Reactive Crowdsourcing

Alessandro Bozzon* Marco Brambilla Stefano Ceri Andrea Mauri

Dipartimento di Elettronica, Informazione e Bioingegneria – Politecnico di Milano
Piazza Leonardo da Vinci, 32 – 20133 Milano, Italy

{alessandro.bozzon, marco.brambilla, stefano.ceri, andrea.mauri}@polimi.it

ABSTRACT

An essential aspect for building effective crowdsourcing computations is the ability of "controlling the crowd", i.e. of dynamically adapting the behaviour of the crowdsourcing systems as response to the quantity and quality of completed tasks or to the availability and reliability of performers. Most crowdsourcing systems only provide limited and predefined controls; in contrast, we present an approach to crowdsourcing which provides fine-level, powerful and flexible controls. We model each crowdsourcing application as composition of elementary task types and we progressively transform these high level specifications into the features of a reactive execution environment that supports task planning, assignment and completion as well as performer monitoring and exclusion. Controls are specified as active rules on top of data structures which are derived from the model of the application; rules can be added, dropped or modified, thus guaranteeing maximal flexibility with limited effort.

We also report on our prototype platform that implements the proposed framework and we show the results of our experimentations with different rule sets, demonstrating how simple changes to the rules can substantially affect time, effort and quality involved in crowdsourcing activities.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

Crowdsourcing, reactive rules, control, social computation.

1. INTRODUCTION

Crowdsourcing is an emerging way of involving humans in performing information seeking and computation tasks; classical platforms, such as Amazon Mechanical Turk (AMT),[1] are marketplaces where tasks created by humans can be posted to humans; other classical platforms allow to post questions or to ask for recommendations, often within specific domains or local contexts. Large crowds may take part to social computations for a variety of motivations, which include non-monetary ones, such as public recognition, fun, or the genuine wish of contributing their knowledge to a social process. In all these cases each responder brings value to the requestor, who wishes to make the best use of responders' availability and reliability so as to get the best possible result for the posted question or job. In particular, requestors would like to dynamically adapt the platform behaviour so as to maximise the quality of results while minimising or focalising the interactions required to responders [11].

Unfortunately, most crowdsourcing systems are not flexible, as they do not support a high level, fine-tuned control upon posting and retracting tasks. The AMT platform offers an API for controlling the posting and evolution of tasks, but the implementation of applications must be done through imperative languages, either with low-level programming or through frameworks like TurkIt [12]. Since crowdsourcing can be performed upon social networking platforms too, one can exploit social networks APIs for programming applications which use them. Some academic works provide control capabilities for crowdsourcing, but with limited and specific control rules (e.g., the DeCo [16] system provides support for closing tasks based on temporal constraints). In summary, in spite of the great importance of crowd control, at the current state-of-the-art designing and deploying crowdsourcing applications with sophisticated controls is very difficult. All the existing platforms and approaches lack methods for systematically designing complex control strategies based on the state of tasks, results and performers.

Objective. This paper is proposing a conceptual framework and a reactive execution environment for modelling and controlling crowdsourcing computations; reactive control is obtained through rules which are formally defined according to a rule specification language and whose properties (e.g., termination) can be easily proved in the context of a well-organized computational framework. As highlighted by [15], several programmatic methods for human computation have been proposed so far [12][9][1][13][14], but they do not support yet the complexity required by real-world, enterprise–scale applications. Due to its flexibility and extensibility, our approach covers the expressive power in reactive control which is exhibited by any of the cited systems.

Our framework applies the principles of *separation of concerns* – which is typical for complex systems design – to the diverse aspects of crowdsourcing design. We adopt an abstract model of crowdsourcing activities in terms of elementary task types (such as: labelling, liking, sorting, grouping) performed upon a data set, and then we define a crowdsourc-

*Alessandro Bozzon is currently affiliated with the *Delft University of Technology*, Mekelweg 4, 2628CD, Delft, The Netherlands. E-mail: a.bozzon@tudelft.nl

[1] https://www.mturk.com

WWW 2013, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2035-1/13/05.

ing task as an arbitrary composition of these task types; this model is not introducing limitations, as arbitrary crowdsourcing tasks can always be decomposed into smaller granularity tasks, each one of the suitable elementary type. Starting from task types, we then define the data structures which are needed for controlling the planning, execution, and reactive control of crowdsourcing applications. Control encompasses the evaluation of arbitrary conditions on result objects (e.g., on their level of confidence and of agreement), on performers (e.g., on the number of performed tasks and their correctness, leading to the classification of performers as experts or spammers) and on tasks (e.g., on their number and duration). Our framework provides a reactive style for specifying these conditions and for defining the actions that must be correspondingly triggered, making decisions about the production of results, the classification of performers the early termination and re-planning of tasks, the dynamic definition of micro-tasks, and so on.

Experiment. Besides the conceptual definitions and the control rule language, our proposal comprises a platform which acts either as a stand-alone system or as an interoperability framework on top of social networks or crowdsourcing systems, which are accessed through their APIs – and therefore are subject to APIs' limitations; the platform includes an engine which executes rules and crowdsourcing tasks. We present a set of experiments with different rule sets, demonstrating how simple changes to the rules can significantly affect the time, effort and quality of tasks.

Added Value. The added value of our work can be summarised as follows:

- We define a design process through a sequence of simple model-driven design and instantiation activities, which simplify crowdsourcing design and enactment;
- We present an application-independent abstraction for crowdsourcing control that uses quantifiable measures of correctness, effort or duration;
- We demonstrate that our rule-based approach provides a high level of automation, flexibility, and expressive power.

Paper Structure. The paper is organised as follows: Section 2 defines our conceptual view of crowdsourcing applications and the method we apply for their design; Section 3 describes in details our reactive language for controlling crowdsourcing tasks; Section 4 reports on our implementation and experiments; Section 5 discusses the related work; and Section 6 concludes.

2. DESIGNING CROWD APPLICATIONS

With our approach, the operations which constitute a task and are assigned to the performers of a crowdsourcing system are described in terms of an abstract model, that was initially presented in [3] and is hereby revised, built after a careful analysis of the systems for human task executions and of many applications and case studies.

The main strength of the model is its extreme simplicity. We assume that each task receives as input a list of items (e.g., photos, texts, but also arbitrarily complex objects) and asks the users to perform one or more operations upon them, which belong to a predefined set of abstract **operations types**; examples are *Like*, for annotating items with a preference tag; *Classify*, for assigning each item to one or more classes; *Order*, for reordering the items in a list. Each operation type is further characterised by *parameters* and *default output variables*; for instance, a *Classify* operation has the list of classes as parameters and the classification(s) of each item as output variables. The full list of currently supported abstract operation types is reported in Table 1.

The model can be extended either by adding custom operation types or by adding custom parameters and output variables to the given types; however, the operation types of the model are supported by automatic model transformations, discussed next, for generating a crowdsourcing application, while custom elements require manual refinement of either the transformations or the constructed models.

Task types are built as lists of operation types and apply to a list of data objects, which have a schema consisting of several attributes. The corresponding concepts are shown in the meta-model ① in the upper left corner of Fig. 1. 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. The meta-model can be used to describe complex and long-duration tasks (e.g., asking for the translation of a long text can be modelled as the insertion of a single item); however, it better suits classical crowdsourcing applications where tasks are split into several micro-tasks (or HITs), each micro-task evaluates a subset of the task objects, and several micro-task answers might be required to decide the results for each data object (e.g., by majority agreement).

In this scenario, the presence of a simple yet formal metamodel is quite useful, as it allows the formalisation of the operation-dependent control variables and aggregations operations required to keep track of micro-task executions and, ultimately, to regulate the task execution.

The progressive specification of crowdsourcing applications consists of the following phases: *i)* **Task design** - deciding how a task is assembled as a sequence of operation types; *ii)* **Object and performer design** - defining the set of objects and performers; *iii)* **Workplan design** - Defining how a task is split into micro-tasks and assigned to objects and to performers; *iv)* **Control Design** - Defining the rules that enable the run-time control of objects, tasks, and performers. These phases are described next.

2.1 Task Design

Task design consists of deciding the main features of a task, i.e. the operation types and the object schema; it is conducted by instantiating the **meta-model** ① in Fig. 1, which contains entities for *Task Type, Operation Type, Operation Parameter, Object Type,* and *Attribute*; in the instance model ② in Fig. 1, we show the design of a simple crowdsourcing application for classifying american politicians as either republican or democratic, where the operation type is *classify*, and classes are the *republican* or *democratic* party. The schema includes *politicians* which are described by their *LastName* and *Photo*.

When the instantiation is completed, the model transformation *MT1* produces the **structural model** of the application, which includes entities for *Task, Performer* and *Politician* (see the model ③ of Fig. 1). The transformation assembles into the application-specific entity *Politician* the attributes of its object type and of the **output schema**,

Table 1: List of the operation types with input parameters and default output variables.

Op.Type	Description	Input Parameters	Default output variables
Choice	The performer selects up to n items	n = max number of items that can be selected	Selected items
Like	The performer adds like (unlike) annotations to some items		Number of likes for each item
Score	The performer assigns a score (in the 1..n interval) to some items.	n = max value for the score	Average score of each item
Tag	The performer annotates some items with tags	n = max number of tags that can be associated	Set of tags for each item
Classify	The performer assigns each item to one or more classes	set of predefined classes	Set of classes for each item
Order	The performer reorders the (top n) items in the input list	n = number of top elements to be ordered	Position of each item
Insert, Delete	The performer inserts/deletes up to n items in the list	n = max number of items that can be inserted/deleted	Inserted objects / Tagging of deleted objects
Modify	The performer changes the values of attributes of some items in the list	Set of modifiable attributes	Set of changes to modifiable attributes
Group	The performer clusters the items into (at most n) distinct groups	n = max number of groups	Assignment of each item to a group

which are inferred from the task operation types (see last column of Table 1). In the specific case, the result will indicate, in the attribute `Party`, whether the politician is *republican*, *democratic*, or *unclassified*.

2.2 Object and Performer Design

Object and process design consists of deciding if the task should be instantiated as several task instances (e.g., for assigning each instance to a different execution platform like AMT, Facebook or Twitter) and then determining which objects and performers are associated to each task instance. Performers can be either pre-selected (e.g. based upon their expertise in the topic) or dynamically determined.

The design is conducted by instantiating the structural model. In our running example, for simplicity, we instantiate a single task, and then we indicate some well-known politicians and some performers (see the instance model ④ in Fig. 1). In the actual design environment, this phase is normally conducted by importing objects and performers from existing sources. In this way, structural information about the crowdsourcing application is fully specified.

2.3 Workplan Design

Workplan design consists of creating micro-tasks for each task and of mapping each micro-task to specific performers and objects. Task design includes task splitting, that should be performed on different task types according to different algorithms. Several articles are dedicated to task splitting algorithms for specific operations, e.g., [18]; we don't discuss task splitting further (although we support it in our system and use it in experiments).

Task planning is performed according to *planning directives*, that drive the second model transformation, MT2, which generates the **workplan model** ⑤ of Fig. 1. The directives indicate:

- *Cardinality*: the number of objects associated to each micro-task.
- *Replication*: the number of copies of each object that should be assigned to micro-tasks.
- *Initiation*: the number of micro-tasks which should be created when the application starts (more micro-tasks can be dynamically created during the execution).

The **workplan model** includes the new entity *MicroTask*; each micro-task instance is associated to one task, one performer, and one or more objects (represented by the *Politician* entity in the example). The workplan model is instantiated by the system, which uses the planning directives and simple mathematic formulas to build the suitable micro-tasks in order to perform the application. In Fig. 1 ⑥ we show the micro-task 723 of type *Classify the political party* assigned to the performer *Mario* and to the objects *Bush* and *Kennedy*. Depending on the underlying execution platform, tasks can be *pushed* to specific performers or performers can *pull* the tasks that they like; in the push model micro-task mapping to performers is static, while in the pull model it is dynamically assigned.

2.4 Control Mart

For effectively monitoring the execution, it is most convenient to track the performer's response for each distinct object which is present in the micro-task. Thus, a final model transformation creates the **control mart**, shown in Fig. 2, where each *microTObjExecution* instance is connected exactly to one task, one performer and one object. The control mart is analogous to data marts used for data warehousing [7], as its central entity represents the facts, surrounded by three dimensions.

A minimal relational representation of the control mart, useful for describing reactive control design in the next section, is described in listing 2.1, and it includes the fact table `Execution` and the dimension tables `Politician, Performer` and `Task`. Each execution requires an `Eid`, as the same performer could be assigned the same politician in different micro-tasks; `Status` attributes trace the application evolution (e.g., a `Performer` can be `'Active'` or `'Spammer'`, a `Task` and an `Execution` can be in the status: `'Planned'`, `'Started'`, `'Completed'` or `'Invalid'`). Additional information of interest is present in the actual data marts, e.g., the starting and ending time of tasks and execution, the number of completed tasks for each performer, and so on.

2.1 Fact Table and Dimension Tables

```
Politician (Oid,Party)
Performer  (Pid,Status)
Task       (Tid,Status)
Execution  (Eid,Oid,Pid,Tid,Status,Party)
```

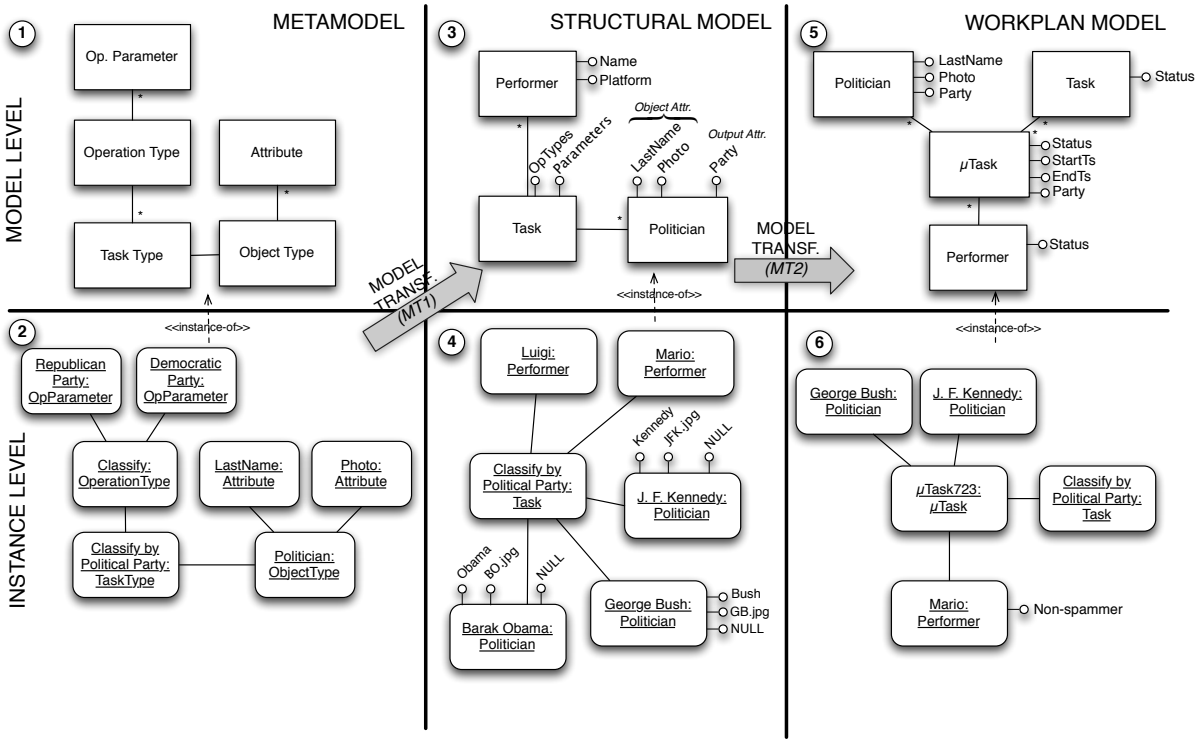

Figure 1: Overview of the concepts and their relations.

2.2 Aggregate Tables

```
Object_CTRL     (Oid,Eval,Dem,Rep,Answer)
Performer_CTRL  (Pid,Eval,Right,Wrong)
Task_CTRL       (Tid,CompExec,CompObj)
```

In addition to the control mart, we derived *aggregate tables* that contain one tuple for each politician, performer and task, and are automatically maintained at each micro-task completion by computing aggregates. As our running example is a classify operation, aggregate tables in listing 2.2 include: *i)* `Object_CTRL`, which contains the number of evaluations performed on each object (`Eval`), a counter for the number of preferences got for each category (`Dem` and `Rep`) and the current value for the object evaluation (`Answer`); *ii)* `Performer_CTRL` which contains the number of micro-tasks executed by the performer (`Eval`), and the number of correct (`Right`) and incorrect (`Wrong`) answers; and *iii)* `Task_CTRL`, which includes the counters of completed executions (`CompExec`) and of evaluated objects (`CompObj`).

Aggregate tables can be derived in an analogous way also for the other operations listed in Table 1: for instance, the `Object_CTRL` table of a *like* operation could feature a `Preferences` attribute to count the preferences obtained by the object.

At workplan enacting, the system creates the instances of the control mart and aggregate tables. Some information might be still undefined (e.g., with a pull model, the identity of performers becomes known during execution), but otherwise we assume that each object, performer and task is associated with suitable entries in the control mart and aggregates tables. The next section is the core of this paper, dedicated to object, performer, and task control.

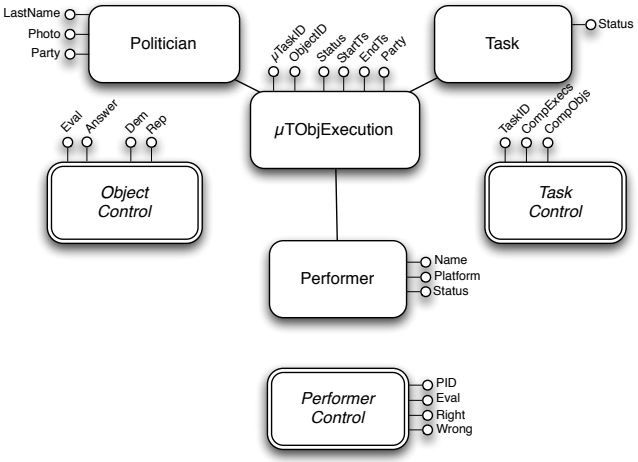

Figure 2: Control mart.

3. REACTIVE CONTROL DESIGN

Control design consists of three activities:

- *Object control* is concerned with deciding when and how responses should be generated for each object.
- *Performer control* is concerned with deciding how performers should be dynamically selected or rejected, on the basis of their performance.
- *Task control* is concerned with completing a task or re-planning task execution.

The control of objects, performers, and tasks is performed by active rules, expressed according to the *event-condition-action* (ECA) paradigm. Each rule is triggered by **events** (`e`) generated upon changes in the control mart or periodically; the rule's **condition** (`c`) is a predicate that must be satisfied on order for the action to be executed; the rule's **actions** (`a`) change the content of the control mart and aggregate tables. This approach has the following advantages:

- *Automation*: most active rules are system-generated.
- *Flexibility*: encoding variants of simple controls require to change specific rules while preserving the rest of the rule set.
- *Power*: rules can be programmed to support arbitrarily complex controls.

3.1 Rule Language

The rule language has been inspired by the long-standing tradition of active databases [19]; its full syntax is reported in Appendix B. Its peculiar syntactic features are the following:

- Rules can be triggered by classical data updates and by periodic `TIMER` events.
- Rules are at row-level granularity. Variables `NEW` and `OLD` denote the *before* and *after state* of each row.
- Special selector formulae are used to express subqueries synthetically; thus, `TABLE[<predicate>].ATTRIBUTE` extracts the same values as `SELECT ATTRIBUTE FROM TABLE WHERE <predicate>`.
- Special *functions* may perform operations on the workplan model (e.g., planning of new micro-tasks).

Two examples of simple active rules for maintaining the counters of 'Rep' and 'Dem' answers for a given Politician are reported in rules 1 and 2; they are triggered by the completion of a micro-task, which in turn consists of an update to the `Answer` attribute of one or more rows in `Execution`.

Rule 1 RepAnswer Rule.

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

Rule 2 DemAnswer Rule.

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

3.2 Active Rule Programming

It is known that active rule programming is rather subtle and unstable: the behaviour of a set of rules may change dramatically as a consequence of small changes in the rules [19]. To overcome this problem, we observe a *best practice* in writing rules. Functionally, we divide rules in three classes:

- *Control rules* for modifying the control tables;
- *Result rules* for modifying the result tables (Politician, Performer, Task);
- *Execution rules* for modifying the execution table - either directly or through replanning of crowdsearching.

Consider the *Precedence Graph* $PG =< N, E >$, where the nodes N are tables; an arc $< N_1, N_2 >$ is drawn when a rule R is triggered by an operation on N_1 and performs an action on N_2. Then, we impose that control and result rules in our system can only generate edges in the PG shown in Fig. 3. Intuitively, this best practice corresponds to propagating rule execution top-down (from execution to control to result tables) and left-to-right (from object to performer to task).

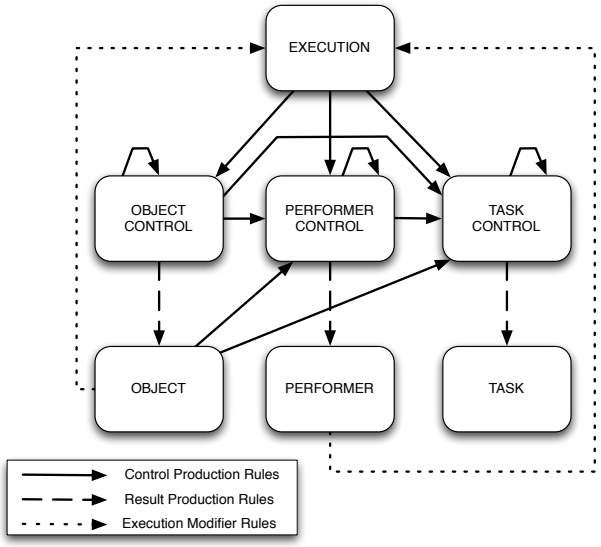

Figure 3: Precedence Graph for rules.

We further assume that only control rules can have cycles in PG, and in such case we assume their triggering graph to be acyclic.[2] Then the following result holds.

Theorem. Any execution of control and result rules terminates.

Proof. The acyclicity of the triggering graph of the rule set, a sufficient condition for termination [19], descends from the acyclicity of the PG graph shown in Fig. 3 (excluding execution rules) and from the acyclicity of ring rules.

Thus, by adopting the best practice for active rule programming described above, we need to worry about rule termination only when we add execution rules, that typically create cycles in the triggering graph of the rule set. These rules have to be carefully considered, as we will show in Section 3.5.

3.3 Control Rules

Control rules maintain the control tables; they are triggered by updates of the execution table (e.g., when a micro-task execution is completed), or of the result tables (e.g., when an object is *closed*, see later) of the control tables themselves – e.g., the rules 1 and 2, which update the aggregate counts `Rep` and `Dem` on the basis of the `Answer` in the

[2]To guarantee this assumption it is sufficient that aggregate computations is performed progressively, from fine to coarse-grain aggregations, as shown by the example rules in Appendix A.

Object_CTRL table. Control rules are automatically generated, due to the fact that operations are typed and to the model-driven application generation process described in Section 2. In addition to rules 1 and 2, several control rules are listed in Appendix A. We next describe the interesting case of rules for spammer control.

We define **spammers** as performers whose answers deviate significantly from correct answers; when performers are rewarded by money, spammers typically give random answers in order to maximise their pay, and providing unreliable answers; our definition of spammer is orthogonal to the reward and is defined just on statistical basis. Spammer detection requires two parts: first computing wrong answers (using a control production rule), and then deciding which performers are spammers based on the amount of wrong answers they provide (using a result production rule). Next we show three different control production rules corresponding to different scenarios; all rules can be automatically produced – the flexibility of the approach is demonstrated by the fact that a rule change is sufficient to change the control policy.

3.3.1 Performers evaluation against golden truth

A typical strategy [17] in crowdsourcing is to set a few *golden answers* known a priori (e.g., predefined by experts) and then check the correctness of performers against them, while tasks are executed. This is possible in our approach by adding the Gold property to the object schema; Gold stores the golden answer when available, and a NULL value otherwise. Then, the rule for managing the aggregate control information of performers is represented in Rule 3.

Rule 3 GoldenTruthRule.

```
e: UPDATE FOR Execution[Answer]
c: Politician[oid=NEW.oid].Gold <> NULL
a: IF(NEW.Answer == Politician[oid==NEW.oid].Gold)
    THEN SET Performer_CTRL[pid==NEW.PID].Right += 1
    ELSE SET Performer_CTRL[pid==NEW.PID].Wrong += 1
```

The rule is triggered by any answer on Execution and simply checks if the answer is about a politician with available golden answer; if so, it updates the counters of Right and Wrong answers of Performer_CTRL depending on the correctness of the given answer against the golden value.

3.3.2 Performers evaluation on object completion

The second possibility for maintaining the performer counters is to wait for an object to be completed (Rule 4) – i.e., for a final evaluation to be produced.

Rule 4 ObjectResultRule.

```
e: UPDATE FOR Politician[Party]
c: NEW.Party <> NULL
a: FOREACH e IN EXECUTION[Oid==NEW.Oid]
     IF (e.Party == NEW.Party)
       THEN SET Performer_CTRL[Pid==e.Pid].Right += 1
       ELSE SET Performer_CTRL[Pid==e.Pid].Wrong += 1
```

The rule is triggered by the completion of an object, which is assigned a non-null Party value as effect of an agreed response. The rule then considers all the past executions of that object and compares the answers of the performers with the answer that has been written in the Party attribute, and updates the Right and Wrong counters of Performer_CTRL.

3.3.3 Performers evaluation on each execution

The third possibility is to maintain the performer's counters at each execution; this anticipates the definition of right and wrong answers even if a final result is not available for the objects. In this case, the Pid attribute of the performer who caused the last change to an object is added to the Object_CTRL table (the content of Pid is copied from Execution by a suitable control rule).

Rule 5 ExcutionResultRule.

```
e: UPDATE FOR Object_CTRL[Answer]
c: Answer<> 'Undefined'
a: SET Performer_CTRL[Pid==NEW.Pid].Right=0
   SET Performer_CTRL[Pid==NEW.Pid].Wrong=0
   FOREACH e IN Execution [Pid==NEW.Pid]
     FOREACH o IN Object_CTRL[Oid==e.Oid]
       IF [e.Answer == o.CurrentAnswer]
        THEN SET Performer_CTRL[Pid==NEW.Pid].Right += 1
        ELSE SET Performer_CTRL[Pid==NEW.Pid].Wrong += 1
```

In Rule 5, at every update of the current Answer in the Object_CTRL table, the specific performer who provided the last Answer in the execution table is considered, and all the past answers of that performer are compared with the corresponding current answers; given that current answers change their value during a crowdsourcing session, incremental maintenance is impossible, and the counters of the affected performer have to be set to zero and recomputed.

3.4 Result Rules

Result rules are triggered by changes in control tables and produce result tables; they express the result control logic, that can be specified through high level directives and be translated to rules. We consider how to decide that an object is closed or that a performer is a spammer.

3.4.1 Closing Objects

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 6.

Rule 6 MajorityResultRule.

```
e: UPDATE FOR Object_CTRL
c: (NEW.Rep== 2) or (NEW.Dem == 2)
a: SET Politician[oid==NEW.oid].Party = 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 two attributes Rep or Dem is equal to 2; then it sets the politician's party 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 ((Rep>0.5*Dem) or (Dem>0.5*Rep))
C2: (Eval>10) and ((Rep>0.8*Dem) or (Dem>0.8*Rep))
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.

3.4.2 Identification of Spammers

Performers are identified as spammers when they are associated with *enough* wrong answers, which have been collected according to anyone of the methods discussed in Section 3.3. A simple rule for identifying spammers is:

Rule 7 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)
```

3.5 Execution Rules

Execution rules respond to the need of altering the execution plan; their action either changes the current micro-tasks or calls for task re-planning, which eventually produces new micro-tasks. These rules are triggered by changes in the control or result tables, and are perhaps the most powerful rules. They must be analysed because they may introduce danger of nontermination of the computation.

3.5.1 Remove Spammer's Micro-Task Executions

Spammer detection results in excluding performers from future assignments. In addition, we may want to propagate the effects of spamming detection upon micro-tasks. Rule 8 selects all the executions of the performer which has been recognised as `'Spammer'`, and checks whether the corresponding objects have been already completed; if not, it logically undoes the spammer's micro-tasks, by first subtracting 1 from `Eval` and either the `Rep` or `Dem` counters of the object control table, and then by deleting the execution tuple. Note that the subsequent propagation to `Answer` in `Object_CTRL` is performed by a rule in Appendix A, with no change.

Rule 8 RemoveMicroTask.

```
e: UPDATE FOR Performer[Status]
c: (OLD.Status != 'Spammer') and (NEW.Status=='Spammer')
a: FOREACH e IN Execution[Pid==NEW.Pid]
     SET e.Status= 'Invalid',
     FOREACH o IN Object_CTRL[o==e.Oid]
       IF (Politician[Oid==o.Oid].Party==NULL)
       THEN
           SET o.Eval -= 1 ,
           IF (e.Answer=='Rep') THEN o.Rep -=1
           IF (e.Answer=='Dem') THEN o.Dem -=1
```

Proving termination requires considering the cycles in the triggering graph and reasoning about their mutual triggering [2]. Updates to `Object_CTRL` may cause the closure of objects (see Sect. 3.4.1), but the condition of the corresponding rule may become true only due to increments of the `Rep` and `Dem` counters (which should be equal to given thresholds), while the above rule performs decrements; thus, the condition of that rule fails. Updates to `Object_CTRL` may also cause updates of `Performer_CTRL` and the re-classification of the performer as a spammer; in such case a second instance of Rule 8 would be triggered, but the condition on the specific performer would fail. Thus, at least one triggered rules along every potential cycle has a false condition, and no cyclic behavior can occur.

Note that termination analysis for this rule must consider the actual rule set with mutual triggering and conditions, as the proof of termination cannot be inferred from the structure of the *PG* graph [19].

3.5.2 Re-planning

Re-planning occurs when, during execution, the answers accumulated so far do not guarantee convergence to a result. For instance, a system execution could require a majority of three responses for a given object and start by planning exactly 3 micro-task executions. Then, on the first conflictual answer, the system could plan to add 3 more micro-tasks, by using the `Plan` function, as shown in Rule 9:

Rule 9 ReplanWhenNoMajority.

```
e: UPDATE FOR Object_CTRL
c: NEW.Eval==3 and NEW.Dem >=0 and NEW.Rep >=0
a: action PLAN(NEW.oid, 3)
```

A different re-planning could be triggered by a timer event, e.g., in the case that the designer wishes to add 10 micro-tasks so as to quickly produce additional executions for each object with less than 30 evaluations, as shown by Rule 10.

Rule 10 ReplanWhenTimeout.

```
e: TIMER for Object_CTRL
c: Object_CTRL.Eval<30
a: action PLAN(Object_CTRL.oid, 10)
```

The `PLAN` function performs the planning of new micro-tasks which are then described through suitable tuples in the control and aggregate tables; this function therefore creates new crowdsearching activities until the condition on evaluations is met, but rule termination is not affected.

Many complex planning rules can be programmed, as demonstrated in the experiments of Section 4; for instance, rules could compute the quality of performers and the difficulty of closing objects, and then assign difficult objects to good quality performers.

4. EXPERIMENTAL EVALUATION

We implemented our reactive control approach in CrowdSearcher,[3] a platform for crowd management presented in [3]. We use the approach presented in Section 2 for application deployment; rules are written in the scripting language Groovy.[4] Each relational control rule is directly translated into a Groovy script; triggering is modelled through internal platform events. We developed three applications,[5] which demonstrate the flexibility and expressive power of reactive crowdsearching for different aspects of application

[3] `http://crowdsearcher.search-computing.com`

[4] `http://groovy.codehaus.org`

[5] `http://demo.search-computing.org/politici`

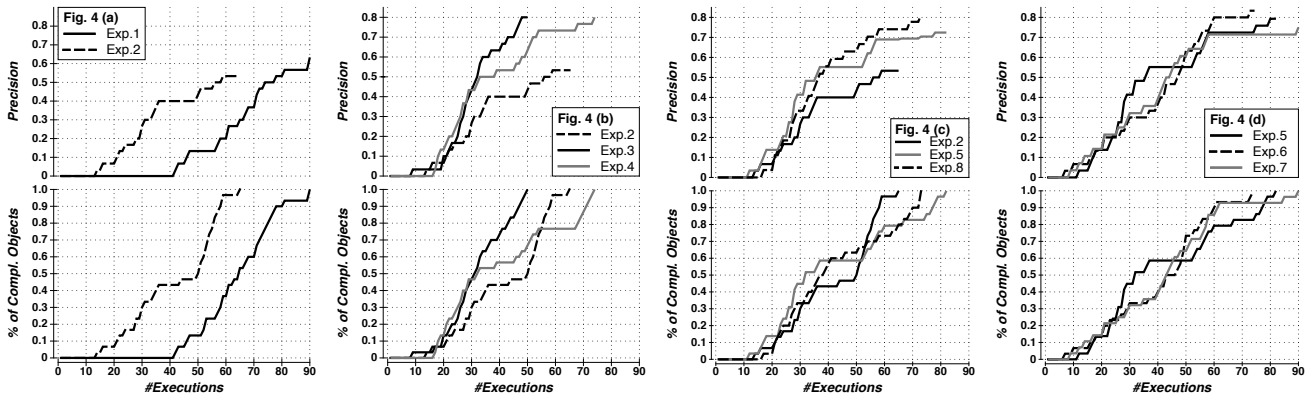

Figure 4: Results precision and percentage of completed objects over micro-task executions.

design and deployment. We recruited (mainly trough public mailing lists and social networks announcements) 284 performers, who performed about 3500 micro-tasks during the third week of Nov. 2012. The code, the dataset, and the anonymized execution logs produced by the experiments are available for download.[6]

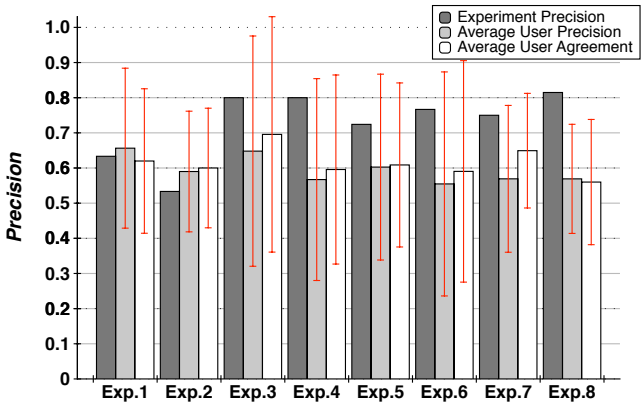

Figure 5: User description of the **politician classification**.

In the **politician's crimes** application, the system presents photos of 50 members of the Italian parliament and asks performers to indicate if they have ever been accused, prosecuted or convicted. From the performers' point of view, this application is a game - of physiognomic nature, given that many faces are not known; each performer sees, in a fixed amount of time, a number of photos which raises as a function of the performer's ability; a higher number of photos gives to players higher possibility of improving their ranking. At the end of each round, the system presents a report with correct answers and the ranking of performers. Rules are used for assessing performers' ability and for creating micro-tasks based on the dynamically evolving quality of performers; we noted that games with higher variability of tasks are most appreciated by users (281 vs. 565 executions), and lead to a considerably higher evaluation precision (65% vs. 82% respectively).

In the **politician ranking** application, the task is to produce a total ranking of 25 politicians; at each interaction, the performer is presented a pair of politicians and is asked to chooses the one she likes the most. In this application, used by 159 distinct performers, the system performs an ordering task by splitting it into micro-tasks with pairwise politician comparisons. Control production rules update the current `score` of politicians after each comparison by using the Elo rating system [5].

Finally, in the **politician classification** application, performers are asked to classify the political affiliation of 30 members of the Italian parliament – this application is similar to the running case study of the paper, with *six* Italian parties instead of two. Performers are provided with a set of photos of politicians, with associated names; there is no time limit, and performers are encouraged to use search engines. This application is therefore an example of human computation whose execution control aims at result precision and spammer detection.

Within the politician classification application we performed eight experiments, each featuring a different set of control rules similar to those presented in Section 3; we executed a total of 593 micro-tasks, involving 105 unique users. Table 2 reports, for each configuration, a short description, the rules that were used in the specific configuration, the number `NE` of executions, the average duration time `AT` of each micro-task, the number `NR` of object re-planning, the number `NP` of performers, the number `NS` of identified spammers, and the precision `PR` at the end of the experiment. All experiments use variations of the control rules 1, 2, and 6, and of the Appendix A; the table reports only the optional rules, specific to the experiment. Tasks are considered closed when all their objects are fully evaluated.

Figure 5 shows the overall experiment's precision, the average performer's precision, and the average degree of agreement of performers. We measured the users' and experiments' precision against the available gold truth (the actual affiliation of each politician), but we did not used it for controlling spammers, so to simulate the worst case scenario were the attainable truth is the one agreed by the involved performers.

Fig. 4 (a) - (d) plot the precision and number of object evaluations as a function of the completed micro-task executions. We stressed four analysis dimensions: *result production*, *object re-planning*, *spammer identification*, and *spammer threshold tuning*.

[6] `http://www.search-computing.org/www2013`

Table 2: Description of experiments of the politician classification experiment.

	Description	Rules	NE	AT	NR	NP	NS	PR
Exp.1	*Maj@7*		90	21	N/A	17	N/A	0.63
Exp.2	*Maj@3/Maj@7*	9	65	29	17	12	N/A	0.53
Exp.3	*Maj@3/Maj@5/Maj@7*	9	50	40	20	11	N/A	0.80
Exp.4	*Maj@3/Maj@7/Maj@15*	9	74	21	20	22	N/A	0.80
Exp.5	*Exp.2,Early Spam@0.5*	5, 7, 9	82	31	24	24	4	0.72
Exp.6	*Exp.2,Early Spam@0.6*	5, 7, 9	74	32	20	27	8	0.83
Exp.7	*Exp.2,Early Spam@0.7*	5, 7, 9	90	48	27	22	8	0.75
Exp.8	*Exp.2,Late Spam@0.5*	4, 7, 9	73	17	23	24	2	0.81

Result production. We experimented with two result production policies. Exp.1 – *Late Policy* – produces object's results as majority answers after 7 evaluations; Exp.2 – *Early policy* – adopts the *Majority Result* policy of Rule 6, where a result is immediately produced if, after three executions, all the involved performers agree; otherwise, 4 additional evaluations are planned. As shown in Table 2 and Figure 4, the *Early policy* is able to considerably reduce the number of executions required for task closing, at the cost of a considerable quality penalization due to the possibility of performers agreement on wrong classification.

Object replanning. In the second set of experiments (Exp.2, Exp.3, and Exp.4), we compared three variations of the control logic expressed by Rule 9 for object replanning. While Exp.2 performs a single stage of replan for objects which fail to have an early majority after 3 evaluations, Exp.3 and Exp.4 adopt a two-staged replanning policies, respectively testing for majority on each object after 5/7 evaluations or after 7/15 evaluations. As displayed in Figure 4 b), both Exp.3, and Exp.4 achieve a considerably higher precision w.r.t. Exp.2; they differ for the number of executions required for completion.

Spammer identification. The third set of experiments, displayed in Figure 4 d), exploited the re-planning policy of Exp.3 while adding spamming detections capabilities. Exp.5 and Exp.8 respectively implement the *early* and *late* evaluation of wrong answers (by rules 5 and 4) and a variant of Rule 7 where spammers are identified as performers with at least 50% of wrong answers. Both experiments required an higher number of micro-task executions compared to Exp.3; Exp.5 detected 8 spammers, while Exp.8 detected only 2 spammers.

Spammer threshold tuning. Finally, we performed a fine tuning of the threshold for judging a performer as spammer. Setting the threshold is critical: while a high threshold value may miss spammers, a low threshold value may detect too many performers as spammers. We respectively required wrong answers to be 60% in Exp.6 and 70% in Exp.7; a close comparison of solutions in Figure 4 shows that the intermediate choice of Exp.6 has better performances.

Figure 6 shows the number of activations of the various classes of rules during the execution of the experiments. Replanning calls for additional triggering of both control rules (more control statistics to update) and result rules (more closures to be done). Execution rules only depend on the re-planning policies, and their executions are slightly higher for spamming control experiments as they have to recompute the aggregates relative to invalid micro-tasks of spammers.

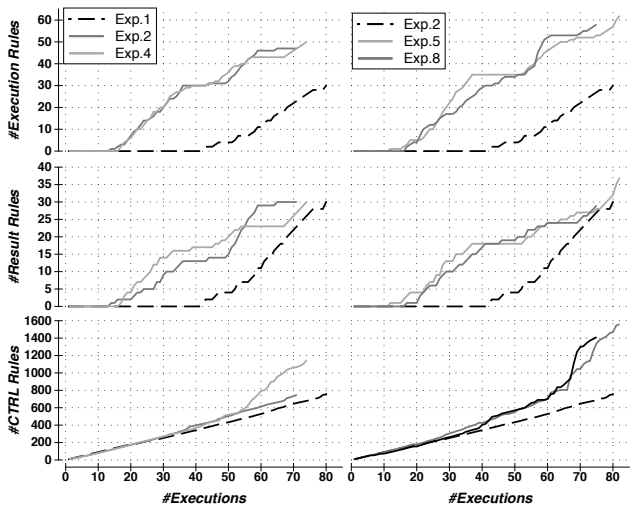

Figure 6: Activations of rules of different classes.

5. RELATED WORK

The increasing importance of high level systems for the programming of human computation tasks is testified by the many startups[7] and crowdsourcing systems [4] that have been proposed in the latest years. Most approaches rely on an **imperative programming model** to specify the interaction with crowdsourcing services. *Turkit* [12] offers a scripting language for programming iterative tasks on Amazon Mechanical Turk. *RABJ* [9] is a proprietary human computation engine developed by Metaweb to enhance the data processing pipelines of Freebase with human judgements. It offers very simple built-in control logic (e.g., limiting the maximum number of votes or the types of responses), while complex controls are externalised to client applications written in HTML and Javascript. *Jabberwocky* [1] transparently manages several crowdsourcing platforms to explicitly manage information about performers and their connections and provides parameterised functions for standard human activities. Jabberwocky can be procedurally programmed and compiles into a functional programming framework inspired by MapReduce. All of the cited frameworks allow defining human computations by procedural programming.

Recent works propose approaches for human computation which are based on **high level abstractions, sometimes of declarative nature**. In [15], authors describe a language that interleaves human-computable functions, standard relational operators and algorithmic computation in a declarative fashion. *Qurk* [13] is a query system for human computation workflows that exploits a relational data model, SQL to express queries, and a UDF-like approach to specify human tasks. *CrowdDB* [6] also adopts a declarative approach by using CrowdSQL (an extension of SQL) both as a language for modeling data and to ask queries; human tasks are modelled as crowd operators in query plan, from which it is possible to semi-automatically derive task execution interfaces. Similarly to *CrowdDB*, the *DeCo* [16] system allows SQL queries to be executed on a crowd-enriched datasource; however, human tasks are defined as *fetch* and *resolution* rules programmed in a scripting language (Python)

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

and defined in the schema of the data source. *CrowdLang* [14] supports workflow design and execution of tasks involving human and machine activities. It incorporates explicit abstractions for group decision processes (e.g., voting and consensus mechanisms) and human computation tasks (e.g., contest, collaboration).

None of the these works dwell into the problem of specifying the control associated with the execution of human tasks, leaving its management to opaque optimisation strategies.

In designing crowdsourcing control, we have been inspired by several applications of human computations. Among them, [20] compares seven strategies for improving the quality and diversity of worker-generated explanations of social analysis tools; [10] presents alternatives in allocating tasks to workers; and [8] compares some alternatives for involving Mechanical Turk users in terms of their cost and quality.

We used methods in active rule design originally defined in [19]; specific rule analysis methods are from [2].

6. CONCLUSIONS

Supporting the dynamic control of crowdsourcing applications is increasingly important; however, most crowdsourcing platforms do not provide adequate solutions. Many platforms hide the control logic, a few expose limited program interfaces. In our framework we focused on reactive control of human computations by designing and deploying active rules for crowdsourcing control; our approach is complemented by a design method for crowdsourcing applications which uses standard operation types and model-driven transformations. In this way, rules have a default version that can be automatically derived from application design, but they can be modified or extended so as to implement arbitrary and sophisticated control policies.

We detailed in a running case study the exact structure of the relational data (control mart) and of rules, showing that simple rule substitutions or condition re-writings enable the encoding of different control policies; these are presented through extensive examples applied to classical human computations. The proposed approach is a good compromise between the conflicting requirements of design automation, flexibility, and expressive power.

7. ACKNOWLEDGMENTS

The authors acknowledge support from *i)* the ERC "Search Computing" project, and *ii)* the EC's FP7 "CUbRIK" project.

8. 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. ACM, 2011.

[2] E. Baralis, S. Ceri, and J. Widom. Better termination analysis for active databases. In *Rules in Database Systems*, pages 163–179, 1993.

[3] 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.

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

[5] A. E. Elo. *The rating of chessplayers, past and present*. Arco Pub., New York, 1978.

[6] 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.

[7] W. H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[8] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with Mechanical Turk. In *SIG-CHI Conf. on Human factors in comp. sys.*, pages 453–456. ACM, 2008.

[9] S. Kochhar, S. Mazzocchi, and P. Paritosh. The anatomy of a large-scale human computation engine. In *HCOMP '10*, pages 10–17. ACM, 2010.

[10] M. Kosinski, Y. Bachrach, G. Kasneci, J. Van-Gael, and T. Graepel. Crowd iq: measuring the intelligence of crowdsourcing platforms. In *Web Science Conf. 2012 (WebSci)*, WebSci '12, pages 151–160. ACM, 2012.

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

[12] 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.

[13] 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.

[14] 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.

[15] A. G. Parameswaran and N. Polyzotis. Answering queries using humans, algorithms and databases. In *CIDR 2011*, pages 160–166, Asilomar, CA, USA, January 2011.

[16] 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.

[17] 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.

[18] 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.

[19] J. Widom and S. Ceri. *Active Database Systems*. Morgan Kaufmann, 1996.

[20] W. Willett, J. Heer, and M. Agrawala. Strategies for crowdsourcing social data analysis. In *SIG-CHI Conf. on Human Factors in comp. sys.*, pages 227–236. ACM, 2012.

APPENDIX

A. RULES FOR THE RUNNING EXAMPLE

The complete rule set of the example is constituted by the seven rules below and by rules 1, 2, 6; and optionally one of (3, 4, 5), 7, 8, and one of (9, 10).

```
rule ObjectEvalCounter
     e: UPDATE FOR Execution[Answer]
     a: SET Object_CTRL[oid==NEW.oid].Eval += 1

rule PerformerEvalCounter
     e: UPDATE FOR Execution[Answer]
     a: SET Performer_CTRL[pid==NEW.pid].Eval += 1

rule TaskEvalCounter
     e: UPDATE FOR Execution[Answer]
     a: SET Task_CTRL[tid==NEW.tid].CompExec += 1

rule CurrentMajorityDem
     e: UPDATE FOR Object_CTRL[Dem,Rep]
     c: NEW.Dem > NEW.Rep
     a: SET NEW.Answer = 'Dem'

rule CurrentMajorityRep
     e: UPDATE FOR Object_CTRL[Dem,Rep]
     c: NEW.Rep > NEW.Dem
     a: SET NEW.Answer = 'Rep'

rule CurrentMajorityTie
     e: UPDATE FOR Object_CTRL[Dem,Rep]
     c: NEW.Rep == NEW.Dem
     a: SET NEW.Answer = 'Undefined'

rule TaskControlOnClosedObject
     e:  UPDATE FOR Politician[Status]
     c: NEW.Status == 'Complete'
     a: SET Task_CTRL[tid==NEW.tid].CompObj += 1
```

B. GRAMMAR OF RULE LANGUAGE

⟨*rule*⟩ ::= 'rule' ⟨*rulename*⟩ 'e:' ⟨*eventclause*⟩ ['c:' ⟨*condition-clause*⟩] 'a:' ⟨*action-clause*⟩

⟨*event-clause*⟩ ::= 'TIMER FOR' ⟨*TABLE*⟩ ⟨*timer-expression*⟩
| 'INSERT FOR' ⟨*TABLE*⟩ | 'DELETE FOR' ⟨*TABLE*⟩
| 'UPDATE FOR' ⟨*TABLE*⟩ ['['⟨*ATTRIBUTE*⟩{,⟨*ATTRIBUTE*⟩}']']

⟨*condition-clause*⟩ ::= (⟨*predicate*⟩) | 'not' ⟨*predicate*⟩
| ⟨*predicate*⟩ 'and' ⟨*predicate*⟩ | ⟨*predicate*⟩ 'or' ⟨*predicate*⟩

⟨*predicate*⟩ ::= ⟨*expression*⟩ ⟨*comp*⟩ ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*expression*⟩ ⟨*op*⟩ ⟨*expression*⟩
| ⟨*op*⟩ ⟨*expression*⟩
| (⟨*expression*⟩) | ⟨*constant*⟩
| ⟨*variable*⟩.⟨*ATTRIBUTE*⟩ | ⟨*selector*⟩.⟨*ATTRIBUTE*⟩

⟨*selector*⟩ ::= ⟨*TABLE*⟩[⟨*condition-clause*⟩]

⟨*action-clause*⟩ := ⟨*statement*⟩ [{,⟨*statement*⟩}]

⟨*statement*⟩ ::= 'IF' ⟨*condition-clause*⟩ 'THEN' ⟨*action-clause*⟩ ['ELSE' ⟨*action-clause*⟩]
| 'FOREACH' ⟨*variable*⟩ 'IN' ⟨*selector*⟩.⟨*ATTRIBUTE*⟩⟨*action-clause*⟩
| 'SET' ⟨*selector*⟩'.'⟨*ATTRIBUTE*⟩ '=' ⟨*expression*⟩
| 'SET' ⟨*variable*⟩'.'⟨*ATTRIBUTE*⟩ '=' ⟨*expression*⟩
| 'DELETE FROM' ⟨*selector*⟩
| 'INSERT INTO' ⟨*TABLE*⟩ (⟨*expression*⟩ {,⟨*expression*⟩})
| ⟨*FUNCTION*⟩ (⟨*parameter*⟩{,⟨*parameter*⟩})

⟨*parameter*⟩ ::= ⟨*variable*⟩ | ⟨*constant*⟩

⟨*variable*⟩ ::= 'NEW' | 'OLD' | ⟨*variable-name*⟩

⟨*comp*⟩ ::= '==' | '>' | '<' | '>=' | '<=' | '!='

⟨*op*⟩ ::= '+' | '-' | '*' | '/' | '+='

⟨*timer-expression*⟩ ::= 'EVERY' ⟨*time-constant*⟩
| 'AT' ⟨*time-constant*⟩

<*TABLE*>, <*ATTRIBUTE*>, <*FUNCTION*>, <*variable-name*>, <*constant*>, <*time-constant*> are strings