Workflow by Example: Automating Database Interactions via Induction

Anthony Tomasic,[*] R. Martin McGuire[*], Brad Myers[**]

September 2006
CMU-ISRI-06-103

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Institute for Software Research International, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
[**]Human Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

This research is supported by grants from DARPA.

Keywords: workflow, database, data integration, data warehouse, automation, induction, programming by example

Abstract

Workflow and data integration engineering are complex and expensive activities. Adding a new workflow to a system often requires lengthy and repeated rounds of software engineering. The time and cost of this work creates an undesirable barrier between users and new system functionality. In the process of creating new workflow, developers frequently reproduce the queries and updates already present in other parts of the system. For example, many IT organizations have a forms-based implementation that allows a user to perform the steps of a workflow by hand. This form system can be leveraged in the creation of workflow procedures. In this paper we describe a system, Workflow By Example (WbE), where developers create workflows (or data integration queries) by demonstrating a workflow to the system. WbE observes the demonstration and automatically constructs a corresponding general workflow script. A performance evaluation of WbE shows that its learning algorithm scales well, and a user study show that if a batch update contains more than 8 form changes, WbE is more efficient to use than manual updates.

1 Introduction

Batch updates to databases are generally implemented using a combination of workflow tools and "extract, transform, and load" (ETL) tools. The addition of a new batch update procedure using these tools requires careful consideration of data cleaning, duplicate record elimination, record linkage, and other issues. Thus, in general, systems are engineered such that "point" updates are handled manually via a form system, and very large updates are handled through workflows. Medium sized updates (between, say, 2 and 50 updates at once, possibly adding up to thousands of updates over time) are often left unsupported. These medium sized updates are therefore done manually through the form system, a process that is slow, expensive, and error prone.

To understand more about ad-hoc batch updates, we interviewed the administrator of a local departmental website. This administrator frequently receives emails requesting lists of changes to the website: combinations of additions, modifications, and deletions of information about employees, events, documents, and more. For example, consider a simple workflow problem where an administrator is given the task of updating a list of office assignments following a recent departmental office shuffle. The administrator received a spreadsheet with rows containing the first name, last name, old office location, and new office location for each person who had moved. Since no batch update facility for office locations was available, the administrator was forced to perform the set of updates manually, i.e., one at a time through a form interface.

Generally, when a list of changes is short, an administrator will simply make the updates using the form interface. However, if a list is very long, or the administrator expects many such updates over time, the administrator may appeal to a development team to provide a specialized workflow for the task. Development teams evaluate such appeals, but must balance the cost of creating the workflow against the expected cost of doing the updates without the workflow. When the cost of creating a workflow is obviously less than the cost of continuing to perform updates by hand, the development team will implement it. Inversely, when the cost of doing something by hand is very low compared to the cost of implementation, the development team will not consider creating the workflow. In between these extremes are workflows which may be beneficial, but appear relatively costly to implement. Many systems would benefit from workflows, such as those for handling medium sized updates, which fall into this 'workflow gap', where they would be beneficial, but are not considered important enough to invest development resources.

In this paper, we propose a system, called Workflow by Example (WbE), that allows a user (a developer) to create *workflow scripts* by demonstrating the workflow necessary to complete a task using familiar applications and an example interaction. The system examines the user's interaction and generates a parameterized script as a result. The user then supplies a list of additional examples to evaluate. The script is executed over these examples, and any exceptions are presented to the user, allowing the user to create new scripts for processing these exceptions. The resulting set of scripts, called a *workflow program*, is then saved for future use. Thus, WbE lowers the cost for developers to create workflows, allowing developer teams to cost-effectively implement workflows that would otherwise languish in the 'workflow gap'.

WbE falls into the general research area of programming by example systems [8]. Programming by example prototypes have been built for many areas: regular expression editing, novice programming, game design, and many more. These prototypes are characterized by the complex-

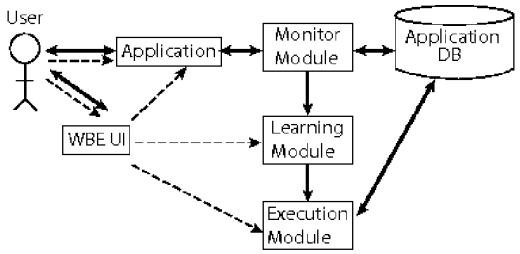

Figure 1: WbE System Architecture. Solid lines indicate data flow. Dashed lines indicate control flow.

ity of the user interaction (e.g., can the user indicate counter-examples?) and the power of the learning algorithm (e.g., does it learn disjunction, negation, sequences, etc.). The power of the learning algorithm directly affects the expressiveness of the programs that can be generated. WbE provides a relatively simple user interaction (upload a file, demonstrate an example, initiate learning and execution). Additionally, should WbE encounter inputs that cause a script to fail, the user is given the opportunity to try recording a new script to handle these failed inputs. The new script is integrated into the existing script as a form of conditional execution. WbE's learning algorithm leverages the input file from the user for hints as to which values in the captured interaction are meant to be parameters for future execution. The scripts created by WbE's learning algorithm accurately reproduce the action in the recorded examples, so each example provided by the user is guaranteed to execute similarly.

WbE also allows the user the ability to test a script, by initially running it in a 'preview' mode, presenting the successes and failures to the user for review before committing changes to the database. Preview mode allows users to experiment and debug scripts without changing dozens or hundreds of production values in a database.

1.1 Architecture

Figure 1 illustrates the architecture of the WbE system. Users interact directly with WbE through the UI by providing input files, examples, and commands (such as "Start Recording"). Users interact indirectly with WbE by completing tasks with an application through a sequence of interactions with the application. This sequence creates an information exchange between the application and the application database, which is exchange is logged by the Monitor Module.

Through the WbE UI, the user can provide input to the system for inference and execution, paste in values copied from the application to describe the output of a task, and initiate script creation and execution.

When a user initiates script creation, the WbE UI passes the input file, output pasted by the user, and the captured log to the Learning Module. The module compares the log with the provided example interaction to create a generalized script for the captured task. After successful script creation, the system passes the script and the remaining input examples to the Execution Module, which executes the script on the examples in 'preview mode', using transactions to test the sequence without committing changes to the database. The user reviews the results of execu-

Roberson	John	6058
Bose	Steven	7094
Bradbury	Mitchell	7158
Butler	Boomer	7108
Chesney	Sarah	7104
Childress	Taylor	7120
Church	Leonard	7068

Figure 2: An example system input containing a list of last names, first names, and new office numbers.

Figure 3: WbE UI Capture Interface

tion, and commits the successful examples if they are satisfied with them. Exceptions to the script can be handled by recursively using the WbE interface to define a new script for dealing with the examples which failed to execute in the first script. Finally, the user can name and store the script permanently for future use. Shared scripts are available to all users, enabling a form of cooperative work.

1.2 Example

In this example task, a user must update a set of employee records in a database with new office room numbers. Figure 2 shows part of an example input file that contains a list of employee names and new office numbers.

The user begins by uploading the input file to the WbE UI, which is a frame in a standard web browser. The WbE UI responds to the input file by presenting to the user the first example in the input file: `('Roberson', 'John', '6058')`. The user navigates in the top frame to the appropriate application. The user then begins task capture by pressing the Start Recording button (Figure 3), causing the WbE system to create a blank log file and begin logging the application's database interaction.

Using the first element in the input list, the user performs the office location change task, exactly as the user normally would. As illustrated in Figures 4 through 6, the user searches for the employee by last name, then selects the employee's details from the search results. For verification, the user captures the employee's old office location as output for the workflow by copying the old office value ('1058') from the employee's record into one of the workflow output fields in the WbE UI (not shown). The user then updates the office number to the new value ('6058') and saves the changes, completing the task.

Once the first employee has been processed, the user presses Stop Recording to end capture and

Figure 4: Demonstrating the Task. The user searches for the employee.

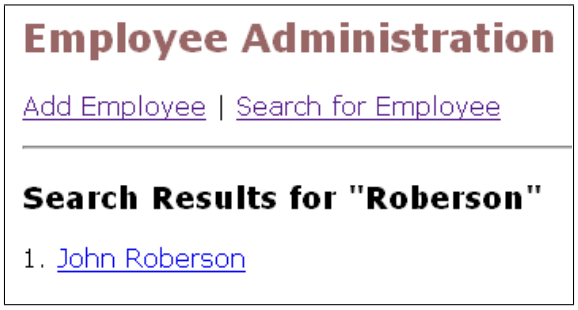

Figure 5: Demonstrating the Task. The user selects the employee's record.

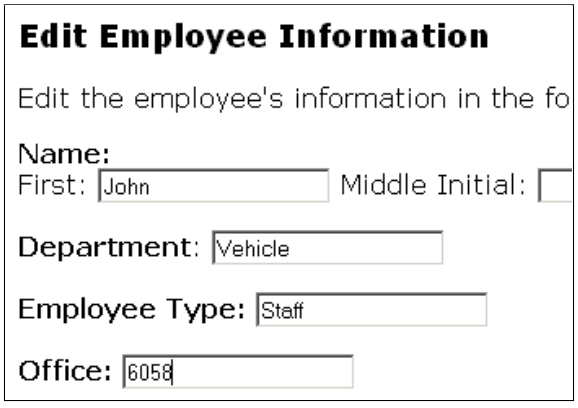

Figure 6: Demonstrating the Task. The user updates the employee's information.

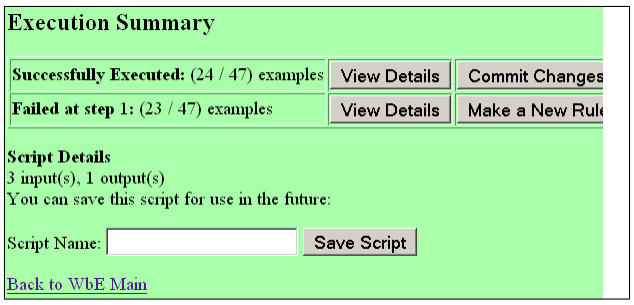

Figure 7: WbE Results Page for the demo task.

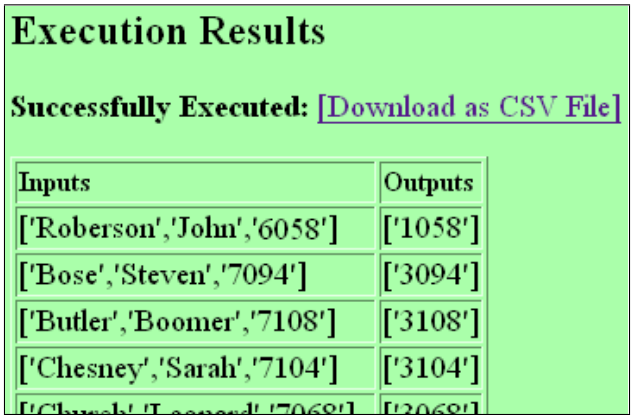

Figure 8: Details of examples on which the script succeeded, including outputs table.

close the application log. WbE now is ready to generate and preview a script using the prepared input file, the output values, and the interaction log.

The user initiates script generation and execution by pressing the Preview button, causing the learning module to generate a script, and execute the script in Preview mode (which executes the script on each example without actually altering the database). During execution, WbE collects information about the success and failure of the script with respect to each example.

After execution, the WbE UI presents the user with a results page (Figure 7) which summarizes the status of all input tuples with respect to the script's success or failure.

For this example, the task yielded 24 successes (including the original example), and 23 failures (all of which failed at the first step of the script). The user examines the details of execution for both the successful examples (see Figure 8) and those that failed at the first step (see Figure 9). The failed inputs were the result of including employees in the input file that were not yet in the database (causing the script to fail when it attempted to look up a non-existant employee). However, the reason for the failure is not immediately visible to the user, as the interface indicates only that the examples failed during the first step of script execution. To discover the reason for failure, the user walks through the workflow manually, finding that searching for the employees in the list of failures turns up nothing. To handle the examples for missing employees, the user presses the Make a New Rule button on the preview page. This action recursively calls WbE to

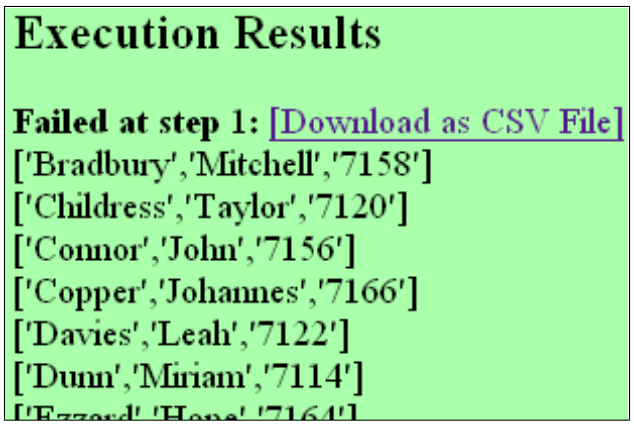

Figure 9: Details of examples on which the script failed.

handle the failed input. Thus, a new WbE window appears with the application's home page at the top and the WbE UI recording controls (Figure 3) at the bottom. The first row of the set of failed examples appears as input for driving the new task demonstration. Beginning by pressing the Start Recording button, the user proceeds through the steps described earlier, replacing the original demonstration of looking up and editing an employee with a demonstration of adding a new employee to the database using the application. After committing all successful changes, the user saves the workflow program. The program consists of two scripts: a script to handle office modification of existing employees and a script to handle adding new employees. The second script is only run for examples which the first script fails to process successfully, effectively creating a conditional execution depending on whether a given employee exists in the database.

1.3 Research Issues and Scope

This section describes the research issues that guided the design and implementation of WbE. Additional open research issues are discussed in the conclusion.

1.3.1 Access Point to Application Behavior

WbE uses the connector between the user's application and the supporting database as an access point. This access point allows WbE to log every interaction between the application and database, filtered through the logic of the application. The log contains the raw SQL requests as they passed from the application to the database, and the results sets that are passed back. The capture is described in more detail in Section 3. Compared to other access points, such as the HTML requests and responses, SQL logs contain precise, typed and formatted information. WbE is the only programming by example system to use the DBMS connector as an access point.

1.3.2 Expressive Power of Learning Module

The goal of WbE is to learn a generalized program from log and the set of input strings that informed the interaction captured in the log. Since the logs are potentially large, the complexity of the learning algorithm is a signficant bottleneck. To this end, we use an inductive algorithm that generalizes only the parameter values of queries and updates that appear in the log. The algorithm allows a script to reproduce database interactions, for each input, which have no internal loops or conditionals. Conditionals are captured through script exception handling, allowing unhandled examples from one script to be passed to another, but are not represented directly in the learning algorithm itself.

1.3.3 Benefits to the User

WbE would not be useful if it performed worse than doing the tasks by hand. WbE is designed to benefit users even for tasks requiring only a few repetitions. The design includes a lightweight upload interface, a lightweight recording interface, and previews of updates to prevent errors. An evaluation of user performance shows that users have no difficulty learning to use WbE. In addition, the evaluation empirically measures the break even point for using WbE verses manual execution of a simple task. Further, a performance evaluation of the generated scripts versus equivalent hand-written Java code shows reasonable performance for medium sized updates.

1.3.4 Scope

The scope of WbE is defined by the algorithm used in the learning module. Each invocation of the algorithm generates a script for one interaction path through an application. The path contains the sequence of SQL queries and updates invoked by the application. Users develop workflow programs that are sets of conditionally executed scripts. The program is driven by a single outer loop that processes the input file. The algorithm does not, however, learn internal loops.

The above issues and others are discussed in the remainder of the paper. Section 2 describes the overall model of WbE's operation. Section 3 elaborates on the WbE method with respect to our implementation. Details are provided with respect to engineering requirements, script creation and execution, exception handling, etc. Section 5 describes the experimental framework and results for system performance experiments and human participant experiments that measure the performance of the system with respect to learning and execution time, user time, and user errors. Section 6 discusses related work. Section 7 discusses future work for WbE. Section 8 concludes the paper with a summary of results.

2 Model

WbE operates on lists of input tuples, each of which provides the information necessary to perform an unknown (to WbE) sequence of actions. The user demonstrates this sequence on the first input tuple in the list, producing an execution log. The execution log represents an example execution of a program in a fixed language, containing both the statements that were executed (queries and

updates) and any results returned. The user may also copy any of the results of their actions into an example output tuple.

WbE uses the input/output tuples and the execution log to create a program which will (a) accurately reproduce the execution history the recorded example and (b) work correctly on the other input tuples. This task is accomplished by comparing the input and output tuple values to the statement parameters and result values in the execution log and inductively constructing a generalized program from the log.

2.1 Language of Logs and Programs

The WbE universe assumes a database D and a fixed language U consisting of a set of parameterized statement types. The ten types are `bind`, `update`, `query`, `result`, `find-one-result`, `read-result`, `commit`, `rollback`, `setAutoCommit` and `end_log`.

The statement `bind(`x`,`y`)` attempts to unify x and y, where x and y are variables or constants. The result of `bind` is *success* and the associated unified variables, if any, or *failure*.

The statement `update(`p`, `v`, `u`)` carries out a parameterized update to a database table, such as SQL INSERT or DELETE statements. The parameters to an `update` are the column names p and values v to be assigned to the SQL update expression u. This allows parameterization of update values and WHERE clause values. The result of `update` is either *success* and D is temporarily modified or *failure* (for example, due to an exception or integrity constraint violation) and D is unchanged.

The statement `query(`p`, `v`, `q`)` performs an SQL SELECT statement, parameterized in the same manner as `update`, allowing additional parameterization of `WHERE` and `ORDER BY` clauses. `query` statements are given unique identifiers to allow them to be associated with the results that they produce. A successful execution of the query results in *success* and the result set of the answers to the query. An error during query processing (e.g., a type error on the argument) results in *failure*.

The statement `read-result(`c`, `d`, `v`)`, which appears only in recorded logs, indicates that the application has read a result row from the results of an SQL query. The tuples describe the column name (c), data type (d), and value (v) from a result row, as it was read by the application. A `read-result` statement also contains an identifier linking it to the query that produced it. Column values for a given row will only appear in a `result` statement if they were actually read from the row by the application.

The statement `find-one-result(`p`, `r`)`, which appears only in WbE scripts, takes as input a result set r returned from a `query` call and a query template p, then attempts to find a single row in that result set, using the query template to `bind` column values. The result matching parameters are tuples consisting of a table column name, a data type string, and a variable or a value which will be passed to `bind`. If all three parts of each tuple in the query template are matched successfully by only one row in the result set, `find-one-result` returns the matched row successfully. Otherwise, `find-one-result` fails.

The statements `commit`, `rollback` and `setAutoCommit` represent the execution of the corresponding SQL statements.

The `end_log` statement is added to the log when the user selects "Stop Recording."

2.2 Log

Execution logs follow a similar format to final WbE programs, with a few additions. Each `update` and `query` statement in an execution log contains a unique id.

A *log* L is a sequence of instances of U. All statements in a log are grounded, that is, the statements do not contain variables. Logs do not contain `bind` or or `find-one-result` statements.

2.3 Correctness

WbE is given an input I consisting of a set of tuples. Each tuple represents a sequence of actions to be performed on D. WbE is also given a set of *examples* E. Each example is the triple (i, l, o) such that $i \in I$, $l \in L$, and o is an output tuple. A program P is *correct*, where $P \subset U$, if P executed on every i does not generate an exception or a failure. This syntactic definition of correctness does not cover any semantic notion of correctness. A user can easily create a rule that, say, replaces last names with office numbers. Thus, preview mode (cf. Section 3.3.3) is essential for the user to have confidence in the workflow scripts generated by WbE.

3 Method

This section describes a WbE prototype implementation, including the Monitor, Learning, and Execution modules. Additionally, the engineering procedure required to connect the prototype to an application is described.

3.1 Monitor Module

When the user initiates recording of a demonstration, the Monitor Module begins recording a new log. As the user demonstrates the task, traffic between the application and its support database are monitored and logged. The Monitor Module currently logs:

- Queries executed by `PreparedStatement` objects. These appear as `query()` actions in the log.
- Updates executed by `PreparedStatement` objects. These appear as `update()` actions in the log.
- Values and column names read by `ResultSet` objects, including the domains of the values. These appear as `read-result()` actions in the capture.
- In addition to those mentioned above, calls to `setAutoCommit()`, `commit()` and `rollback()`. These appear unchanged in the log and give WbE the ability to preview database updates safely.

As mentioned earlier, queries and `ResultSet` reads are tagged with shared IDs in order to link them together for execution. The need for this will be explained during the induction step (Section 3.2).

For the example described in the introduction, the Monitor Module recorded the log shown in Table 3.1. This log shows the application searching a database table `employees` for a record whose `lastname` column matches the search string (`'Roberson'`), provided by the user. Next, a result is read containing the matching `lastname`, as well as the associated `firstname` and `eid`, the surrogate key for the `employee` table. This information is used by the application to display the results of the search (Figure 5). When the user clicks on the search result, the application generates a new query to get the detailed information about the selected employee. This query, and the returned values read from the result, are indicated in the log. Finally, the user makes the change to the employee record and the application performs an update, which is the last event in the log before the user closes it. The system is now ready to generate a script from this captured data and the input provided by the user.

```
query(6623322, ['SELECT * FROM employees WHERE lastname LIKE ''%'' || ',
   var(lastname, 'Roberson'), ' || ''%''']),
read-result(6623322, [value(eid, eid, '1'), value(firstname, firstname, 'John'),
   value(lastname, lastname, 'Roberson')]),
query(1813781, ['SELECT * FROM employees WHERE eid=', var(eid, '1'), '']),
read-result(1813781, [value(bio, bio, ''), value(dept, dept, 'Vehicle'), value(eid, eid, '1'),
   value(email, email, 'jroberson@ardra.com'), value(firstname, firstname, 'John'),
   value(lastname, lastname, 'Roberson'), value(mi, mi, ' '),
   value(office, office, '1058'), value(phone1, phone1, '412 281 1346'),
   value(phone2, phone2, ''), value(type, type, 'Staff')]),
setAutoCommit('false'),
update(3032088, ['UPDATE employees SET firstname=', var(firstname, 'John'), ', mi=',
   var(mi, ' '), ', lastname=', var(lastname, 'Roberson'), ', dept=',
   var(dept, 'Vehicle'), ', type=', var(type, 'Staff'), ', office=',
   var(office, '7100'), ', phone1=', var(phone1, '412 281 1346'),
   ', phone2=', var(phone2, ''), ', email=', var(email, 'jroberson@ardra.com'),
   ', bio=', var(bio, ''), ' where eid=', var(eid, '1'), '']),
commit,
setAutoCommit('true'),
end_log
```

Table 1: Log file of the captured task.

3.2 Learning Module

The task for this component of the system is to generate a script based on the input example and task capture that can then be applied to any similar task. The module begins script creation by reading the example (the first row) from the input file. This example is our input array, i, and it will be used to replace the specific values used in the capture log with variables that can be bound to new values. The workflow output values provided by the user are also put into an array, o. Finally, the log is read as an array, S, with each query, update, result read, or transaction management statement making up an element in the array.

To create a general script from the specific example, the WbE system uses an inductive algorithm, $make_script$, that generalizes the specific information in the example to a set of variable relationships. Script induction is a process that takes as input the following information: an array of input constants, i, and an array of output constants, o, both provided by the user, and the recorded log, S. It produces: sets of variables, $\hat{i}$ and $\hat{o}$, which will be bound to new inputs and outputs during execution, a generalized script, $\hat{S}$, and a set of `result templates`, Q, describing how each query's results should be handled. We refer to each element of S as S_1, S_2, etc, with a similar notation for elements of the other arrays and lists.

$$make_script(i, o, S) \rightarrow (\hat{i}, \hat{o}, \hat{S}, Q) \tag{1}$$

Induction begins by creating the $\hat{i}$ and $\hat{o}$ arrays. These arrays are of the same length as their counterparts, i and o, with each member being a unique new variable. Additionally, two temporary lists, r and $\hat{r}$, are created for storing all tuples read from `ResultSet` objects. Once the replacement variables and local variable lists have been created, for each action S_i in the capture log, create a script action, $\hat{S}_i$, according to these rules:

> If the action is any of `setAutoCommit()`, `commit`, or `rollback`, copy the action verbatim into $\hat{S}_i$.
>
> If the action is a `query()` or an `update()`, copy the action, its ID, and its non-parameterized contents over to $\hat{S}_i$. Then, process the parameters in the action according to the algorithm in Table 3.2.
>
> If the action is a `read-result()`, process its contents, r_j, according to the algorithm in Table 3.2 to create its counterpart $\hat{r}_j$, by replacing all result values with variables. Add these tuples to their respective lists, r and $\hat{r}$. The action is not copied into $\hat{S}_i$.
>
> If the action is an `end_log`, finalize the script and create the result templates, Q, for handling query results. Result templates are described, below.

3.2.1 Result Templates

Our prototype implementation makes a simplifying assumption about each query in a recorded log. Namely, it is assumed that each query produces zero or one "useful" result rows. Following from this assumption, the "useful" result row, r_j, from a query provides a template for future execution in the form of it's generalized counterpart, $\hat{r}_j$. This template, chosen according to the algorithm in Table 3.2, provides future matching information in its `value()` tuples, which have had all recognizable values replaced with variables. At runtime, these variables may be bound to new input rows (creating constants with which to match query result values). Otherwise, they will be bound to the query result value of the proper row at runtime, "filling in" values that may occur later in the workflow script. Due to limitations in the present induction algorithm, if more than one possible template is found, script creation fails and the user is offered the chance to attempt to create a script using the next example in the input file, hopefully providing a better

For each `var(`$domain$`, `$value$`)` tuple contained in an `update()` or `query()` action:

1. Attempt to replace $value$ by finding a match in i or o. If a match is found, use the corresponding variable from $\hat{i}$ or $\hat{o}$ to replace $value$ in $\hat{S}_i$.
2. If no match was found, repeat the search using all local values stored so far in r. Matching for these values is more restricted, since $domain$ must match the data domain of the result, as well as the value itself. Should a match be found, $value$ will be replaced in $\hat{S}_i$ with the appropriate variable from $\hat{r}$. Additionally, the matching result in r will be flagged as having been referenced.
3. Finally, if there is still no match, then $value$ is copied into $\hat{S}_i$ verbatim.

Table 2: Algorithm for processing parameters in `update()` and `query()` tuples

For each `value(`$column$`, `$domain$`, `$value$`)` tuple in log result tuple r_j, create a counterpart in $\hat{r}_j$:

1. Attempt to replace $value$ by finding a match in i or o. If a match is found, then $value$ is replaced in $\hat{r}_j$ by the corresponding value from $\hat{i}$ or $\hat{o}$. If the match was from o, then the result will be flagged as having been referenced.
2. If no match was found, then $value$ is considered to be new information. It is given a new variable in $\hat{r}_j$.
3. Finally, add r_j and $\hat{r}_j$ to their respective lists.

Table 3: Algorithm for processing parameters in `read_result()` tuples

demonstration. If no template is added to Q, the query is effectively ignored at runtime. Because the `read-result()` used to create the template has been matched against the input and output tuples provided by the user, some relationships may be established which effectively cause the resulting program to expect a result set value to match the one passed in by the user for all future steps. This works in addition to matches on values `query()` and `update()` to create extra constraints for row matching during execution. These additional constraints imply that, for our example, the first name in a result row must match the one in the input tuple, though the user never specifies this explicitly. Further examples which violate these extra constraints will cause the script to fail. This has the implication that extra information in the input file may be used to create constraints that insure a correct workflow.

To discover the query template $q_i \in Q$ for a query, iterate over each tuple in r:

- If r_j and q_i have non-matching IDs, ignore r_j.
- If r_j and q_i have matching IDs, but the `value()` tuples in r_j were not referenced by any `query()` or `update()`, and did not match any values in o, ignore r_j.
- If r_j and q_i have matching IDs, and at least one `value()` tuple in r_j was referenced by a `query()` or `update()`, or matched any value in o:
  - If no template q_i has been chosen, assign q_i to be equal to $\hat{r}_j$.
  - If q_i has already been chosen, then this query produces more than one "useful" answer, which this algorithm cannot cope with. Script generation fails.

If q_i remains unassigned, then no template exists for this query.

Table 4: Algorithm for creating result templates

3.2.2 Implications

This *naive* induction algorithm is sufficient for dealing with small logs, such as those generated from simple database management forms using search-based navigation. More complex systems, such as those which present elements to the user in a dropdown list, produce many more query results in the execution log. This result 'noise' greatly slows down the linear search algorithm described in Table 3.2 as each new value must be compared against all previous results until a match is found or no results are left. With this in mind, a second *indexed* induction algorithm was implemented so as to store `read-result` tuples in a hash table, indexed by the string value they produced from the database. These two methods are compared in Section 5.

Table 3.2.2 shows the results from applying this induction procedure to part of the log. At this point, the execution module is ready to execute the script over the rest of the data in the input file to generate a preview of the script's effects.

3.3 Execution Module

Rule execution begins by combining the generalized script $\hat{S}$ with new input values i. Each variable in $\hat{i}$ is bound to it's new value from i, propagating this new value throughout the script. The Execution Module then proceeds to iterate through $\hat{S}$, executing each action as follows:

> `update()` actions are executed by creating and executing an SQL string. The string is created by iterating over each element in the `update()` tuple, combining the raw strings (in single quotes) with the (quoted) values from each `var(`*domain*`,` *value*`)` tuple. The resulting string is executed against the database. If the update should fail for any reason, the input tuple is marked as having failed at this step and execution halts for the failing example.

$\hat{i}, \hat{o}$:	(A,B,C) (D)
$\hat{S}$	query(6623322, ['SELECT * FROM employees WHERE lastname LIKE ''%'' || ', var(lastname, A), ' || ''%''']), query(18137981, ['SELECT * FROM employees WHERE eid=', var(eid, E), '']), setAutoCommit('false'), update(3032088, ['UPDATE employees SET firstname=', var(firstname, B), ', mi=', var(mi, J), ', lastname=', var(lastname, A), ', dept=', var(dept, G), ', type=', var(type, M), ', office=', var(office, C), ', phone1=', var(phone1, K), ', phone2=', var(phone2, L), ', email=', var(email, I), ', bio=', var(bio, F), ' where eid=', var(eid, H), '']), commit, setAutoCommit(true)
Q	result-template(18137981, [value(bio, bio, F), value(dept, dept, G), value(eid, eid, H), value(email, email, I), value(firstname, firstname, B), value(lastname, lastname, A), value(mi, mi, J), value(office, office, D), value(phone1, phone1, K), value(phone2, phone2, L), value(type, type, M)]), result-template(6623322, [value(eid, eid, E), value(firstname, firstname, B), value(lastname, lastname, A)])

Table 5: Rule and associated variables from inference procedure.

`query()` actions are executed much like `update()` actions. A string is built from the strings and `var()` values in the `query()` tuple, and the resulting string is executed against the database. A failure of the query also results in aborting execution for the failing example, and the input tuple is marked with the step at which it failed. `query` actions also implicitly call `find-one-result` using the result template for this query, which is stored in Q.

`find-one-result()` is called implicitly after a `query` and uses a passed-in result set and a result template to match against rows in the result set. For a given `value(`*column*`,` *domain*`,` *value*`)` tuple in the result template, the value of the column named *column* is read from the result set, and a `bind` is attempted against the value in the result template. If `bind` fails for any of the tuples in a template, the row is considered a non-match. If only one row is found which matches the template, the call to `find-one-result` is successful. If no rows match, or more than one matches the template, the input tuple is marked as having failed at this step, and execution halts for the failing example.

`commit`, `rollback`, and `setAutoCommit()` are executed immediately (unless the script is being run in preview mode, which is described later). If one of these actions fail for any reason, the input tuple is marked as having failed at this step and execution halts for the failing example.

3.3.1 Execution Results

As each tuple from the input file is executed, information about its success or failure, and any output values that have been bound are collected for display to the user. All examples are partitioned into groups based on the last script action that succeeded. Examples which reach the end of the script are labeled as successful, and their outputs are stored together in an outputs table. The remaining groups of examples are labelled as failed.

3.3.2 Rule Failure Example

Not all tuples in the example input file executed successfully. For example, in Figure 9, the tuple `('Bradbury', 'Mitchell', '7158')`, as well as the other 22 failures, failed at the initial step of the script. This initial step was the query step which performs a search for an employee based on last name, and none of the 23 failure cases returned a successful match in their result sets. While in our example all the tuples failed in the same way, many failure modes are possible: misspelled names, last name collision with different first names, etc. Failed examples are grouped by the step in the script at which they failed, and can be passed along to the WbE UI if the user chooses to make a new script for handling them.

3.3.3 Preview Implementation

Rule execution takes place in one of two modes: preview and commit. In preview mode, execution proceeds normally for all actions except the `commit` action. When `commit` is encountered in preview mode, a `rollback` is executed in its place. As a side effect, this effectively causes each execution of the script to occur 'in a vacuum' with respect to the others, such that later examples cannot count on the outcome of prior execution. Commit mode is identical to preview mode, except that it honors the `commit` actions, allowing changes to be committed to the database.

4 Engineering Procedure

In this section we document the engineering procedure required to make an application WbE compliant.

4.0.4 Database Driver Replacement

Our implementation uses a modified Java Database Connectivity (JDBC) driver [12] to intercept, log, and pass along database queries, updates, and results. This driver acts as an intermediary between the application and the original driver through which it accesses its database. It is necessary for WbE compatible applications to be configured to use this augmented driver, rather than their normal one.

4.0.5 Use of Prepared Statements

To aid WbE in generalizing only the correct portions of logged SQL statements, compatible applications must interact with the database using `PreparedStatement` objects. Leveraging the existing JDBC API calls for specifying parameter values for portions of SQL statements driver to mark for replacement only those portions of the query specified by the developer. As a side effect, this design restricts some of the types of inference that can be done by WbE, as SQL restricts the use of PreparedStatement parameters to values in SQL updates and queries, not for table or column names.

4.0.6 Data Domains

The JDBC API within our driver has been extended to include the concept of *data domains* to guide the induction algorithm described below. These domains are specified by the developer using altered `getX()` and `setX()` calls of `PreparedStatement` and `ResultSet` objects. For each call to `getString()`, `setString()`, etc., the developer specifies an extra string, indicating the domain to which this value belongs. For example, to fetch a string phone number, the `String pn = rs.getString(pnatt, "phone number");` method is called, where `pnatt` is the schema attribute name and `phone number` is the label of the domain. Using data domains, WbE can easily differentiate between values expected to be used as, for example, office numbers, rather than employee ID numbers or telephone extensions. A special 'null' domain of `none` is understood by WbE to indicate data that WbE should ignore during rule generation. The `none` domain can be used to 'hide' data from WbE that would be useless to learn. For example, a form system which maintains data about itself in the same database that it maintains might mark that all values related to the forms themselves with the `none` domain, to prevent WbE from attempting to match this 'meta' form information.

4.0.7 Using Transactions

In working with early prototypes of WbE, we determined that the ability to preview effects of the tool's use was critical to user acceptance, since the consequences of script execution can be large. In order to support WbE's preview ability, applications must explicitly wrap statements that will alter the underlying database (such as `INSERT`, `UPDATE`, `DELETE`) with transaction calls via the JDBC driver. For example, such updates should be preceded by a call to `setAutoCommit(false)` and followed by calls to `commit()` and `setAutoCommit(true)`. The driver logs these transaction management calls so that the execution module may later use them to roll back updates when a user wishes to preview the effects before committing to a set of changes. Explicit `BEGIN`, `COMMIT`, `ROLLBACK` requests made via SQL statements are not supported by this version of WbE.

4.0.8 Result Batching

To preserve logical grouping of results read from `ResultSet` objects into rows, the logging driver does not actually log the data read from a particular row until a call is made to `next()` or

`close()`. This limitation should encourage developers to group reads from `ResultSet` object into batches, followed closely by calls to `next()` or `close()` before values from those objects are used in further SQL statements.

4.0.9 Limiting Application Logic

WbE has access to application behavior through one mechanism: the JDBC driver. Thus, any application logic that occurs on the database side of the driver (through stored procedures or other mechanisms) is handled automatically. Any application logic not visible through the database interaction which affects query or update parameters (such as math or string operations) will render WbE unable to learn scripts involving these steps. Future work for WbE may include an API for revealing application logic to WbE with respect to data transformations.

5 Results

In order to understand the potential impact of WbE, we conducted experiments to measure WbE's performance in various learning conditions, in an execution condition, and in comparison with performing an experimental task "by hand", using a web-based forms system. A "deployment evaluation" was also performed, to explore the impact of attempting to use WbE on existing systems.

5.1 Learning Algorithm Evaluation

As mentioned earlier, the initial induction algorithm used by our prototype performed poorly as the number of query results increased. To evaluate this, and to compare the iterative induction method with the hash-based induction method, we compared the script-learning times for both algorithms in various learning situations. Each algorithm learned a simple script for looking up a person's information amongst the results of a query listing all people in the database, using their first and last name as identifiers, then to collect various pieces of information about the person, before updating the person's phone number in the database. The algorithms performed this task in 5 conditions with increasing amounts of 'result noise'. The 5 experimental conditions contained 0, 25, 50, 100, and 200 non-target results which must be considered by each algorithm. Along with varying the number of 'extra' results that must be handled, each algorithm was run in a 'best case' and 'worst case' condition, where the target result was either the last result seen (best case), or the first result seen (worst case).

5.2 Execution Algorithm Evaluation

In addition to understanding learning performance time, we wanted to evaluate the overhead of executing these WbE scripts within an interpretive environment, versus a hand-constructed program for performing the same task. Using the same 'look up, gather information, update' task described above, a WbE script was constructed and run with varying amounts of 'result noise'. Additionally,

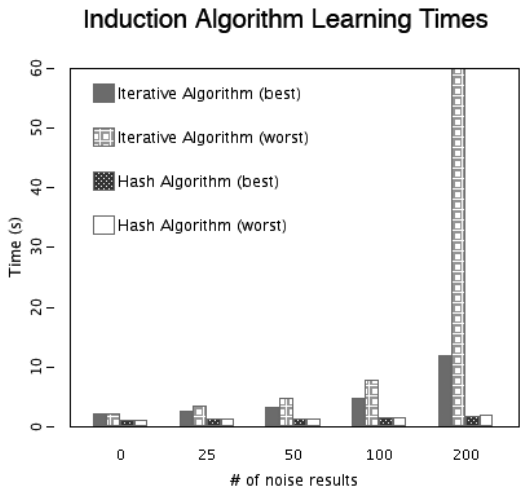

Figure 10: Iterative vs. Hashing algorithm in best-case and worst-case learning conditions

a Java program was hand-crafted to mimic the behavior of the WbE script, and measured in the same conditions.

Results of the learning algorithm evaluation can be seen in Figure 10. The graph shows that the iteration-based induction algorithm can take exponential time in the number of results that it must consider, a fact that is especially evident in the worst case. Additionally, the hash-based algorithm takes slightly more time as the number of results that it must store increases, but this effect doesn't seem to present a scaling issue.

Results of the execution algorithm evaluation can be seen in Figure 11. This graph shows that the scripting environment overhead adds significant time to each example as the number of 'noise results' increases. Future work will decrease this overhead by improving the handling of noise results.

5.3 User Experience Evaluation

To understand the potential impact of WbE for users, we conducted a pilot study on 8 human participants. The experiment was a between-subject design, conducted by presenting each participant with the same task of 16 updates to the database, in one of two conditions: either using a form interface to update the database directly, or using the forms augmented by WbE. The main task consists of performing two types of updates to the database, all in the same format. Nine of the 16 updates consist of looking up a person in the database and updating their information according to the values of a spreadsheet. The people listed in the other 7 updates did not yet exist in the database, requiring the user to add them, using the forms. Each condition included a training example to teach users how to perform updates and additions using the forms, in the forms-only case, and to perform the task using WbE and WbE's exception handling feature in the WbE case.

Participant performance was measured on completion time and errors. We chose these metrics to evaluate WbE because, from the user's point of view, script induction and execution are effec-

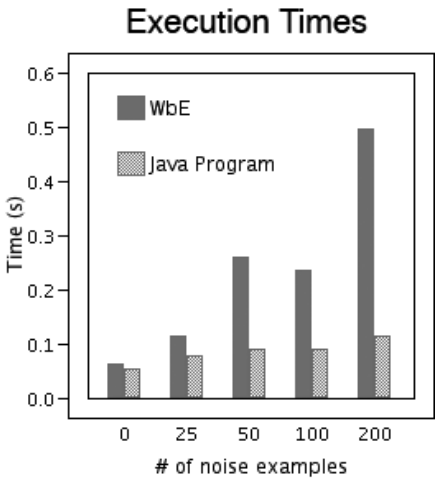

Figure 11: Execution times for WbE script vs. Handwritten Java Program for one example

tively instantaneous. The primary bottleneck to the system is the use of the tool and the class of scripts that can be generated by our induction procedure. We did not extensively evaluate the user interface nor the behavior of the system given dirty input; these issues are future work.

Results of our user evaluation revealed an average time of 3.19 minutes to complete the task with WbE, and an average time of 7.5 minutes to complete the task in the 'by hand' condition. A two-tailed, unequal means, t-tailed test comparing the completion times of the participants indicates that the mean time to complete the tasks is significantly different in the two tasks (p value < 0.01), and show that WbE provides a significant speed increase over performing the task by hand. There were no errors made by participants in either condition, so no comparison can be made between the systems with respect to errors.

The times recorded in our user evaluation show that WbE provided a 235% speed increase over performing 16 updates by hand. From this result, it can be seen that WbE provides a speed increase over the manual condition after only 8 updates.

5.4 Deployment Evaluation

WbE was deployed on a web-based database forms system as a means to connect two applications. The first application was a scheduling optimizer, which dealt with constraint-based planning of events. The second application was a database-backed website which used a set of web-based forms to maintain the database. The problem that WbE solved was that of connecting these two applications. When the optimizer made changes to the schedule, the changes could be output in a format readable by WbE, giving details about the name, location, date, start time, and duration for each event. The form system for maintaining the schedule website was altered such that it would produce logs for WbE. Once these two pieces were in place, WbE could be used to create a workflow program that would take a schedule delta file from the optimizer as input, and would update the website's database to reflect the changes that had been made. This deployment resulted

in some interesting discoveries and solutions about the effective workings of WbE.

The use of WbE as the "glue" to synchronizing two systems revealed a couple of interesting insights into WbE. The first issue encountered came as a result of the design of the form system used for maintaining the web site. To facilitate selection of events, the forms displayed all event names in a list, from which the user would choose. The display of this list resulted in WbE having to filter out many 'noise results' when trying to match the chosen event with the proper row in the result set. The discovery of this issue led to the hash-based learning algorithm discussed earlier in the paper.

The source of the second issue was WbE's learning algorithm itself, and its feature of learning relationships between spreadsheet inputs and expected values for rows in query results. For example, consider an event update where only the location of an event has changed. In this case, WbE would learn a relationship between the date, start time, and duration listed in the spreadsheet with that in the database. Future execution of this rule on other updates would fail if any of the three date, start time, or duration values in the spreadsheet were different from that in the database. This led to an exponential branching in the number and type of exception cases for each example, as an example which had a different date would be in a different failure group than an example which had the same date but a different start time. One way to avoid this situation would be to ensure that the training example used to teach WbE was one which required changing all four key fields. In this case, WbE would learn no connection between the spreadsheet values and the initial date, location, etc. values for the event it is updating.

As it was unlikely or impossible that any event would be both different for all four values, and also be chosen as the example for WbE, a different solution had to be constructed. This solution required altering the forms system to have it report the location, date, start time, and duration as having a data domain of 'none', which would cause them to be ignored entirely by WbE. This change had the benefit of allowing WbE to handle rules for updating these four values. However, because WbE was forced to ignore these values, it could never usefully reason about them, either. For instance, a script could not be constructed to output the location, date, start time, or duration information for a given event. Similarly, any updates made to events through WbE would be forced to include all four pieces of information, or WbE would simply set them all to a constant value (the value used in the example). Overall, this issue represents future work WbE in terms of the types of reasoning it can do about relationships between values, as well as the methods used for grouping failed examples into common exception cases.

6 Related Work

6.1 Programming by Example Systems

Unlike other Programming by Example (PbE) systems [8], WbE uses an application's interactions with its support database as an access point for capturing user behavior. Like some PbE systems designed for tasks like web browsing [13] and information extraction from the web [2], WbE's interface is in the web browser, and the user interacts with it by copying and pasting values between the application's web interface and WbE's web interface. However, while these other systems focus

on the HTML as the access point for performing inference on the user's actions, WbE looks to the actions performed on the database itself, operating on sequences of SQL rather than portions of HTML.

WbE is similar to programming by example systems like SMARTedit [5], which capture a sequence of actions in a fashion similar to a macro recorder, then reason on these sequences in the background. WbE again differs from such systems based on the access point for interaction capture, but also based on the type of learning used. While SMARTedit uses a formalized version space algorithm with multiple examples for searching and pruning the generalization space, WbE currently relies upon a more biased set of heuristics to drive generalization based on only one example. Lau and Weld [6] discuss an extension to the version space algorithm that may apply to WbE.

In terms of the examples in this paper, which use web-based forms systems to perform queries and updates, WbE competes with web-based PbE systems such as Chickenfoot [3]. Both WbE and Chickenfoot can be used to construct workflows over web-based forms, and both systems use user interfaces which are embedded in the web browser. The difference between the systems comes primarily from the access point. Chickenfoot uses textual relationships, combined with the browser Document Object Model (DOM) to perform information extraction and query submission via web pages. This method is very complex, requiring strong text parsing, and suffers from brittleness on systems with interfaces that are dynamic or prone to redesign. On the other hand, WbE's access point between the application and database makes information extraction and workflow execution much easier, thanks to the relatively simple structure of the SQL query language, and programmatic access available through JDBC.

While participants our pilot produced no errors using WbE, they were using clean data, with no typos or wrong information. Presently, WbE's exception mechanism provides some handling for these types of dirty data, but still allows other types through. Outlier finding, such as described by Miller and Myers [10], can focus user attention on possible erroneous matches or mismatches over the inputs provided, assisting them in cleaning up dirty input.

6.2 Workflow and Other Database Systems

Workflow by Example is unique among database manipulation and workflow systems. While it shares some similarities with QbE [15], such as constructing generalized queries from user examples, it differs greatly in that user input takes the form of whole sequences of SQL, captured out of the sight of the user. These sequences are generalized in ways that go beyond data retrieval, to allow for the creation of workflow scripts.

Other intelligent workflow composition systems, such as [4], combine ontological knowledge about the inputs and outputs of available workflow services with AI planning techniques to construct workflows. These systems require knowledge-rich descriptions of the available systems, including constraints. WbE focuses on creating workflows from captured information at the database level, requiring a change to low-level driver interactions, rather than high-level workflow descriptions of each component.

Some workflow systems use a process mining [1, 14] approach which is similar to WbE in that process logs are examined in order to create models of observed behavior. These systems differ

from WbE in their access points (recorded workflow data, rather than SQL), and in their exception-learning methods, which require many execution examples to learn conditional structures for a process graph.

Existing data integration systems, such as Clio [9], use data and schema examination methods and graphical browsing tools to give developers a visual understanding of the nature of their data sources and operation mapping. This understanding aids developers in creating functions for converting between schemas. Unlike these tools, WbE lets the user create mappings for each relation through a familiar interaction, using the existing infrastructure.

Unlike standard workflow development processes [11], tools, or languages [7], WbE puts the creation of workflow in the hands of the end user, allowing them to construct and automate more complex behaviors from those built into a given system.

7 Future Work

Our prototype needs improvement in several areas. One problem area stems largely from the opacity of application logic and user logic with respect to our system's ability to compare and generalize values. For example, the prototype does not support any methods of value processing, such as substring matching. By disallowing substring matches, the prototype cannot associate a single cell containing "John Doe" with two cells containing "John" and "Doe". A lack of substring matching also prevents users from performing 'lazy searches' by entering only parts of strings, so while typical applications allow searching via partial terms, the WbE system would not properly associate a partial term with a log value. Future work will include giving WbE the ability to parse and combine both input and database values to deal with data extraction.

Another problem area is the decoupling of WbE from the user browser experience. In particular, some implementations of navigation in a system are invisible to WbE because they do not generate database calls. Further work is needed in this area.

In future versions of our WbE system, we plan to remove the Start Recording and Stop Recording buttons, moving towards more of an user observation based model. This step allows for truly ad-hoc automation, as the system could prompt the user when it believes it has found a pattern that it can automate.

The WbE user interface also needs improvement. The main issues include user trust and exploring how WbE can understand the actions and results of a script, as well as how it can inform the user of the consequences of running a script. This information will help the user to properly manage scripts and to have a clear understanding of the result of executing them. Additionally, work will be done in exploring ways of allowing the user to view and edit the rules created by WbE, allowing a deeper understanding of WbE scripts, which in turn should allow for greater reuse and extension. The UI will also be extended to allow for user defined functions, which allows a user to teach WbE to perform data transformation and constraint checks. These changes, combined with outlier finding for bringing the user's attention to unusual examples, WbE should be a system the user can trust to properly automate workflow.

Finally, the ability to share workflow programs among users will provide a powerful method for collaborative work processes, and we look forward to exciting results in this area.

8 Conclusion

Maintaining a database can be a challenge for any organization. While bulk updates can be simplified by implementing workflow systems, and point updates can be completed with form systems, many types of updates exist in a 'workflow gap'. These updates are a strain on the user who must perform them by hand, but are not enough of a strain to warrant a developer writing a specialized workflow. In this paper we describe the WbE prototype, a system that attempts to narrow the 'workflow gap': the workflow processes that are difficult to perform manually, but too expensive or rare to implement in the workflow system. WbE narrows the workflow gap by allowing developers to define workflows by example, bypassing the need for complicated workflow or ETL tools. WbE allows the creation of workflows to automate tasks such as those that can be accomplished with common form-based database management applications. These tasks include repetitive sequences of updating, querying, adding, and deleting information from the database. WbE facilitates these tasks over lists of input data, using exception handling to deal with inputs that vary with respect to the types of actions that must be performed to complete a task. WbE's unique access point, watching the interaction between an application and its database, allows for powerful data manipulation via easily parameterized SQL. Altogether, WbE gives users the ability to automate certain kinds of data integration or database workflow tasks in an ad-hoc manner.

An evaluation of the WbE prototype shows that the prototype reduces the time required to perform simple tasks compared to manually completing the task. If the task contains more than 8 tasks to be performed, WbE is more time efficient than manual updates. A user study indicates that the mean completion times in the WbE case and the manual case are significantly different ($p < 0.01$). In addition, evaluation of the learning algorithm and the execution algorithm indicate that the algorithms scale reasonably well as the number of input examples increases. Finally, a deployment of WbE as 'glue' between two existing systems shows that WbE can be used to augment real-world data management systems.

References

[1] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. Mining process models from workflow logs. *Lecture Notes in Computer Science*, 1377:469–483, 1998.

[2] Mathias Bauer, Dietmar Dengler, and Gabrielle Paul. Trainable information agents for the web. In Henry Lieberman, editor, *Your Wish Is My Command: Programming by Example*, pages 88–89. Academic Press, 2001.

[3] Michael Bolin et al. Automation and customization of rendered web pages. In *UIST' 05*, pages 163–172. ACM Press, 2005.

[4] Jihie Kim, Marc Spraragen, and Yolanda Gil. An intelligent assistant for interactive workflow composition. In *Proceeings of the International Conference on Intelligent User Interfaces (IUI-2004)*. ACM Press, 2004.

[5] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Learning repetitive text-editing procedures with SMARTedit. In Henry Lieberman, editor, *Your Wish Is My Command: Programming by Example*, pages 210–225. Academic Press, 2001.

[6] Tessa A. Lau and Daniel S. Weld. Programming by demonstration: An inductive learning formulation. In *Proceedings of IUI '99*, pages 145–152. ACM Press, 1999.

[7] Frank Leymann. Web services flow language (WSFL 1.0). Technical report, IBM Corporation, May 2001 2001.

[8] Henry Lieberman, editor. *Your Wish is My Command: Programming by Example*. Academic Press, 2001.

[9] Renee J. Miller, Mauricio A. Hernandez, Laura M. Haas, Lingling Yan, C. T. Howard Ho, Ronald Fagin, and Lucian Popa. The Clio project: Managing heterogeneity. In *Proceedings of ACM SIGMOD*, pages 78–83, March 2001.

[10] Robert C. Miller and Brad A. Myers. Outlier finding: Focusing user attention on possible errors. In *Proceedings of UIST 2001*, pages 81–90, Nov 2001.

[11] C. Mohan. Workflow management in the Internet Age (abstract). In *NGITS*, page 237, 1999.

[12] P6spy. http://www.p6spy.com/.

[13] Atsushi Sugiura. Web browsing by example. In Henry Lieberman, editor, *Your Wish Is My Command: Programming by Example*, pages 62–85. Academic Press, 2001.

[14] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. In *IEEE Transactions on Knowledge and Data Engineering*, number 9, pages 1128–1142, September 2004.

[15] Moshé M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.