The Knowledge Engineering Review, Vol. 22:3, 281–295. © 2007, Cambridge University Press
doi:10.1017/S0269888907001154 Printed in the United Kingdom

PrediCalc: a logical spreadsheet management system

MICHAEL KASSOFF and MICHAEL R. GENESERETH

Gates Hall, Computer Science Department, Stanford University, Stanford, CA 94305, USA
e-mail: mkassoff@stanford.edu, genesereth@stanford.edu

Abstract

In this article, we describe PrediCalc, a logical spreadsheet that allows for many-to-many constraints and propagation in all directions. We explain PrediCalc's update mechanism and PrediCalc's unique approach to handling inconsistencies between the spreadsheet values and the spreadsheet formulas. We have developed a paraconsistent entailment relation for the purpose of computing the consequences of PrediCalc's value assignments under inconsistency.

We close with thoughts on the prospects of logical spreadsheets on the World Wide Web, and describe our initial *Websheet* prototypes.

1 Introduction

Computerized spreadsheets are a great success. They are often touted in newspapers and magazine articles as the first 'killer app' for personal computers. Over the years, they have proven their worth time and again. Today, they are used for managing enterprises of all sorts—from one-person projects to multi-institutional conglomerates. Their applications range from financial planning to scientific data analysis to maintaining shopping lists.

The power of computerized spreadsheets derives in large part from two primary features—the automatic calculation of the values by spreadsheets and the use of mathematical formulas to specify those calculations. The automatic calculation of values frees the user from the tedious task of doing those calculations manually. The automatic *re*calculation of values upon changes to the spreadsheet is an even more powerful feature, allowing for easy 'what if' analysis. Once the initial formulas are entered, exploring different scenarios is as simple as changing the initial parameters. This power is easy to take for granted, but consider that before electronic spreadsheets existed calculations were carried out on paper spreadsheets, and if one wanted to change his assumptions, or if an error was detected early in the computation, then essentially all of the computation would have to be carried out again.

The support for mathematical formulas simplifies the task of setting up the calculations and makes spreadsheet technology accessible to a broad class of users, including those with no background in programming. This ease-of-use derives largely from the fact that spreadsheets are programmed using familiar mathematical notation, such as $C1 = A1 + B1$. Furthermore, unlike a traditional programming environment, all of the intermediate computation steps are displayed for the programmer. Programming a spreadsheet is like being in 'debug mode' all the time, where one can see all computation steps at once.

Despite their successes, computerized spreadsheet systems today have significant and unnecessary restrictions that limit their usefulness. One significant restriction is that the formulas used to specify calculations must be functions. Spreadsheet systems generally do not provide the ability to encode many-to-many relationships across cell values. For example, one cannot say that the

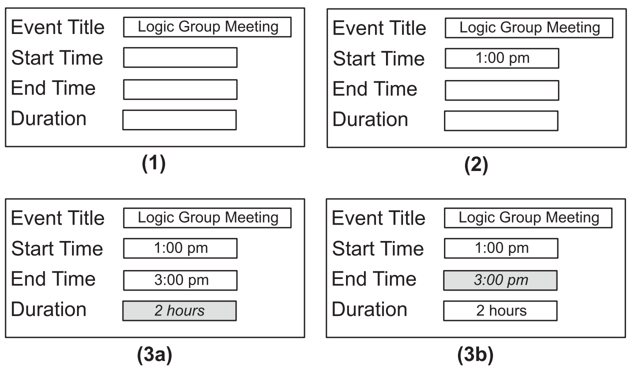


Figure 1 Creating an event. (1) The user first sets the title for the event. (2) The user then sets the start time for the event to 1:00 pm. (3a) In one scenario, the user then sets the end time for the event to 3:00 pm. The duration is then automatically filled in as 2 hours. (3b) In an alternate scenario, the user instead chooses the duration of the event to be 2 hours. The end time of the event is then automatically filled in as 3:00 pm.

values in cells A1 and A2 are different. Another unnecessary restriction is that propagation can only occur in one direction in a traditional spreadsheet. For example, if one defines B2 = B1, then one can specify a value for B1 and B2 will obtain that same value, but one cannot then assign a value to B2 and expect B1 to obtain that value as well even though the cells have equal values.

If these restrictions are lifted, then spreadsheets gain additional utility. Consider, for example, the following spreadsheet that allows for the creation of an event. This spreadsheet has four cells, which allow for the specification of the event's name, start time, end time, and duration (Figure 1.1). The spreadsheet also has some formulae (not shown), which specify that 'the start time must be before the end time' and that 'the start time plus the duration equals the end time'. Note that the first formula is many-to-many, and that the second formula specifies each of the start time/end time/duration cells in terms of the other two. Thus, if the user were to specify 1:00 pm as the start time of the Logic Group Meeting (Figure 1.2) and 3:00 pm as the end time, then the duration can be automatically filled in as 2 hours (Figure 1.3(a)). Or, since the system can propagate values in any direction, the user could instead specify the start time to be 1:00 pm and the duration to be 2 hours and have the end time automatically filled in as 3:00 pm (Figure 1.3(b)). Similarly, the user could first fill in the duration to be 2 hours and the end time to be 3:00 pm, and have the start time automatically filled in as 1:00 pm (not shown).

To allow for many-to-many relationships to be expressed between cells, we generalize the spreadsheet formula language from function definitions to logical formulae. For example, we might formalize the constraint 'the start time must be before the end time' by the formula $startTime(S) \wedge endTime(E) before(S, E)$.

While traditional spreadsheets were built to perform mathematical calculations, logical spreadsheets are ideal for handling symbolic data. For example, it is easy to express the rule 'only senior managers can reserve the third floor conference room' using the formula $eventOwner(O) \wedge \neg seniorManager(O) \neg eventRoom(room301)$.

In addition, it is easy to handle relational data in a logical spreadsheet. For example, given three relational database tables, *income(id, income)*, *age(id, age)*, and *status(id, status)*, we could write a rule specifying that 'insurance applicants who make at least $60,000 and are under 50 yrs old are approved' as $income(ID, I) \wedge age(ID, A) \wedge I \geq 60000 \wedge A < 50 status(ID, approved)$.

There are some interesting design decisions to be made when building a logical spreadsheet management system. In particular, there is the question of what to do when an update to the spreadsheet would be inconsistent with a formula. For example, what should happen if, in the example above, the user were to specify a start time of 2:00 pm and an end time of 1:00 pm? Or, what if the user were to specify a start time of 1:00 pm, an end time of 3:00 pm, and a duration of 5 hours? We present our solution to these problems in the body of this article.

In this article, we describe PrediCalc, a logical spreadsheet management system that allows for many-to-many logical formulae and propagation in all directions. We begin by describing applications of PrediCalc, and then we describe some design decisions we made when developing the system. We continue with a detailed example of PrediCalc in operation. We conclude with a description of future work, in particular applications to the World Wide Web.

2 Applications

Before we delve into the particulars of our system, we should first answer the question, ‘What are logical spreadsheets useful for?’ Logical spreadsheets are indeed useful for a wide variety of applications that are beyond the reach of traditional spreadsheets. Some broad classes of applications are as follows.

Smart forms. A spreadsheet with an HTML front end would allow users to fill out online forms in which data are checked for semantic well-formedness. The event creation example from the previous section is an example of this. Another example is the constraint for an airline travel site that ‘the number of lap infants traveling must not be greater than the number of adults’. Or, consider the many-to-many relationship between US cities and ZIP codes—a city may have many associated ZIP codes, and a ZIP code may be associated with multiple cities.

Data management. Logical spreadsheets facilitate the entry and editing of symbolic data governed by symbolic constraints. ‘Correct on capture’ data entry systems and resource management systems are examples of this capability. The constraint ‘only senior managers can reserve the third floor conference room’ is an example of this. Since laws and policies can be expressed using logic, a logical spreadsheet can be thought of as a simple kind of *computational law* system (Love & Genesereth, 2005).

Design and configuration. Configuration systems are good examples of the use of logical spreadsheets in design. Consider, for example, a configuration system to help consumers design their own cars, which might have the constraint ‘if the car’s exterior color is blue, then the car interior color may be gray, tan or black’. Or consider a student designing his course schedule, which might have the constraint ‘students must take at least two math courses to graduate’.

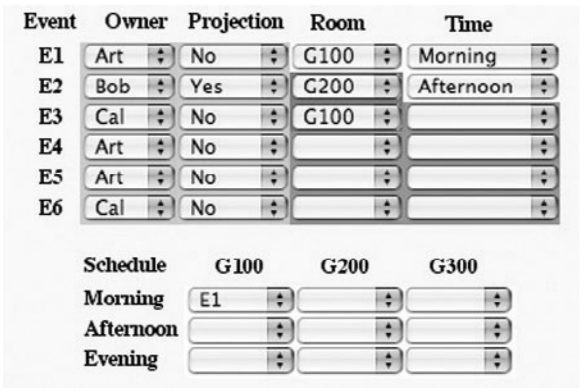
Interactive documents. Systems can return ‘interactive answers’ to users, e.g., simulations, which allow a user to experiment by varying certain parameters while the system automatically propagates the consequences of those variations. Consider, for example, a student learning how lenses refract light by experimenting with different lens shapes. Or consider a spreadsheet used by an insurance agent to determine if a client is eligible for a particular kind of insurance. The rule that ‘insurance applicants who make at least $60,000 and are under 50 yrs old are approved’ is an example of this. Essentially, an interactive document allows one to perform the ‘what if’ analyses that spreadsheets are famous for, although there need not be a distinction between the cells used as input parameters and the cells used to output results.

Logical spreadsheets also have many applications to the World Wide Web. We describe these applications in Section (10).

3 The user experience

We have developed a fully functional prototype logical spreadsheet prototype called PrediCalc. PrediCalc is a desktop application written in Common Lisp, built on top of the Epilog Model Elimnation theorem proving system (Genesereth, 1995).

The user interface of PrediCalc is somewhat different from that of traditional spreadsheets. A user creating a new spreadsheet document with PrediCalc is greeted with a blank canvas, a textual constraint editor, and a domain editor. The experience is similar to that of other canvas-based WYSIWYG tools, such as visual Web page designers. The user begins by placing cells and textual



Event	Owner	Projection	Room	Time
E1	Art	No	G100	Morning
E2	Bob	Yes	G200	Afternoon
E3	Cal	No	G100	
E4	Art	No		
E5	Art	No		
E6	Cal	No		

Schedule	G100	G200	G300
Morning	E1		
Afternoon			
Evening			

Figure 2 Part of a room management system created using PrediCalc

labels on the canvas. The user may also place static text onto the canvas, change the color scheme, etc.

A cell may have any number of modalities, such as a drop-down list or a type-in field. In addition, cells may be arranged into tables, complete with row and column names. This arrangement of cells into tables serves not only to visually organize cells but also allows cells to be given names based on their rows and columns. Two tables created in PrediCalc are shown in Figure 2. The first table has six rows, representing events that need to be scheduled, and four columns, containing some properties of the events, namely their owner, whether a projector is required, their room, and their time. The second table represents the event schedule, where each cell contains the event scheduled in a given room at a given time.

Once the cells and tables are laid out, the user can create formulae that express relationships between cells. The formulae are written textually using a variant of first-order logic (the formula language is explained in more detail in Section (4)). The user can also create domains for cells using a textual editor and associate each cell with a domain. These domains are used to populate cell drop-down lists.

Finally, the user may proceed to use the newly created spreadsheet. As the user enters and deletes values from cells, the values in other cells are changed automatically. PrediCalc's approach to update is described in the Section (6).

4 Formal definitions

A *spreadsheet* consists of a finite set of *cells* together with a set of *formulae* that relate the cells. An instance of a spreadsheet is a mapping from cells to *values* for those cells. Informally, when a cell is mapped to a value we say that the cell contains that value. In a traditional spreadsheet, a cell contains either 0 or 1 values, and those values are typically literal values, for example, numbers, strings, dates or times. Note that this definition of a traditional spreadsheet is equivalent to the definition of a constraint satisfaction problem (Russell & Norvig, 2003).

We represent the assignment of an atomic value a to a cell c by the logical formula $c(a)$. The formulae which we use to relate the cells of our spreadsheet to each other are function-free universal formulae, that is, quantifier-free formulae of first-order logic without function symbols. We allow for interpreted predicates such as $+$ and $>$ as well as $=$. We use a traditional logical semantics such as that found in Enderton (2000).

5 Cell names

In addition to simple, atomic names for cells, our system allows for cells to be given structured names. When a cell is part of a *spreadsheet table*, it is addressed by its position within the table,

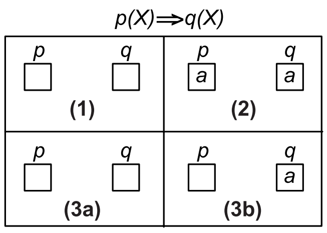


Figure 3 (1) A spreadsheet with two empty cells, *p* and *q*, and the constraint *p*(*X*)*q*(*X*). (2) The user places the value *a* into cell *p*. The value *a* is automatically placed in cell *q*. (3a) The user removes value *a* from cell *p*. The value *a* is automatically removed from cell *q*. (3b) Alternatively, when the user removes value *a* from cell *p*, the value *a* remains in cell *q*.

similar to cells in a traditional spreadsheet. For example, in Figure 2, the structured name event [e2, g100] refers to the cell in the event table in the e2 row and the g100 column. This sort of structured name allows rows and columns to be quantified over. These structured names are similar to the structured predicate names allowed in the logic programming language HiLog (Chen *et al.*, 1993).

In addition to improving the user experience by reducing the replication of formulas typically required in a traditional spreadsheet, structured names allow spreadsheet tables to be queried in a manner similar to relational database tables. Indeed, since all rows in a spreadsheet table are named, one can either treat a row as a tuple with attributes named by the columns, or treat a column as a tuple with attributes named by the rows.

Examples of spreadsheet formulas with structured names are found in Figures 6 and 7.

6 Update

In this section, we discuss the decisions we made when designing the update mechanism of our logical spreadsheet system. As we shall see, the generalization from unidirectional functions to many-to-many relationships leads to many choices. We describe PrediCalc's handling of value assignments in detail; we omit a description of PrediCalc's value deletion semantics, which are analogous.

6.1 Bilevel update

In a traditional spreadsheet, the formulae partition the cells into 'input cells' and 'output cells', where the output cells are functions of the input cells. In a logical spreadsheet, this is no longer the case: regardless of the formulae, any cell can be an input cell or an output cell.

Although a logical spreadsheet has this additional functionality, it should still behave in the same way that a traditional spreadsheet would behave in the case that this additional functionality is not used. Consider a simple spreadsheet with two cells, *p* and *q*, together with the formula *p*(*X*)*q*(*X*), shown pictorially in Figure 3.1. If the user were to place value *a* into cell *p*, then the formulae tell us that *q* must contain value *a* as well (Figure 3.2). What should happen, then, if the value *a* is removed from cell *p*. In a traditional spreadsheet, *a* would automatically be removed from cell *q* as well, since it is no longer computable by the 'input cell', namely *p*, and the formulae (Figure 3.3(a)). The user would be surprised if that the value *a* were to remain in cell *q* (Figure 3.3(b)), because this value is no longer supported by any other input values in the spreadsheet.

To support this behavior, the spreadsheet must keep track of which values were supplied by the user and which values were not. We shall therefore separate the values into *base* values and *computed* values, where the base values are supplied by the user and the computed values are entailed by the base values and the spreadsheet formulae.

Thus, in the example above, *p*(*a*) would be a base-value assignment and *q*(*a*) would be a computed-value assignment.

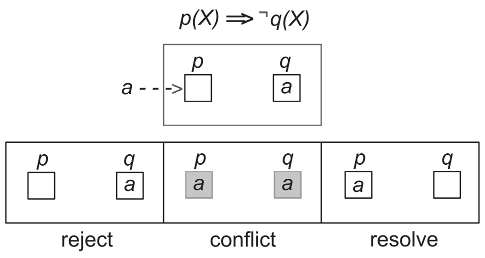


Figure 4 A spreadsheet with two cells, p and q, and the constraint $p(X) \neg q(X)$. At first, cell q contains value a. The user then attempts to place value a into cell p. Three possible outcomes are depicted: at left, the attempt is rejected; at center, the update results in a conflict between $p(a)$ and $q(a)$; at right, the conflict is automatically resolved.

In the sequel, we will represent the set of base-value assignments by Λ and the set of formulae by Ω. The computed-value assignments are those entailed by Λ and Ω.

6.2 Dealing with unambiguous conflicts during update

An interesting possibility raised by multidirectional update is how to deal with the case that the value being entered into the spreadsheet conflicts with the values currently in the spreadsheet. For example, consider a spreadsheet with two cells, p and q, and one constraint, $p(X) \neg q(X)$. The user first places value a into cell q, as depicted at the top of Figure 4. What should happen if the user then attempts to place value a into cell p, directly conflicting with value a in cell q ? We consider three possibilities, shown at the bottom of Figure 4.

1. The request is rejected by the system.
2. The request is executed by the system, resulting in a conflict between cells p and q.
3. The request is executed by the system and, in addition, the value a is automatically removed from cell q, thus resolving the conflict.

In a traditional spreadsheet, when a user places a value into a cell, the result is that (1) the cell has that value, and (2) all consequences of that value are propagated throughout the spreadsheet. We take the same approach. Thus, since a consequence of cell p having value a is that cell q cannot have value a, the system automatically removes value a from q, resulting in the third scenario above.

Formally, if λ is the set of current value assignments, λ is a newly assigned value assignment, and Λ' is the resulting set of value assignments after the update, then we define:

$$\begin{aligned} \Lambda' = \Lambda \cup \phi \in \mathit{ValueAssignments} | \lambda \cup \Omega \models \phi \\ - \neg\psi \in \mathit{NegValueAssignments} | \lambda \cup \Omega \models \neg\psi \end{aligned} \qquad (*)$$

where *ValueAssignments* is the set of all possible value assignments and *NegValueAssignments* is the set of all possible negated-value assignments. The restriction of ϕ to *ValueAssignments* ensures that we do not add into Λ logical sentences such as disjunctions, negations, and non-ground sentences, as well as ground atoms $r(b)$ such that r is not a cell name. Note that the corresponding restriction of Ψ to *NegValueAssignments* is actually not necessary, but is included for symmetry.

6.3 Dealing with ambiguities during update

Things are not always so simple. It may be the case that a new value placed into the spreadsheet does not directly conflict with any value currently in the spreadsheet, but instead conflicts with some combination of values.

For example, consider that we have a spreadsheet with three cells, $d1$, $d2$, and $d3$, representing the departments of the courses a student is taking. Suppose also that the spreadsheet contains a formula stating that ‘a student cannot take three physical education classes’, written as

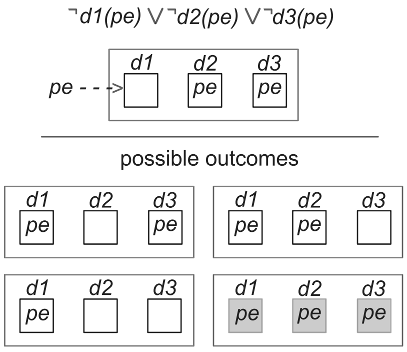


Figure 5 A spreadsheet with three cells, *d*1, *d*2, and *d*3 and a constraint stating that they cannot all have the value *pe*. At first, *d*2 and *d*3 contain the value *pe*. The user then places the values *pe* into cell *d*1. Four possible outcomes are displayed.

$\neg d1(pe) \vee \neg d2(pe) \vee \neg d3(pe)$. So, what should happen when two of the courses are in the physical education department, and the student attempts to take a third course in the physical education department? Figure 5 displays four alternatives.

We would like to obey the user's request and, at the same time, we would like to resolve the conflict as we did in Section (6.2). There are three possible ways to resolve the conflict, namely to remove *pe* from cell *d*2, to remove *pe* from cell *d*3, and to remove *pe* from both *d*2 and *d*3. The system has no reason to prefer either *d*2 or *d*3 over the other. Furthermore, while the third option is fair, it is fair at the loss of information. While this example is small, with only three cells, a larger example with, say, 100 cells and a similar constraint would result in 99 deletions, a completely unacceptable loss of information. Thus, we choose the final option: place *pe* into *d*1, and leave the other cells alone.

Note that definition (*) does exactly this, with no modification.

6.4 Dealing with inconsistency in a static spreadsheet

Our decision to allow conflicts between values in our spreadsheet has an unfortunate consequence: we cannot simply define the computed values as the logical consequences of the base-value assignments and the spreadsheet formulae. To see why, recall that a set of sentences Π logically entails a sentence Ψ if and only if every model of Π is a model of Ψ (see Enderton, 2000). However, if the base-value assignments are inconsistent with the spreadsheet formulae, then this set of sentences has no models, and therefore every model of these sentences is a model of any sentence. Therefore, when the base-value assignments are inconsistent with the spreadsheet formulae, *every value is logically entailed in every cell*, thus rendering the spreadsheet useless.

We must therefore turn to a paraconsistent notion of entailment. The notion of entailment used by PrediCalc is as follows. Let Λ be a set of base-value assignments and let Ω be the spreadsheet formulae. Then we say that a computed-value assignment $\rho(\tau)$ is existentially Ω-entailed by Λ if there is some subset $\lambda \subseteq \Lambda$ such that $\lambda \cup \Omega$ is consistent and $\lambda \cup \Omega$ logically entails $\rho(\tau)_1$.

Note that existential ω-entailment behaves the same as logical entailment as long as the base-value assignments are consistent with the spreadsheet formulae. However, when the base values are inconsistent with the spreadsheet formulae, the number of entailed computed values is kept in check.

For example, consider that in addition to cells *d*1, *d*2, and *d*3, we have a fourth cell *m*, which is meant to contain the student's major. If cells *d*1, *d*2, and *d*3 all contain value *pe*, then $\{d1(pe), d2(pe), d3(pe)\} \cup \{\neg d1(pe) \vee \neg d2(pe) \vee \neg d3(pe)\}$is inconsistent and logically entails *m*(*pe*), *m*(*cs*), *m*(*art*), and so on. However, note that any subset of $\{d1(pe), d2(pe), d3(pe)\}$ taken together with the formula $\neg d1(pe) \vee \neg d2(pe) \vee \neg d3(pe)$ does not entail any value for *m*, and therefore no value of *m* is existentially Ω-entailed by this spreadsheet.

```
schedule[T,R](E) <=> event[E,time](T) and event[E,room](R)
event[E,projection](yes) and event[E,room](R) => room[R,projector](yes)
event[E,owner](P) and person[P,faculty](no) => not event[E,room](g100)
```

Figure 6 Formulae for the room manager. The first formula relates the schedule table to the event table. The second states that events that require a projector must be scheduled in a room with a projector. The third states that only faculty members can reserve room g100. Note that free variables are considered to be universally quantified.

```
event[W,Z](X) and not X=Y => not event[W,Z](Y)
schedule[W,Z](X) and not X=Y => not schedule[W,Z](Y)
room[W,Z](X) and not X=Y => not room[W,Z](Y)
person[W,Z](X) and not X=Y => not person[W,Z](Y)
```

Figure 7 Formulae which specify that the cells in each of the tables are single-valued. These constraints are automatically generated by the system.

6.5 *Dealing with single values*

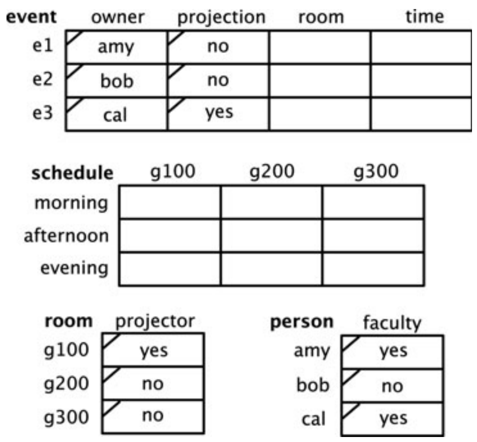
The update formula (*) plus existential entailment are the core ideas from PrediCalc's update mechanism that we would like to impart to the reader. However, for completeness we will mention one final complication. In PrediCalc, cells can hold at most one value. What should happen, then, if there is more than one value computable for a cell? For example, consider a spreadsheet with five cells, p_1, q_2, p_2, q_2, and r, and two formulas, $p_1(x) \wedge q_1(x) \Rightarrow r(x)$ *and* $p_2(x) \wedge q_2(x) \Rightarrow r(x)$. Say that the spreadsheet now has base-value assignments $\{p_1(a),q_1(a,p_2(b)\}$ and therefore has computed-value assignment $r(a)$. If the user places value b into cell q_2, then the base-value assignments will be $\{p_1(a),q_1(a,q_1(b)q_2(b)\}$, and there will be two computed-value assignments, $r(a)$ and $r(b)$. However, since PrediCalc's cells are single-valued, we must choose a single value to display for the cell. Our criterion is to prefer the value that was in the cell before the update, in this case $r(a)$. We therefore say that our update mechanism is *intertial*, since the values currently in computed cells remain in the cell until they are no longer computable. This is to be compared with an *recency-preferential* semantics in which the most recently computed value always appears in a computed cell, or a *fair* semantics in which no value is shown in a computed cell if multiple values are computed for it.

6.6 *Example*

We illustrate our approach using the room management system of Figure 2 as an example. The room manager consists of four tables, shown schematically in Figure 8. The top table contains event requests, each of which has an owner, a specification of whether a projector is needed, a room, and a time. The center table contains a schedule of the events. The information is redundant with the first table, but is useful because it offers a different view. The bottom-left table lists whether or not each room has a projector. The bottom-right table lists whether each person is a faculty member or not. All cells are single-valued. To distinguish base values from computed values, we will place a triangle in the upper left-hand corner of a cell containing a base value.

This spreadsheet has three formulae that were entered by the user who created the spreadsheet, shown in Figure 6. The first of these formulae relates the event table to the schedule table. The second states that events that require a projector must be scheduled in a room with a projector. The third states that only faculty members can reserve room g100.

The spreadsheet also has formulae, which state that each of the cells can contain at most one value (Figure 7). These formulae are generated automatically by the system.



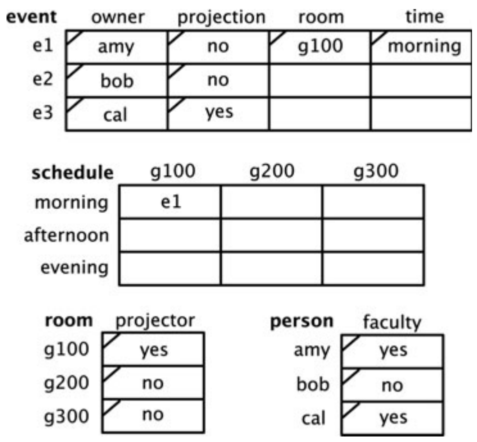
event	owner	projection	room	time
e1	amy	no		
e2	bob	no		
e3	cal	yes		

schedule	g100	g200	g300
morning			
afternoon			
evening			

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 8 After creating three events



event	owner	projection	room	time
e1	amy	no	g100	morning
e2	bob	no		
e3	cal	yes		

schedule	g100	g200	g300
morning	e1		
afternoon			
evening			

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 9 After scheduling e1

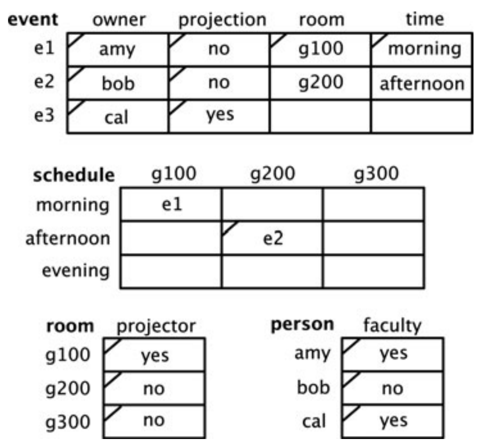
We consider an administrator whose task is to assign three new events, a room, and a time. The administrator starts with an empty schedule table and event table. He/She creates three new events in the event table and, for each, fills in the event owner's name and whether a projector is needed (Figure 8).

The administrator then selects g100 as the room for event e1 and morning as the time for e1 in the event table, causing e1 to show up in the corresponding cell in the schedule table (Figure 9). The administrator then assigns e2 to g200 in the afternoon by modifying the schedule table directly, causing the corresponding values to appear in the event table. This leads to the state shown in Figure 10. This illustrates our system's ability to do propagation in multiple directions.

Next, the administrator places e1 to g100 in the evening by modifying the schedule table, resulting in e1's time being changed to the evening in the event table and being removed from the morning slot in the schedule table (Figure 11). This illustrates how the update algorithm deals with direct conflicts.

The administrator then sets the room assignment for e3 to g200. Since e3 requires a projector but g200 lacks a projector, this leads to a conflict. As shown in Figure 12, our system marks the conflicting cells in red. This shows how our system deals with conflicts caused by multiple cells.

The administrator does not have to resolve the conflict immediately. He/She instead proceeds to change e2's room to g100 (Figure 13). This leads to yet another conflict, since Bob is not a faculty member and only faculty members can reserve g100. At this point, there are six cells colored red. So that the user can distinguish between the different sources of conflict, the values involved in the conflict are highlighted when the mouse pointer is placed over a conflicted value (Figure 13).



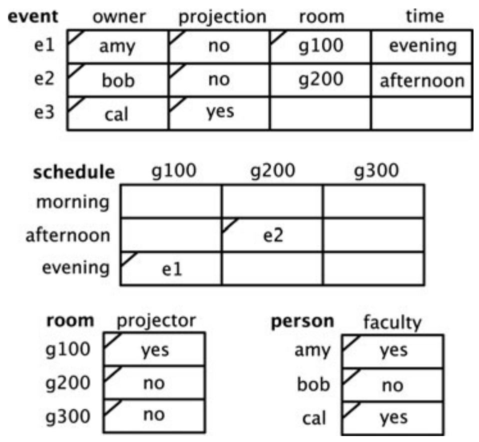
event	owner	projection	room	time
e1	amy	no	g100	morning
e2	bob	no	g200	afternoon
e3	cal	yes		

schedule	g100	g200	g300
morning	e1		
afternoon		e2	
evening			

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 10 After scheduling e2



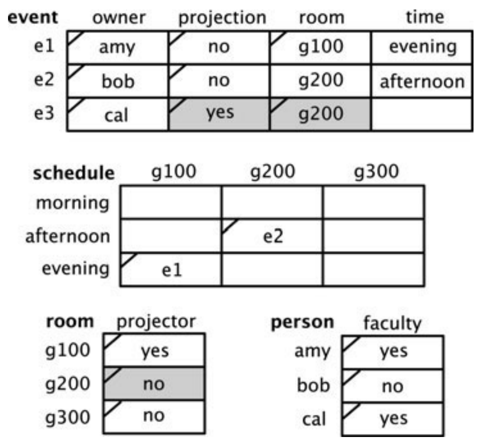
event	owner	projection	room	time
e1	amy	no	g100	evening
e2	bob	no	g200	afternoon
e3	cal	yes		

schedule	g100	g200	g300
morning			
afternoon		e2	
evening	e1		

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 11 After moving e1 to the evening



event	owner	projection	room	time
e1	amy	no	g100	evening
e2	bob	no	g200	afternoon
e3	cal	yes	g200	

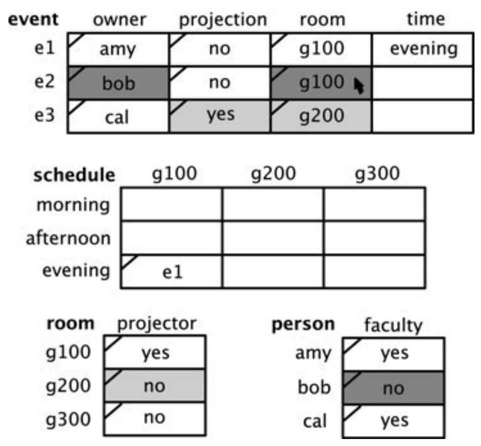
schedule	g100	g200	g300
morning			
afternoon		e2	
evening	e1		

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 12 Spreadsheet with inconsistency

The administrator next sets the time of e3 to the morning. The event e3 then appears in the schedule table, as shown in Figure 14. This demonstrates our system's use of existential Ω-entailment to show the consequences of the (inconsistent) base assignments. Finally, the administrator moves the projector from g100 into g200 (by setting the g100 projector cell to yes and the g100 projector cell to no) and places e2 in g200 in the afternoon, removing the conflicts and resulting in a complete assignment to all events (Figure 15).



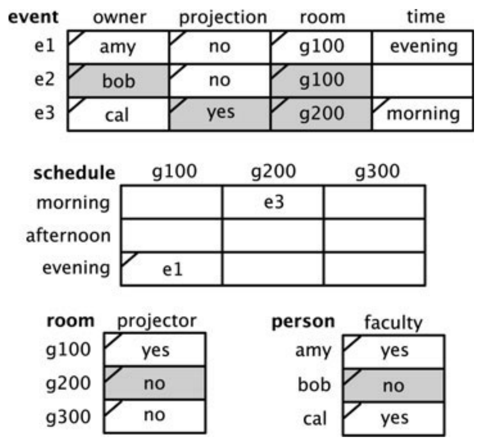
event	owner	projection	room	time
e1	amy	no	g100	evening
e2	bob	no	g100	
e3	cal	yes	g200	

schedule	g100	g200	g300
morning			
afternoon			
evening	e1		

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 13 Distinguishing between conflicts



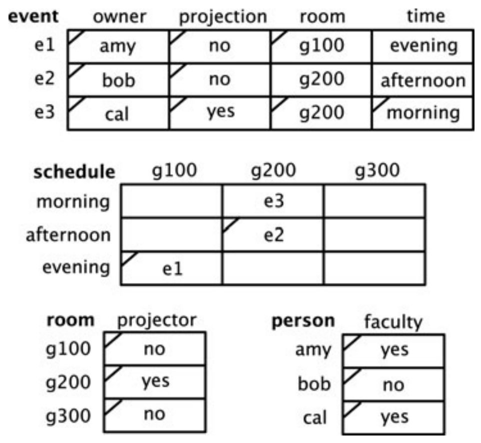
event	owner	projection	room	time
e1	amy	no	g100	evening
e2	bob	no	g100	
e3	cal	yes	g200	morning

schedule	g100	g200	g300
morning		e3	
afternoon			
evening	e1		

room	projector
g100	yes
g200	no
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 14 Showing consequences under inconsistency



event	owner	projection	room	time
e1	amy	no	g100	evening
e2	bob	no	g200	afternoon
e3	cal	yes	g200	morning

schedule	g100	g200	g300
morning		e3	
afternoon		e2	
evening	e1		

room	projector
g100	no
g200	yes
g300	no

person	faculty
amy	yes
bob	no
cal	yes

Figure 15 A completed spreadsheet

7 Update rules

The update semantics described in the last two sections are not appropriate for all situations. For example, there are situations in which one might prefer to retain a value in a cell even if its negation is entailed by the latest value assignment and the constraints. This would occur if one does not wish to give preference to the most recent value assignment. For example, say that we have a spreadsheet that contains user data obtained from two different sources: cell c_1 holds a customer's business address, obtained from source #1, and say that cell c_2 also holds that customer's business address, obtained from source #2. If the data were correct, the two cells should hold the same value, and therefore we assume that the spreadsheet has a formula $c_1(x)\langle=\rangle c_2(x)$ stating this relationship. However, in the case that the two addresses are not the same, we do not want one to overwrite the other: we wish to let the spreadsheet user decide which is correct, and make the change himself/herself.

In this case, PrediCalc's update semantics is too aggressive. In other cases, PrediCalc may not be aggressive enough. For example, consider the room management example in the previous section. If the administrator were to place event e1 in g100 in the morning in the event table, and were to subsequently place event e2 in g100 in the morning in the event table, the resulting situation would be a conflict between these four cells. However, we might want to specify that the most recent assignment wins, and so e1 would be removed from both g100 and from the morning, leaving the spreadsheet conflict free.

To accommodate such departures from the standard update practice, we can allow the administrator to instead specify logical update rules that specify the desired behavior. Our update rules use the following modalities:

- $c(a)$ means that before the update request, cell c has value a.
- $pos(c(a))$ means that the user has requested that cell c have value a.
- $neg(c(a))$ means that the user has requested that cell c no longer have value a.
- $plus(c(a))$ means that after the update, cell c will have value a
- $minus(c(a))$ means that after the update, cell c will not have value a

An update rule is of the form $plus(t) \Leftarrow body$ or $minus(t) \Leftarrow body$, where t is a term and $body$ is a conjunction of literals composed of atoms that may contain the *pos* or *neg* modalities. For example, the update rule $plus(p(x)) \Leftarrow pos(q(x)) \wedge s(a)$ says that when the user requests that cell q have some value v, and cell s currently has value a, then cell p will also be assigned value v.

Let Γ be a set of update rules, Λ be our current set of value assignments, and μ be an update request (a fact of the form $pos(v)$ or $neg(v$. We can define the resulting set of value assignments, Λ', as follows:

$$\Lambda' = \Lambda \cup \phi | \{\mu\} \cup \Lambda \cup \Gamma \models plus(\phi)$$
$$- \psi\{\mu\} \cup \Lambda \cup \Gamma \models minus(\psi)$$

We have created a prototype system, analogous to PrediCalc, which uses update rules to define the spreadsheet's update behavior instead of the update semantics described in the previous sections.

We are currently investigating how to best combine the two approaches. For example, one can generate update rules that specify the default update semantics by differentiating the spreadsheet formulas (Orman, 1998) and allow the user to subsequently modify these update rules. One question that needs to be answered is what should happen when the formulas are changed—how then should the update rules be automatically modified to reflect these changes?

8 Related work

This paper is an expanded version of Kassoff *et al.* (2005).

There have been several other systems that allow for multidirectional, many-to-many constraints, including LogiCalc (Kriwaczek, 1988), FINANZ (Fischer & Rathke, 1988), PERPLEX (Spenke & Beilken, 1989), Knowledgesheet (Gupta & Akhter, 2000), and CsSolver (Felfernig *et al.*, 2003). None of these systems allow for propagation under inconsistency or use structured names for cells. For a survey of these systems, see Kassoff & Valente (2006).

In addition to these systems, PrediCalc has a relationship to both relational databases and constraint satisfaction problems. PrediCalc can be thought of as a logical database with constraints, with a user interface that gives immediate feedback. Of course, a traditional spreadsheet can also be thought of in this way. The line between what constitutes a database and what constitutes a spreadsheet is not a sharp one. In Kassoff *et al.* (2005), we defined a spreadsheet as a single-valued, unary database. However, we have since generalized our notion of a spreadsheet system, allowing for multivalued, *n*-ary cells, and still claim that it is a spreadsheet. We have thus modified our definition of spreadsheet to encompass all systems in which there are *cells* related by *formulae*, where the system *propagates* the consequences of the *cell values*, displaying the current-value assignments in some kind of *user interface*.

PrediCalc can also be thought of as an interactive constraint satisfaction problem (Pu & Faltings, 2002), whereas a traditional CSP is solved completely automatically; in an interactive CSP, the user and system cooperate to produce a solution. PrediCalc does not currently have automatic constraint-solving functionality, though such a feature could be added straightforwardly.

PrediCalc's primary contribution is its update semantics, and its handling of inconsistency. In particular, none of the above systems handle updates that are inconsistent with the current spreadsheet assignments. Its notion of consequence, Existential Ω-entailment, is a new paraconsistent entailment relation, developed to meet the needs of spreadsheet users.

9 Limitations of Predicalc

Our research prototype has proven the feasibility of our approach on spreadsheets containing thousands of cells and hundreds of constraints. In fact, the limiting factor is not the underlying algorithms, but is instead the GUI, which is built in a naive way and does not scale well. One can, of course, devise examples where a small number of interconnected constraints will result in an unacceptably slow response time for the spreadsheet; however, simple examples with little interconnection allow for large numbers of constraints to be handled quickly. Our prototype uses a naive implementation of Existential Ω-entailment; however, we have found significantly faster algorithms that scale with the amount of inconsistency; details will be provided in a forthcoming paper. An open question is how to incrementally maintain materialized Existential Ω-entailment consequences as the cell values are updated. Also, our experiments indicate that unary relations do not scale particularly well, and so we will extend our model to allow for *n*-ary cells as well as unary ones in future versions of our spreadsheet.

While our user-interface works well for spreadsheets that fit on a single screen, more innovation is required to properly support larger spreadsheets. For example, our highlighting of conflicting values via a mouseover (illustrated in Figure 13) works only if all conflicting values are visible at once. Furthermore, extending our prototype to allow for multiple values in one cell will require more complex user feedback mechanisms—e.g., the spreadsheet should be able to highlight an individual value within a cell, and should be able to indicate which values in a cell are computed and which were directly specified by the user.

10 Websheets

As mentioned in Section (2), spreadsheets have several applications to the World Wide Web. We refer to a logical spreadsheet with a Web interface (normally HTML) as a *Websheet*. In this case, the form fields on the Web page serve as the cells of the spreadsheet.

The propagation of values in a Websheet may be done through a combination of client-side scripting and server-side processing. There are several trade-offs between client-side and server-side processing.

A Websheet that runs completely within the browser is ideal for the situations in which the user is disconnected from the Internet. In addition, a user can treat such a Websheet like any other electronic document—he/she can save multiple versions of such a Websheet to his hard drive, archive backup copies of the Websheet, send the Websheet through email, etc. In this sense, a client-side Websheet is quite similar to Hilliger von Thile & Melzer's (2005) *smart files*.

In the interest of exploring the feasibility of building a completely client-side Websheet, we created a model-elimination JavaScript Theorem Prover to do the processing. However, our experiments showed that such an approach would be unacceptably slow for all but the smallest of Websheets, due to the interpreted nature of JavaScript in modern browsers.

We have created two prototype Websheets that do have acceptable performance. In the first, the system compiles the formulas into JavaScript on the server-side when the Websheet is created. The result is a completely standalone client-side Websheet. In the second, the update computation takes place largely on the server side, using AJAX to update the Web page each time an update is made. This has the advantage of allowing the update computation to rely upon large amounts of data—megabytes or larger—which might be changing.

We are currently investigating how to integrate Websheet capabilities into Infomaster (Genesereth *et al.*, 1997). Infomaster is a semantic database management system with a Web interface. A Websheet interface would serve as a correct-on-capture tool for Infomaster, ensuring that data entered are semantically sound.

Of course, the power behind Web pages is that they can link to one another. Spreadsheets can also support collaborative applications if they are linked, with automatic propagation of values and constraints among the connected spreadsheets. Linked spreadsheets of this sort would support a wide variety of applications in cooperative design and collaborative management.

For example, consider a group of engineers collaboratively designing an airplane. As wires are added and windows are placed, the consequences of those decisions are propagated to the spreadsheets of all the engineers working on the design. Or consider a set of professors and administrators collaboratively designing the course schedule for the next year.

A publicly available set of Websheets could form a World Wide Websheet. These spreadsheets could reference arbitrary conditions on the Web, and be referenced by arbitrary other Websheets and applications.

Indeed, the World Wide Web is rapidly becoming integrated into desktop applications. More and more applications are depending on the Internet for some purpose—to deliver software updates, to download song titles and album covers, to send emails, to browse documentation, to share files, and so on. At the same time, the Semantic Web is slowly becoming a reality. RDF is beginning to catch on as a standard way of encoding semantic relationships. Although it is not clear precisely when this will happen—5 yr? 10 yr?—in the future there will be a plethora of semantic data on the Web.

A logical spreadsheet would make an excellent 'data browser' for the Semantic Web. In particular, a logical spreadsheet could be used to integrate data from different sources and then translate data from one schema to another. Furthermore, a logical spreadsheet could be used to analyze this data, combining it with the user's personal data stored on his/her local machine.

Consider, for example, a user who is trying to determine which digital camera to buy. The user obtains product information from a few manufacturer Websites, collates it with opinions and ratings found at a camera review Website. He/She then queries the data ('give me all compact 5-megapixel cameras that got a dependability rating of at least three stars'), and then adds to this data his/her personal thoughts.

The possibilities of logical spreadsheets on the Web are great, and we hope that the future work of ourselves and others will make these possibilities a reality.

11 Conclusion

In this paper, we have presented a foundation for future work in logical spreadsheets. Much work remains to be done. As indicated in the previous section, there are many directions in which this work could be taken. One could design a traditional-looking spreadsheet with these extended capabilities, or create a tool to build Websheets, or build a collaborative multi-user Websheet, or create a logical spreadsheet that acts as a data browser. It is also clear that different applications—for example, resource scheduling, could justify building specialized logical spreadsheet systems to suit their needs, with built in predicates and relations and a user interface specialized for that task.

Spreadsheets are known for their ease-of-use and their accessibility to non-programmers. Logical spreadsheets should bring additional power into the hands of the general public. They should allow people to solve problems in a short amount of time that currently take hours or longer.

Acknowledgments

Lee-Ming Zen was instrumental in building the JavaScript Theorem Prover. We thank him for this, and also for helping to design the first user interface for our PrediCalc system, together with Ankit Garg (Kassoff *et al.*, 2005). We thank the attendees of WOLS'05 and the anonymous reviewers for their valuable feedback.

References

Chen, W., Kifer, M. and Warren, D. 1993 HILOG: a foundation for higher-order logic programming. *Journal of Logic Programming,* **15**(3), 187–230.

Elvang-Gøransson, M. and Hunter, A. 1995 Argumentative logics: reasoning with classically inconsistent information. *Data and Knowledge Engineering,* **16**(2), 125–145.

Enderton, H. 2000 *A Mathematical Introduction to Logic*, 2nd edn. Academic Press, New York 2000.

Felfernig, A., Friedrich, G., Jannach, D., Russ, C. and Zanker, M. 2003 *Developing Constraint-Based Applications with Spreadsheets*. IEA/AIE 2003, pp. 197–207.

Fischer, G. and Rathke, C. 1988 *Knowledge-Based Spreadsheets*. AAAI 1988, pp. 802–807.

Genesereth, M. R. 1995 *Epilog for Lisp 2.0 Manual*. Palo Alto, CA: Epistemics Inc.

Genesereth, M. R., Keller, A., & Duschka, O. 1997 *Infomaster: An Information Integration System*. SIGMOD1997, pp. 539–542.

Gupta, G. and Akhter, S. 2000 *Knowledgesheet: A Graphical Spreadsheet Interface for Interactively Developing a Class of Constraint Programs*. PADL 2000, pp. 308–323.

Hilliger von Thile, A. and Melzer, M. 2005 *Smart Files: Combining the Advantages of DBMS and WfMS with the Simplicity and Flexibility of Spreadsheets*. BTW 2005, pp. 175–184.

Kassoff, M. and Valente, A. 2007 An Introduction to Logical Spreadsheets. *The Knowledge Engineering Review*, **22**, 213–219.

Kassoff, M., Zen, L., Garg, A., & Genesereth, M. R. 2005 *PrediCalc: A Logical Spreadsheet Management System*. VLDB 2005, pp. 1247–1250.

Kriwaczek, F. 1988 LogiCalc: a prolog spreadsheet. *Machine Intelligence*, **11**, 193–208.

Love, N. and Genesereth, M. R. 2005 *Computational Law*. ICAIL 2005, pp. 205–209.

Orman, L. V. 1998 *Differential Relational Calculus for Integrity Maintenance*. IEEE Trans. Knowl. Data Eng. **10**(2): 328–341.

Pu, P. and Faltings, B. 2002 Effective interaction principles for user-involved constraint problem solving. In *Second International Workshop on User-Interaction in Constraint Satisfaction*, CP 2002, pp. 77–91.

Russell, S. and Norvig, P. 2003 *Artificial Intelligence: A Modern Approach*. 2nd edn. Englewood Cliffs, NJ: Prentice-Hall.

Spenke, M. and Beilken, C. 1989 *A Spreadsheet Interface for Logic Programming*. CHI 1989, pp. 75–80.