SpLyCI: Integrating Spreadsheets by Recognising and Solving Layout Constraints

Dirko Coetsee[1,2,3] 0000-0002-1898-1279, Steve Kroon[2] 0000-0001-5625-8623, McElory Hoffmann[2,3] 0000-0002-8735-6282, and Luc De Raedt[1] 0000-0002-6860-6303

[1] KU Leuven, Belgium
[2] Stellenbosch University, South Africa
[3] Praelexis, South Africa

Abstract. Valuable data are often spread out over different similar spreadsheets. Consolidating this data for further analysis can take considerable effort for a spreadsheet user without programming skills. We introduce Spreadsheet Layout Constraint Integration (SpLyCI), a system to semi-automatically merge multiple spreadsheets and lay the result out in a single output spreadsheet. SpLyCI takes advantage of the observation that spreadsheet users lay out their spreadsheets with certain implicit constraints in mind. For example, certain cells should be in the same rows or columns as other cells, or formulae should be repeated over all numbers. The system therefore identifies these implicit layout constraints, combines them, and then solves the resulting constraint satisfaction problem. The solution yields a new spreadsheet that contains the same information and the same relationships between cells as the inputs, but with formulae that are present only in some sheets extended to cover data from other sheets.

Keywords: Relational Learning · Constraint Learning · Spreadsheets.

1 Introduction

Semi-structured data sources often contain information that would be useful if they could be converted into the right format. Spreadsheets, in particular, often have their data spread over different sheets or even files, each corresponding to a different data source or some other partitioning of the data. To work with such spreadsheets can take considerable effort, as there is not yet any tool to help consolidate data into a single sheet before further analysis can be done.

As part of a larger project to make data science more accessible to spreadsheet users [6], we aim to create a tool that can semi-automatically merge multiple spreadsheets, while handling repeated formulae correctly. A user specifies multiple input sheets and the tool (semi-)automatically transforms the input sheets into a single output sheet that non-redundantly captures the same information as the input sheets.

There are a few characteristics of spreadsheets that make them difficult to integrate: First, the same challenges that occur in relational schema matching might occur, such as changing of the schema over time or between sheets. In addition, sheets might not be in first normal form. Users might further use idiosyncratic layouts that use empty cells, text formatting, and pivoting in creative ways to represent complicated hierarchical schemas. Important information might be captured with implicit or explicit layout constraints over rows or columns. This not only includes primary-key and foreign-key constraints also present in relational databases, but also sorting on one or more row or columns, or formulae over rows or columns.

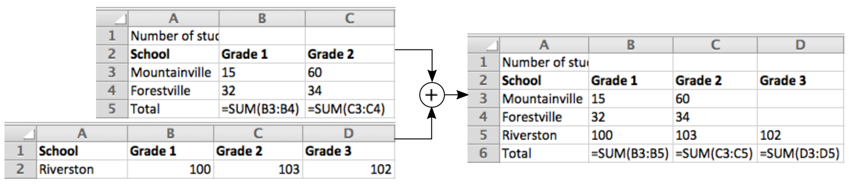
Some of these problems have been tackled in different contexts in isolation; however, as far as we know, there is as yet no end-to-end solution. This paper takes a step towards spreadsheet integration by concentrating on the last aspect—integration of spreadsheet layout constraints. To the best of our knowledge this aspect has not received any attention in the existing literature.

We propose SpLyCI—Spreadsheet Layout Constraint Integration—a framework for spreadsheet integration that is based on the view that spreadsheet integration can be represented as a constraint satisfaction problem. The layout and formula constraints in each input spreadsheet are first inferred. Next these constraints are combined, and finally the resulting constraint satisfaction problem is solved, yielding a new spreadsheet that adheres to all of the input constraints. The user is able to choose which constraints are active during the process and so can help disambiguate difficult cases or choose between contradictory constraints. We implement a prototype that is able to handle some illustrative constraints, and evaluate it on real-world data available on the web.

We contribute a description of a new task, that of interactive spreadsheet integration with formulae, a dataset and metrics to evaluate a system's performance on the task, a representation for spreadsheets based on layout constraints, and a system that uses this representation to tackle the task.

2 Problem

Figure 1 illustrates some aspects of the problem. Two spreadsheets contain similar information that the user wants to consolidate. Note the description in Cell `A1` that will frustrate a naive concatenation of rows. Also note that the user would want the formulae in the first input sheet to be expanded to cover the cells introduced by the second sheet.



	A	B	C
1	Number of stu		
2	**School**	**Grade 1**	**Grade 2**
3	Mountainville	15	60
4	Forestville	32	34
5	Total	=SUM(B3:B4)	=SUM(C3:C4)

	A	B	C	D
1	**School**	**Grade 1**	**Grade 2**	**Grade 3**
2	Riverston	100	103	102

	A	B	C	D
1	Number of stu			
2	**School**	**Grade 1**	**Grade 2**	**Grade 3**
3	Mountainville	15	60	
4	Forestville	32	34	
5	Riverston	100	103	102
6	Total	=SUM(B3:B5)	=SUM(C3:C5)	=SUM(D3:D5)

Fig. 1. Example input spreadsheets and the desired result after merging. The formulae in the first sheet have been expanded to cover new data from the second sheet.

We define *layout constraints* as the user-intended constraint on the relative row or column positions between cells. In the example in Figure 1, the user probably intends `Grade 3` to be in the same column as the number `102`, because `Grade 3` is (implicitly) its header. *Formulae constraints* are user-intended constraints on the extent to which the same formula is repeated in neighbouring cells, or the ranges of their arguments. For example, `=SUM(.)` above is repeated over all `Grade`s, and sums over all schools for each `Grade`. *Match constraints* are constraints on the row or column positions between sheets. For example, the cells below `Grade 1` in the first input sheet are probably intended to be in the same column as the cells below `Grade 1` in the second sheet.

At a high level, the problem can now be described as follows: *Given multiple input spreadsheets, combine them so that the output satisfies the layout and formulae constraints present in each input sheet, while also satisfying match constraints between the sheets.*

3 Method

There are two main parts to our proposed technique: first the constraints present in the input sheets are recognised, and then they are combined and solved. We distinguish between two types of constraints which can be handled in two separate steps. In our system[4], formula constraints are implemented in Prolog because we represent formula generalization with logical rules which fits logic programming, and layout constraints are translated to MiniZinc[5] because the resulting constraint satisfaction problem fits the constraint programming paradigm.

We next discuss the representation and heuristics we use to solve the two subproblems in more detail. We first discuss the process for simple spreadsheets without formulae, and then show how to extend the process to handle the more interesting case where formulae are present.

Our basic notation and key concepts are defined below. Note that we will sometimes only define the columnwise concept but the corresponding rowwise concept is defined similarly.

Cuts Each cell is indexed by a column i and row j identifier, where i or j is a unique identifier like `1`, `A`, or $\mathtt{A}_*$. Cells sharing an index represent the knowledge that the user intends those cells to be in the same column or row. To ensure that identifiers are unique across sheets, we add the sheet number as a superscript to some of the examples below, for example $\mathtt{A}^2$ is a column in the second sheet.

Adjacent cells have the same column or row identifier by default, except if separated by a *cut*. A cut represents the knowledge that the cells on either side of the cut are not required to be in the same row or column of the output even

4 https://github.com/dirko/splyci
5 https://www.minizinc.org/

if they are observed to be in the same row or column in one of the input sheets. Figure 2 shows an example of a cut.

We are unsure about what cuts the user intends without further user input. A probability distribution over cut locations could potentially be constructed, but in this paper we recognize cuts by a heuristic. We add a cut when two cells are separated by the border property at least one empty cell. This is intended to separate independent tables embedded in a single spreadsheet.

Matches A *match* `match`(i, i'), where i and i' come from two different sheets, represent the knowledge that the user wants to align two identifiers between sheets. They can be interpreted as fields or entities that represent the same thing between sheets. A *matching algorithm* is an algorithm that takes a set of spreadsheets as input and produces a set of matches as output. The process of matching column identifiers is related to the mature fields of *schema matching* [3], and matching rows is related to *record linkage* or *entity resolution* [4], depending on the orientation of the cells in the spreadsheet.

In this paper we use a simple algorithm that matches indices when their *headers* match exactly. There is work that automatically identifies header cells in spreadsheets [7], but as a simple baseline the work in this paper takes the topmost or leftmost cells as far as necessary to produce a unique match.

Once the matching algorithm is done, for all `match`(i, i') we replace i' with i so that across all sheets we use a common set of identifiers.

Blocks Working on the cell level is, unfortunately, slow. The number of constraints (discussion below) grows quadratically in the number of cells, which becomes impractical for large spreadsheets, even if the overall structure is simple. We therefore partition each sheet into rectangular *blocks* of cells. Each block b is associated with its position and dimensions with the `block`(b, i, j, w, h) predicate, where i is a column identifier, j a row identifier, and w and h the width and height of the block in number of cells.

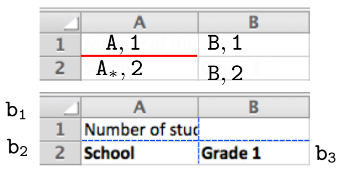


Fig. 2. An example spreadsheet (below) with a cut indicated by the red line. The column and row identifiers are shown inside each cell above. Note that `A` and `A`$_*$ are different. Below, blue dotted lines indicate the partitions induced by the cut, namely (`1`, `2`), (`A`, `B`), and (`A`$_*$, `B`), and the corresponding blocks are labelled around the figure.

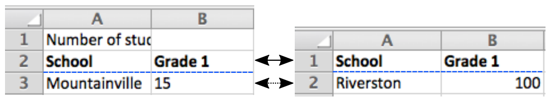
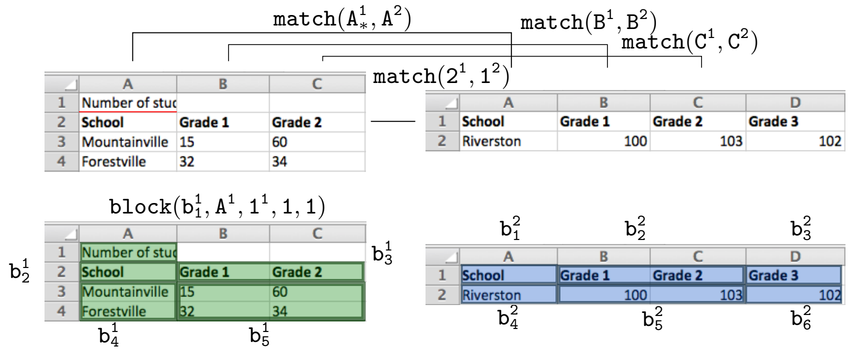


Fig. 3. Two example sheets where row 2 (left) and row 1 (right) match, but row 3 (left) and row 2 (right) do not match. This produces a partition between the two rows, represented by the blue dotted line, as the cells on either side of these rows should be in different blocks.


$\mathtt{match}(\mathtt{A}^1_*, \mathtt{A}^2)$
$\mathtt{match}(\mathtt{B}^1, \mathtt{B}^2)$
$\mathtt{match}(\mathtt{C}^1, \mathtt{C}^2)$
$\mathtt{match}(\mathtt{2}^1, \mathtt{1}^2)$
$\mathtt{block}(\mathtt{b}^1_1, \mathtt{A}^1, \mathtt{1}^1, 1, 1)$
$\mathtt{b}^1_2$
$\mathtt{b}^1_3$
$\mathtt{b}^1_4$
$\mathtt{b}^1_5$
$\mathtt{b}^2_1$
$\mathtt{b}^2_2$
$\mathtt{b}^2_3$
$\mathtt{b}^2_4$
$\mathtt{b}^2_5$
$\mathtt{b}^2_6$

	A	B	C
1	Number of stuc		
2	School	Grade 1	Grade 2
3	Mountainville	15	60
4	Forestville	32	34

	A	B	C	D
1	School	Grade 1	Grade 2	Grade 3
2	Riverston	100	103	102

Fig. 4. Example of the proposed spreadsheet representation for two spreadsheets without formulae. The two sheets with corresponding row and column indices are given above, with the extracted blocks below. Note that the sheet is partitioned both by the cut below Cell `A1` in the left sheet and the matches. Of the block facts, only the $\mathtt{b}^1_1$ block is shown fully.

In general, cuts do not divide a spreadsheet into rectangular blocks, as can be seen in the example in Figure 2. We therefore add *sheet partitions*, which we define as column or row pairs, to ensure that each spreadsheet is partitioned into blocks. The block-level representation of a spreadsheet should keep some of the properties created by cuts and matches on the cell level. For cuts, cells on either side of the cut have different indices. There should therefore be different blocks on either side of a cut. We therefore add a partition along each cut, and perpendicular to cut ends to divide the spreadsheet into four blocks, as is illustrated in Figure 2. For matches, cells containing a matching index should be in a block that is free to align with matching cells in another block. We therefore also add partitions between indices where one of the indices match an index in another sheet but the other one does not match the adjacent index in the other sheet, as shown in Figure 3.

Constraint satisfaction problem (CSP) For each input column i or row j we construct a decision variable x_i or x_j that can take values in $\{1, \ldots, m_i\}$ and $\{1, \ldots, m_j\}$ respectively, where m_i and m_j are maximum dimensions for the output spreadsheet, which we set to the sum of all block widths and heights respectively. We avoid re-using the column and row identifiers as decision variables directly and rather create new variables to make a distinction between the identifiers that are used to index cells, and variables with domains in the positive integers.

A major constraint on the values of the index variables is that the blocks should not overlap. We therefore add a $\mathtt{disjoint\_rectangles}(R)$ constraint on the set of rectangles $R = \{(x_i, x_j, w, h) \mid \mathtt{block}(b, i, j, w, h)\}$. Note that x_i and x_j are decision variables, but w and h are constants in terms of the CSP.

Other than this non-overlapping constraint, we add match constraints and the layout constraints present in the input spreadsheets. Our system currently recognises the following constraints, but more could be added: $\texttt{left}(i, i')$ if column i is to the left of i' in an original sheet, and $\texttt{above}(j, j')$ if row j is above j'. This defines a partial ordering of row and column identifiers.

The constraint satisfaction problem is summarised as follows:

$$\begin{array}{ll}
\text{variables} & X = \{x_i\} \cup \{x_j\}, \quad i, j \text{ over all input sheets,} \\
\text{domains} & x_i \in \{1 \ldots m_i\}, x_j \in \{1 \ldots m_j\}, \\
\text{subject to} & \texttt{disjoint\_rectangles}(R), \\
 & R = \{(x_i, x_j, w, h) \mid \texttt{block}(b, i, j, w, h)\} \\
 & \wedge (\texttt{above}(i, i') \rightarrow x_i < x_{i'}) \wedge (\texttt{left}(j, j') \rightarrow x_j < x_{j'}) \,.
\end{array}$$

For example, the constraint satisfaction problem for Figure 4 is

$$\begin{array}{ll}
\text{variables} & X = \{x_{\texttt{A}^1}, x_{\texttt{A}^1_*}, x_{\texttt{B}^1}, x_{\texttt{D}^2}, x_{1^1}, x_{2^1}, x_{3^1}, x_{2^2}\}, \\
\text{domains} & x_i \in \{1 \ldots 15\} \quad \text{for } x_i \text{ in} \{x_{\texttt{A}^1}, x_{\texttt{A}^1_*}, x_{\texttt{B}^1}, x_{\texttt{D}^2}\} \\
 & x_j \in \{1 \ldots 13\} \quad \text{for } x_j \text{ in} \{x_{1^1}, x_{2^1}, x_{3^1}, x_{2^2}\} \\
\text{subject to} & \texttt{disjoint\_rectangles}(R), R = \{(x_i, x_j, w, h) \mid \texttt{block}(b, i, j, w, h)\} \\
 & \wedge\, x_{\texttt{A}^1_*} < x_{\texttt{B}^1} \wedge x_{1^1} < x_{2^1} \wedge x_{2^1} < x_{3^1} \wedge x_{\texttt{B}^1} < x_{\texttt{D}^2} \wedge x_{2^1} < x_{2^2}.
\end{array}$$

Note that $x_{\texttt{A}^2}$, $x_{\texttt{B}^2}$, $x_{\texttt{C}^2}$, and x_{1^2} are not present because they were replaced by matching identifiers in the other sheet, and that $x_{\texttt{C}^1}$ and x_{4^1} could be removed from the CSP as an optimisation because they occur only inside blocks.

Since the `disjoint_rectangles` constraint is usually built-in using efficient internal representations, standard constraint satisfaction solvers can be used to solve this layout problem. Our implementation is in MiniZinc.

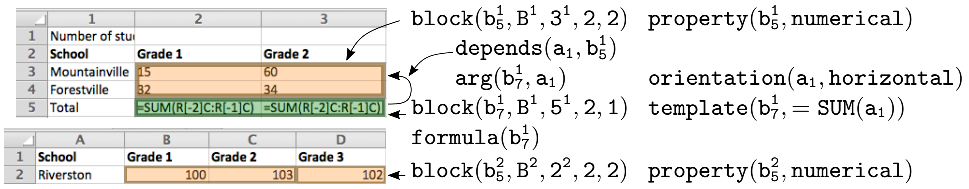
An instantiation of the decision variables represents a possible output layout. Once we have an instantiation, the output spreadsheet is constructed by looking up the values in the original cells and copying them to the output sheet in their new position. Note that this scheme copies each cell in the original sheet to a single cell in the output sheet: it only represents a reordering of cells and cannot duplicate cells.

Formula blocks The scheme described above is able to represent and merge simple spreadsheets without any formulae. We now turn to the problem of representing and merging spreadsheets containing formulae.

The following steps encode formula blocks:

1. Find formula blocks. Add partitions to create blocks where all the cells have the same R1C1[6] representation. When formulae are repeated over different cells with just the row or column incrementing, the R1C1 representation will be the same for all of them.

[6] https://wiki.openoffice.org/wiki/Documentation/How_Tos/Calc:_R1C1_notation Absolute column or row values are replaced with values relative to the formula cell, for example "`=SUM(R[-2]C[0]:R[-1]C[0])`" instead of "`=SUM(B3:B4)`" in the `B5` cell.



	1	2	3
1	Number of stu		
2	**School**	**Grade 1**	**Grade 2**
3	Mountainville	15	60
4	Forestville	32	34
5	Total	=SUM(R[-2]C:R[-1]C)	=SUM(R[-2]C:R[-1]C)

	A	B	C	D
1	**School**	**Grade 1**	**Grade 2**	**Grade 3**
2	Riverston	100	103	102

$\texttt{block}(\texttt{b}^1_5, \texttt{B}^1, 3^1, 2, 2)$
$\texttt{property}(\texttt{b}^1_5, \texttt{numerical})$
$\texttt{depends}(\texttt{a}_1, \texttt{b}^1_5)$
$\texttt{arg}(\texttt{b}^1_7, \texttt{a}_1)$
$\texttt{orientation}(\texttt{a}_1, \texttt{horizontal})$
$\texttt{block}(\texttt{b}^1_7, \texttt{B}^1, 5^1, 2, 1)$
$\texttt{template}(\texttt{b}^1_7, = \texttt{SUM}(\texttt{a}_1))$
$\texttt{formula}(\texttt{b}^1_7)$
$\texttt{block}(\texttt{b}^2_5, \texttt{B}^2, 2^2, 2, 2)$
$\texttt{property}(\texttt{b}^2_5, \texttt{numerical})$

Fig. 5. Example sheets where one sheet now contains a repeated formula. The formula block is colored green and all the blocks with the `numerical` property are colored orange. Note that only one property of $\texttt{b}^1_5$ and $\texttt{b}^2_5$ is listed, as all the various properties of each of the blocks would take too much space to show here.

2. Denote a block b to be a formula with the `formula`(b) predicate, and its template with `template`(b, t), where t is a string where the formula argument ranges are replaced by placeholders, so "`=SUM(R[-2]C:R[-1]C)`" becomes "$= \texttt{SUM}(\texttt{a}_1)$" for example.
3. Associate the formula block b with its argument identifiers with `arg`(b, a), where a is a unique argument identifier, for example $\texttt{a}_1$.
4. Create argument blocks by adding partitions between two cells if they are arguments to two different formulae.
5. Associate the argument identifiers a with argument blocks b_a, `depends`(a, b_a).
6. Associate each argument with its orientation, `orientation`(a, r), where r is either "`vertical`" or "`horizontal`". This determines how block-level arguments map to cell-level arguments. A horizontal argument means that argument cells increment horizontally as formula cells increment horizontally. The left-most cell in the argument block is the argument to the left-most cell in the corresponding formula block, the cell to the right of that is the argument to the formula cell to the right, and so forth.

The representation is summarized in the example in Figure 6.

Properties Raw cells have properties like color, font, type, and other implicit properties like their semantic categories. Our main assumption when generalizing formulae is that all cells with a certain property should be the argument to a certain formula template. Block properties, denoted `property`(b, p), where b is a block and p a property, are the properties that all the cells in that block have in common. The system currently supports cell color, cell type, and properties for each column or row.

Formulae generalisation Now we can construct rules that generalise formulae to new data. For each formula block, we assume that if it has arguments, the height or width of its arguments determine its own height or width. At a high level, we therefore search for rules of the form,

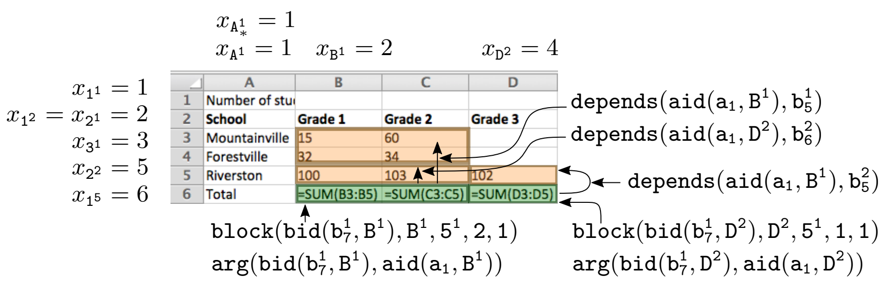
$$\{\text{formula block facts}\} \leftarrow \{\text{argument block facts}\}. \tag{1}$$

$\texttt{block}(f, i, \texttt{5}^1, w, 1) \leftarrow \texttt{block}(b, i, j, w, h), \texttt{property}(b, \texttt{numerical}), f = \texttt{bid}(\texttt{b}_7^1, i).$

$\texttt{arg}(f, a) \leftarrow \texttt{block}(b, i, j, w, h), \texttt{property}(b, \texttt{numerical}), f = \texttt{bid}(\texttt{b}_7^1, i), a = \texttt{aid}(\texttt{a}_1, i).$

$\texttt{depends}(a, b) \leftarrow \texttt{block}(b, i, j, w, h), \texttt{property}(b, \texttt{numerical}), a = \texttt{aid}(\texttt{a}_1, i).$

(a) Prolog rules that replace some of the facts in Figure 5. These rules add formula blocks in the same columns as all numerical blocks. The first line states that, if there exists a block at some location and with some width and height, and that block has the `numerical` property, then a block f in the same column and with the same width, but in row $\texttt{5}^1$ should also exist, where f is an identifier that combines the original block identifier with the column it is in so that f will be unique for each block-column combination produced by the rule.


$x_{\texttt{A}^1_*} = 1$
$x_{\texttt{A}^1} = 1$
$x_{\texttt{B}^1} = 2$
$x_{\texttt{D}^2} = 4$
$x_{1^1} = 1$
$x_{1^2} = x_{2^1} = 2$
$x_{3^1} = 3$
$x_{2^2} = 5$
$x_{1^5} = 6$
$\texttt{depends}(\texttt{aid}(\texttt{a}_1, \texttt{B}^1), \texttt{b}_5^1)$
$\texttt{depends}(\texttt{aid}(\texttt{a}_1, \texttt{D}^2), \texttt{b}_6^2)$
$\texttt{depends}(\texttt{aid}(\texttt{a}_1, \texttt{B}^1), \texttt{b}_5^2)$
$\texttt{block}(\texttt{bid}(\texttt{b}_7^1, \texttt{B}^1), \texttt{B}^1, \texttt{5}^1, 2, 1)$
$\texttt{arg}(\texttt{bid}(\texttt{b}_7^1, \texttt{B}^1), \texttt{aid}(\texttt{a}_1, \texttt{B}^1))$
$\texttt{block}(\texttt{bid}(\texttt{b}_7^1, \texttt{D}^2), \texttt{D}^2, \texttt{5}^1, 1, 1)$
$\texttt{arg}(\texttt{bid}(\texttt{b}_7^1, \texttt{D}^2), \texttt{aid}(\texttt{a}_1, \texttt{D}^2))$

	A	B	C	D
1	Number of stu			
2	**School**	**Grade 1**	**Grade 2**	**Grade 3**
3	Mountainville	15	60	
4	Forestville	32	34	
5	Riverston	100	103	102
6	Total	=SUM(B3:B5)	=SUM(C3:C5)	=SUM(D3:D5)

(b) The facts produced by the rules above on the two spreadsheets in Figure 5, with a solution to the accompanying CSP. Since $\texttt{B}^2$ matches $\texttt{B}^1$ and was therefore replaced by it, the rule produces a block $\texttt{bid}(\texttt{b}_7^1, \texttt{B}^1)$ that depends on two blocks, namely b_5^1 and b_5^2.

Fig. 6. Example of the proposed representation where one sheet contains a formula.

Since the head is a conjunction with free variables which cannot directly be implemented in Prolog, we implement these rules by splitting the conjunction into separate rules and creating compound terms *block ID* $\texttt{bid}(f, i)$ and *argument ID* $\texttt{aid}(a, i)$ to represent identifiers for the new formula blocks and arguments. f is the original formula block, i is the column of the argument, and a the original argument identifier. These terms are necessary because a rule can generate new formula blocks and their corresponding arguments, and we need a unique way to identify each block and argument. See Figure 6a for an example of a rule that covers a single formula block and its implementation in Prolog.

Techniques from *inductive logic programming* [10] can be used to learn these rules to cover the spreadsheet formulae, but since our desired rule format is fixed we implement a custom search algorithm. The main challenge is in finding the property of the argument blocks that "explains" each formula. For a specific formula block, we first find all its argument blocks with the same width or height as itself. For each of these argument blocks, which we assume are the blocks that determine the formula's extent, we find the set of properties that uniquely identify it by taking the properties of the argument block that are not properties of any other blocks.

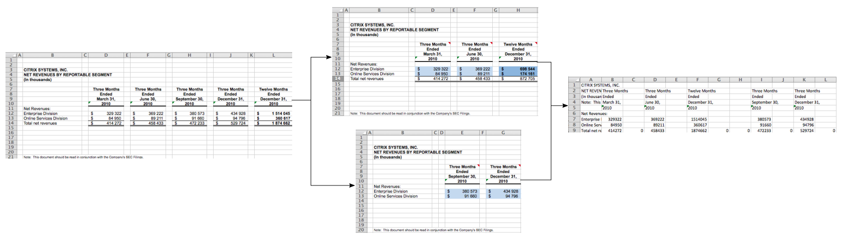
The facts and rules that represent each sheet are now merged by taking their union, before grounding and finding all `blocks` to pass to the constraint solving phase. Note that rules can depend on the result of other rules, as formulae can be arguments to other formulae.

Constraint satisfaction problem With formulae, the CSP step is the same as previously, as the generalisation of formulae happens independently of the assignment of values to columns and rows. The argument templates of formulae are then filled in according to their orientations.

4 Experiments

The prototype is evaluated by comparing its output to that obtained by manually integrating multiple spreadsheets.

The Fuse corpus[7] [1] is a collection of publicly available spreadsheets scraped from the web. Since not every spreadsheet in this corpus is mergeable with every other spreadsheet (because of different domains and contents), we manually constructed a set of 15 mergeable spreadsheet pairs. We draw a random sample of spreadsheets, split suitable sheets horizontally or vertically to create two mergeable sheets, and remove formulae from the second sheet. The smallest sheet in the sample is $8 \times 17 = 136$ cells and the largest $12 \times 512 = 6144$. The experimental process is illustrated with an actual sheet from the evaluation set in Figure 7.



	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3		CITRIX SYSTEMS, INC.										
4		NET REVENUES BY REPORTABLE SEGMENT										
5		(In thousands)										
6												
7				Three Months		Three Months		Three Months		Three Months		Twelve Months
8				Ended		Ended		Ended		Ended		Ended
9				March 31,		June 30,		September 30,		December 31,		December 31,
10				2010		2010		2010		2010		2010
11		Net Revenues:										
12		Enterprise Division		$ 329 322		$ 369 222		$ 380 573		$ 434 928		$ 1 514 045
13		Online Services Division		$ 84 950		$ 89 211		$ 91 660		$ 94 796		$ 360 617
14		Total net revenues		$ 414 272		$ 458 433		$ 472 233		$ 529 724		$ 1 874 662
15–20												
21		Note: This document should be read in conjunction with the Company's SEC Filings.										

	A	B	C	D	E	F	G	H
1								
2								
3		CITRIX SYSTEMS, INC.						
4		NET REVENUES BY REPORTABLE SEGMENT						
5		(In thousands)						
6								
7				Three Months		Three Months		Twelve Months
8				Ended		Ended		Ended
9				March 31,		June 30,		December 31,
10				2010		2010		2010
11		Net Revenues:						
12		Enterprise Division		$ 329 322		$ 369 222		$ 698 544
13		Online Services Division		$ 84 950		$ 89 211		$ 174 161
14		Total net revenues		$ 414 272		$ 458 433		$ 872 705
15–20								
21		Note: This document should be read in conjunction with the Company's SEC Filings.						

	A	B	C	D	E	F	G
1							
2							
3		CITRIX SYSTEMS, INC.					
4		NET REVENUES BY REPORTABLE SEGMENT					
5		(In thousands)					
6							
7					Three Months		Three Months
8					Ended		Ended
9					September 30,		December 31,
10					2010		2010
11		Net Revenues:					
12		Enterprise Division			$ 380 573		$ 434 928
13		Online Services Division			$ 91 660		$ 94 796
14–19							
20		Note: This document should be read in conjunction with the Company's SEC Filings.					

	A	B	C	D	E	F	G	H	I	J	K	L
1	CITRIX SYSTEMS, INC.											
2	NET REVEN	Three Months		Three Months		Twelve Months			Three Months		Three Months	
3	(In thousan	Ended		Ended		Ended			Ended		Ended	
4	Note: This	March 31,		June 30,		December 31,			September 30,		December 31,	
5		2010		2010		2010			2010		2010	
6	Net Revenues:											
7	Enterprise	329322		369222		1514045			380573		434928	
8	Online Serv	84950		89211		360617			91660		94796	
9	Total net re	414272	0	458433		1874662	0	0	472233	0	529724	0

Fig. 7. The experimental process illustrated with a spreadsheet from the evaluation set. A sheet is manually split, then the two sheets are manually annotated as necessary to merge with the system.

We annotate the sheets with the necessary information if the default heuristics fail and note the number of annotations that is necessary as a measure of the manual effort that can be saved in the future with better property, cut, and constraint recognisers. Cut annotations, implemented as comments in the spreadsheet file, are added between multiple tables embedded in a single sheet,

[7] http://static.barik.net/fuse/

	Total	Correct	Avg. %
Cuts	15	8	30%
Possible matches	3588	3498	73%
Properties	169	1	1%
Blocks correctly laid out	1155	870	90%
Sheets layout	15	7	47%
Sheets layout (ignoring order)	15	10	67%

(a) Integration results. The average column is the micro average, in other words the average percentage correct per sheet. Note that the match heuristic is much more successful than the cut or property heuristics at avoiding user input.

# annotations	# sheets
None	4
1 to 5	5
6 to 15	2
More than 15	4

(b) A frequency table of the number of annotations necessary to solve each spreadsheet pair.

Table 1. Result summary

and between descriptive cells and headers. Matches are annotated by adding matching column or row identifiers as comments when the match heuristic fails, while cells are colored manually when there is no other default property that uniquely covers a formula argument. We count a contiguous rectangle of cells that had to be colored as one annotation.

The results are summarised in Table 1. Over half of the sheets were solvable with 5 or fewer annotations. We distinguish between two types of correct layout results: fully correct layout and correct when ignoring the ordering of some blocks. In some cases, like in Figure 7, the system produced a layout that had the correct meaning but not exactly the same column or row order as the original. A human would put the year-total to the right of all the months, but the system added the total in the middle. Such cases could be handled within the current framework by allowing more specific layout constraints such as that a formula is to the right, or below, its arguments.

On average, it took 22 seconds to process a sheet pair, with most pairs correctly merged within a few seconds, which means that a system like this has the potential to be useful in an interactive setting where it takes longer to add annotations or manually merge sheets than the system takes to compute merge proposals.

5 Related work

Chen et al. [5] were the first to undertake automatic spreadsheet integration. They separate the problem into *extraction* and *integration* phases. They concentrate on the extraction of relational data from spreadsheets and use off-the-shelf database integration tools for the integration phase. To extract relational data from spreadsheets, they identify pivoted tabular data with machine learning techniques. They do not, however, address the integration of formulae or other constraints that we address in this work, and do not present the result as a spreadsheet again for the end-user.

Other approaches for extracting relations from spreadsheets include FlashRelate [2], which allows a user to visually specify transformations, Foofah [8], which learns transformations by example, and the predictive program synthesis approach in [12], which searches for type-consistent relations with a zero-information-loss constraint. None of these extraction techniques take formulae into account, in contrast with our work.

Kolb et al. [9] developed a system that is able to automatically extract constraints from spreadsheets where the formula information is not present. It searches for possible constraints over blocks of cells, limiting the search space to only sub-blocks of the correct shape. Their view that spreadsheets can contain constraints inspired the current work, although we take the idea further by including layout constraints.

There is a vast literature on schema matching and mapping for relational data [3]. A recent take on the problem is in the context of data science, where Sutton et al. [11] created a tool to synthesise executable summaries of changes between tables. In contrast to our work, they do not take spreadsheet formulae into account, consider layouts other than relational data that are already in first normal form, or do the merge after calculating the changes.

6 Conclusions and future work

We have introduced the spreadsheet integration problem as the problem of merging multiple spreadsheets to obtain a new spreadsheet while generalising formulae so that they take data from the other sheets into account. We proposed a layout constraint representation in which merging sheets is achieved by solving the resulting constraint satisfaction problem. We present results of a system prototype that uses baseline heuristics to solve some of the sub-problems and show that it is able to correctly recover artificially split spreadsheets. We use the number of manual annotations that had to be added as a measure of user effort.

Going forward, the validation framework should be extended to include naturally occurring spreadsheet pairs, and the measure should be refined to better reflect saved user effort. We also made some strong assumptions such as identical column or row headers, and the system could be extended in a natural way allowing for soft matches between headers and determining their effects. This is a challenge that is shared with database integration.

In the future, the performance of the system can be improved by using better heuristics or machine learning approaches to lower the number of manual annotations required. The use of soft constraints can be investigated to handle duplicate or contradictory data gracefully, and under-specified constraints can be disambiguated by using a cost function to rank solutions.

Seeing spreadsheets as user-intended layout constraints opens up the possibility of investigating exactly which layout constraints are used in typical spreadsheets. So far, we have identified only simple same-column and same-row constraints together with some ordering on the columns and rows. But a more complete understanding of the constraints users typically use in spreadsheets will enable better integration of sheets and might enable other automated applications on spreadsheets.

Acknowledgements This work was funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No [694980] SYNTH: Synthesising Inductive Data Models). LDR is also funded by the Flemish Government (AI Research Program).

References

1. Barik, T., Lubick, K., Smith, J., Slankas, J., Murphy-Hill, E.: Fuse: A reproducible, extendable, internet-scale corpus of spreadsheets. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. pp. 486–489 (2015)
2. Barowy, D.W., Gulwani, S., Hart, T., Zorn, B.: FlashRelate: Extracting relational data from semi-structured spreadsheets using examples. SIGPLAN Notices **50**(6), 218–228 (Jun 2015)
3. Bernstein, P.A., Madhavan, J., Rahm, E.: Generic schema matching, ten years later. Proceedings of the VLDB Endowment **4**(11), 695–701 (2011)
4. Brizan, D.G., Tansel, A.: A survey of entity resolution and record linkage methodologies. In: Communications of the International Information Management Association. vol. 6 (Jan 2006)
5. Chen, Z., Cafarella, M.: Integrating spreadsheet data via accurate and low-effort extraction. In: Proceedings of the 20th ACM SIGKDD. pp. 1126–1135. KDD '14, ACM, New York, NY, USA (2014)
6. De Raedt, L., Blockeel, H., Kolb, S., Teso, S., Verbruggen, G.: In: 17th International Symposium, Advances in Intelligent Data Analysis XVII (2018)
7. Doush, I.A., Pontelli, E.: Detecting and recognizing tables in spreadsheets. In: Proceedings of the 9th IAPR International Workshop on Document Analysis Systems. p. 471–478. DAS '10, ACM, New York, NY, USA (2010)
8. Jin, Z., Anderson, M.R., Cafarella, M., Jagadish, H.V.: Foofah: Transforming data by example. In: Proceedings of the 2017 ACM SIGMOD. pp. 683–698. SIGMOD '17, ACM, New York, NY, USA (2017)
9. Kolb, S., Paramonov, S., Guns, T., De Raedt, L.: Learning constraints in spreadsheets and tabular data. Machine Learning **106**(9-10), 1441–1468 (Oct 2017)
10. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. The Journal of Logic Programming **19-20**, 629 – 679 (1994)
11. Sutton, C., Hobson, T., Geddes, J., Caruana, R.: Data Diff: Interpretable, executable summaries of changes in distributions for data wrangling. In: Proceedings of the 24th ACM SIGKDD. pp. 2279–2288. KDD '18, ACM, New York, NY, USA (2018)
12. Verbruggen, G., De Raedt, L.: Towards automated relational data wrangling. In: Proceedings of AutoML@PKDD/ECML 2017, Skopje, Macedonia, September 22, 2017. pp. 12–20 (2017)