Business Processes for the Crowd Computer

Pavel Kucherbaev[1], Stefano Tranquillini[1], Florian Daniel[1], Fabio Casati[1], Maurizio Marchese[1], Marco Brambilla[2], and Piero Fraternali[2]

[1] University of Trento, Via Sommarive 5, 39123 Povo (TN), Italy
surname@disi.unitn.it
[2] Politecnico di Milano, Piazza L. da Vinci 32, 20133 Milano, Italy
name.surname@polimi.it

Abstract. Social networks and crowdsourcing platforms provide powerful means to execute tasks that require human intelligence instead of just machine computation power. Especially crowdsourcing has demonstrated its applicability in many fields, and a variety of platforms have been created for delegating small tasks to human solvers on the Web. However, creating applications that are structured, thus applications that combine more than a single task, is a complex and typically manual endeavor that requires many different interactions with crowdsourcing platforms. In this paper, we introduce the idea of a *crowd computer*, discuss its properties, and propose a programming paradigm for the development of crowdsourcing applications. In particular, we argue in favor of business processes as formalism to program the crowd computer and show how they enable the reuse of intricate crowdsourcing practices.

1 Introduction

The ability to connect a large number of people and to lower the effort barrier to collecting input from them in all sorts of contexts - while in the office, home, or while waiting in line at the grocery store - facilitates the involvement of humans in computations and information sourcing and processing tasks. The process of involving humans in computations is typically referred to as *crowdsourcing*, *social computing*, or variations thereof based on the aspect of human information processing one wants to emphasize. As common in a relatively novel area of research, there are a number of variations of the interpretation of these terms, but, in general, they refer to the process of outsourcing task solving to a possibly unknown and large number of people - the crowd [5], thereby harvesting the collective intelligence to realize greater value from the interaction between users and information [11].

Many social computing systems are already available on the market, some of them born before the term "social computing" became widely known and used. If we consider them based on the kind of computations they support, we can notice that there are essentially two kinds of platforms: horizontal platforms, allowing people to post different kinds of computing problems, and applications, tailored at a specific kind of crowdsourcing task. An example of the former is Amazon Mechanical Turk (MTurk), that gives the ability to post generic tasks to users

and collects results. In a way, MTurk– together with its user base – resembles the notion of a traditional computer, that is, something we can program to execute a task. An example of crowdsourcing application is Wikipedia, where the information is indeed crowdsourced but the task is restricted to that of providing and managing information for an encyclopedia. In IT terms, this is indeed more similar to a packaged application more than a computer.

Despite the early success of some of these platforms and applications, creating a crowdsourcing application – or even a task on platforms such as MTurk – is still an art, and in practice it is unfeasible to leverage a (social) *computing platform* such as MTurk to create an *application* such as Wikipedia or others. Indeed, MTurk is mostly used for very simple tasks, many of which oriented to research studies on people's behavior [10]. Indeed, social computing is still in its infancy as a scientific discipline. The crowd can be a terrific source of information and of "computing power", able to execute some "computations" that a computer (or a crowd of computers – the cloud) cannot do (or cannot do as effectively and efficiently). However, there is no consensus or understanding of what a social computer is, which its fundamental concepts, components and functionality are, and how it can be "programmed".

In this paper we discuss the characteristics of a particular instance of social computer, a *crowd computer*, that is, a platform that can be programmed, with a flexibility conceptually comparable to that of a traditional computer, to create crowdsourcing applications. We propose and sketch-out a separation between what are the basic functions of a crowd computer (conceptually analogous to the instruction set of a microprocessor) and what should be instead specified by programs that leverage these functions to generate applications. We then identify programming language templates, expressed as process skeletons, that can be reused by programmers to develop their applications. The goal is to capture practices for the various aspects of business logic commonly needed in crowdsourcing applications, simplifying the programming of the crowd computer and laying a foundation the actual social computer.

This is a preliminary study based on our earlier analysis of what can be achieved by social/crowd platforms and derived from some recent work (e.g., the work on CrowdSearch [2] and the work appearing at BPM2012 [12]).

Notice that this vision, while opening up a plethora of new application scenarios and implementation possibilities, also implies addressing a wide set of related problems, spanning from ethical issues related to using human brains as "components" of a computation platforms, to security, trust and performance of this new platforms. In this paper we don't address these issues directly, but we design our solution also considering the need of addressing them in the future.

2 Scenario

For demonstration and explanation purposes, let's imagine we want to organize and manage a photo contest for a given thematic area, e.g., crowdsourcing. The

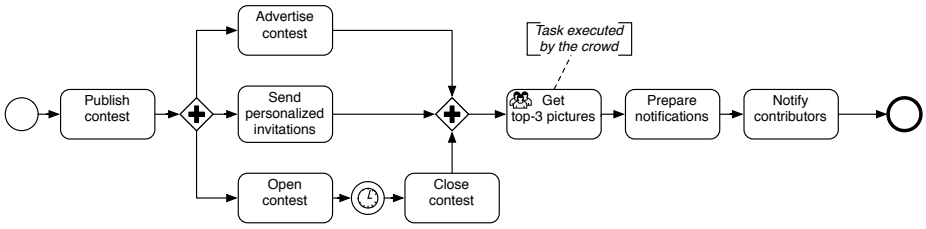


Fig. 1. A simple photo contest crowdsourcing the selection of the top-3 contributions

idea is to publish a simple website for the submission of photos and to advertise the contest via journals, magazines, and newspapers related to photography, as well as via direct contacts (e.g., emails, Facebook contacts, Google+ circles, etc.) to potentially interested photographers. The submission system is to be kept open for one month, in parallel to all advertising activities. Upon the closure of the submission system, we simply want to crowdsource the selection of the best three contributions, that is, we do not want to create an internal jury for the selection of the best three photographs and instead delegate the task to the crowd. Once the three best photos have been determined by the crowd, we prepare and send out the notifications to photographers. The process is illustrated in Figure 1.

While the overall process is a traditional BPM problem, the crowdsourcing of the selection of the top-3 contributions is not. There are many possible crowdsourcing platforms we can use and many different ways we can use them to obtain the ranking of photos. For instance, if we use Amazon Mechanical Turk (MTurk) we must split the task into smaller chunks, in order to have tasks of reasonable size (asking for a ranking of all submissions in one single task would be too complex for a single worker). For example, a set of 10 photos could be big enough to provide a reasonable choice to the worker and small enough to allow fast decisions. We ask each worker for a ranking of the best three photos out of their set, then we aggregate all results into one global ranking and select the best three. Of course, we could also have another intermediate selection step before the global ranking, split the photos differently, ask workers to order *all* of their photos, etc. Specifying a good logic can be a complex, iterative task.

Whatever logic we adopt, the above idea of splitting the ranking into subtasks is relatively naive, in that it does not consider possible quality issues regarding the feedback that can be obtained from crowsourcing platforms like MTurk (and others). In order to grant a better quality of the final result, we could for instance assign each chunk to two different workers and then average their rankings, or we could have chunks that partially overlap, or we could ask to workers to agree on a common ranking, and so on. Instead of focusing on the quality of the feedbacks, we could also try to select only workers that we trust are able to judge photos, e.g., because they are photographers themselves. Doing so would require us to set up a suitable qualification test and to admit to the "crowdsourced jury" only those that pass the test. We could get this information

either by looking at their profiles or by simply asking them. But, again, there are many possibilities. In summary, several options are at hand and it is not trivial to understand which solution suits best which task.

3 The crowd computer: architecture and instruction set

In the following, we present and discuss a conceptual architecture of a *crowd computer*, an information collection and processing system that can be programmed to execute crowd computing applications.

3.1 Architecture

The crowd computer can be described based on an analogy and comparison with a traditional computer or a cloud computing cluster. In both cases we have computing systems: the main difference being that in the crowd computer the "hardware" also includes the crowd, in addition to the traditional elements of a computer (CPU, memory, etc.). This means that the information sources, sinks, and processors can be humans, and therefore each processor operates at will and in a rather non-deterministic fashion. Correspondingly, the *instruction set* of a crowd computer also needs to be extended to interact with this new type of processing entities, for example to distribute the work (possibly in a redundant fashion), accept or reject it, remind workers, maintain profile and rating information, and the like. These instructions are conceptually analogous to an API for accessing the crowd (or, in other words, they represent a *crowd programming interface*, or CPI).

Figure 2 shows the main components of the ***crowd computer***, namely:

- two kinds of *computing components*: i) a traditional computer (in Von Neumann's terms, the arithmetic logic unit) and ii), the crowd;
- the *crowdsourcing engine*, that is, in terms of the Von Neumann machine, the control unit that coordinates the execution of social computing programs. These programs are in turn composed of the instructions within the instruction set, executed either by a CPU or by the crowd;
- a *storage* which, besides memory for data and instructions, includes data on the crowd (members, their execution performance and history, ratings, payment information, etc.);
- the *crowd interaction component* (a Graphical User Interface – GUI) that connects the engine with the crowd. Because the crowd is made of humans, interactions always occur through some (typically graphical) interface, which can be a traditional desktop UI, a mobile UI, sensor-based, and the like. For instance, in the case of Amazon Mechanical Turk, the GUI is the directly the website mturk.com. In the general case, the GUI would adapt and provide the needed human-computer interaction for the specific social application.

In this conceptual architecture we focus on the crowd aspect, assuming that applications will obviously require traditional functionalities too. We also observe

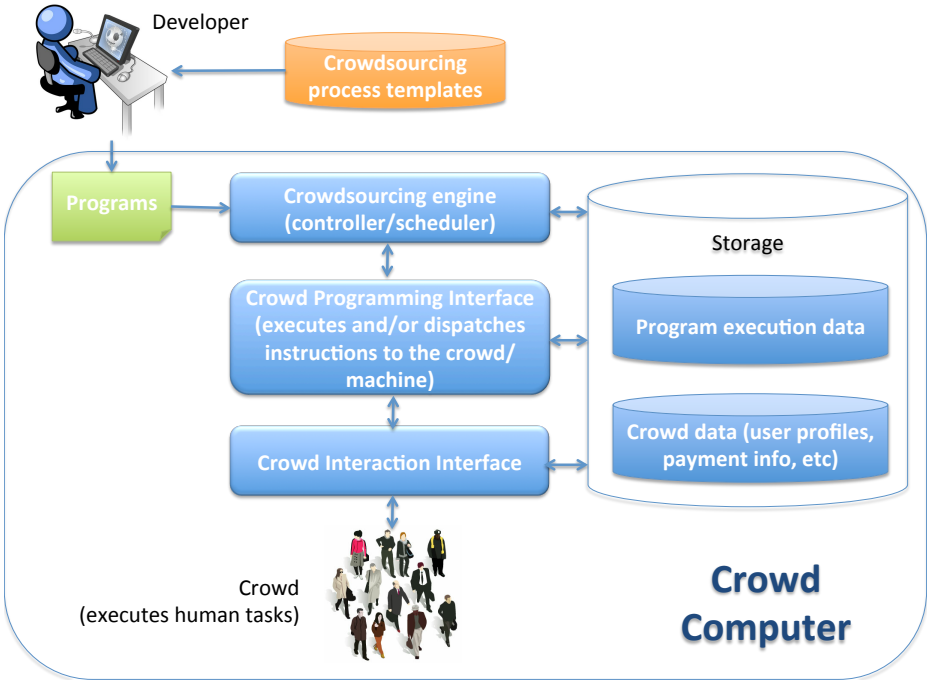


Fig. 2. Conceptual architecture of the crowd computer

that a crowd computer is naturally more open and dynamic than a traditional one in that: the members and the profile of the crowd can change over time; the specification of the tasks assigned to the crowd and the way they are executed are more flexible; and and even the UI to communicate with the crowd may be programmable as each application may want or need to have its own mechanisms for communicating or collecting information from the crowd.

3.2 The Crowd Programming Interface (CPI)

When designing a crowd computer, a key decision lies in identifying the instruction set that it should have to support a reasonably large class of programs while keeping the instruction set simple, manageable, and efficient. Our approach starts from providing a minimal instruction set, which is then extended once we get an understanding of the most commonly used patterns of invocation. A program combines the instructions to implement specific behaviors or policies. We take a similar stand here, briefly listing below which are the operations that should be provided to the crowd programmer to implement crowd computing behaviors. We then present process templates that can combine these instructions to define behaviors and policies. In the following we will use the terms crowd instructions, CPI (Crowd Programming Interface) or API interchangeably. The definition of these functions stays at a high level of abstraction, trying to emphasize the

needed functionality of API. As the reader will notice, a crowd computer with the proposed instruction set will be more flexible (or less "hardcoded") than current crowdsourcing applications and even of current crowdsourcing platforms that tend to hardcode some of the behaviors, which we instead see as being specified at the programming (process) level. The crowd computer design relies on the following definitions:

Human Task. This set of operations (and of corresponding data, stored as part of the crowd computer storage) manages the human tasks lifecycle, essentially submitting tasks to users and collecting replies. It provides several functions like: *create task* - function that creates a new task with metadata about it; *read task* - returns data and metadata about the task; *update task* - updates task parameters (like deadline, maximum executors per task); *delete task* - deletes information about the task; *activate task* - makes the task visible for crowd workers; *cancel task* - makes task invisible for crowd workers; *assign task* - assigns task with a list of crowd workers to be performed; *connect task* - connects task with another task, making possible creation of processes; *get list of tasks* - returns a list of tasks filtered by parameters; *perform task* - executor saves results of solving task; *submit task* - executor submits results of solving task; *confirm task* - requester confirms submitted results.

Profile. This CPI block stores and manages personal and skills information of crowd workers. It contains a list of functions like: *register profile* - creates a new profile of a crowdworker or a requester; *authorize profile* - authorization of user profile by login and password; *update profile* - changes personal information in profile or user skills information; *delete profile*; *read profile information*; *get list of profiles* - returns a list of user profiles filtered by some parameters (age, skills, education, experience); *connect profiles* - connects one profile with others for creating groups, teams or substitutable crowd workers; *update workers' experience*.

Qualification. This CPI block takes care of qualification tests in the crowdsourcing process to make sure crowd workers have enough expertise to solve a given task. Here the list of core functions: *create test* - create a new qualification test to check crowd worker experience; *update test*; *delete test*; *connect test with task*; *validate test*; *create experience parameter*.

Payment. This block holds the payment basic functions, like: *create reward* - creates an information in the payment system about an account of the requester and the reward amount; *propose reward* - connects the proposed reward with the task; *pay reward* - after results are confirmed the requester pays the executor.

In addition, a crowd computer would offer management operations that provide runtime and statistical information on task completion status, performances, and the like. Notice that the strategy on how to design and monitor the crowd computing behaviour (starting from the definition of which tasks are performed by humans and which ones by actual machines) lays in the hand of a human designer. In this first version of the work we don't address the problem of dynamic or collaborative design of the crowd computer.

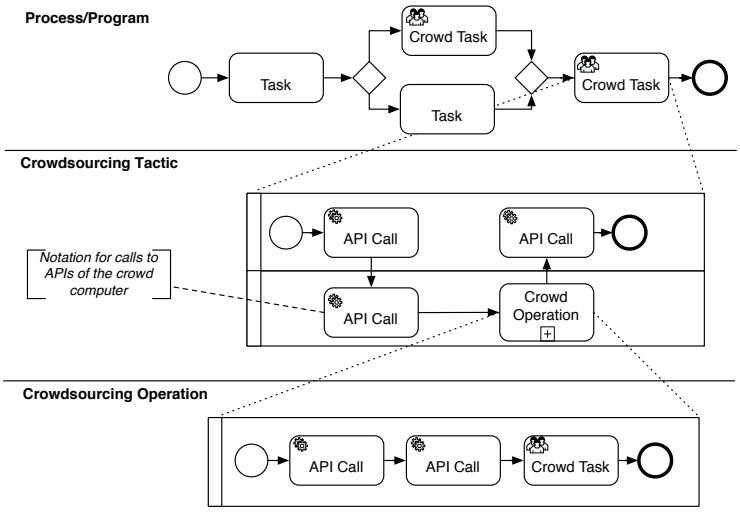


Fig. 3. A three-layered approach to assisted programming of the crowd computer

4 Programming the crowd computer

What does it now mean to *program* a crowd computer? In essence, we need to program the APIs exposed by the underlying crowdsourcing platform in such a way that the resulting "program" (the set of API invocations) solves the task we would like to crowdsource. In our reference scenario, this task is the ranking of the top-3 photographs. We have seen that crowdsourcing that task requires spitting the problem into sub-tasks, deciding on the order in which to execute sub-tasks, aggregate feedbacks, control quality, possibly pre-select workers, etc. As a matter of fact, this is like programming. In fact, we envision a *process-driven paradigm* to "programming" the crowd computer, as such seems a natural fit for structuring and managing both work and people. In line with this design choice, as crowdsourcing engine (see Figure 2) we envision a business process engine with suitable crowdsourcing extensions.

We have also seen that integrating crowd tasks into a common business process logic is not trivial at all. For this integration, we identify three conceptual layers of abstraction that help understand, modularize, and program crowdsourcing applications (see Figure 3):

- At the top-most level, we have the ***process/program*** layer. This is the place where we model the actual process logic, such as the one illustrated in Figure 1. We use the BPMN to express processes plus add a new construct to it, i.e., the crowd task (labeled with a crowd icon), to tell which tasks are to be delegated to the crowd. This layer talks about tasks and crowd tasks.
- Next, we have the ***crowdsourcing tactic*** layer. This is where we decide how to approach the crowd and how to manage the overall crowdsourcing process of each individual crowd task. For instance, MTurk implements a

so-called *marketplace* tactic. If we used other platforms, we could have, for instance, organized a *contest* tactic (like in Prizes.org), or similar. The tactic decides how work is assigned, how workers are motivated, and how they are remunerated. This layer shows API invocations and uses sub-processes exposed by the lower layer.
– At the lower-most level, we have the ***crowdsourcing operations***. This is where we concretely decide about how to pre-select workers, how to control quality, how to aggregate feedback, how to split input data, and similar. For each aspect (e.g., pre-selection) there are typically several options for how to implement it. This is the lowest level of detail and shows how to operatively enact the different choices in terms of API calls or crowd tasks.

The three layers are tightly integrated with each other, yet they provide for the necessary abstractions and flexibility to configure each crowd task according to its very own characteristics and goals. The use of tactics on top of a crowd computer allows us to abstract away from individual platform specifics (today, each platform typically implements one tactic) and instead to focus on what is best for each individual crowd task. Modular crowdsourcing operations enable the flexible configuration of the different tactics and foster reuse.

In the following, we specifically look into the details of these latter two layers, leaving the details of the process/program layer to future work. Specifically, we express the two layers in terms of reusable BPMN patterns (sub-processes), which assist the developer in programming the crowd computer. We choose BPMN as a design notation because of its widespread adoption and rather solid semantics (also considering that our patterns are quite simple and do not make use of the most controversial aspects of the BPMN notation).

5 Crowdsourcing patterns

We start by explaining the logic of the tactics, focusing on two of the most used approaches, i.e., *marketplace* and *contest*, then we show the operations that are needed to turn the tactics into concrete API calls and crowd tasks.

5.1 Crowdsourcing tactics

In the general case, a crowd task is not an atomic action that can be executed by one single worker, but rather a combination of steps and multiple workers. The tactic tells which steps to use and which and how many workers to involve. Yet, in current crowdsourcing platforms these tactics are typically applied at the level of the individual task executed by a crowd worker. Therefore, before applying a tactic to a given simple task, we split the complex crowd task into smaller tasks. Only then we apply a tactic to each of the small tasks, and finally merge all results into a final result for the crowd task. This logic is illustrated in Figure 4(a). Different tactics to crowdsourced work exist; for space reasons, we only illustrate the two most common ones, the *market* and the *contest*.

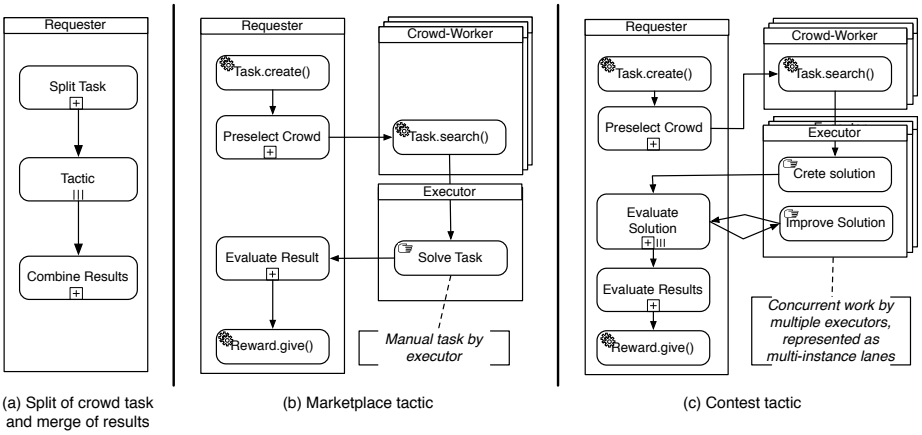


Fig. 4. Splitting a crowd task, choosing a tactic, and merging the results

In Figure 4(b) we model the ***marketplace*** tactic. The marketplace is based on the idea of a shared place where multiple requesters offer work for a given reward, and workers can choose among the offers the one they like most. The crowd worker who executes a task is called executor. Once the task is taken by an executor it is removed from the platform. The executor is paid when he submits his answer and when the answer is accepted by the requester.

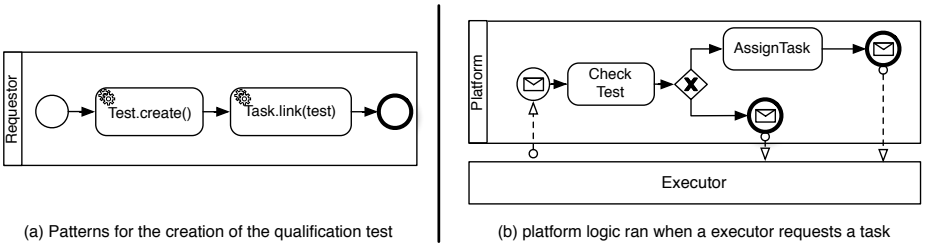
Figure 4(c) models the ***contest*** tactic. The marketplace and the contest share common parts, the beginning, where there is the creation and pre-selection, and the end, where there is the validation and reward. Yet, in the contest, a task is made available for a given amount of time, and many different executors may perform it and submit answers. They are aware that they are in competition with each other and that in the end only one (or some) will get the reward. Executors can improve their solution as long as the task is active by taking into account comments from the requester or by looking at the solutions by others.

Typically, the difference of these two tactics manifests itself also in the reward. In the marketplace, solving a task is generally rewarded only with some cents, and an executor is almost sure to receive this small amount. In the contest, the reward is higher (up to hundreds or thousands of dollars), but only one or few executors receive the reward.

5.2 Crowdsourcing operations

The tactics described in Figure 4 require the expansion of four sub-processes for the splitting of the crowd task, the pre-selection of crowd workers, quality control, and the final merge of feedbacks. Again, for each of these we may have different typical solutions. We exemplify the most interesting ones here.

Preselection. Worker selection is one of the crucial steps in creating a crowd task. Pre-selecting workers means defining minimum requirements to be eligible



(a) Patterns for the creation of the qualification test

(b) platform logic ran when a executor requests a task

Fig. 5. Qualification test pre-selection pattern with requester and platform concerns

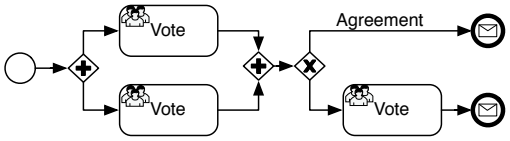


Fig. 6. The agreement operation for quality control

to perform a task. This operation is fundamental, since pre-selecting the right crowd may increase the quality of results. We have different pre-selection options:

- *Implicit by platform choice.* Each crowdsourcing platform available today is different, with different workers, having different skills, expecting different behaviors, etc. Choosing which platform to use implicitly limits the worker basis to the audience of that platform only.
- *User profile and performance data.* We can inspect the workers' profile and pre-select the characteristics of the desired workers by specifying constraints, e.g., on nationality, age, and the like. In addition to user profile data, also the historical performance of workers can be taken into account.
- *Qualification tests.* We can also ask workers to pass a task-specific qualification test before being able to take and solve a task. We depict this pattern in Figure 5, which creates the test and links the test with the corresponding task. Part (b) of the figure shows how the platform internally implements and manages the test: if the worker has passed the test, the system assigns the task, otherwise the worker is notified with a message.

Quality control. Quality control is the evaluation of the executors' results. It decides about the acceptance or rejection of performed work and about the payment. We again can implement this operation in multiple ways:

- *Requester evaluation.* This is the base case. The requester evaluates by himself the work deciding if it is acceptable or not.
- *Expert evaluation.* Similar to the previous case, only one person evaluates the work, but an expert is selected from the crowd via a suitable pre-selection. The expert evaluation is a crowd-task.

- *Top-k*. This patterns asks a set of executors to vote and rank results, accepting only the best k results. The top-k patterns is a combination of crowd tasks for collecting votes from the crowd.
- *Agreement*. The agreement pattern involves two judges, chosen from the crowd, which evaluate the solution. If the two judges are in an agreement, their answer is taken as the official one. If they disagree, a third judge gets involved. The third judge basically decides. The pattern can be modeled by asking directly to three judges, while the model in Figure 6 is cheaper in case of immediate agreement. The pattern requires three crowd tasks.
- *Automatic*. This practice injects control questions into the response form. Control questions are questions whose answer is known a priori. Based on the quality of the answers to the control questions, the system can automatically assess the quality of the actual work.
- *Average*. Following the idea of *wisdom of the crowd* [4], we can also compute the average of all collected answers (in case of numeric answers), assuming that the average (or any other aggregation) represents the best result.

Sensibly using reusable crowdsourcing operations to instantiate a crowdsourcing tactic means "programming" the crowd in an easy and effective way and allows the developer to leverage on common crowdsourcing practice.

6 Related Work

Complex crowd tasks are generally dealt with by ***splitting*** them into smaller tasks, combining their solutions, and constructing the integrated solution, as in the MapReduce algorithm know from distributed computing [3]. MapReduce applied to crowdsourcing has been proposed by Kittur et al. [6] and Kulkarni et al. [7]. The authors propose two similar systems, allowing a crowd worker to decide whether to solve a task as is or to split it into smaller chunks for other workers. The worker who splits a task has the duty of recomposing the solutions of the smaller tasks, practically providing an aggregated solution to the task he spitted. Little et al. [9] use iterative and parallel tasks to structure complex crowd tasks, which however does still not provide enough flexibility.

Programming the crowd like a computer is a research trend that is growing fast. A set of purpose-built languages and solutions to program the crowd have been proposed so far. For instance, Turkit [8] is a scripting language that allows one to create applications for solving tasks with the help of the crowd. The approach conciliates human labor of Amazon Mechanical Turk and computations performed by a machine. The language is equipped with an execution engine able to run crowdsourcing applications. In order to achieve a deterministic behavior (like applications on a regular computers), the system provides the possibility to store the results of the executors' works for later reuse. e.g. to restart after a crash. This crash-and-rerun approach can save time and money and guarantees re-produceable results in case of interruptions. Jabberwocky [1] is another interesting approach. In this work, Ahmad et al. introduce a social

computing stack that consists of a platform (Dormouse) that interacts with the crowd, a method for splitting and aggregating tasks called ManReduce, inspired by [6], and a scripting language (Dog) to specify the crowdsourcing logic and also the graphical interface of the application. These scripting languages indeed allow one to implement crowdsourcing applications from scratch. However they do not come with reusable patterns and are not suitable for integration with business process management practices, feasible through BPMN instead.

7 Conclusions

We leverage on the intrinsic process nature of crowdsourcing and propose a *process-based programming paradigm* for what we call the *crowd computer*, i.e., an information collection and processing system for crowd computing applications. The core of the proposal is an extensible set of reusable crowdsourcing practices, which we group into *tactics* (telling how to approach the crowd) and *operations* (telling how to manage the crowd). The work is in an early stage, and we still have to formalize a varied set of crowdsourcing patterns and to implement an own crowd computer.

Acknowledgements. This work is partially funded by the BPM4People project (`http://www.bpm4people.org`) of the EU FP7 SME Capacities program.

References

1. S. Ahmad, A. Battle, Z. Malkani, and S. Kamvar. The jabberwocky programming environment for structured social computing. In *UIST'11*, pages 53–64, 2011.
2. A. Bozzon, M. Brambilla, and S. Ceri. Answering search queries with crowdsearcher. In *World Wide Web Conference 2012, WWW, Lyon, France*, pages 1009–1018. ACM, 2012.
3. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
4. F. Galton. Vox populi (the wisdom of crowds). *Nature*, 75, 1907.
5. J. Howe. The rise of crowdsourcing. *October*, 14(14):1–7, 2006.
6. A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut. Crowdforge: crowdsourcing complex work. In *UIST'11*, pages 43–52, 2011.
7. A. P. Kulkarni, M. Can, and B. Hartmann. Turkomatic: automatic recursive task and workflow design for mechanical turk. In *CHI EA'11*, pages 2053–2058, 2011.
8. 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, 2009.
9. G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Exploring iterative and parallel human computation processes. *CHI EA'10*, page 4309, 2010.
10. J. Ross and B. Tomlinson. Who are the crowdworkers? shifting demographics in mechanical turk. *CHI 10*, pages 2863–2872, 2010.
11. J. Surowiecki. *The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies and Nations*, volume http://www. Doubleday, 2004.
12. S. Tranquillini, P. Spieß, F. Daniel, S. Karnouskos, F. Casati, N. Oertel, L. Mottola, F. Oppermann, G. Picco, K. Römer, and T. Voigt. Process-Based Design and Integration of Wireless Sensor Network Applications. In *BPM 2012*, 2012.