CrowdSheet: An Easy-To-Use One-Stop Tool for Writing and Executing Complex Crowdsourcing

Rikuya Suzuki, Tetsuo Sakaguchi(✉), Masaki Matsubara, Hiroyuki Kitagawa, and Atsuyuki Morishima

University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan
rikuya.suzuki.2015b@mlab.info, {saka,masaki,mori}@slis.tsukuba.ac.jp, kitagawa@cs.tsukuba.ac.jp

Abstract. Developing crowdsourcing applications with dataflows among tasks requires requesters to submit tasks to crowdsourcing services, obtain results, write programs to process the results, and often repeat this process. This paper proposes CrowdSheet, an application that provides a spreadsheet interface to easily write and execute such complex crowdsourcing applications. We prove that a natural extension to existing spreadsheets, with only two types of new spreadsheet functions, allows us to write a fairly wide range of real-world applications. Our experimental results indicate that many spreadsheet users can easily write complex crowdsourcing applications with CrowdSheet.

Keywords: Rapid development · Complex crowdsourcing
Expressive power analysis

1 Introduction

Crowdsourcing involving dataflow among tasks (i.e., the results of some tasks affect other tasks) is called *complex crowdsourcing*, and is a promising approach for a wide range of applications [14] such as writing articles and filling in tables. Complex crowdsourcing is ubiquitous even in real-world applications. For example, asking people to provide photos that contain suspected "red imported fire ants" and a location and to filter out obviously different ones, and then asking experts to check whether the remainder really are red imported fire ants, is complex crowdsourcing.

As crowdsourcing allows us to realize things previously impossible with computers only, enabling many people to easily *develop* complex crowdsourcing will have an impact on people's problem-solving abilities. However, the development is not an easy task for many people at present.

As a running example, suppose that we are holding an academic conference with parallel programme sessions (e.g., session 1 A and 2A in room A; session 1B and 2B in room B; session 1 A and 1B at the same time), and want to estimate

J. Krogstie and H. A. Reijers (Eds.): CAiSE 2018, LNCS 10816, pp. 137–153, 2018.
https://doi.org/10.1007/978-3-319-91563-0_9

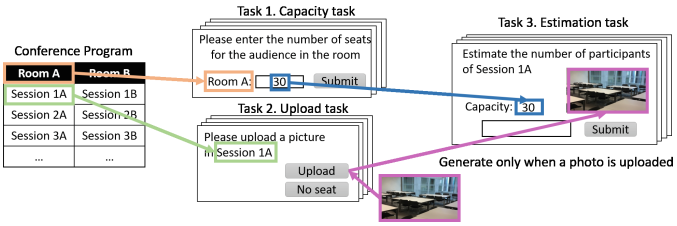

Fig. 1. Flow for complex crowdsourcing with running example: (1) Capacity task; enter the number of audience seats in the room. (2) Upload task; upload a photo taken in the session or report "no seat". (3) Estimation task; estimate the number of attendees in the session.

the number of attendees in each session except demos and posters. To this end, we solve our problem using three types of microtasks. The underlying idea is that we may be able to estimate the number of attendees in each session, if we know the room capacity (i.e., the number of audience seats in each room) and have photos taken in the session. Figure 1 illustrates the flow. First, for each room, we make a "capacity task" to ask workers to enter the number of audience seats in the room. Second, for each session of the program, we create an "upload task" to ask workers to upload a photo taken in the session if the audience seats exist (e.g., in general presentation sessions), or to report that there is no audience seats if the seats are removed in some sessions such as demos and posters. If a worker uploads a photo with audience seats, we submit the third task, called the "estimation task" to ask workers to analyze the picture and the room capacity and to estimate the number of attendees in the session. If it is reported that there is no audience seats (e.g., in demo & poster sessions), we do not issue the estimation task.

The flow can be implemented with current crowdsourcing services as follows: (1) Submit the capacity and upload tasks to the crowdsourcing service: (2) wait for the workers to complete the tasks, and then download the results in some manner: (3) check whether each uploaded picture was taken in a room with audience seats and, if so, submit an estimation task for the session and download the results. Using this procedure, we need to submit tasks to crowdsourcing services, obtain results, write programs, and use tools to process the results. This process is often repeated. If we want to improve data quality, we have to implement our own code for it (e.g., duplicated tasks), which is not separated from the essential logic of the application.

This paper proposes CrowdSheet, an easy-to-use, one-stop tool for implementing complex crowdsourcing (Fig. 2). CrowdSheet provides a spreadsheet interface for easily writing complex dataflows with a variety of microtasks, with crowdsourcing services such as Amazon Mechanical Turk (MTurk) in its backend. Whereas many other solutions focus on how to optimize the execution plans, our focus is on making it easy for a wide range of people to exploit the power of complex crowdsourcing. Our design principle separates the concerns on aspects

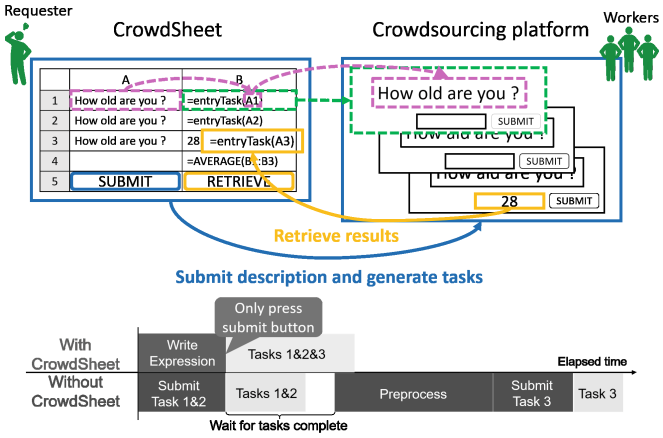

Fig. 2. (Top) CrowdSheet Overview: Tasks are submitted to crowdsourcing platforms according to the dataflow description in spreadsheets. (Bottom) Execution Process: Without CrowdSheet, the user has to wait for the first set of tasks to be completed, download the result, often filter it, and upload the second set of tasks based on the results. The process becomes more complicated if the user has more than three sets of tasks. With CrowdSheet, the user is released immediately after writing expressions and pressing the submit button and the spreadsheet cells will be filled with the results of completed tasks.

such as quality and costs from the essential logic of applications; CrowdSheet accepts independent modules implementing techniques for generating alternative execution plans [24] (omitted in this paper).

With CrowdSheet, the user is released after writing simple expressions and pressing the submit button. Tasks are *automatically* submitted and the spreadsheet cells are *gradually filled* with the results of completed tasks.

Because of its simplicity, the spreadsheet paradigm has been widely accepted by people who are not IT experts. Each cell contains either a value (numerical or string) or a function to compute a value whose parameters are often taken from other cells. CrowdSheet builds on this paradigm and provides two predefined functions to define and invoke tasks, whose parameters are often taken from other cells.

Although the idea is simple, coming up with a *good design* was non-trivial. We carefully designed CrowdSheet as a natural extension of existing spreadsheets with only two additional spreadsheet functions, while guaranteeing reasonable expressive power with our theoretical analysis results. As shown in Sect. 6, more than 60% of 33 participants who usually had experience using spreadsheets could implement complex crowdsourcing using CrowdSheet.

The contributions of this paper are as follows.

Spreadsheet Paradigm for Crowdsourcing. This paper shows a set of functions allows us to implement complex crowdsourcing with the spreadsheet paradigm. The design conforms to a common spreadsheet framework and extends

it naturally. A running example is used to demonstrate how CrowdSheet easily facilitates the implementation.

Theoretical Analysis. Theoretical analysis is conducted to identify the expressive power of CrowdSheet. More specifically, a class of programs in another language that is equivalent to CrowdSheet expressions is identified and used to demonstrate its limitation. In addition, we explain that some of the crowdsourcing applications surveyed in [26] can be implemented as long as certain conditions are satisfied.

Implementation. We give constructive proof of our idea by implementing CrowdSheet with Microsoft Excel and two crowdsourcing platforms, although our concept is not restricted to particular spreadsheets and crowdsourcing platforms. Through the explanation, we not only show that the concept is feasible, but also identify the detailed semantics of the extended functions for CrowdSheet.

Usability Evaluation. We evaluate whether ordinary spreadsheet users can use CrowdSheet to write complex crowdsourcing applications by recruiting users via a crowdsourcing service and asking them to write applications with CrowdSheet.

2 Related Work

Complex crowdsourcing beyond simply submitting a set of tasks leads to many attractive applications [5,7]. Although many tools have been proposed for *optimizing* complex crowdsourcing [10,18,21], which try to find execution plans considering execution times, monetary costs, and data quality, only a few abstractions that support the *development* of complex crowdsourcing applications have been proposed. For example, CrowdForge is MapReduce abstraction [14], CrowdLang uses control flow diagrams [19], CyLog uses a logic-based abstraction [11,20], and Lukyanenko and others discuss conceptual modeling principles for crowdsourcing [16]. Our spreadsheet paradigm is unique in that it not only is different from the others but can also foster *end-user development* of complex crowdsourcing applications, which we believe can make a meaningful difference in the real world. CrowdSheet has the same spirit as the system presented by Kongdenfha et al. [15], although the domain and problems are different. In addition, this paper discusses the appropriateness of the proposed design through theoretical analysis (that shows a natural and small extension of existing spreadsheets leads to wide ranging of expressive power) and usability studies (that show that many ordinary spreadsheet users can implement complex crowdsourcing with CrowdSheet). It is worth noting that the design of CrowdSheet is independent of the optimization techniques, in the sense that it can be combined with existing techniques as long as the description can be mapped to the abstractions used in the techniques. Such mappings can be automatically generated because CrowdSheet has formal semantics.

Various proposals have been made as regards office applications with crowdsourcing platforms in their backends [8,23]. In contrast to them, CrowdSheet

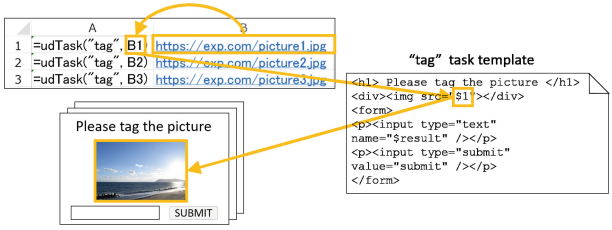

Fig. 3. `udTask` functions and the generated tasks by user-defined task template

is a *general-purpose* engine for implementing complex crowdsourcing. Google Spreadsheet and CrowdFill [22] actualize another approach related to spreadsheets and crowdsourcing. Whereas in this approach, the spreadsheet interface helps *workers* to enter data efficiently, CrowdSheet helps *requesters* to implement their complex crowdsourcing.

Tyszkiewicz [25] showed that a relational database can be implemented using only spreadsheet functions. Our theoretical result is compatible with it in the sense that the expressive power of CrowdSheet does not allow transitive closure.

Data quality is always an important issue in crowdsourcing [6,12,17]. CrowdSheet adopts a declarative feature that separates the application logic and mechanisms for improving data quality [24], where independent modules rewrite the code generated by CrowdSheet to propose alternative execution plans for improving data quality. Consequently, CrowdSheet can be combined with a variety of techniques (e.g., worker selection and majority consensus) for improving the data quality in crowdsourcing. Incorporation of related novel notions, such as social computing units [9], into our framework is an interesting future work.

3 CrowdSheet

CrowdSheet is a spreadsheet with a crowdsourcing platform in its backend (Fig. 2). It allows requesters to use functions to invoke microtasks and to control dataflows that can be used in the same manner as other spreadsheet functions (e.g., sum and average)[1].

TI-Functions. Task Invocation (TI) Functions invoke tasks. Each TI-function generates a task with specified parameters and submits it to the crowdsourcing platform. Subsequently, it receives the task result when the task is completed. Figure 2 shows an example of this. `entryTask` (String *question*) generates a task with a specified *question* and a text field, and submits it to the crowdsourcing platform. Another example is `choiceTask` (String *question*, Range *choices*), which generates and submits a task with a specified *question* and a list of *choices*, and returns the result.

[1] In our current implementation, TI-functions cannot be combined with other functions in the same cell. However, this is not an essential limitation because expressions in the cell can refer to the results of TI-functions in other cells.

CrowdSheet also allows us to invoke microtasks with user-defined task templates (Fig. 3). `udTask` (String *templateID*, Range *parameters*) is a TI-function that submits a task with a specified task template and returns the result. A requester can easily register their user-defined task templates to CrowdSheet. First, the requester writes a task template in HTML containing place holders where the values of *parameters* will be embedded. Next, they upload the file to our system with an associated *templateID* to be used when requesters call the function[2].

C-Functions. CrowdSheet has one C-function, called `evaluateIf`, which controls task invocations. `evaluateIf` (String *condition*, String *then-value*, String *else-value*) returns the *then-value* if the specified *condition* holds and returns the *else-value* otherwise. The two values can be specified by cell references. For example, the *condition* could be "A3 < 5," which means cell "A3" is smaller than 5, and the *then-value* refers to a cell containing a TI-function, and the *else-value* could be a value "None". A C-function is different from the usual if-function of spreadsheets in that it allows for lazy evaluation of tasks if it involves references to task invocations. Therefore, the TI-function referred to by the C-function is evaluated only when the C-function finds that the condition is true. C-functions are useful if a requester wants to conditionally invoke tasks, as in the estimation tasks in the running example.

How to Use CrowdSheet. The running example can be easily implemented as shown in Fig. 4. We assume that the sheet already has columns that store rooms

	A	B	C	D	E	F	G	H	I
1	Session	Room	Capacity	Picture	Estimation	Evaluate		Room	Capacity
2	Session 1A	=H2	=VLOOKUP($B2,$H$2:$I$4,2)	=udTask("upload",$A2)	=udTask("estimation",$A2:$D2)	=evaluateIf($D2="No seat", "None", $E2)		Room A	=udTask("capacity", $H2)
3	Session 1B	=H3	=VLOOKUP($B3,$H$2:$I$4,2)	=udTask("upload",$A3)	=udTask("estimation",$A3:$D3)	=evaluateIf($D3="No seat", "None", $E3)		Room B	=udTask("capacity", $H3)
4	Session 1C	=H4	=VLOOKUP($B4,$H$2:$I$4,2)	=udTask("upload",$A4)	=udTask("estimation",$A4:$D4)	=evaluateIf($D4="No seat", "None", $E4)		Room C	=udTask("capacity", $H4)
5	Session 2A	=H2	=VLOOKUP($B5,$H$2:$I$4,2)	=udTask("upload",$A5)	=udTask("estimation",$A5:$D5)	=evaluateIf($D5="No seat", "None", $E5)			
6								Submit	Retrieve

Fig. 4. CrowdSheet description in running example

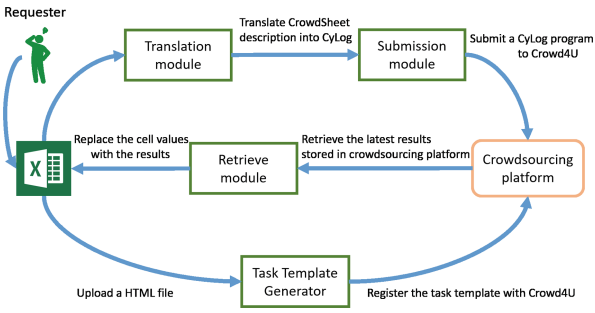

Fig. 5. CrowdSheet implementation

[2] Each task template is also associated with the payment information for workers.

(Column H) and sessions (Column A). First, a requester writes and uploads task templates for the three types of tasks. Then, they encode the logic in CrowdSheet as follows: (1) Put TI-functions into Column I of each room (row) for invocation of the capacity task, (2) put references to the results of the capacity tasks in Column C, (3) put TI-functions into Column D of each session (row) for invocation of the upload tasks, and (4) put TI- and C-functions into Columns E and F for invocation of the estimation tasks if the room for the session has audience seats. Note that a user can write TI- and C-functions for only one session and one room, and then drag the mouse to copy them for others.

Then, the user clicks on the "submit" button to submit tasks. When the user clicks on the "retrieve" button, the cells containing TI- or C-functions change to the result value if the task is complete at that time. Naturally, the user can click on the "retrieve" button at any point to obtain the intermediate results.

4 Implementation

Figure 5 shows the main components of our CrowdSheet implementation. We used Microsoft Excel and Crowd4U [13], but can adopt other services, such as Google Sheets and MTurk (We have already implemented modules for MTurk [24]).

In order to use CrowdSheet, the user has to download an Excel file with EXCEL add-ins for CrowdSheet. The sheet contains "submit" and "retrieve" buttons in it.

The submission module (implemented as an EXCEL add-in) passes the CrowdSheet description (i.e., all cells in the spreadsheet) to the translation module, which then translates it into an intermediate expression written in CyLog, a programming language for human-machine computations [11,20]. The logic encoded in the expression is executed by Crowd4U, and the tasks are submitted to crowdsourcing services such as MTurk and the Crowd4U native task pool. Workers then perform tasks on crowdsourcing services. If specified in the expression, Crowd4U dynamically generates tasks using the results of other tasks.

```
Pre1: Session(sid:1, rid:1, sname:"Session 1A");
Pre2: Room(rid:1, rname:"Room A");
Pre3: Session(sid:2, rid:2, sname:"Session 1B");
Pre4: Room(rid:2, rname:"Room B");
Pre5: Session(sid:3, rid:3, sname:"Session 1C");
Pre6: Room(rid:3, rname:"Room C");
Pre7: SRoom(sid:s, rid, sname, rname) <- Session(sid:s, rid, sname), Room(rid, rname);

R1: Capacity(rid, capacity)/open <- Room(rid, rname);
R2: Picture(sid, rid, sname, rname, URI)/open <- SRoom(sid, rid, sname, rname);
R3: !CapacityTask(rid, rname) <- ?Capacity(rid, capacity), Room(rid, rname);
R4: !UploadTask(sid, rid, sname, rname) <- ?Picture(sid, rid, sname, rname);
R5: Capacity(rid, capacity), Picture(sid, rid, URI){
      URI == "no_seat" {
                Estimation(sid, rid, result:0);
          } else {
                Estimation(sid, rid, result)/open;
          }
    }
R6: !EstimationTask(sid, rid, sname, rname, capacity, picture)
<- ?Estimation(sid, rid, result), Capacity(rid, rname, capacity), Picture(sid, rid, sname, URI);
```

Fig. 6. Fragment of a CyLog program

When the CrowdSheet user clicks on the "retrieve" button, the retrieve module (implemented as an EXCEL add-in) retrieves the latest results stored in crowdsourcing services at that time and inserts the retrieve results into the cells containing TI- or C-functions.

The user can also write and upload HTML files for user-defined task templates. The task template translator is an independent module that takes each HTML file as input, assigns a template id to it so that the template can be used in TI-functions.

4.1 CyLog Overview

As a means to define the formal semantics and discuss its expressive power, we use CyLog [20], a rule-based language for crowdsourcing. The basic data structure in CyLog is a *relation*, which is a table to deal with a set of *tuples* that conform to the *schema* of the relation. A program written in CyLog consists of three sections: `schema`, `rules` and `views`. The `schema` section describes the schema of the relations. The `rules` section has a set of rules, each of which *fires* (is executed) if its condition is satisfied. The `views` section describes HTML templates to be used as the interface with workers. In the following discussions, we explain only the `rules` section. The `schema` section is straightforward and we assume that they are appropriately given. The HTML templates in the `views` section are supplied by the task template translator in Fig. 5.

Facts and Rules. The main component of a CyLog program is the set of *statements*. Figure 6 shows a set of statements, each of which is preceded by a label (such as `Pre1`) for explanation purposes. A statement is either a *fact* or a *rule*. A rule has the form *head* ← *body*. In the figure, `Pre1` to `Pre6` are facts. `Pre7` and `R1` to `R6` are rules. Each *fact* or *head* is given in the form of an *atom*, while each atom consists of a predicate name (e.g., `Session`) followed by a set of *attributes* (such as `sid`, `rid`, and `sname`). Optionally, each attribute can be followed by a colon with a value (e.g., :"`Session 1A`") or an alias (e.g., :`s`). Each *body* consists of a sequence of atoms.

A fact describes that the specified tuple is inserted into a relation. For example, `Pre1` is a fact that inserts a tuple whose values for attributes `sid`, `rid`, and `sname` are `1`, `1`, and "`Session 1A`" respectively into relation `Session`[3].

A rule specifies that, for each combination of tuples satisfying the condition specified in the *body*, the tuple described in *head* be inserted into a relation. Atoms in the body are evaluated from left to right and variables are bound to values that are stored in the relation specified by each atom. For example, `Pre7` is a rule that inserts a tuple having `sid`, `rid`, `sname`, and `rname` into relation `SRoom` if `sid`, `rid`, and `sname` are in `Session` and the `rid` and `rname` in `Room`. In other words, for each combination of a tuple in `Session` and a tuple in `Room`

[3] CyLog adopts the *named perspective* [3], which means that the variables and values in each atom are associated with attributes by explicit *attribute names*, not by their positions.

whose `rid` attributes match each other, it inserts a tuple having `sid`, `rid`, `sname`, and `rname` values into relation `SRoom`.

Open Predicates. CyLog allows predicates to be `open`, which means that the decision as to whether a tuple exists in the relation is performed by humans when the data cannot be derived from the data in the database. For example, the head of `R1` is followed by `/open` and is an open predicate. If a head is an open predicate, CyLog asks humans to give values to the variables that are not bound to any values in the body (e.g., `capacity` in `R1` and `URI` in `R2`).

Task Predicates. A task predicate (preceded by `!`) represents a relation implementing a task pool, in which each tuple corresponds to a task instance. For example, `R2` defines a task predicate `!CapacityTask` in which each tuple corresponds to one capacity task. In general, each tuple in a task predicate is created for a combination of tuples including (1) ones to supply data to be presented to workers in the task screen and (2) open tuples to store the task results. For example, `R3` creates a capacity task instance for a combination of (1) a tuple in `Room(rid, rname)`, whose `rname` is used to show workers the name of the room in the task screen, and (2) a tuple in `Capacity(rid, capacity)`, where `capacity` contains an open value. In the rule body, `?Capacity(rid, capacity)` is a predicate that holds when it contains an open tuple. Note that open tuples for `Capacity` are supplied by `R1`.

Block Style Rules. Each rule $P \leftarrow P_1, P_2, \ldots, P_n$ can be written in the *block style* $P_1\{P_2\{\ldots\{P_n\{P;\}\ldots\}$. For example, `Pre7` in Fig. 6 can be written as follows:

```
Session(sid:s, rid, sname) {
      Room(rid, rname) {
        SRoom(sid:s, rid, sname, rname);
   }
}
```

where `SRoom(sid, rid, sname, rname)` is the head of the rule. The block style provides a concise expression when we have many rules that have the same body

```
SR-1: Cell(loc, value)/open <- Ready(loc);
SR-2: !Task_k(loc, d_1, ..., d_{n_k}) <- ?Cell(loc, value), Param_k(loc, d_1, ..., d_{n_k});

TP-1: Param_k(loc:(x,y), d_1, ..., d_{n_k}) <- Cell(loc:(x_1,y_1), value:d_1), ...,
                                               Cell(loc:(x_{n_k},y_{n_k}), value:d_{n_k});
TP-2: Ready(loc)<-Param_k(loc:(x,y), d_1, ..., d_{n_k});

CP-1: Cell(loc:loc_1, value:d_1), ...., Cell(loc:loc_m, value:d_m), cond{
CP-2:       Ready(loc:(x_b,y_b)) <- Param_k(loc:(x_b,y_b), d_1, ..., d_{n_k});
CP-3: } else {
CP-4:       Cell(loc:(x_b,y_b), value) <- Cell(loc:(x_c,y_c), value);
CP-5: }
```

Fig. 7. Patterns for CyLog code generation: SR (`SR-`*i*s), TP (`TP-`*i*s), and CP (`CP-`*i*s)

atoms (e.g., $P_1\{P_2; P_3;\}$ for $P_2 \leftarrow P_1; P_3 \leftarrow P_1;$). In addition, the block style allows us to use the `else` clause. For example, `R5` inserts a tuple having the result "0" into `Estimation(sid, rid, result)` if the `URI` value is "`no_seat`".

4.2 Semantics of CrowdSheet Descriptions

CyLog Program for the Running Example. `R1` to `R6` in Fig. 6 constitute a set of rules that is equivalent to the CrowdSheet description shown in Fig. 4. (Rules `Pre1` to `Pre7` are rules we intend to use to explain CyLog and do not have expressions in the figure.)

`R1` to `R4` generates two types of tasks for asking workers to count the number of seats in the room and take a picture of the session. `R5` states that if the results of the capacity task is not "0", the estimation result of the number of audiences is open; otherwise, we take the value "0" as the estimation result. `R6` generates a task for estimation of the number of people in the audience in each session, if the estimation result is open.

Mapping to CyLog Rules. The semantics of CrowdSheet descriptions is defined by a method that maps a CrowdSheet description to CyLog rules. A CrowdSheet description is mapped to CyLog rules consisting of the following three components (Fig. 7).

Shared Rules (SR) a set of rules to give common functionalities to generate tasks.
TI-function Pattern (TP) a set of rules generated for each TI-function appearing in the sheet.
C-function Pattern (CP) a set of rules generated for each C-function in the sheet.

The TP or CP for a function specified in the cell at $(x,\ y)$ is fired only once when their parameter values are ready. Note that $(x,\ y)$ is a pair of constant values, not variables.

Shared Rules and TI-function Pattern. Assume that we have a TI-function with Template k in a cell at $(x,\ y)$, and its parameters are stored in cells at $(x_1,\ y_1), \ldots, (x_{n_k},\ y_{n_k})$. Then, the translator generates a one SR (consisting of `SR-1` and `SR-2`) and one TP (consisting of two rules: `TP-1` and `TP-2`) for the function.

SR works as follows. First, `SR-1` makes the cell value open if the task is ready. Then, `SR-2` generates an instance of the Template-k task if the value is open.

TP works as follows. First, `TP-1` constructs a tuple to store parameters for the TI-function at $(x,\ y)$. Here, $Param_k$ is a predicate to store parameters for TI-functions with Template k, where the key *loc* stores the locations of the cells at which TI-functions are placed on the spreadsheet. As a result, `TP-1` inserts a tuple containing parameters for the TI-function at $(x,\ y)$ into $Param_k$. The predicate $Cell(loc : (x,\ y),\ value)$ stores the value in the cell at $(x,\ y)$.

Next, TP-2 states that the task is ready to be evaluated if we have all parameters to invoke the task. Here, the predicate $Ready(loc)$ stores the locations of those TI-functions that are ready to be evaluated.

If we had more than one TI-function for the same Template K, the translator would only generate TPs for the TI-functions. The shared rules do not depend on the locations of cells and are commonly used by every TP for Template k.

Rules for C-functions. Assume that we have a C-function in the cell at $(x,\ y)$, and the condition specified in the C-function requires values in cells at $(x_1,\ y_1), \ldots, (x_m,\ y_m)$. Note that the "then" and "else" clauses can refer to values in other cells. Here, we assume that the *then-value* parameter of the C-function refers to a TI-function located at $(x_b,\ y_b)$ and the *else-value* parameter refers to a value v at $(x_c,\ y_c)$.

Then, the translator generates rules in the following two steps. First, it generates the CP for the C-function, as shown in Fig. 7. CP-1 to CP-2 make the task at $(x_b,\ y_b)$ ready only when the condition holds and we have all of its parameters. CP-4 copies the value at $(x_c,\ y_c)$ to the cell at $(x,\ y)$ if the condition is not satisfied. Second, for each reference to a TI-function contained in the *then-value* or *else-value* parameters, the original rule for creating Ready tuples (i.e., TP-2) is removed, because the tuple is created by CP-2 when the condition is satisfied.

5 Expressive Power

In this section, we prove that there is a class $\mathcal{P}$ of CyLog programs such that every program in $\mathcal{P}$ has a CrowdSheet description equivalent to it, and vice versa.

5.1 CrowdSheet to CyLog

First, we identify the conditions satisfied by any CyLog program that defines the semantics of a CrowdSheet expression.

Theorem 1. *Let p be a CyLog program converted from a CrowdSheet description. Then, p satisfies all of the following conditions.*

C1. Every task generated by p returns only one value[4]*.*
C2. There exists a natural number N such that p generates at most N tasks.
C3. p generates tasks of predefined task templates only.

Proof. The conditions are derived from the limitations of spreadsheets and the CrowdSheet design. The first condition is derived from the fact that the output of each spreadsheet function is inserted into one spreadsheet cell. The second states that an infinite number of functions can not be written in a sheet because TI-functions have to be explicitly written in cells. The third is derived from the design policy stating that the task-design functionalities are out of

[4] Tasks can return more than one value with composite values (omitted owing to lack of space).

the spreadsheet paradigm, i.e., new templates of tasks can not be defined with spreadsheet functions. Every program mapped from a CrowdSheet description (Sect. 4.2) satisfies the three conditions. □.

Definition 1. *We define $\mathcal{P}$ as a class of CyLog programs that satisfies all conditions C1, C2, C3 identified by Theorem 1.*

5.2 CyLog to CrowdSheet

Second, we prove that every CyLog program in class $\mathcal{P}$ can be converted into a set of CyLog fragments of shared rules, TPs, and CPs. The fundamental idea of the proof is to map each task invocation to a virtual spreadsheet space, such that every task result is associated with a unique location (corresponding to a cell in the virtual spreadsheet). In this manner, every TI-function or C-function can refer to the results of other tasks in the mapped spreadsheet.

First, we consider a simple case; we prove that every program in $\mathcal{P}'$ can be converted into a set of CyLog fragment patterns.

Definition 2. *$\mathcal{P}'$ is a class of CyLog programs in which each program in $\mathcal{P}'$ satisfies all of C1, C2', and C3, where,*

- *C2' p generates exactly N tasks and no function uses the results of other functions as its parameters.*

Theorem 2. *Every program p in $\mathcal{P}'$ can be converted into a set of TPs and shared rules.*

Proof. Let $pred\_seq_{q,k}$ be a sequence of CyLog predicates (filled with constant values) that computes a set of parameters $d_1, \ldots, d_{n_k}$ to be used for invoking tasks with Template k. Here, q is a unique number associated to such a sequence, independent of k (i.e., q is unique in all $pred\_seq_{q,k}$ with any ks). Because the total number of task invocations is fixed by N, the number of task invocations generated by each $pred\_seq_{q,k}$ is also fixed. Let $N_{q,k}$ be that number. In addition, as no task uses the results of other tasks as its parameters, all parameters are constant values. Therefore, we can construct a set of cells in a virtual spreadsheet to store those parameters generated by $pred\_seq_{q,k}$. For that purpose, we "expand" $pred\_seq_{q,k}$ by generating the following CyLog fragments $N_{q,k}$ times.

$pred\_seq_{q,k}$, $cond_i$ {
  $Cell(loc : l_{new_1},\ value : d_1)$;
    ...
  $Cell(loc : l_{new_k},\ value : d_k)$;
}

Here, $cond_i (1 \leq i \leq N_{q,k})$ is a condition to choose a tuple for one task, and l_{new_i} is a constant value to represent a new cell location that has not been used yet. In this way, we obtain all "virtual" cells that can be referred to by task invocations.

Then, we generate the following CyLog fragment for each sequence of parameters for invoking a task with Template k.

TP-1: $Param_{k,q}(loc : l_{new},\ d_1,\ \ldots,\ d_{n_k})$
<- $Cell(id : l_1,\ value : d_1),\ \ldots,\ Cell(id : l_k,\ value : d_k)$,
TP-2: $Ready(loc)$ <- $Param_{k,q}(loc : l_{new},\ d_1,\ \ldots,\ d_{n_k})$;

Where l_i is the location of the parameter used for the ith parameter to invoke the task, and l_{new} is the new location to store the task result. The combination of the rules and shared rules is equivalent to CyLog rules generated from a CrowdSheet description. Therefore, there is a CrowdSheet description that is equivalent to any p in $\mathcal{P}'$. □

Note that each *loc* has a unique value such that each task result can be referred to by other tasks as a parameter. *loc* does not have to be in the form $(x,\ y)$, as long as the values serve as unique identifiers for spreadsheet cells.

This way, any value to be used as a parameter is assigned to a unique location. The following lemma is important.

Lemma 1. *For every task t in any program in the class $\mathcal{P}$, there exists a number $N_t < N$ s.t. t is ready after N_t tasks are completed.*

Proof. If this does not hold, then N is not a fixed number, which is a contradiction. □

Note that for every task t in programs in the class $\mathcal{P}'$, $N_t = 0$. We use Lemma 1 and gradually increment N_t to prove the following theorem (proof omitted).

Theorem 3. *Every p in $\mathcal{P}$ can be converted into a set of TPs, CPs, and SRs.* □

5.3 Expressive Power in Terms of Applications

An important question that entails from this is the question of what the expressive power means for real-world applications. To answer this question, we reviewed a survey [26] to ascertain the expressive power of CrowdSheet in terms of real-world applications. Note that being able to write an application does not equate to being able to implement it as is. We are interested in whether we can use CrowdSheet to implement applications that are functionally equivalent to the applications in surveyed.

There are two major limitations in CrowdSheet. The first is that each task is explicitly distinguished with others and must be an instance of a task template. This hinders CrowdSheet from implementing certain kinds of games where the interactive user interface is important. Secondly, it does not allow recursive task invocations so that any CrowdSheet expression cannot issue an infinite number of microtasks.

The survey [26] grouped crowdsourcing applications into four groups: voting systems, information sharing systems, games, and creative systems. First, most of the voting systems operated on MTurk and, in most cases, it is implemented with microtask interfaces. Therefore, as the number of tasks is fixed,

CrowdSheet can implement voting systems. Second, information sharing systems, including Wikipedia and YouTube. In this case, the ability to issue an infinite number of tasks is essential. Therefore, we surmise that CrowdSheet is not appropriate to implement such systems. Third, games are crowdsourcing applications that have been developed in line with ESP games [4]. For many games, an interactive interface is essential and a microtask interface only is not enough to implement effective game experiences. CrowdSheet is not appropriate to implement such games. Finally, creative systems, such as The Sheep Market [1] can be implemented if the number of tasks is limited. There are systems that can be implemented with the microtask interface, and we believe that certain kinds of creative systems can be implemented with CrowdSheet.

Our conclusion is that although CrowdSheet is not a perfect tool, it has a reasonable expressive power to be able to implement some of real-world applications.

6 Usability Evaluation

We recruited workers with prior experience using Microsoft Excel and spreadsheet functions and asked them to write CrowdSheet expressions to implement an application that is isomorphic to the running example presented in Sect. 1.

6.1 Settings

Worker Recruitment. We recruited workers as follows. First, we submitted 300 tasks to Yahoo! Crowdsourcing [2] to ask workers to answer questions on how to use Microsoft Excel, in order to find workers who can use spreadsheet functions, i.e., workers who are potentially able to write CrowdSheet expressions. The questions included how to use Excel's `if` function to solve a given problem in Excel. Three hundred (300) workers participated in the task, with each worker receiving 2 JPY (about 0.02 USD). 110 out of the 300 participants gave correct answers to the questions. Next, we submitted the "CrowdSheet" tasks (which we will explain next) to Yahoo!Crowdsourcing, with the condition that each worker could accept to perform only one of the submitted CrowdSheet tasks and would receive 10 JPY (about 0.09 USD) for the task. The tasks were visible only to the 110 workers who gave correct answers to the first task.

CrowdSheet Tasks. The CrowdSheet task consisted of two parts. In the first part, the workers were asked to write CrowdSheet expressions with only TI-functions for issuing `entryTask`s and `choiceTask`s having data flow among them. In the second part, the workers were asked to write CrowdSheet expressions with the combination of TI- and C-function to implement the running example in Fig. 1. Each part began with a tutorial to explain TI-functions `entryTask` and `choiceTask` (in the first part) and C-funcion `evaluateIf` (in the second part), and then asks workers to write CrowdSheet descriptions in the matrix of text forms simulating the CrowdSheet interface.

6.2 Results

Table 1 shows the results obtained. 33 out of the 110 workers accepted to perform the CrowdSheet tasks. 19 workers did not accept to perform the tasks. 58 workers did not access to the task at all. If we look at the numbers for Part 2, which asked workers to write expressions for a more difficult application, 15 workers out of the 33 workers gave correct answers and 6 workers gave correct answers with minor mistakes (such as typo and lack of the closing parenthesis). The remaining 12 workers made essential mistakes. An example of a mistake is a wrong choice of TI-function. For example, a worker issued `choiceTask` task instead of an estimation task, because `choiceTask` was used in the tutorial part. Another example is missing parameters of TI-tasks, which could be corrected if the spreadsheet supported interactive syntax error check. The sum of workers in the first two categories was 21, which is 63.6% of the workers who performed the CrowdSheet tasks. The results suggest that many workers with experience using Excel and its spreadsheet functions were able to use CrowdSheet, even though they were using it for the first time and had received only a simple CrowdSheet functions tutorial.

Table 1. Evaluation Result

	Correct (minor error)	Incorrect	Did not accepted	Ignored	Total
Part 1 (only TI-functions)	31 (3)	2	19	58	110
Part 2 (TI- and C-functions)	21 (6)	12			

The average of the time spent by participants for performing a CrowdSheet task (reading the tutorials and writing CrowdSheet descriptions for two scenarios) was about 10 min (598.2 s). For reference, a skilled programmer who use MTurk needed 28 min to implement tasks with the same data flow as the part 2 (submitting tasks and processing data) with MTurk, excluding the time required for him to wait for workers to complete the first step tasks. This suggests that we can expect many spreadsheet users can quickly learn and easily use CrowdSheet.

7 Summary

This paper proposed CrowdSheet, a spreadsheet for writing and executing complex crowdsourcing applications. We explained the design and a possible implementation, gave a theoretical analysis of its expressive power, and discussed the power and limitations for implementing real-world applications. Further, we presented experimental results showing that many users with spreadsheet experience are able to write complex crowdsourcing applications with CrowdSheet. In future work, we plan to make it generate codes in the form closer to that described by humans. This will make it easier to use CrowdSheet as a tool for implementing the basic functions of applications and rewriting the generated code to add functionalities beyond the expressive power of CrowdSheet.

Acknowledgement. This work was partially supported by JST CREST GrantNumber JPMJCR16E3, Japan.

References

1. The sheep market. http://www.thesheepmarket.com/
2. Yahoo! crowdsourcing. http://crowdsourcing.yahoo.co.jp
3. Abiteboul, S., et al.: Foundations of Databases: The Logical Level, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
4. von Ahn, L., et al.: Labeling images with a computer game. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 319–326. ACM (2004)
5. Akiki, P., et al.: Crowdsourcing user interface adaptations for minimizing the bloat in enterprise applications. In: Proceedings of ACM SIGCHI 2013, pp. 121–126 (2013)
6. Allahbakhsh, M., et al.: Quality control in crowdsourcing systems: issues and directions. IEEE Internet Comput. **17**(2), 76–81 (2013)
7. Artikis, A., et al.: Heterogeneous stream processing and crowdsourcing for urban traffic management. In: Proceedings of EDBT 2014, vol. 14, pp. 712–723 (2014)
8. Bernstein, M.S., et al.: Soylent: a word processor with a crowd inside. Commun. ACM **58**(8), 85–94 (2015)
9. Candra, M.Z.C., Truong, H.-L., Dustdar, S.: Provisioning quality-aware social compute units in the cloud. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 313–327. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45005-1_22
10. Franklin, M., et al.: Crowddb: answering queries with crowdsourcing. In: Proceedings of ACM SIGMOD 2011, pp. 61–72. ACM (2011)
11. Fukusumi, S., Morishima, A., Kitagawa, H.: Game aspect: an approach to separation of concerns in crowdsourced data management. In: Zdravkovic, J., Kirikova, M., Johannesson, P. (eds.) CAiSE 2015. LNCS, vol. 9097, pp. 3–19. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19069-3_1
12. Hung, N.Q.V., et al.: Erica: expert guidance in validating crowd answers. In: Proceedings of ACM SIGIR 2015, pp. 1037–1038. ACM (2015)
13. Ikeda, K., et al.: Collaborative crowdsourcing with crowd4u. PVLDB **9**(13), 1497–1500 (2016)
14. Kittur, A., et al.: Crowdforge: crowdsourcing complex work. In: Proceedings of the 24th Annual ACM UIST, pp. 43–52. ACM (2011)
15. Kongdenfha, W., et al.: Rapid development of spreadsheet-based web mashups. In: Proceedings of WWW 2009, pp. 851–860. ACM (2009)
16. Lukyanenko, R., et al.: Conceptual modeling principles for crowdsourcing. In: Proceedings of the 1st International Workshop on Multimodal Crowd Sensing, pp. 3–6. ACM (2012)
17. Lukyanenko, R., et al.: The IQ of the crowd: understanding and improving information quality in structured user-generated content. Inf. Syst. Res. **25**(4), 669–689 (2014)
18. Marcus, A., et al.: Crowdsourced databases: query processing with people. In: CIDR (2011). http://hdl.handle.net/1721.1/62827

19. Minder, P., Bernstein, A.: *CrowdLang*: a programming language for the systematic exploration of human computation systems. In: Aberer, K., Flache, A., Jager, W., Liu, L., Tang, J., Guéret, C. (eds.) SocInfo 2012. LNCS, vol. 7710, pp. 124–137. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35386-4_10
20. Morishima, A., et al.: Cylog/Game aspect: an approach to separation of concerns in crowdsourced data management. Inf. Syst. **62**, 170–184 (2016)
21. Park, H., et al.: Deco: a system for declarative crowdsourcing. Proc. VLDB Endow. **5**(12), 1990–1993 (2012)
22. Park, H., et al.: Crowdfill: collecting structured data from the crowd. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 577–588. ACM (2014)
23. Quinn, A.J., et al.: AskSheet: efficient human computation for decision making with spreadsheets. In: Proceedings of CSCW 2014, pp. 1456–1466. ACM (2014)
24. Suzuki, R., et al.: Crowdsheet: instant implementation and out-of-hand execution of complex crowdsourcing. In: Proceedings of ICDE 2018 (2018)
25. Tyszkiewicz, J.: Spreadsheet as a relational database engine. In: Proceedings of ACM SIGMOD 2010, pp. 195–206. ACM, New York (2010)
26. Yuen, M.C., et al.: A survey of crowdsourcing systems. In: 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), pp. 766–773. IEEE (2011)