XcelLog: A Deductive Spreadsheet System

C.R. Ramakrishnan, I.V. Ramakrishnan, and David S. Warren

Department of Computer Science, Stony Brook University
Stony Brook, NY 11794-4400
{cram,ram,warren}@cs.sunysb.edu

Abstract. The promise of rule-based computing was to allow end users to create, modify, and maintain applications without the need to engage programmers. But experience has shown that rule sets often interact in subtle ways, making them difficult to understand and reason about. This has impeded the wide-spread adoption of rule-based computing. This paper describes the design and implementation of XcelLog, a user-centered *deductive spreadsheet system*, to empower non-programmers to specify and manipulate rule-based systems. The driving idea underlying the system is to treat sets as the fundamental data type and rules as specifying relationships among sets, and use the spreadsheet metaphor to create and view the materialized sets. The fundamental feature that makes XcelLog suitable for non-programmers is that the user mainly sees the *effect* of the rules; when rules or basic facts change, the user sees the impact of the change immediately. This enables the user to gain confidence in the rules and their modification, and also experiment with what-if scenarios without any programming. Preliminary experience with using XcelLog indicates that it is indeed feasible to put the power of deductive spreadsheets for doing rule-based computing into the hands of end users and do so without the requirement of programming or the constraints of canned application packages.

1 Introduction

The defining problem: Rule-based specifications are used in a wide variety of applications. Examples include business rules (e.g. [20]), authorization rules for scalable access control and distributed trust management (e.g. [17, 11]), and configuration management of complex systems (e.g. system administration [2], security policy configuration [15], and vulnerability analysis [19, 18]). Also automated support for decision making is by and large based on rule-based systems. However, a major factor that hampers their large-scale adoption is the difficulty of developing, understanding and modifying the (rule-based) specifications. In general, it is not easy to infer the effect of a rule from the way in which it is written. Rule systems need a "programmer" to specify the rules and additional tools to analyze the rules in order to convince the users of their soundness and completeness; a case in point is the SELinux security policies [14] and the variety of tools that have been developed to analyze these policies [7, 6, 24]. This raises the question: how do we empower end-users to develop rules-driven application without having to know programming?

Deductive Spreadsheets: The electronic spreadsheet, as exemplified by Excel®, is a spectacularly popular application program that is widely used by the masses. Every spreadsheet user effectively creates a program to process data without having to be trained as a programmer. The large-scale adoption of spreadsheets as a programming tool by the masses (albeit for particular classes of problems) is mainly because computations are specified by examples. A user specifies an instance of a computation (e.g. sum of two cells in a row); subsequently by copying and filling, the user specifies that the other cells (the destination of the filling gesture) are computed in a "similar" manner. This allows the user to not have to think about abstractions and general parameterized operations, but instead concentrate on multiple concrete operations. Moreover, the spreadsheet user interface shows the results of computation directly and changes the results whenever the underlying data changes. This direct interaction with data eliminates the line between code development and testing.

Analogous to traditional numerical spreadsheets the idea of *deductive spreadsheets* (DSS) is to bring the power of rules-driven computing within the familiar paradigm of spreadsheets — specifically empower end users (who are not programmers) to write and maintain rules, not in an ad hoc language, but in terms of the effect of the rules on an underlying sample of data using the classic 2-D graphical spreadsheet metaphor.

An Example: We illustrate the idea of DSS using a simple example from Trust Management (following [12]).

1. A publisher, PUB, wants to give a discount to their member, which is anyone who is both a student and a preferred customer.
2. PUB delegates the authority over the identification of preferred customers to its parent organization ORG.
3. ORG has a policy of treating IEEE members as preferred customers.
4. PUB also delegates the authority over the identification of students to accredited universities.
5. The identification of accredited universities, in turn, is based on credentials issued by the University Accreditation Board, UAB.

These rules, which form a deductive system, have been traditionally written in a special syntax specific to the trust management system; the meaning of the rules is usually given in terms of the set of all logical inferences that can be drawn from these rules. Using a DSS, the same rules can be specified and their impact can be more directly visualized as in Figure 1.

Following traditional spreadsheets, a DSS is a two dimensional array of cells. However, columns and rows in a DSS are labeled by symbolic values. For instance, rows in the DSS shown in Figure 1, labeled PUB, ORG, ..., represent entities referenced in the example. Columns correspond to properties of these entities. The value in a cell at row r and column c (denoted by $r.c$, or in a functional notation cr) represents the value of property c of entity r. For instance, the cell "IEEE.member" represents the set of IEEE members; and the cell "UAB.member" represents the set of universities accredited by UAB. To manipulate multiple interrelated DSSs, we use the notation $s!r.c$ to denote the cell $r.c$ in sheet s.

	member	preferred	student	univ
PUB	*PUB.preferred && PUB.student* /* Rule 1 */ {Amy}	*ORG.preferred* /* Rule 2 */ {Amy, Joe}	*PUB.univ.student* /* Rule 4 */ {Amy, Bob}	*UAB.member* /* Rule 5 */ {ESU, USB}
ORG		*IEEE.member* /* Rule 3 */ {Amy, Joe}		
IEEE	{Amy, Joe}			
UAB	{ESU, USB}			
ESU			{Amy}	
USB			{Bob}	

Fig. 1. Deductive Spreadsheet for Discount Eligibility

Note that, unlike in a traditional spreadsheet (and indeed in other logical spreadsheets, e.g. [5, 10]), each DSS cell contains a *set* of values. Cell references correspond to expressions that evaluate to a set. In the figure, expressions (called *intensions*) are shown in *italics* and their values (called *extensions*) are shown in teletype. In the figure the cell expressions and comments (enclosed between "/*" and "*/") are shown for illustration only. Following traditional spreadsheets, the user specifies only the intensions; the DSS system computes the extensions, shows only the extensions in the cells, and recomputes them whenever cell values change.

Now consider the encoding of Rule 3 of the example above, which states that every IEEE member is a preferred customer of ORG. This is specified in DSS using a *cell reference*: the cell *ORG.preferred* contains a reference to another cell *IEEE.member*, indicating that whatever occurs in *IEEE.member* must also occur in *ORG.preferred*. This is analogous to the idea in traditional spreadsheets of referring in one cell to the numeric value in another cell. Rules 2 and 5 can be similarly encoded. Rule 4 states that PUB delegates the identification of students to recognized universities. Note that *PUB.univ* contains the set of all universities recognized by PUB and hence $u.student \subseteq PUB.student$ whenever $u \in PUB.univ$. This (rather complex) rule can be specified by "lifting" the dot notation to sets: for example, $a.b.c$ represents $\bigcup y.c$ for every y in $a.b$. In the example, the cell *PUB.student* contains the expression *PUB.univ.student*. Finally, Rule 1 states that *Pub.member* consists of entities that are in both *PUB.preferred* and *PUB.student*.

These two ideas: (1) allowing cells to contain multiple values and (2) permitting cell references that make the statement that a cell must contain all the elements of another cell, bring the power of deduction into a simple spreadsheet framework. Thus they provide the foundation for our vision of DSS.

As a natural consequence of set-valued cells, DSS permits a cell $a.b$ to contain multiple cell references: the meaning of such an expression is that the value of $a.b$ is a set that contains the union of all the values of the referred cells. Moreover, the cell references may be recursive in general. The meaning of recursive references is given in terms of least fixed points [13]. Set-valued cells and recursive definitions provide a

powerful platform for encoding complex problems involving deduction. Nevertheless, from an end-user's perspective, these are relatively simple extensions to the traditional spreadsheet paradigm, thereby adding the power of deduction without compromising the simplicity of defining and using spreadsheets. The interesting problem now is to realize a functional DSS system based on the above ideas.

The rest of this paper describes our technical approach to the design and implementation of the DSS system envisioned above. The starting point is the design of the DSS expression language and intuitive gestures for specifying contents of cells and relationships between them (see Section 2). In general DSS expressions can involve circular references as is typical when computing with logical relations. We give least-model semantics [13] to DSS expressions by translating them into Datalog programs, a subclass of logic programs, which is used in deductive databases [3]. In Section 3 we describe the implementation of XcelLog, our prototype DSS system, with Excel as it's front-end and our XSB tabled logic programming system as the backend deduction machine [22]. We have encoded problems drawn from a number of application domains including logistics, combinatorial optimization, and network security in XcelLog. We illustrate the encoding of one such problem in Section 4. There have been a large number of proposals to combine logic with the spreadsheet metaphor. Ours differs in a fundamental way from all others that we know by supporting set-valued cells and meaningful recursive definitions. In Section 5 we describe related work in more detail. Discussion appears in Section 6.

2   The Deductive Spreadsheet Language

The primary design criterion for the DSS language was simplicity: the users should be able to construct and manipulate deductive spreadsheets with gestures, operators and expressions that are easy to learn and intuitive to use. Abstraction is one of the fundamental aspects of programming, and also one of the most difficult aspects to learn and master. User-level programming in spreadsheets cleverly circumvent this problem by letting the user program by example (e.g. specifying an expression in a specific cell) and then generalize the program (e.g. by filling cells with expression from another cell). Thus users never deal directly with the notion of variables. We have followed the same philosophy by designing an expression language without variables. The following is a brief summary of the salient aspects of the language.

A deductive spreadsheet contains a grid of cells, each of which contains a set of elements. A spreadsheet may also refer to an external database table. Thus tables, spreadsheets, cells, and elements are the four classes of entities that will be defined and manipulated by our language. We classify the operators based on the entities they produce, as follows:

1. *Element operators:* Elements can be atomic values (strings, integers, etc.) or formed using *tuple construction*, *tuple projection*, *arithmetic*, *aggregation* (such as SUM, MIN, MAX, etc.) and conditional operators. The tuple construction and projection operations offer a way to create and access data structures.
2. *Cell operators:* Cell expressions evaluate to sets of elements. The contents of a cell may be specified by explicitly listing a set of elements, and/or by expressions

constructed using *cell reference*, *selection*, *difference* and *lifted operators* that lift
tuple construction, tuple projection, aggregation and conditionals to sets.

3. *Sheet operators* to construct a sheet from other sheets or from database tables.
4. *Abstraction operators:* "*Ctrl-Shift-C*" and "*Ctrl-Shift-V*" are the copy and paste
   gestures respectively in DSS that permit the user to first specify a computation on
   concrete instances, then copy the specification and "fill" other cells, which causes
   similar specifications to be generated for the destination cells. In particular a user
   in DSS can bulk copy a subset of cells and paste it into a target cell.

These operators will be illustrated in the encoding exercises (in Section 4). As is the
case with traditional spreadsheets DSS users also type in simple expressions either in
the cells directly or in the function box f_x. More complex expressions get created by
gestures such as copy, paste and fill.

Semantics: The semantics of a DSS expression is given by translation to Datalog pro-
grams, i.e., Prolog programs without function symbols [16]. A Prolog program consists
of rules of the form *head :- body* where *head* is a literal and *body* is a conjunct of lit-
erals. The *head* is true whenever the *body* is true. A *head* with an empty *body* is a fact
that is unconditionally true.

A set of spreadsheets defines a 4-ary relation:*sheet(Name,Row,Column,Contents)*,
where *sheet(Sht, Ro, Co, Ent)* is true iff *Ent* is in the cell at the intersection of the row
Ro and column *Co* in sheet *Sht*. For example, the upper left cell in the DSS table named
say 'discount' in Figure 1, is defined by the Prolog rule:

```
sheet(discount,'PUB',discount,X) :-
    sheet(discount,'PUB',preferred,X),
    sheet(discount,'PUB',student,X).
```

The meaning of the spreadsheet is the least fixed point of the Datalog program
defined in this way.

3   The XcelLog DSS System

A deductive engine becomes a core computational infrastructure component for imple-
menting a DSS system that is predicated on translating DSS expressions into Datalog.
A key requirement for such an engine is that it completely and efficiently evaluate Dat-
alog programs. Our XSB Tabled Logic Programming system is well suited for this
purpose [22]. It is a high-performance deductive rule-based engine that uses tabling to
implement a more complete version of resolution-based query answering. In contrast a
standard Prolog system would loop infinitely when given cyclic definitions, which can
easily arise.

We implemented XcelLog – a prototype DSS system with an Excel front end and
XSB with its tabling machinery as the backend deductive engine to correctly and finitely
compute the DSS semantics. Below we provide an overview of the XcelLog system. It
was engineered as an Excel "add-in", i.e. the implementation of deductive spreadsheets

was encapsulated within the Excel environment; Excel served as the frontend while the DSS expressions were evaluated by XSB in the backend. This way XcelLog users would continue getting the benefits of traditional Excel along with the added power of deduction.

Cells in XcelLog are of two types – traditional Excel cells and DSS (deductive spreadsheet) cells. Deduction expressions are specified only within DSS cells. All DSS expressions are enclosed within [ ]. E.g. the DSS expression, using functional notation for a cell reference, corresponding to the (intensional) Rule 2 is: [preferred ORG] while the (extensional) set of values computed by this expression in cell at row *PUB* and column *preferred* is the DSS expression: [Amy,Joe]. DSS cell contents of the form [...] are automatically translated for Excel as =DSS("..."). So a DSS expression in Excel's function box f_x is enclosed within "=DSS( )". These correspond to the intensional view of rules associated with DSS cells. The cells themselves display materialized views of the effect of the rules, just as in regular Excel.
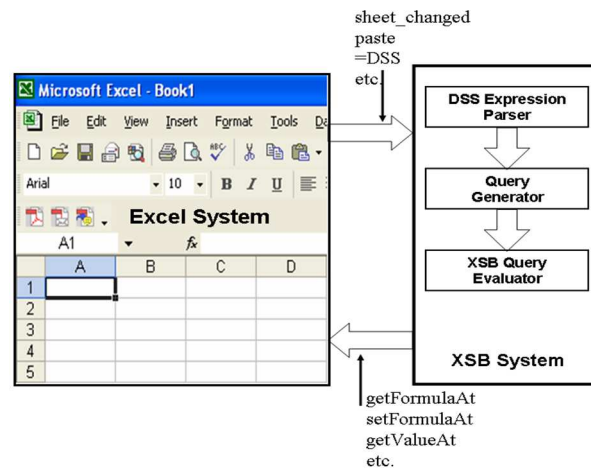
Fig. 2. The XcelLog DSS System

The architectural schematic of our XcelLog prototype for evaluating DSS expressions is shown in Figure 2. Notice that Excel and the XSB Tabled LP system are the two main components making up the XcelLog Deductive Spreadsheet System. In this architecture users see only the Excel front end. They interact with the system via Excel's familiar interface. Excel cells are used in the traditional style. Cells with expressions within "[ ]" are treated as DSS cells, and the expressions are evaluated by the XSB system. Note the flexibility afforded by this system combining traditional Excel-style computing (embodied within Excel cells) intermixed with deduction specified in DSS cells.

Evaluation of a DSS program in XcelLog requires bi-directional communication between Excel and XSB. This is facilitated via the XLL-Add-in component. The XLL-Add-in component is set up to recognize certain events such as when a cell's content
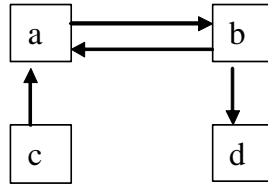
is changed, when DSS expressions are entered into a cell, etc. Whenever such an event occurs, control is passed to XSB via this component, using sheet changed, paste, =DSS operations in Figure 2 (among others.) XSB does the needed processing, perhaps using call-back functions that Excel provides (e.g. getFormulaAt, setFormulaAt, getValueAt operations in the figure), and then returns control back to Excel. In the basic operation of materializing a DSS cell, XSB parses the cell expression, translates it to a Prolog goal, and then simply uses `call/1` to invoke the XSB query evaluator to evaluate the goal and produce the extensional values. These are the materialized sets that get displayed in the cells. For example the materialized set corresponding to Rule 1 in Figure 1 that is computed by XcelLog is Amy as shown in row `PUB` and column `member` in Figure 1. Note that in our XcelLog implementation we use the functional notation for cell references so as to be consistent with Excel. So for example "`PUB.preferred`" in functional notation is "`preferred PUB`".

4 Encoding Exercises in XcelLog

We have encoded a number of problems, drawn from varied areas such as logistics, combinatorial optimization and network security, in XcelLog. Here we illustrate two such problems to demonstrate its deductive power and versatility. The first example deals with directed graphs, and determines, for each node in the graph, the set of all nodes reachable from it. This example shows the need for recursive definitions, and the naturalness of the least fixed point semantics. It also illustrates the example-based mechanism for defining new relations. The second example is a more complex one, of finding optimal purchase strategy in a supply chain. This example illustrates features of XcelLog that were described but not illustrated before: (a) the use of tuple values in cells, (b) aggregation operations over cell values, and (c) abstraction. Moreover, this example also shows the power of XcelLog to encode complex problems of this nature with few relatively simple gestures.

(1) The reachability problem in graphs: The problem here is to compute the set of reachable nodes from every node in the graph in Figure 3(a). This is the canonical transitive closure example that is used for illustrating deduction through recursive rules.

Figure 3(b) depicts a fragment of the encoding in XcelLog. The rows represent the 4 nodes in the graph. The edge column for a row contains the set of nodes directly reachable from that node through some edge. The f_x box associated with a cell show the DSS definitions of the cell contents and the cell itself shows its computed contents of these definitions. In Figure 3(b), the DSS expression =DSS("a,d") in the f_x box is associated with the highlighted cell in row b and column edge. This expression indicates that nodes a and d are targets of edges from node b. The f_x box in Figure 3(c) is the DSS expression associated with the reach cell in row b. The =DSS("edge b, edge reach b") cell expression indicates that there are two ways to get an entry in this highlighted cell: "edge b" indicates that every entry in the cell at column edge and row b must be in this cell; "edge reach b" indicates that we take each entry in the cell at column reach and row b (a node reachable from b), and using that value as the row indicator in column edge, we add the entries in *that* cell to the current cell (i.e. those reachable by taking

(a) Directed graph

	A	B	C
B3		fx =DSS("a,d")	
1		edge	reach
2	a	b	[a, b, d]
3	b	[a, d]	[a, b, d]
4	c	a	[a, b, d]
5	d		[]

(b)

	A	B	C	D	E
C3			fx =DSS("edge b, edge reach b")		
1		edge	reach		
2	a	b	[a, b, d]		
3	b	[a, d]	[a, b, d]		
4	c	a	[a, b, d]		
5	d	a	[a, b, d]		

(c)

	A	B	C	D
C3			fx =DSS("(edge b), (edge (reach b))")	
1		edge	reach	
2	a	b	[a, b, c, d]	
3	b	[c, d]	[a, b, c, d]	
4	c	a	[a, b, c, d]	
5	d		[]	

(d)

Fig. 3. Reachability problem

one more edge). This is an example of a cyclic specification: the reach column of row b contains a cell expression that refers to itself.

The user sees the effect of the rules (which are the materialized sets) rather than the rule itself. In addition when the rules or base facts change the user can immediately

see their effect. For example, if we remove the edge from b to a in Figure 3(a) and add the edge from b to c instead then XcelLog immediately recomputes the new set of reachable nodes (see Figure 3(d)). Thus it can provide immediate feedback to "what-if" scenarios.

It is noteworthy pointing out how the user creates this DSS. First he types in all the entries into the edge column. These are the direct edges between nodes (see the edge column in Figure 3(b)). Next he creates the reach expression for a cell say in row b and column reach (see the f_x box in Figure 3(c)). Then he copies this expression (using DSS copy) and fills (using the DSS paste operation) all the cells in the reach column with the copied expression. The system automatically inserts the appropriate row number into the expression for the corresponding reach cell in that row. This is similar in terms of user experience with traditional spreadsheets. However the idea of allowing set valued cell expressions and cyclic references as illustrated in this example, has enabled the user to perform more complex computations.

The DSS expression in the reach column in Figure 3(c) gets translated to the following left-recursive Datalog rules:

```
graph(b,'reach',X) :- graph(b,'edge',X).
graph(b,'reach',X) :- graph(b,'reach',Y),
                      graph(Y,'edge',X).
```

The XSB system evaluates the recursive rules and passes the computed result, which in this case is the set a, b, d of reachable nodes, to Excel for display at that cell. Note that traditional Prolog systems will go into an infinite loop on this example. XSB's tabling machinery ensures termination of evaluation in the presence of such recursive definitions

(2) A complex logistics example: the supply chain problem. The supply chain of a manufacturer is a complex network of suppliers, retailers, distributors, transporters and storage facilities that participate in the sale, delivery and production of finished goods. Analysis of the behavior of the supply chain provides important information to the manufacturer for contingency planning, resource optimization, etc. Such an analysis can be facilitated by specifying the rules governing the supply chain parameters and relationships in DSS as illustrated below. A switch assembly consists of two major components: a limit switch and an actuator subassembly; the actuator in turn is made up of a sensor and a transmitter. Each part can either be bought off the shelf or, if a composite part, can be acquired by assembling it from its components. Figure 4(a) shows the DSSs that contain data about (a) the cost of buying a part off the shelf from different vendors (the "supply costs" sheet); (b) the cost of assembling a part (given its components) at different manufacturing sites ("assembly costs" sheet); (c) the cost of shipping a part from one site to another ("ship" sheet); and the composition of a part ("subpart of" sheet).

The purchase policy is to acquire a part from the cheapest source: either off-the-shelf, or by custom assembly. In order to compute the least cost, we create a new sheet (the "acquire" sheet shown in Figure 4(b)) with the different vendors and assemblers as rows, and the parts as columns. The value of a cell *dest.part* in this sheet represents, for a given part and the destination where it should be shipped to, the best price and a

(a) base costs

(b) computing acquisition costs

Fig. 4. Supply chain problem

supplier that provides the given part at that price. That is, it is a pair *(v, p)* where *p* is the price that *dest* has to pay to acquire the part *part* from vendor *v*. We now describe how the values in this sheet are calculated.

Consider the purchase of the switch assembly from *acme*: this part can either be (i) bought from *dunrite* and shipped to *acme*; or (ii) its parts can be bought, shipped to one of the assemblers (*precision* or *allied*), and the final assembly shipped to *acme*. In order to encode this complex decision process, we create an intermediate sheet "_acq_acmeswitch" to represent the different ways of getting the switch assembly to *acme*. The basic idea purpose behind an intermediate sheet is to specify, step by step, a complex expression that defines a single selected cell's value. In the above example, we use the sheet "_acq_acmeswitch" to define the cell *acme.switch_assm* in the acquire sheet.

The rows of "_acq_acmeswitch" represent all the possible sources and the columns correspond to different kinds of costs. The cells in row *gyro* in this sheet represent the following. The cell at column *buy* represents the cost of purchasing the switch assembly off the shelf from *gyro*. The value of this cell is the sum of the price at which *gyro* sells

switch assemblies (i.e. the value at *gyro.switch_assm* cell in the "supply_costs" sheet) and the cost to ship from *gyro* to *acme* (i.e. the value at *gyro.acme* in the "ship" sheet). Note in this case the cell is empty since *gyro* does not sell switch assemblies. For every row **R**, the value of **R**.*buy* is given by the expression *supply_costs!***R**.*switch_assm* + *ship!***R**.*acme*. This "rule" is specified in XcelLog by first specifying an instance for *gyro*, and then filling all the cells in the buy column with the expression in *gyro.buy*.

The cell at column *get_comp* represents the cost of acquiring and sending the components of the switch assembly to *Gyro*. This cell has a more complex definition. We first need to find all the components of the switch assembly. That is given by the value in the cell *switch_assm.part_of* in the "subpart_of" sheet. For each such part **P**, the best way to get that part to *gyro* is in the cell *gyro.***P** in the "acquire" sheet. Hence the expression: "*acquire!gyro.(subpart_of!switch_assm.part_of)*" represents a set of values: each element in the set represents the best way to get some subpart of switch assembly to *gyro.*. Summing over all elements of this set therefore gives the best way to get all the components of the switch assembly. This is done by applying an aggregation operation to reduce a set to a single element (in this case, *sum*).

The cell at *build* adds the cost of assembly at *gyro* to the cost of acquiring the components (it is empty in this row since *gyro* does not assemble switch assemblies). Finally, the cell at *total_cost* represents the sum of all costs of costs for *acme* to get the switch assembly from *gyro*.

Observe that the values in column *total_cost* in "`_acq_acmeswitch`" sheet represent the best ways of obtaining a switch assembly to *Acme* from different vendors. The minimum over all these values is therefore the best way for *Acme* to acquire a switch assembly. This is specified in XcelLog by selecting the column values and "filling" the cell acme.swiTch_assm in the "acquire" sheet (see the arrow denoting this operation in Figure 4(b)). Note that in traditional spreadsheets, the destination of a "fill" operation must be at least as large as the source. In XcelLog, an entire column (more generally, any rectangular area in a sheet) can be used to fill a single cell. The meaning of this operation is to set the destination cell's value to the union of all the contents of the source cells. In this example, we modify this default meaning by selecting an aggregation operation — *minimum* — to apply to this union.

Consider a cell $r.c$ filled with values from a column, say cells $r_1.c', r_2.c', \ldots, r_2.c'$. Instead of generating $[r_1.c', r_2.c', \ldots, r_n.c']$ as the expression for $r.c$, we generate a more abstract expression $[exists(R, R.c')]$ where R is a variable that ranges over rows, i.e. the set $r1, r2, \ldots, rn$. This *abstraction* is useful in two contexts. First of all, when the rows of a sheet change (e.g. when a new row is added), the abstraction remains unchanged but still retains the original intention of the user: that the cell contain all values in column c'. Note that without the abstraction, the user will have to change the expression of $r.c$ to add the new row. Secondly, the abstraction permits us to have non-materialized sheets. For instance, the sheet ''`_acq_acmeswitch`'' is used to compute the values of *acme.switch_assm* in the "acquire" sheet, but there is no such sheet corresponding to the other cells of the "acquire" sheet. Using abstractions with inlining (where a cell reference $r.c$ is replaced by the cell expression at $r.c$) we can compute the values at the other cells of the "acquire" sheet without explicitly creating the corresponding intermediate sheets.

Observe from the above examples that the spreadsheet metaphor was used to create the rules without the user having to specify any expression with variables. The traditional copy and fill gestures are used to abstract "rules" from one instance and to apply them to other instances. In this example, the only cells whose intensions were entered explicitly and not by filling (other than the four base tables) were the four cells in the *gyro* row of the `_acq_acmeswitch` table. The abstractions may introduce variables into DSS expressions but the user never needs to deal with them directly. Complex rule systems can be constructed with relatively simple interactions. We have thus introduced deduction into spreadsheets without compromising on its basic simplicity and ease-of-use from the end user perspective.

A noteworthy feature is that cell references can be recursive (unlike traditional spreadsheets); this enables the user to specify dynamic programming solutions. In this example, the cheapest cost of a part is defined in terms of the cheapest costs of its sub-parts. Indeed shortest path in a cyclic graph (with nonnegative edge weights) can be specified easily in a DSS. The specification is example based and yet at a high level: specifying only how the different cell values are related. Moreover, numeric and symbolic computations are seamlessly combined. Finally, the user sees the effect of the specifications directly and immediately; a change in a component's price, for example, would immediately propagate to all the dependent cells (as in a traditional spreadsheet.) This permits the user to experiment with what-if scenarios: e.g. the impact of a supplier ceasing to sell a particular part.

5   Related Work

Spreadsheets based on Logic: There have been a great many proposals for combining the spreadsheet metaphor with logic. A recent survey is available at `http://www.ainewsletter.com/newsletters/aix_0505.htm`. We will describe in more detail recent proposals that are most similar to ours.

Knowledge-sheet [5] and *PrediCalc* [10] extend traditional spreadsheets by allowing the user to specify constraints on the values of cells. Cells are still required to contain unique values, but those values may be partially (or totally) determined by constraints. In *Knowledgesheet* finite-domain constraints are associated with cells and specify combinatorial problems. On user request, the system converts these constraints into a CLP(FD) program, executes it, and returns the solution as cell values. In *PrediCalc* the constraint-solving engine is more integrated into spreadsheet interaction, and issues addressed include how to handle over-specified (or inconsistent) values and under-specified values. *PrediCalc* is similar to our proposal in that rows and columns of spreadsheets are given names and individual cells are referenced by providing the sheet name, the row name, and the column name. Our approach differs from these in a fundamental way in that these approaches maintain the functional aspect of traditional spreadsheets, in that each cell contains a unique value. We allow cells to contain sets of values, and cell references specify subset constraints. This means that recursively defined cells don't make sense in their functional framework but are perfectly meaningful in our relational one. *This is what really allows our spreadsheets to support full deduction.* These approaches add constraints to the functional framework, which as

they have shown can be very useful, and constraints can also be added to our relational framework. Another interesting combination of rules and spreadsheets is ARulesXL (`http://www.arulesxl.com/`). ARulesXL allows users to define WHEN rules that specify cell contents using defined variables. The use of logic is interesting, but it retains the functional aspects of traditional spreadsheets and does not support recursive definitions. Deductive spreadsheets can be understood as specifying subset relationships among sets. There have been several proposals for programming languages that support such set specifications [8, 25]. Our DSS might be viewed as a visual interface to a language like that of [8], however the other language is much more powerful than ours; we can define only Datalog programs, whereas the other language is Turing complete. Our focus is less on the power of the underlying language and more on its presentation and usability in the tabular spreadsheet form.

Visual Programming: There are a number of research as well as commercial systems that call themselves "Visual Prolog" or "visual logic programming" systems. We can distinguish these works into two broad categories. The first one describes technologies and systems, mainly commercial ones, which provide an integrated programming environment to develop and debug Prolog programs very much akin to Forte which provides a development environment for Java programming. An example of such a system is in `http://www.visual-prolog.com`. The second group of works focuses on graphical interfaces to create logic programs. Examples include [1], which provides graphical symbols via which one can create Prolog terms, Prolog clauses and Prolog programs. These kinds of work are more along the lines of visual programming languages. However unlike the vision espoused in DSS, users of such systems are required to be knowledgeable of logic programming. Finally we point out a recent work that describes extensions to the Excel spreadsheet that integrate *user-defined (non-recursive) functions* into the spreadsheet grid, rather than treating them as a "bolt-on" [9]. What they have achieved is a way to specify user defined functions visually with a spreadsheet. But each cell still possesses a unique value. We can lift these point-wise user-defined functions to work over cells representing sets of values as in XcelLog.

6 Discussion

The synergy between spreadsheets and rule-based computing has the potential to put into the hands of end users (ranging from novices to power users) technology to create and manage their own automated decision support applications with the same ease with which they are currently able to create financial applications of varying complexity with traditional spreadsheets. Our XcelLog system demonstrates that technology to create rules-driven applications with the spreadsheet metaphor is feasible. Nevertheless our experience with using XcelLog suggests that there is considerable scope for further research and development. The most immediate one concerns generating and visualizing explanations of the computational behavior of DSS expressions. This is useful not only for debugging the application but also for analyzing "what if scenarios". Our work on generating explanations for deduction [4, 21] and Excel's color-coded outlines denoting cell dependencies offers a suitable starting point for this problem. Another interesting and useful problem that has emerged from our XcelLog encoding exercises is to develop

a DSS methodology that will aid end users to conceptualize, and systematically develop DSS encodings for their problems. On the computing infrastructure side efficiency can be improved by developing incremental algorithms for (re)evaluating DSS expressions when cell content is changed. Our work on incremental algorithms for logic programs [23] seems to be well suited for this purpose. Progress on these fronts will accelerate the acceptance of DSS as a mainstream technology.

7 Acknowledgments

References

1. J. Augusti, J. Puigsegur, D. Robertson, and W. Schorleme. Visual logic programming through set inclusion and chaining. In *CADE 13 Workshop on Visual Reasoning*, 1996.
2. A. L. Couch and M. Gilfi. It's elementary, dear watson: Applying logic programming to convergent system management processes. In *Proceedings of the 13th USENIX Conference on Systems Administration (LISA)*, pages 123–138, 1999.
3. S. K. Das. Deductive databases and logic programming. Addidon-Wesley.
4. Y. Dong, C. R. Ramakrishnan, and S. A. Smolka. Evidence explorer: A tool for exploring model-checking proofs. In *Fifteenth International Conference on Computer Aided Verification (CAV)*, pages 215–228, 2003.
5. G. Gupta and S. F. Akhter. Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs. In *Practical Aspects of Declarative Languages (PADL), volume 1753 of Lecture Notes in Computer Science*, pages 308–323, 2000.
6. J. D. Guttman, A. L. Herzog, and J. D. Ramsdell. Information flow in operating systems: Eager formal methods. In *Workshop on Issues in the Theory of Security (WITS)*, 2003.
7. T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the selinux example policy. In *USENIX Security Symposium*, 2003.
8. B. Jayaraman and K. Moon. Subset logic programs and their implementation. *J. Log. Program*, 42:71–110, 2000.
9. S. P. Jones, A. Blackwell, and M. Burnett. A user-centered approach to function in excel. In *ICFP*, 2003.
10. M. Kassoff, L.-M. Zen, A. Garg, and M. Genesereth. Predicalc: A logical spreadsheet management system. In *31st International Conference on Very Large Databases (VLDB)*, 2005.
11. N. Li, B. Grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *IEEE Symposium on Security and Privacy*, pages 27–42, 2000.
12. N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11:35–86, 2003.
13. J. Lloyd. Foundations of logic programming. 2nd Edition, Springer-Verlag.
14. P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *FREENIX track of the 2001 Usenix Annua; Technical Conference*, 2001. Available from http://www.nsa.gov/selinx/.
15. P. Loscocco and S. Smalley. Meeting critical security objectives with security-enhanced linux. In *Proc. of 2001 Ottawa Linux Symposium*, 2001. Available from http://www.nsa.gov/selinx.

16. D. Maier and D. S. Warren. *Computing with Logic: Logic Programming and Prolog*. Benjamin/Cummings Publishers, Menlo Park, CA, 1988. 535 pp.

17. N.Li, J. Mitchell, and W. Winsborough. Design of a role-based trust-management framework. In *Proceedings of 2002 IEEE Symposium on Security and Privacy*, pages 114–130, May 2002.

18. X. Ou, S. Govindavajhala, and A. W. Appel. Mulval: A logic-based network security analyzer. In *14th Usenix Security Symposium*, 2005.

19. C. R. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security (JCS)*, 10:189–209, 2002.

20. D. M. Reeves, M. P. Wellman, and B. N. Grosof. Automated negotiation from declarative contract descriptions. In Müller, E. Andre, S. Sen, and C. Frasso, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 51–58, Montreal, Canada, 2001. ACM press.

21. A. Roychoudhury, C. Ramakrishnan, and I. Ramakrishnan. Justifying proofs using memo tables. In *Proc. of Principles and Practice of Declarative Programming*, 2000.

22. K. Sagonas, T. Swift, D. S. Warren, J. Freirre, and P. Rao. Xsb programmers manual, 2001. http://xsb.sourceforge.net/.

23. D. Saha and C. Ramakrishnan. Incremental evaluation of tabled logic programs. In *Proc. of Intl. Conf. on Logic Programming*, 2003.

24. B. Sarna-Starosta and S. D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, 2004. Available at http://www.cs.sunysb.edu/ stoller/WITS2004.html.

25. J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag, New York, NY, USA, 1986.