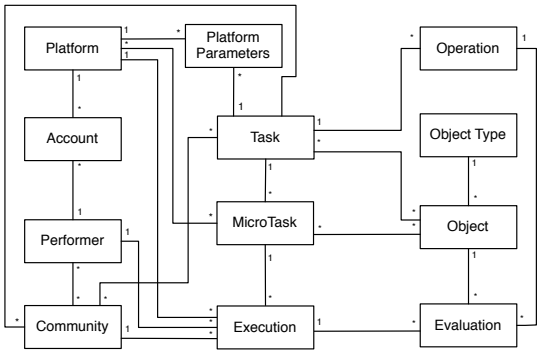


Fig. 1. Metamodel of task properties.

Task Type	Performer Action
Choice	Selects up to n items
Like	Adds like/unlike annotations to some items
Score	Assigns a score (1..n interval) to some items
Tag	Annotates some items with tags
Classify	Assigns each item to one or more classes
Order	Reorders the (top n) items in the input list
Ins./Del.	Inserts/deletes up to n items in the list
Modify	Changes the values of some items attributes
Group	Clusters the items into (at most n) groups

Fig. 2. List of the crowdsourcing operation types.

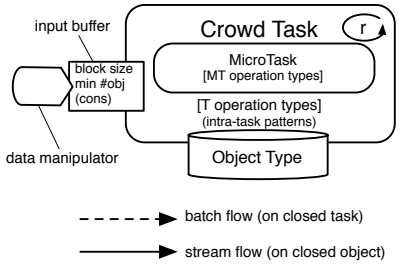


Fig. 3. Notation for crowdsourcing workflows.

The main strength of the model is its extreme simplicity. 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 Figure 2; they can be extended by adding custom types. For instance, a task may consist in choosing one photo out of an input list of photos, writing a caption for it, and then adding some tags.

Task management requires specific sets of objects to be assembled into a unit of execution, called **MicroTask**, that is associated with a given **performer**. Each MicroTask 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 Amazon Mechanical Turk). A **performer** may be registered on several platforms (with different accounts) and can be part of several communities. MicroTask **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.

Figure 3 depicts the graphical notation adopted on our work to describe a *Crowd Task*.

Next sections describe each component of a task model, explaining their design rationale and principles, and detailing their representation in the adopted notation.

3.2 Task Design

The design of each task in a crowd-based application consists of a progression of six phases reported in Figure 4, namely ***1*) Operation design**, i.e. deciding how each task is assembled as a set of **MicroTask operation types**; ***2*) Object design**, i.e. defining the **Object Type** and preparing the actual set of objects to be evaluated, which may be extracted from different data sources; ***3*) Performer design**, i.e. selecting the performers that will be asked to perform the application; ***4*) Workplan design**, i.e. defining how each task is split into micro-tasks, and how micro-tasks are assigned to objects; ***5*) Platform selection**, i.e. defining the *invitation platforms*, where performers are recruited, and the *execution platforms*, where performers execute tasks; many different platforms may be involved in either roles; ***6*) UI design**, i.e. defining the front end aspects of the task execution. We next define each of these phases.

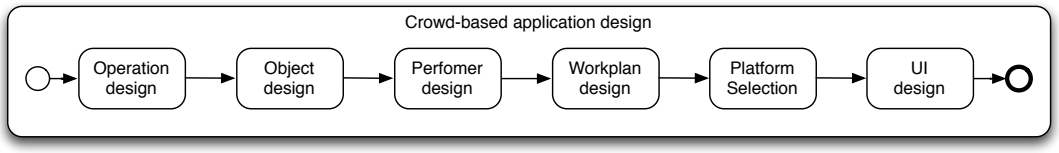


Fig. 4. Development process of crowdsourcing tasks.

3.2.1 Operation Design

Each task consists of one or more MicroTasks; each MicroTask performs a **MicroTask Operations** upon objects of a given object type; the parameter r in the notation of Figure 3 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.

In a typical example of data analysis, that we will use as running use case, performers are presented scenes of movies and must position them in the movie timeline, understand if the images are spoilers (i.e. images that should not be disclosed as they suggest the movie's plot) and identify the actors in each scene. This can be organised in two tasks, the first one for identifying the positioning of images and whether they are spoilers or not, and the second one for annotating images with the actor names.

3.2.2 Object Design

Object design consists of defining the dataset which is subject to the analysis. In particular, object design entails: *1*) the definition of the schema of the objects to be analysed; *2*) the extraction or collection of the instances (e.g., from web sources, proprietary databases, plain text, or multimedia repositories); and *3*) the cleaning of the extracted objects, so as to make them conforming to the defined schema.

Tasks have an input buffer that collects incoming data objects, described by two param-

eters: *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. Prior to task execution, a **data manipulator** may be used to compose the objects in input to a task, possibly by merging or joining data incoming from different sources.

3.2.3 Performer Selection

In crowdsourcing, strategies for task assignment can be roughly classified into two main categories: *push* and *pull* assignment. With *pull* assignments, tasks are published on an open board, and performers select them. This strategy is used by most crowdsourcing platforms, and is well-suited for simple repetitive tasks. With *push* assignments, tasks are routed either to individuals or to communities based upon trust, knowledge, or expertise. Performers are either pre-selected or dynamically assigned to executions, depending on the task content.

Social platforms, such as Facebook, Twitter and LinkedIn, provide their members with several hundreds of known contacts, with variable expertise about various topics, and with varying availability and responsiveness. In [**?**] we studied **social expert finding**, i.e. the process of identifying potential domain-specific experts in the crowd by mining their social profiles. **Social expert finding** can be characterised by the following question: *given a task relative to a given domain, who are the most knowledgeable users for it, and which social network is most appropriate for approaching them?*
While traditional methods rely on profile information [**?**], in our approach we considered also the users' social relationships, social activities, and socially shared content. In our study we discovered that user profiles alone are not too effective in determining expertise, as they contain few resources compared to those that are found in the candidate's pages; resources of connected persons are also quite informative. We observed that, in our experimental setting, Twitter is the most effective social network for expertise matching, and sometimes outperforms the combined use of all social networks. It resulted most effective to identify experts in the domains: Computer Engineering, Science, Technology, Games, and Sport. Facebook, on the other hand, proved more effective in complementary domains, such as Locations, Sport, Movies, TV, Music. Interestingly, despite its focus on work-related expertise, the resources contained in LinkedIn were rarely useful for expert identification purposes. These insights are extremely helpful for determining the best suited communities of performers for domain-specific crowdsourcing activities. Indeed, as we have shown in our experimental evaluation of crowdsourcing based on expert communities, the selection of the right set of performers have a relevant impact on the quality of the results [8].

3.2.4 Workplan Design

Workplan design consists of creating micro-tasks for each task and of mapping each micro-task to specific performers and objects. A typical workplace design operation is *task splitting* [**?**], whose strategies must be designed by taking into account the different operation types

involved in the task, the number of objects to be evaluated in the task, and the number of replicas that are requested for each object. The latter value is specified in the task model notation as a circular arrow located on the top-right of the task box. Repeated executions on the same object are customary in crowdsourcing, for the purpose of collecting responses from various performers and then aggregating them by considering their agreement. This allows higher confidence on the final outcome of the task. A splitting strategy must be provided with a corresponding aggregation strategy that is able to re-compute the final result of the task.

Plat.Type	Examples	Inv.	Exec.
Social Network	Facebook, Twitter, G+, LinkedIn	YES	Limited
Question & Answer	Quora, Doodle	Limited	YES
Crowdsourcing	AMT, CrowdFlower	YES	YES
Proprietary UI	Custom developed application	Limited	YES
Email or messaging	Mailing lists, personal email, phone messages	YES	Limited

Fig. 5. Taxonomy of platforms and use in invitation and execution.

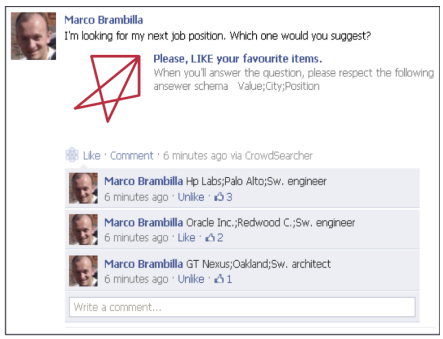


Fig. 6. Native Facebook "*like*" operation.

3.2.5 Platform Selection

At this stage, after a platform-independent design, deployment platforms must be chosen. A variety of systems are offered, and it is crucial to understand how they can be crafted to reflect the application needs. We distinguish *invitation* from *execution*, the former process is concerned with inviting people to perform tasks, the latter is concerned with executing tasks. In general it is possible to use different systems for invitation and execution. Figure 5 shows how the different kinds of platform support them.

Social Networks provide powerful interfaces for crowd selection and invitation, and limited support for execution. Note that deployment on social network can be of two kinds [4]:

- *Native implementations* use the features of a specific social network for task execution.
- *Embedded implementations* use the execution of user-defined code from within the social network.

For instance, Figure 6 shows how Facebook can be natively used for implementing *like* operations: data objects are posted on a wall and users simply click the Facebook *Like* button. Certain operations, such as liking and tagging, are best supported by native interfaces.

Question-Answering Systems are dual [**?**, **?**], as they normally cannot support invitation (or provide invitation mechanisms that require to provide the list of invitees from a personal contact list), while they can support execution, although some of them only support free text responses.

Crowdsourcing Systems support both invitation and execution, but they do it within the context of a given platform, which acts as a market place, where invitations include the

monetary reward for each task, and after execution each performer is credited. Invitations from other platforms are typically not allowed.

Email and other messaging systems support the invitation phase, that can be routed to specific groups, mailing lists, or enumerated list of targets. Execution is typically delegated to other platforms.

The most typical form of deployment consists in delivering a UI crafted around the features of the specific objects to be shown to performers, as discussed in the next section.

3.2.6 UI Design

UI design plays an important role in crowdsourcing, as it produces the user interfaces that permit the actual execution of tasks by performers. UIs can be designed in three main ways: *1*) By exploiting default, basic UIs made available by crowdsourcing platforms (e.g., AMT); *2*) By exploiting the conceptual task model for generating a simple custom UI; *3*) By manually implementing an ad-hoc user interface most suited to the task.

Finally, no UI is needed if tasks are natively performed on social platforms. For instance, this is the case when a task is performed by directly exploiting the *like* mechanism in Facebook.

3.3 Crowd 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 data-flows between tasks, distinguishing streams and batches.

A **crowdsourcing workflow** is defined as a control structure involving two or more interacting tasks performed by humans. Tasks communicate with each other with **data flows**, produced by extracting objects from existing data sources or by other tasks. Data flows are of two kinds:

- **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 the computations of the task.

Flows can be constrained based on a condition applied on the arrow. In our model, conditions are specified as `OCL` statements. The condition applies on properties of the produced objects and allows transferring only the instances that satisfy the condition.

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.
- **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.

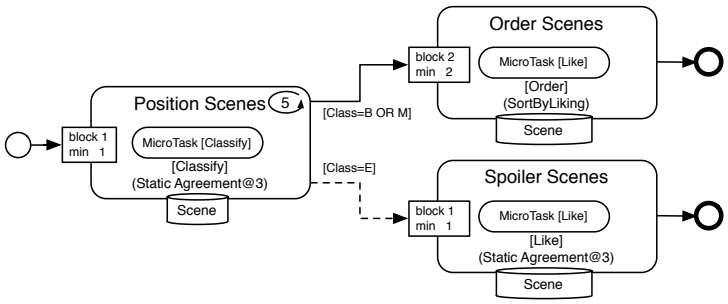


Fig. 7. Example of crowd flow.

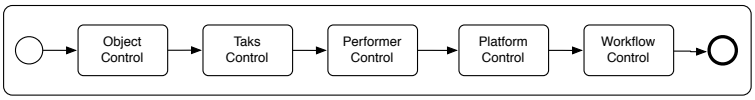


Fig. 8. Control steps of crowdsourcing applications.

Note that loosely coupled tasks have independent **control** management, as described in Section 3.4. Figure 7 shows a simple workflow example in the domain of movie scenes annotation. The *Position Scenes* task 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 (more on agreement control rules in Section 3.4). Scenes in the ending part of the movies are transmitted to the *Spoiler Scenes* task, which asks performers whether the scene is a spoiler[d]or not; 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 Control

The behaviour of the crowd-based application is hardly predictable, therefore control aspects are an important part of the design. It consists of four phases, shown in Figure 8: ***1*) Object control**, i.e. assembling the responses relative to each object and deciding if the work relative to each object is either completed or need to be replanned; ***2*) Task control**, i.e. assembling statistics about responses relative to each task and deciding if the task is either completed or needs to be replanned; ***3*) Performer control**, i.e. deciding how performers should be dynamically selected or rejected, on the basis of their performance; ***4*) Platform and community control**, i.e. deciding how and when a crowd platform or community should be activated or deactivated, based on its aggregated performance.

For monitoring task execution, a data structure called **control mart** was introduced in [5]; control marts are analogous to data marts used for data warehousing [**?**], as their central entity represents elementary activities, surrounded by dimensions representing objects, tasks,

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

and performers. 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 proven in the context of a well-organised computational framework. For a deeper description of the rule grammar and structure see our previous work [5]. Section 5 provide examples of control rules, described in the context of their implementation within the Crowdsearcher framework.

3.4.1 Object Control

Object control monitors the responses for each objects, assembling them into an **object state**. When enough evaluations are available, an object is *closed*; when evaluations are not converging, an object can be *replanned*. Normally, an object is closed when agreeing answers go beyond a given threshold or form a majority, and several majority schemes are possible (e.g., 3/5/7 agreeing votes).

3.4.2 Task Control

Task control monitors the responses received for all the objects of the task, objects, assembling them into a **task state**. When enough evaluations are available, a task is *closed*; when evaluations are not converging, a task can be *replanned*. Task quality can be progressively monitored and corrective actions may be taken, including adding performers or changing the involved platform or community.

3.4.3 Performer Control

Performer control allocates responders to performers, assembling them into a *performer state*; such activity guarantees that performers are rewarded by the system. The most important aspect of performer control is **spammer detection**, i.e. determining the performers who produce wrong answers in a statistically significant way. A typical strategy for spammer detection in crowdsourcing is to use *golden questions* (or honeypots), i.e. tasks whose responses is predefined by experts, and then test performers against them [**?**]. In our approach, this requires adding properties to the object schema, so that the corresponding items can be identified and used. Spammer detection results in excluding them from future assignments. In addition, the effects of their activity may be removed from the state of affected objects.

3.4.4 Platform and Community Control

Platform and community control allows adapting a crowd-based application; it requires **Replanning**, i.e. the process of generating new microtasks; and **Re-invitation**, i.e. the process of generating new invitation messages for existing or replanned microtasks.

Crowd-based applications can be deployed over multiple systems or communities and can dynamically adapt to their behaviour. With **Cross-Platform Interoperability**, applications change the underlying social network or crowdsourcing platforms, e.g., from Facebook to Twitter or to AMT. With **Cross-Community Interoperability**, applications change the performers' community, e.g., from the students to the professors of a university. Adaptation can be applied at different granularity levels:

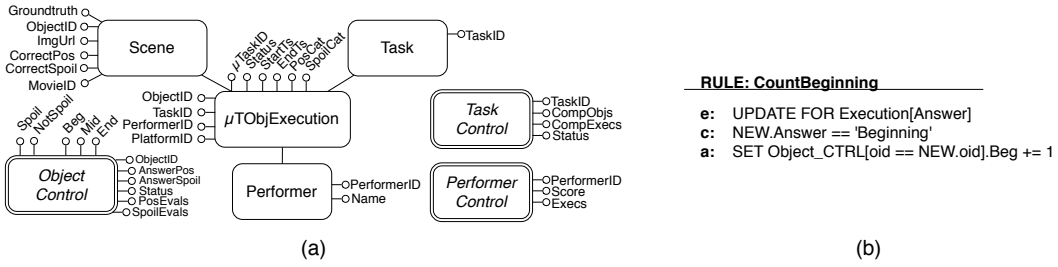


Fig. 9. (a) Example of control mart for the tasks of Figure 7; (b) Example of control rule that updates the number of responses in the *Position of Scenes* task.

Task granularity, when the re-planning or re-invitation occur for the whole task.

Object granularity, when the re-planning or re-invitation is focused on one (or a few) objects.

Adaptation at execution time requires a **switch-over**, which denotes the time interval during which adaptation occurs. A switch-over can be **continuous** (i.e., results from previously involved platforms or communities continue to be accepted), **instantaneous** (i.e., results of initiated tasks of previous platforms/communities are considered if they were produced before the switch-over and either blocked or disregarded afterwards), or **reset** (i.e., all the results from previous platform or community are disregarded).

3.4.5 Workflow Control

Workflow control, as the name suggests, allows the coordination of tasks in a crowd workflow. It builds upon the same rule based mechanism described in [5] and exploited for the previously described control aspects.

Control is performed 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 managed through rules that trigger the creation of new micro-tasks and their executions, upon production of new results.

Figure 9(a) shows a sample control mart for the three tasks tightly connected *Position of Scene* and *Order of Scene* 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 9(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 performs 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 accessed with the dot notation).

Loosely coupled tasks have independent control marts, hence their interaction is more

complex. As loosely couple tasks do not share a data space, we rely on an event-based publish-subscribe communication model: upon changes in its state (e.g. task termination, object evaluation completed), a task emits an event, which carries information about the changed state (e.g. the ID of the completed object), and it is then captured by the subscribed tasks. Each subscribed task reacts to the event according to the business logic contained in its data manipulation stage. This event-based mechanism is described in more details in Section 5.

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 four classes:

- **Intra-Task Patterns.** They are typically used for executing a complex, single 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 to describe a crowd application from a *control-flow* perspective, to represent the execution order of tasks.

- **Crowd-flow Patterns.** They are used to structure the solution of problems that require the coordination of different tasks, each demanding for 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 their results.

- **Workflow Patterns.** (e.g. sequence, parallelism, join synchronisation, etc.) are common in disciplines like (business) process modelling and service orchestration; therefore, we omit their description and refer the reader to relevant works in literature [**?**][**?**].

Next, we analyse the other pattern class separately.

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 10 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 stratgy*, 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.

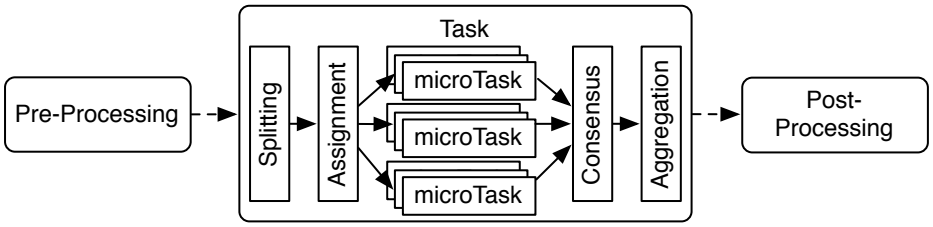


Fig. 10. Building blocks of an Intra-Task Pattern.

4.1.1 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, referred to as *consensus* or *agreement patterns*, assume the existence of a population of workers that collaboratively produce the correct answer. Typical consensus patterns are: ***1)*** **StaticAgreement** [5]: a simple approach that 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; ***2)*** **MajorityVoting** [**?**]: an agreement approach that accepts a response only if a given number of performers produce the same response, given a fixed number of total executions; ***3)*** **ExpectationMaximisation** [**?**]: an adaptive approach that alternates between estimating correct answers from task parameters (e.g. complexity), and estimating task parameters from the estimated answers, eventually converging to maximum-likelihood parameter and answer values.

4.1.2 Join Patterns

Crowd join patterns, studied in [**?**], are used to build an equality relationship between matching objects in the context of crowdsourcing tasks. Examples of join patterns are: ***1)*** **SimpleJoin**, which 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; ***2)*** **OneToManyJoin** is a simple variant that includes in the same microtask one left object and several right candidates to be joined; ***3)*** **ManyToManyJoin** includes in the same microtask several candidate pairs to be joined.

4.1.3 Sort Patterns

Sort patterns determine the total ordering of a set of input objects. Examples of sort patterns are: ***1)*** **SortByGrouping** [**?**] orders a large set of objects by aggregating the results of the ordering of several small subsets of them; ***2)*** **SortByScoring** [**?**] asks performers to rate each item in the dataset according to a numerical scale; **SortByLiking** [5] 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. ***3)*** **SortByPairElection** [5] asks workers to perform a pairwise comparison of two items and indicate which one they like most. Ranking algorithms (such as Elo [**?**]) calculate their ordering. ***4)*** **SortByTournament** [**?**],

presents to performers a tournament-like structure of sort tasks; each tournament elects its champions that progress to the next level, eventually converging to a final order.

4.1.4 Grouping Patterns

Grouping patterns are used in order to classify or clustered several objects according to their properties. Examples of sort patterns are: ***1)*** **GroupingByPredefinedClasses**[**?**] occurs when workers are provided with a set of known classes. ***2)*** **GroupingByPreference** [**?**] 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.

4.1.5 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 common patterns for performer control include: ***1)*** **QualificationQuestion** [**?**], at the beginning of a micro-task, for assessing the workers expertise and deciding whether to accept his contribution or not. ***2)*** **GoldStandard**, [**?**] for both training and assessing worker's quality through a initial subtask whose answers are known (they belong to the so-called *gold truth*. ***3)*** **Majority-Comparison**, [5] for assessing performers' quality against responses of the majority of other performers, when no gold truth is available.

4.1.6 Community Allocation Patterns

These patterns enable the spawning of crowdsourcing tasks upon multiple communities of performers, thus leveraging the peculiar characteristics and capabilities of the community members. By community we mean a set of people that share common interests (e.g., football club fans, opera amateurs,...), have some common feature (e.g., leaving in the same country or city, or holding the same degree title) or belong to a common recognised entity (e.g., employee in an office, workgroup or employer; students in a university; professionals in a professional association; ...). Leveraging communities for crowdsourcing includes both the possibility of statically determining the target communities of performers, and also dynamically changing them, taking into account how the community members behave when responding to task assignments. Dynamic adaptation of crowdsourcing campaigns to community behaviour is particularly relevant because it can be very effective for obtaining answers from communities, with very different size, precision, delay and cost, by exploiting the social networking relations and the features of the task. Examples of community allocation patterns are: ***1)*** **Expertise Level**, i.e. the re-planning of a crowdsourcing task to different platforms, each featuring performers with a different level of expertise w.r.t. the task's topic. For instance, a task could be re-planned from a human computation platform to a social network when the performers of the former community provide unsatisfactory quality performance. ***2)*** **Crowd Capacity Relocation**, i.e. re-planning due to unsatisfactory execution time performance, i.e. when a task initially planned for execution in a social network takes too long to be completed.

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 10. *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* are

typically devoted to the refinement or transformation of the task outputs into their final form. Examples of auxiliary patterns are: ***1)*** **PruningPattern** [**?**], 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. ***2)*** **TieBreakPattern** [**?**], 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 Crowd-Flow 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 **Crowd-Flow Pattern** is a workflow of heterogeneous crowdsourcing tasks with co-ordinated goals. Several crow-flow patterns defined in the literature are next reviewed; they are comparatively shown in Figure 11.

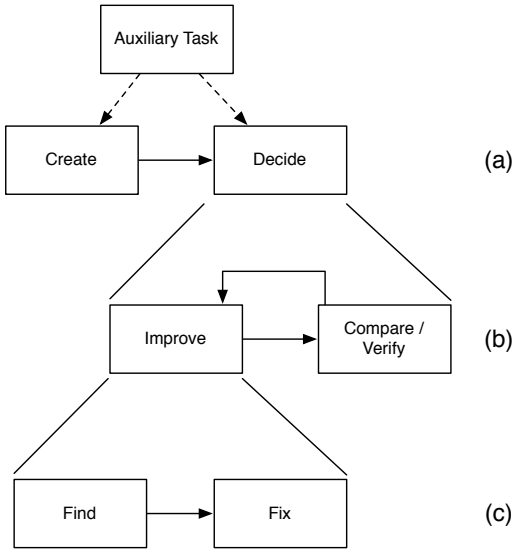


Fig. 11. Template for complex task patterns.

Create/Decide [**?**], shown in Figure 11(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. The *create* and *decide* steps can be combined in arbitrary ways. For instance, to provide an alternative characterisation of the decision process, multiple executions of the *create* task can be performed in a *sequential* or *parallel* fashion. In such cases, the content generation task is generated multiple times, and workers are asked to compare the content generated by previous tasks in order to make a decision about which content to produce in output. Such a comparison can be performed sequentially or in parallel at multiple stages.

Improve/Compare [**?**], shown in Figure 11(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.

Find/Fix/Verify [1], shown in Figure 11(c), further decomposes the *improve* step, by splitting the task of finding potential improvements from the task of actually implementing them.

4.4 Auxiliary Crowd-Flow Patterns

Auxiliary tasks can be designed to support the creation and/or the decision tasks. Examples of auxiliary crowd-flow patterns are: ***1)*** **AnswerBySuggestion** [**?**]: 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. ***2)*** **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 Implementation

The design process is supported by a platform prototype, called CrowdSearcher.[e] The software is written in JavaScript and running on the *Node.js*[f] server; a demonstration video of the platform is available on YouTube.[g] Figure 12 depicts the architectural organisation of CrowdSearcher: the main modules, described in details in the next sections, are the *Configuration And Management Interfaces*, the *Task Execution Framework*, and the *Reactive Environment*. CrowdSearcher has been designed with modularity and extensibility in mind: therefore, a set of APIs simplify it evolution with new functionalities, but also the integration with third-party application.

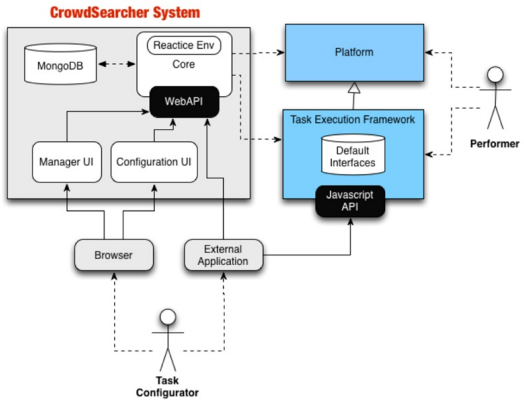


Fig. 12. Architecture of the Crowdsearcher platform.

5.1 Configuration And Management Interfaces

CrowdSearcher include several interfaces to support the specifying and monitoring of crowd-based applications through a step-by-step approach, where each step is supported by suitable wizards. Figure 13 shows four design steps for our running case study: *1*) Definition of a basic task, the analysis of scenes for the movie *007 Skyfall* using a *fuzzyclassify* method, subdividing scenes in three classes (*begin, middle, end*) and with *spoiler detection*. *2*) Platform selection, selecting *Amazon Mechanical Turk (AMT)* with a given *template interface, url, price* (1 cent) and *duration* (60 minutes). *3*) Object control strategy, with the *level of agreement*, the *number*

[e] http://crowdsearcher.search-computing.com
[f] http://nodejs.org
[g] http://www.youtube.com/watch?v=wX8Dvtwyd8s

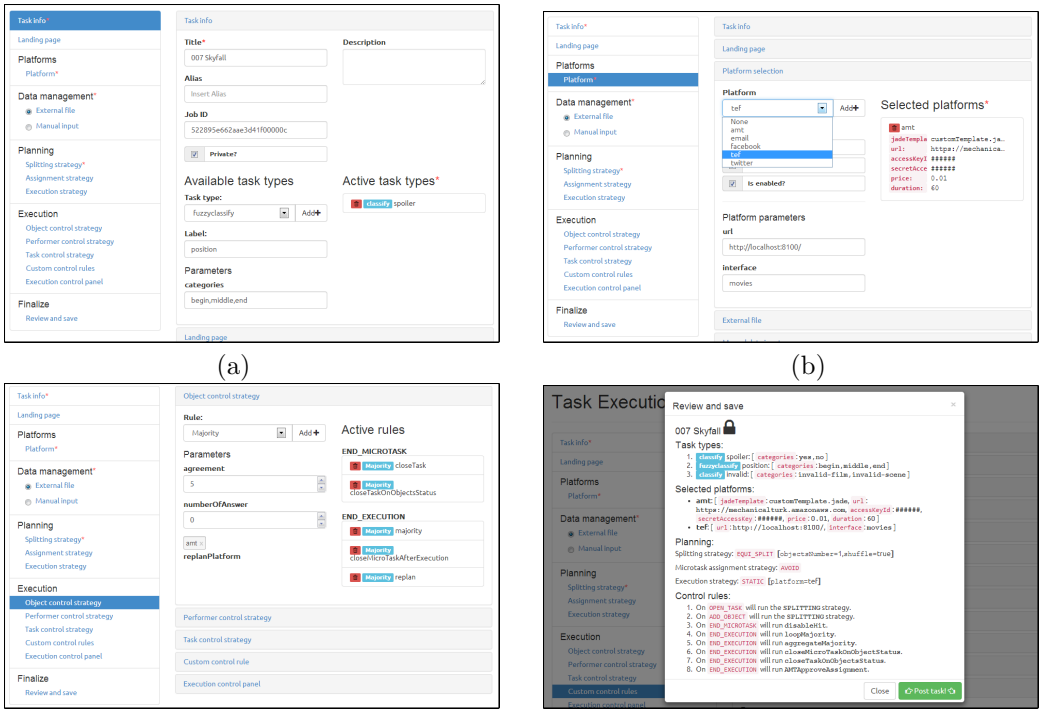


Fig. 13. Four design steps: (a) task design; (b) platform selection; (c) object control; (d) task summary (including the list of control rules).

of awaited answers, the *platform where to replan. 4*) Task configuration summary (including list of control rules, see later), to be used for a final control before deployment.

Note that the design framework supports a *declarative style of control*, whereas a designer can indicate how objects, tasks, performers, platforms and communities should be controlled. These declarations are automatically translated into active rules, as discussed in Section 5.3.

5.2 Task Execution Framework

CrowdSearcher offers a plug-in environment for transparently deploying crowd-based applications over several social networks and crowdsourcing platforms. 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 or Facebook. Alternative approaches for the implementation of operations on these crowd-based systems are discussed on [4]. 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.

Figure 14 shows the platforms currently supported by the CrowdSearcher. In particular, *Invitation* indicates if our tool is able to use the platform for inviting performers to the task, *Execution (TEF UI)* indicates if the TEF interface is embedded in the platform environment, and *Execution (Native UI)* indicates if CrowdSearcher is able to deploy the task using the

Platform	Invitation	Execution (TEF UI)	Execution (Native UI)
AMT	N/A	yes	yes
Facebook	yes	yes	yes
Twitter	yes	N/A	N/A

Fig. 14. Deployment platforms supported by the tool. "N/A" means that the particular type of deploy is not supported by the platform itself.

native interface of some platform (for example deploying the task on AMT using its native proprietary interface).

5.3 Reactive Environment

Most crowdsourcing systems only provide limited and predefined controls; in contrast, our approach provides fine-level, powerful and flexible controls. We define high-level abstractions for declaring task control, as well as low-level rules for implementing such control, which typically encompasses the evaluation of arbitrary conditions on objects, performers, tasks, platforms, and communities.

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 [5].

Task interactions are implemented differently depending on whether interacting tasks are tightly coupled or loosely coupled. 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.

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 all 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.

Rules are triggered by changes in control tables and express the result control logic, that can be specified through high level directives. Typical examples of rules consider how to decide that an object is closed or that a performer is a spammer.

Objects are closed when they are associated with *enough* evaluations to provide a conclusive response, i.e. a majority of equal answers. The smallest possible majority calls for two equal answers, and is recognised by Rule 1.

Rule 1 MajorityResultRule.

```
e: UPDATE FOR Object_CTRL
c: (NEW.Begin== 2) or (NEW.Middle == 2) or (NEW.End == 2)
a: SET Scene[oid==NEW.oid].Position = NEW.Answer,
   SET Task_CTRL[tid==NEW.tid].CompObj += 1
```

This rule is triggered by any change of the object control table, and simply checks that one of the attributes `Begin`, `Middle` or `End` is equal to 2; then it sets the scene position equal to the current answer and increases the number of completed objects in `Task_CTRL`.

Of course, different majority conditions are possible, which can be arbitrarily complex and depend also on the number of evaluations, e.g.,

```
C1: (Eval>5) and ((Begin>0.5*Middle) or (Middle>0.5*Begin) ...)
C2: (Eval>10) and ((Begin>0.8*Middle) or (Middle>0.8*Begin)...)
C3:  Eval>15
```

The above cases denote three distinct rule conditions; they can either be embedded into three different rules or their disjunction could be embedded into a single rule. The effect is to close the object as soon as one of the three conditions is true. With enough micro-task completions, the condition `Eval>15` becomes eventually true.

Another case of control rule is identification of spammers. Performers are identified as spammers when they are associated with *enough* wrong answers. A simple rule for identifying spammers is:

Rule 2 SpammerIdentificationRule.

```
e: UPDATE FOR Performer_CTRL
c: (NEW.Eval > 10) and (NEW.Wrong > New.Right)
a: SET Performer[Pid==NEW.Pid].Status = 'Spammer'
```

This rule is triggered by any change of the performer control table, and simply checks that after 10 evaluations the number of wrong answers exceeds the number of right answers; then it sets the performer's status to `'Spammer'`.

Of course, different spammer identification conditions are possible, e.g., condition C1 identifies as spammer whoever performs 4 errors, condition C2 selects as spammer anyone who has given more than 20% of wrong answers, condition C3 uses two thresholds.

```
C1: Wrong == 4
C2: Wrong > 0.2*Eval
C3: ((Eval>10) and (Wrong>3)) or (Wrong>Right)
```

In the control framework, relational rules are translated into JavaScript; triggering is modelled through internal platform events. Precedence between rules reacting to the same event is implicitly obtained by orderly translating all such rules into a single script. For example, rule 3 computes majority of answers for the classify operation.

numberOfAnswers is the minimum number of answers needed and agreement is the number of performers that must agree on a particular category. The rule has three main parts: *(i)* lines 15–20 updates the control variable (total number of answer and the count of the selected

category); *(ii)* lines 23–41 select the category that currently has the higher count and set the control variables; *(iii)* lines 43–49 close the current object if termination conditions are met.

Rule 3 Calculation of the majority for a classify operation.

```
var performRule = function( data, config, callback){
  // Array of categories
  var categories = data.task.operation.params.categories;
  // Minimum number of answers
  var numberOfAnswers = config.numberOfAnswers;
  // Agreement needed
  var agreements = config.agreement;
  var applyMajority = function(annotation,callback){
    var object = annotation.object;
    // Updating the metadata
    var count = object.getMetadata('count');
    object.setMetadata('count',count+1);
    var selectedCategoryCount = object.getMetadata(annotation.response);
    object.setMetadata(annotation.response,selectedCategoryCount+1);
    // Build the data structure [category, count]
    var categoriesMetadata = [];
    _.each(categories,function(category){
      var count = {
      category:category,
      count: object.getMetadata(category)
    };
    categoriesMetadata.push(count);
    // Selecting the category with maximum count
    var max = _.max(categoriesMetadata,function(categoryCount){
      return categoryCount.count;
    });
    // Verifying that the maximum is unique
    var otherMax = _.where(categoriesMetadata,{count:max.count});
    if(otherMax.length==1){
      object.setMetadata('result',max.category);
    }
    // Checking if the object should be closed
    // If numberOfAnswers is equal to 0, then ignore the parameter
    if(count === numberOfAnswers || numberOfAnswers=== 0){
      if(max.count >= agreement){
      object.setMetadata('status','CLOSED');
      }
    }
    return object.save(domain.bind(callback));
  )};
    //Call the applyMajority function of each annotation
(object)
  return async.eachSeries(annotations,domain.bind(applyMajority),callback);
};

var params = {
  agreement: ['number'],
  numberOfAnswers: ['number']
};
```

6 Experience and Evaluation

The implementation of Crowdsearcher granted the opportunity to put our approach at work in the development of several real-world applications. In Section 6.1 we will describe some

of our experience in modelling real-world case studies. Section 6.2 and Section 6.3 report on the experiments conducted with 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.

6.1 Approach in use

Our approach has been used in several real-world projects, for instance:

Multimedia Analysis and Search [7][9]: in the context of the FP7 CUbRIK Integrating Project, we introduced a conceptual and architectural framework for addressing the design, execution and verification of tasks by a crowd of performers. By combining CrowdSearcher with an infrastructure for multimedia analysis, we created several applications (e.g. trademark and logo detection in video) that demonstrates how the contribution of (expert) crowds can improve the recall of state-of-the-art traditional algorithms, with no loss in terms of precision.

Cultural Heritage Items Annotation [13]: in the context of the COMMIT Dutch national program, several experiments of crowdsourced cultural heritage collection annotations were performed. In collaboration with several institutions such as the Rijkmuseum Amsterdam, we instrumented several experiments aimed at showing the performance of alternative crowd-flow configurations for expert annotation of artworks.

In addition, we instrumented several experiment scenarios, aimed at supporting our research activities, while validating the applicability and flexibility of our approach. For instance:

Politician party [5]:In this experiment we asked the crowd to classify the political affiliation of 30 members of the Italian parliament. To single performer is presented a set of photos and names and has to match the single politician to the correct political party.

Politician law [5]: In this experiment we presented photos of 50 members of the Italian parliament and asked the crowd to indicate if they have ever been accused, prosecuted or convicted. Each performer sees, in a fixed amount of time, a number of photos which raises as a function of the performers ability, and, after he give his answer, the system presents a report with correct answers and the ranking of the other performers.

Politician order [5]: The objective of this experiment was to produce the total ranking of 25 politicians. At each performer is presented a pair of politicians and is asked to choose the one he likes the most.

Model search (1) [2]: In this experiment we asked the crowd to evaluate the results of a query performed on a model repository. Given the query, and two possible results, the performers had to choose which one is better.

Model search (2) [3]: In this experiment we asked the crowd to evaluate the ranked results of a query performed over a model repository. Given the query and two possible ranks, the performer had to decide which one was better.

Transportation: In this experiment we asked the crowd to validate the classification of tweet related to public transportation made by an automatic tool. The performers had to evaluate the correctness of the topic, geo-localization and polarity of the tweets.

Movie Scenes [6]: In this experiment we asked the crowd to classify images taken from movie scenes. Each performer had to tell if an image belonged to the initial, middle or final part of the film, and, in the latter two cases, if the image was a spoiler.

Movie actors [6]: In this experiment we asked the crowd to identify actor in movie scenes. In particular this scenario was divided in two parts: in the first the performers had to insert

Experiment	Dataset preparation	UI generation	Monitoring	Tuning
Politicians Party	1.5	4	5	1
Politicians Law	1.5	2	7	1.5
Politicians Order	1.5	2	3	1
Model Search 1	3	3	0 (*)	1
Model Search 2	3	3	0 (*)	1.5
Transportation.	1	5	0 (*)	1
Movies Scenes	1	4	2	1
Movies Actors	2	3.5	0.5 (*)	0.5
Image Select	1.5	4.5	1.5	1

Fig. 15. Development effort for different applications (man/days). (*) = high reuse of existing rules.

the name of the actors present in the image and in the second they had to validate the answers given by the others.

Professors images [8]: In this experiment we asked the crowd to evaluate of relevance images about the professor of retrieved through the Google Image API. We asked the performers to specify whether each image represents the professor himself, some relevant people or places, other related materials (papers, slides, graphs or technical materials), or it is not relevant at all.

Among the various aspects we studied, we considered the cost of development of the different applications. Figure 15 reports the approximate development effort of nine recent application. Note that data preparation and UI generation (ad hoc) were required regardless of the adopted method. Our approach to monitoring required large efforts for the first applications, which was well compensated by a high reuse of rules in the subsequent applications. Note also that our method enables fast prototyping of applications in the small scale, with small crowds who give interaction feedbacks; tuning is quite efficient, as it can be done by changing configuration parameters from within the design framework.

6.2 Experiments With Intra-Task and Crowd-Flow Patterns

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 ground-truth dataset regarding temporal positioning and actors playing in the scenes. Each configuration stresses a different combination of *Intra-Task* and *Crowd-Flow* patterns. We compare cost and quality of executions for different workflow configurations.

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 spoilers. Two variants of the scenario have been tested, as shown in Figure 16: 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.

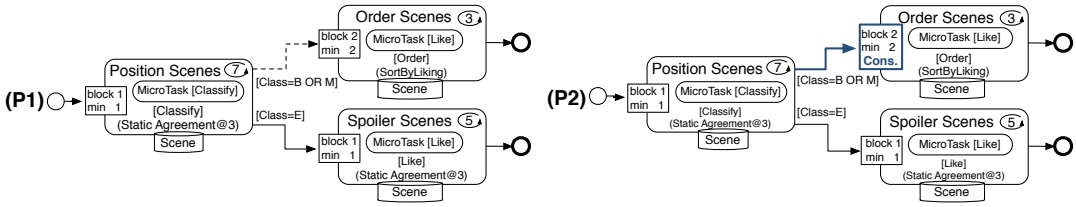


Fig. 16. Flow variants for the Positioning scenario.

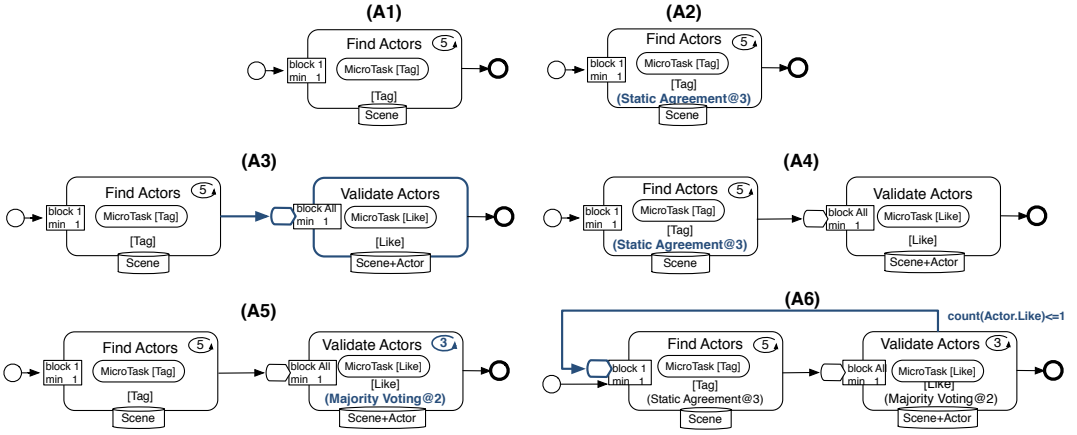


Fig. 17. Flow variants for the Actor scenario.

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.

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 transforms 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 17, 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;
- 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.

	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

Fig. 18. *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).

	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

Fig. 19. *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).

6.2.1 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. Figure 18 and Figure 19 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 20(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. Figure 21 shows the precision of the classification results of task *Position Scenes*. Figure 22 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 figures show that the attained quality was not significantly influenced by the different task activation modes.

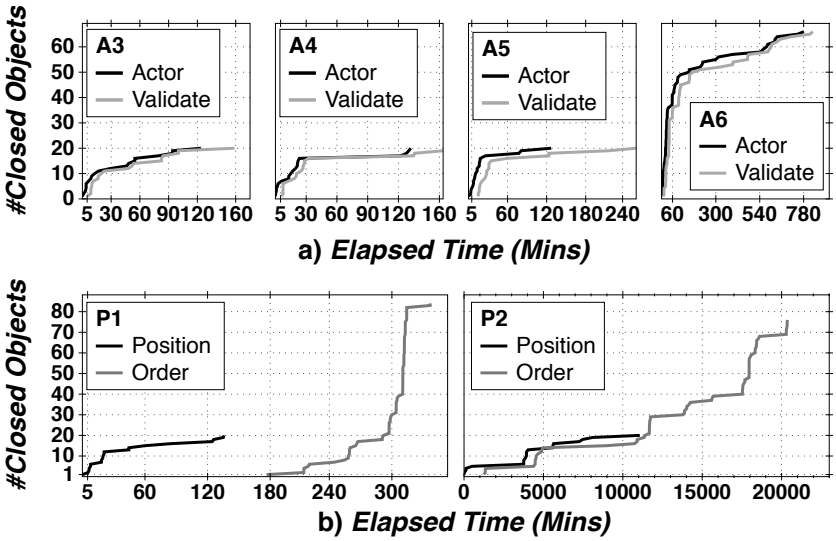


Fig. 20. Temporal distributions of closed objects.

In summary, we didn't note a different behaviour due to streaming. One possible reason is that in the batch configuration the entire set of assignments are 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 [10]), while in a stream execution a small number of assignments is posted on AMT at every closing event of objects from the previous tasks.

	PBeginning	P Middle	P End
P1	0.50	1	0.11
P2	0.50	0.80	0.33

Fig. 21. Scenario 1 (Positioning): Precision of the *Position Scenes* classification task.

	Spearman Beginning	Spearman Middle
P1	0.500	0.543
P2	0.900	0.517

Fig. 22. Scenario 1 (Positioning): Spearman's rank correlation for the *Order Scenes* order task.

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

Fig. 23. Scenario 2 (Actor): Precision, Recall, and F-score of the six tested configurations.

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 20(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 20(c)) is significantly slower due to the feedback loop, which also generates a much higher cost of the campaign, as reported in Figure 18. Indeed, due to the feedback, many tasks are executed several times before converging to validated results.

Quality. Figure 23 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 to be 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.

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.

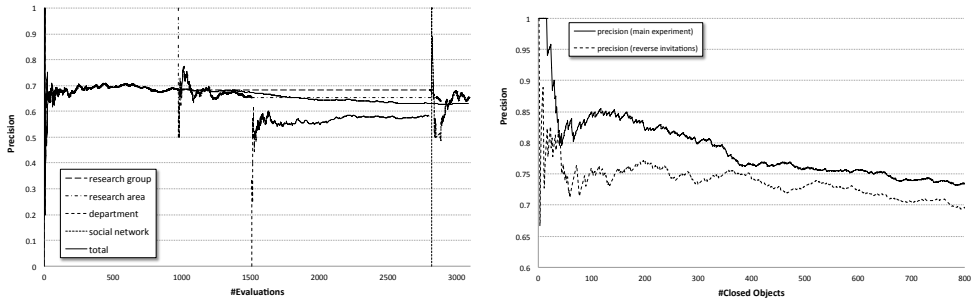
6.3 Experiment With Community Allocation Patterns

The flexibility of our approach allows to cover dynamic planning of tasks also across different communities of experts. This means to switch the focus from cross-platform crowdsourcing to community-based crowdsourcing applications, In a separate experiment, we focused on community allocation patterns through a scenario concerned with *image classification*. The dataset consists of images about professors of our department retrieved through the Google Image API. In the crowdsourcing campaign we asked the performers to specify whether each image represents the professor himself, some relevant people or places, other related materials

(papers, slides, graphs or technical materials), or it is not relevant at all. The experimental settings are thoroughly described in [8]: we selected 16 professors within two research groups in our department (DB and AI groups) and we downloaded the top 50 images returned by the Google Image API for each query; each microtask consisted of evaluating 5 images regarding a professor. Results are considered accepted (and thus the corresponding object is closed) when some agreement level on the class of the image is reached among performers. We defined 3 types of communities as: the research group of the professor (e.g., Databases); the research area containing the group (e.g., Computer Science); and the whole department (which accounts for more than 600 people in different areas).

We devised two experiments addressing the **Expertise Level** community allocation pattern. In the first one, named *inside-out*, we started with invitations to experts, e.g. people the same groups as the professor (DB and AI), and then expanded invitations to Computer Science, then to the whole Department, and finally to open social networks (Alumni and PhDs communities on Facebook and Linkedin). In the second one, named *outside-in*, we started with the Department members, then restricting to Computer Scientists, and finally to the group's members. Our assumption is that researchers that work closer to the person mentioned in the query know him better and are more able to recognise relevant images.

All invitations (except for the social networks in the first experiment) were automatically sent via email. The communities were not overlapping: every performer received only one invitation. The members of the Department, of Computer Science area, and of the DB Group were randomly split into two sets. Invitations have been implemented as a set of dynamic, cross-community interoperability steps, with task granularity and with continuous switch-overs starting one working day after a community was idle (stopped to produce results). A rule invites the performers of a broader community when the current crowd ceases to produce answers, for instance after one hour of *idle* time, i.e. when the last execution occurred more than one hour ago in the smaller community (DB-Group). A rule focuses on re-planning the crowdsourcing task on a specific object when the performers of a community are in disagreement, e.g., if there are votes on every category of the classify operation.



(a) (b)

Fig. 24. Precision of evaluations by community (a) and comparison of the precision for the inside-out and outside-in approaches (b).

Figure 24 (a) shows the precision of evaluations by community and Figure 24(b) shows the final precision on closed objects. Figure 24(b) compares also the precisions of the *inside-out* and *outside-in* experiments, and shows that former performs better than the latter in terms of quality of results. This is quite evident in the initial phases (when the first half of the objects close), as the performance of experts (research group) is much higher than performance of the people of the rest of department.

7 Conclusions

In this paper, we presented a comprehensive approach to the modeling, design, and pattern-based specification of crowd-based workflows. We discussed how crowd-based tasks communicate by means of stream-based or batch data flows, and we defined the concepts of loose and tight coupling. We also discussed known patterns that are used to create crowd-based computations either within a task (in order to solve a complex task by using simple operations) or between tasks (in order to decompose a cognitively complex task into a progression of simple tasks, possibly with retroactions). We showed how the workflow model is translated to executable specifications which are based upon control data, reactive rules, and event-based notifications.

Part of these concepts has been implemented in the CrowdSearcher framework that supports declarative style for defining crowdsourcing applications and provides runtime environment for transparently deploying crowd-based applications over several social networks and crowdsourcing platforms.

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

As future work, we plan to broaden our research to studying patterns not only focusing on the crowdsourcing. We will analyze patterns for engaging social networks users in content production and enrichment campaigns, as well as in brand awareness and information diffusion. A typical use case scenario is defining games or challenges and trying different methods for keeping the users involved in time. We will also address the problem of monitoring the execution of a crowdsourcing task, by providing advanced dashboards where to compare different task configurations and to see how the crowd reacts to different solicitation events.

References

1. S. Abiteboul and V. Vianu. Collaborative data-driven workflows: think global, act local. In *Proceedings of the 32nd symposium on Principles of database systems*, pages 91–102, New York, NY, USA, États-Unis, 2013. ACM.
2. G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734–749, 2005.
3. S. Ahmad, A. Battle, Z. Malkani, and S. Kamvar. The jabberwocky programming environment for structured social computing. In *UIST '11*, pages 53–64. ACM, 2011.
4. O. Alonso, D. E. Rose, and B. Stewart. Crowdsourcing for relevance evaluation. *SIGIR Forum*, 42(2):9–15, Nov. 2008.
5. M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich. Soylent: a word processor with a crowd inside. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 313–322, New York, NY, USA, 2010. ACM.
6. B. Bislimovska, A. Bozzon, M. Brambilla, and P. Fraternali. Graph-based search over web application model repositories. In S. Auer, O. Díaz, and G. Papadopoulos, editors, *Web Engineering*, volume 6757 of *Lecture Notes in Computer Science*, pages 90–104. Springer Berlin Heidelberg, 2011.
7. B. Bislimovska, A. Bozzon, M. Brambilla, and P. Fraternali. Textual and content-based search in repositories of web application models. *ACM Trans. Web*, 8(2):11:1–11:47, Mar. 2014.
8. A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with crowdsearcher. In *21st Int.l Conf. on World Wide Web 2012*, WWW '12, pages 1009–1018. ACM, 2012.
9. A. Bozzon, M. Brambilla, S. Ceri, and A. Mauri. Reactive crowdsourcing. In *22nd World Wide Web Conf.*, WWW '13, pages 153–164, 2013.

10. A. Bozzon, M. Brambilla, S. Ceri, A. Mauri, and R. Volonterio. Pattern-based specification of crowdsourcing applications. In *Web Engineering, 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings*, pages 218–235, 2014.
11. A. Bozzon, M. Brambilla, S. Ceri, M. Silvestri, and G. Vesci. Choosing the right crowd: expert finding in social networks. In *16th International Conference on Extending Database Technology*, EDBT '13, pages 637–648, New York, NY, USA, 2013. ACM.
12. A. Bozzon, I. Catallo, E. Ciceri, P. Fraternali, D. Martinenghi, and M. Tagliasacchi. A framework for crowdsourced multimedia processing and querying. In *Proceedings of the First International Workshop on Crowdsourcing Web Search, Lyon, France, April 17, 2012*, pages 42–47, 2012.
13. M. Brambilla, S. Ceri, A. Mauri, and R. Volonterio. Community-based crowdsourcing. In C. Chung, A. Z. Broder, K. Shim, and T. Suel, editors, *SOCM Workshop, 23rd International World Wide Web Conference, WWW '14, Seoul, April 7-11, 2014, Companion Volume*, pages 891–896. ACM, 2014.
14. S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *Proceedings of the 16th International Conference on Database Theory*, ICDT '13, pages 225–236, New York, NY, USA, 2013. ACM.
15. A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY*, 39(1):1–38, 1977.
16. D. E. Difallah, G. Demartini, and P. Cudré-Mauroux. Pick-a-crowd: tell me what you like, and i'll tell you what to do. In *22nd international conference on World Wide Web*, WWW '13, pages 367–374, 2013.
17. A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, Apr. 2011.
18. A. E. Elo. *The rating of chessplayers, past and present.* Arco Pub., New York, 1978.
19. M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *ACM SIGMOD 2011*, pages 61–72. ACM, 2011.
20. P. Fraternali, M. Tagliasacchi, D. Martinenghi, A. Bozzon, I. Catallo, E. Ciceri, F. Nucci, V. Croce, I. S. Altingovde, W. Siberski, F. Giunchiglia, W. Nejdl, M. Larson, E. Izquierdo, P. Daras, O. Chrons, R. Traphoener, B. Decker, J. Lomas, P. Aichroth, J. Novak, G. Sillaume, F. S. Figueroa, and C. Salas-Parra. The cubrik project: Human-enhanced time-aware multimedia search. In *Proceedings of the 21st International Conference Companion on World Wide Web*, WWW '12 Companion, pages 259–262, New York, NY, USA, 2012. ACM.
21. C. Grady and M. Lease. Crowdsourcing document relevance assessment with mechanical turk. In *Proceedings of the NAACL HLT 2010 Workshop*, CSLDAMT '10, pages 172–179, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.
22. W. H. Inmon. *Building the Data Warehouse.* John Wiley & Sons, Inc., New York, NY, USA, 1992.
23. S. Kochhar, S. Mazzocchi, and P. Paritosh. The anatomy of a large-scale human computation engine. In *HCOMP '10*, pages 10–17. ACM, 2010.
24. E. Law and L. von Ahn. *Human Computation.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2011.
25. C. H. Lin, Mausam, and D. S. Weld. Crowdsourcing control: Moving beyond multiple choice. In *UAI*, pages 491–500, 2012.
26. G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *HCOMP '09*, pages 29–30. ACM, 2009.
27. G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Exploring iterative and parallel human computation processes. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 68–76, New York, NY, USA, 2010. ACM.
28. A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *Proc. VLDB Endow.*, 5(1):13–24, Sept. 2011.
29. A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR 2011*, pages 211–214. www.cidrdb.org, Jan. 2011.

30. P. Minder and A. Bernstein. How to translate a book within an hour: towards general purpose programmable human computers with crowdlang. In *WebScience 2012*, pages 209–212, Evanston, IL, USA, June 2012. ACM.
31. M. Minsky. *The society of mind*. Simon & Schuster, Inc., New York, NY, USA, 1986.
32. A. Nigam and N. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.
33. S. Nowak and S. Rüger. 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 '10, pages 557–566, New York, NY, USA, 2010. ACM.
34. O. M. G. (OMG). Business process model and notation (bpmn) version 2.0. Technical report, jan 2011.
35. J. Oosterman, A. Nottamkandath, C. Dijkshoorn, A. Bozzon, G.-J. Houben, and L. Aroyo. Crowdsourcing knowledge-intensive tasks in cultural heritage. In *Proceedings of the 2014 ACM Conference on Web Science*, WebSci '14, pages 267–268, New York, NY, USA, 2014. ACM.
36. A. Parameswaran, M. H. Teh, H. Garcia-Molina, and J. Widom. Datasift: An expressive and accurate crowd-powered search toolkit. In *1st Conf. on Human Computation and Crowdsourcing (HCOMP)*, 2013.
37. A. G. Parameswaran and N. Polyzotis. Answering queries using humans, algorithms and databases. In *CIDR 2011*, pages 160–166, Asilomar, CA, USA, January 2011.
38. H. Park, R. Pang, A. G. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. *PVLDB*, 5(12):1990–1993, 2012.
39. V. S. Sheng, F. Provost, and P. G. Ipeirotis. Get another label? improving data quality and data mining using multiple, noisy labelers. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 614–622, New York, NY, USA, 2008. ACM.
40. W. M. P. van der Aalst. Business process management demystified: A tutorial on models, systems and standards for workflow management. *Lectures on Concurrency and Petri Nets: Advances in Petri Nets — LNCS Vol. 3098*, pages 1–65, June 2004. InternalNote: Submitted by: hr.
41. P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW '12*, pages 989–998, New York, NY, USA, 2012. ACM.
42. J. Wang and A. Kumar. A framework for document-driven workflow systems. In *Proceedings of the 3rd international conference on Business Process Management*, BPM'05, pages 285–301, Berlin, Heidelberg, 2005. Springer-Verlag.
43. J. Yang, C. Hauff, A. Bozzon, and G. Houben. Asking the right question in collaborative q&a systems. In *25th ACM Conference on Hypertext and Social Media, HT '14, Santiago, Chile, September 1-4, 2014*, pages 179–189, 2014.
44. J. Yang, K. Tao, A. Bozzon, and G. Houben. Sparrows and owls: Characterisation of expert behaviour in stackoverflow. In *User Modeling, Adaptation, and Personalization - 22nd International Conference, UMAP 2014, Aalborg, Denmark, July 7-11, 2014. Proceedings*, pages 266–277, 2014.