# Towards a Language for Non-Expert Specification of POMDPs for Crowdsourcing

**Christopher H. Lin  Mausam  Daniel S. Weld**
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{chrislin,mausam,weld}@cs.washington.edu

**Introduction** Crowdsourcing requesters are trapped between a rock and a hard place. Typically they specify their crowdsourcing workflows procedurally, but current languages commit them to overly strict and static policies that waste human effort. While optimizing workflows with more sophisticated tools like POMDPs can significantly reduce labor costs, such advanced AI techniques are hard to use and understand. We report on our progress in developing CLOWDER, a recently proposed system that would allow crowdsourcing requesters to achieve both their desires (Weld, Mausam, and Dai 2011). Such a system frees requesters from needing to resort to sub-optimal techniques that use approximate heuristics or hire a planning expert to formally define and solve their problems.

CLOWDER provides the user with an adaptive programming language (extending (Pinto et al. 2010; McAllester 1999) to handle partial observability and non-expert usability) that looks and feels like Lisp, yet abstracts over POMDPs so that non-experts can write POMDPs without knowing anything about them. For instance, suppose a requester would like to write a dynamic workflow that uses crowdsourcing to label training data. An adaptive program that achieves this goal might poll crowd workers for labels until the system is "confident" it can stop and return a label. However, for this program to make optimal decisions, or in other words, for it to know when it is confident enough to stop, it needs to both maintain some state that represents a current belief about what the correct label is, and know how to update this belief after every label observation. Therefore, for requesters to write such programs, they must understand not only how to model some world dynamics, but also how to update probabilistic beliefs. Instead of hiring a planning expert to write such a program or to handcraft a custom POMDP for this simple voting problem (Dai, Mausam, and Weld 2010; Kamar, Hacker, and Horvitz 2012), requesters who are unfamiliar with AI should be able to write a very simple program that abstracts away from state variables, probabilities, and notions of "confidence": either ask another worker for another label and recurse, or return the label with the most number of votes. CLOWDER provides such functionality.

Figure 1 shows a CLOWDER program for labeling (voting) that implements the algorithm we just described. It assumes there are two possible labels and reposes the problem as one of discovering if the first label is better than the second. Notice that the program makes no reference to any POMDP components in its definition. Any requester who can program can write the program. The programmer does not need to specify some hidden state that represents the correct answer. Instead, the programmer provides a choice point in the program, and CLOWDER automatically compiles the program into a POMDP, and then produces an optimal policy that determines the optimal branch to take at runtime.

```
(define (vote q a0 a1 c0 c1)
  (choose
    (if  (crowd-vote q a0 a1)
         (vote q a0 a1 (+ c0 1) c1)
         (vote q a0 a1 c0 (+ c1 1)))
    (if (> c0 c1) #t #f)))
```

Figure 1: A CLOWDER program for labeling that manages uncertainty without exposing it to the user. `q` is an input question, `a0,a1` are two possible answers, and `c0,c1` count the number of votes for each choice.

Suppose a requester wanted to write the iterative-improvement workflow (Little et al. 2009). Figure 2 shows a program written in the CLOWDER language that uses iterative-improvement to crowdsource a caption for an image. There are three choice points. The program can either improve the best caption so far, or it can ask a worker about which of the current captions is better and recurse with the new information, or it can return the best caption. Again, notice that the program contains no references to uncertainty of any kind. It also allows the user to use the already written `vote` program, just like an ordinary programming language.

**The High Level Details** We now use a simple example program (Figure 3) to provide a high level understanding of how CLOWDER works. The program, `improve`, might be one that a crowdsourcing expert would write for improving a piece of text. It is a simplified version of iterative-improvement that removes voting. Indeed, it just repeatedly improves the text until it decides the text is improved enough, and then terminates by returning the text.

The semantics of a CLOWDER program incorporate un-

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

```
(define (it-i image worse-text better-text)
  (choose
    (it-i image better-text
              (c-imp better-text))
    (if (vote image better-text
              worse-text 0 0)
       (it-i image worse-text better-text)
       (it-i image better-text worse-text))
    better-text))
```

Figure 2: A CLOWDER program for iterative-improvement on descriptions for images.

```
(define (improve text)
  (choose
    (improve (c-imp text))
    text)))
```

Figure 3: A CLOWDER program for improving a piece of text. `text` is the current text.

certainty. While to the non-expert user the behavior is as expected, the user with knowledge of AI understands that in the execution of a CLOWDER program, all variables are actually bound to two values, and thus all expressions evaluate to two values. The first value, the *Normal value*, is the usual value that the non-expert user sees and understands, and is the same as it would be in any other programming language. For example, the argument `text` is bound to a string. The second value is a *Clowder value* that can be unobservable, and hence will be represented by a distribution in the system. This value is the value of a state variable in the POMDP that CLOWDER compiles from the program.

Since users can not be expected to define the domains of CLOWDER values, CLOWDER bootstraps by relying on contributions from experts. The CLOWDER system contains a library of *primitives*, which experts may contribute to. Primitives are essentially probabilistic models of functions. For instance, `improve` uses the primitive `c-imp`. To the non-expert user, `c-imp` is an API call to some labor market that will return an improved piece of text. Since `c-imp` is a primitive, an expert has defined a model for it. The model both describes the domain of the argument, and provides a stochastic description of the output given the inputs. For instance, the expert can define the CLOWDER value of the argument to `c-imp` to be some $q \in [0, 1]$ to represent the unobservable quality of the text. Further, the expert can specify the probability that an output text has quality $q = 0.5$ given that the input text has quality $q = 0.2$ to be 0.8. Then, CLOWDER infers that the domains of all the `text` variables are also $[0, 1]$. CLOWDER will also be able to maintain a distribution over the possible qualities that `text` can have at all points in the program. When defining a primitive, the expert must also specify a cost for the primitive (*e.g.*, the cost of `c-imp` can be 5 cents).

**Utilities and Goals** We now address how users can tell CLOWDER what they believe is optimal behavior. Since optimality is different for every user, we need the flexibility to construct different utility functions or goals for individual users. CLOWDER assumes that executing a primitive incurs a cost defined by the primitive, but that the user uses one of CLOWDER's goal or utility eliciation modules to provide information about the overall program objective.

For instance, for the voting program, CLOWDER provides an *accuracy module* that simply asks the user for a desired accuracy, and converts the desired accuracy into a goal belief state. Such a module can work equally well for any program that outputs a "correct answer." In CLOWDER, the user can simply specify this module as the goal module for their program. A user can also limit the amount of money that is spent, by writing a budget into the program.

However, we note that this accuracy module is actually unable to guarantee the desired accuracy. Since CLOWDER does not know the ground truth, the best it can do is guarantee an expected accuracy. The user may actually end up with much worse results. If such behavior is unacceptable to the user, CLOWDER can also provide a *best-effort module* that does not require input from the user, but simply attaches a positive utility for returning the correct answer. Such a module can be useful, for example, when the user has many labelling tasks that need to be solved. If the user includes a budget in the program, CLOWDER then spends the entirety of the budget and does the best it can by dynamically figuring out how much money to put into each task so that harder tasks receive more of the budget.

**Final Thoughts** We note that CLOWDER is still a work-in-progress, and requires a user study to prove ease-of-use. Additionally, CLOWDER would benefit from a wider variety of modules that address different kinds of programs, like those that return artifacts with intrinsic qualities. CLOWDER would also benefit from a typed language, so that users can more easily use primitives. We end by observing that since CLOWDER relies on expert-defined primitives and goal/utility modules, it is coincidentally a crowdsourced system that delivers crowdsourcing systems.

**Acknowledgements** We thank the anonymous reviewer for their comments. This work was supported by the WRF / TJ Cable Professorship, Office of Naval Research grant N00014-12-1-0211, and National Science Foundation grants IIS 1016713 and IIS 1016465.

## References

Dai, P.; Mausam; and Weld, D. S. 2010. Decision-theoretic control of crowd-sourced workflows. In *AAAI*.

Kamar, E.; Hacker, S.; and Horvitz, E. 2012. Combining human and machine intelligence in large-scale crowdsourcing. In *AAMAS*.

Little, G.; Chilton, L. B.; Goldman, M.; and Miller, R. C. 2009. Turkit: tools for iterative tasks on mechanical turk. In *KDD-HCOMP*, 29–30.

McAllester, D. 1999. Bellman equations for stochastic programs.

Pinto, J.; Fern, A.; Bauer, T.; and Erwig, M. 2010. Robust learning for adaptive programs by leveraging program structure. In *ICMLA*.

Weld, D. S.; Mausam; and Dai, P. 2011. Human intelligence needs artificial intelligence. In *HCOMP*.