The Knowledge Engineering Review, Vol. 22:3, 269–279. © 2007, Cambridge University Press
doi:10.1017/S026988890700118X Printed in the United Kingdom

XcelLog: a deductive spreadsheet system

C.R. RAMAKRISHNAN, I.V. RAMAKRISHNAN and DAVID S. WARREN

Department of Computer Science, Stony Brook University, NY 11794;
e-mail: cram@cs.stonybrook.edu, ram@cs.stonybrook.edu, warren@cs.stonybrook.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 widespread 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. Reeves *et al.*, 2001), authorization rules for scalable access control and distributed trust management (e.g. Li *et al.*, 2000, 2002), and configuration management of complex systems (e.g. system administration (Couch & M. Gilfix, 1999), security policy configuration (Loscocco & Smalley, 2001b), and vulnerability analysis (Ramakrishnan & Sekar, 2002; Ou *et al.*, 2005)). 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 (Loscocco & Smalley, 2001a) and the variety of tools that have been developed to analyze these policies (Guttman *et al.*, 2003; Jaeger *et al.*, 2003; Sarna-Starosta & Stoller, 2004). 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

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press

	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}	

Figure 1 Deductive spreadsheet for discount eligibility

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 (Li *et al.*, 2003)).

1. A publisher, PUB, wants to give a discount to its 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 the 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 follows:

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 the 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 ***c r***) 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*.

Note that, unlike in a traditional spreadsheet (and indeed in other logical spreadsheets, e.g. Gupta & Akhter, 2000; Kassoff *et al.*, 2005), each DSS cell contains a *set* of values.

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press

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 U 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 which 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 (Lloyd). 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 (Lloyd) to DSS expressions by translating them into Datalog programs, a subclass of logic programs, which is used in deductive databases (Das). In Section 3 we describe the implementation of XcelLog, our prototype DSS system, with Excel as its front-end and our XSB tabled logic programming system as the backend deduction machine (Sagonas *et al.*, 2001). 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

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press

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 programs, that is, Prolog programs without function symbols (Maier & Warren, 1988). A Prolog program consists of rules of the form *head :- body* where *head* is a literal and *body* is a conjunct of literals. 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 if *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 implementing 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 Datalog programs. Our XSB Tabled Logic Programming system is well suited for this purpose (Sagonas *et al.*, 2001). 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.

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press

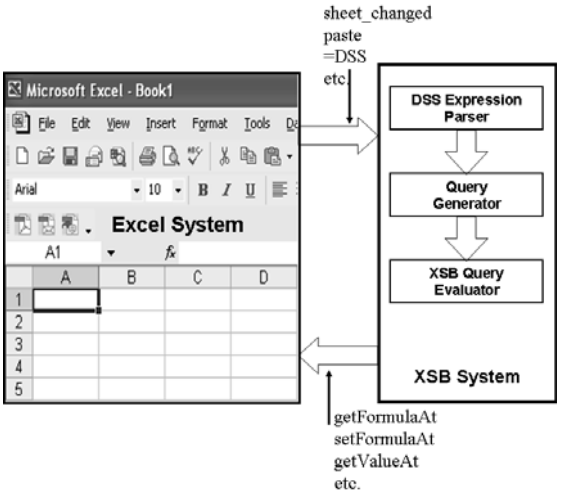


Figure 2 The XcelLog DSS system

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', that is, the implementation of deductive spreadsheets was encapsulated within the Excel environment; Excel served as the front end while the DSS expressions were evaluated by XSB in the backend. By 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 [ ]. For example, 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.

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, which is computed by

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press

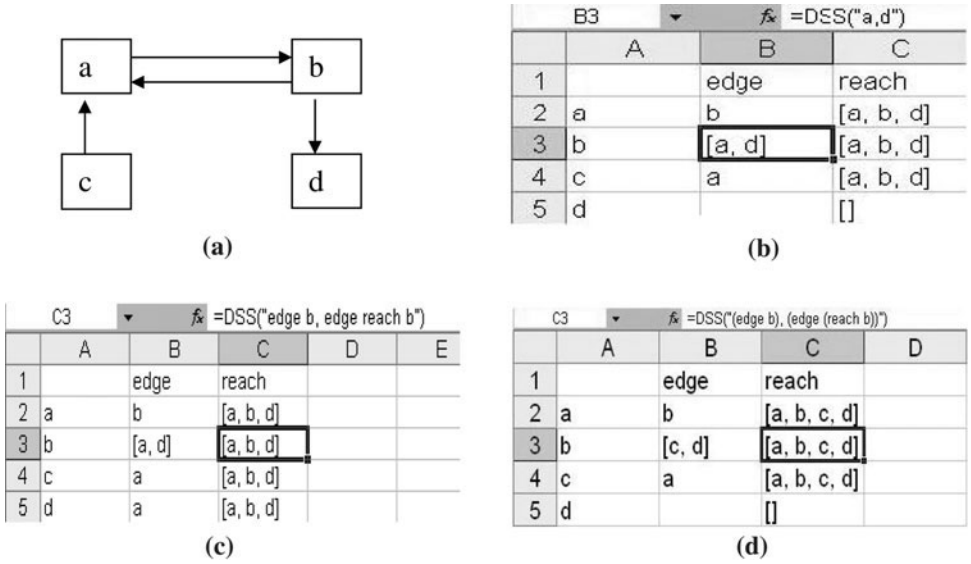


Figure 3 Example of a reachability problem in graphs

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 the 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 four 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 shows 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 one more

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press

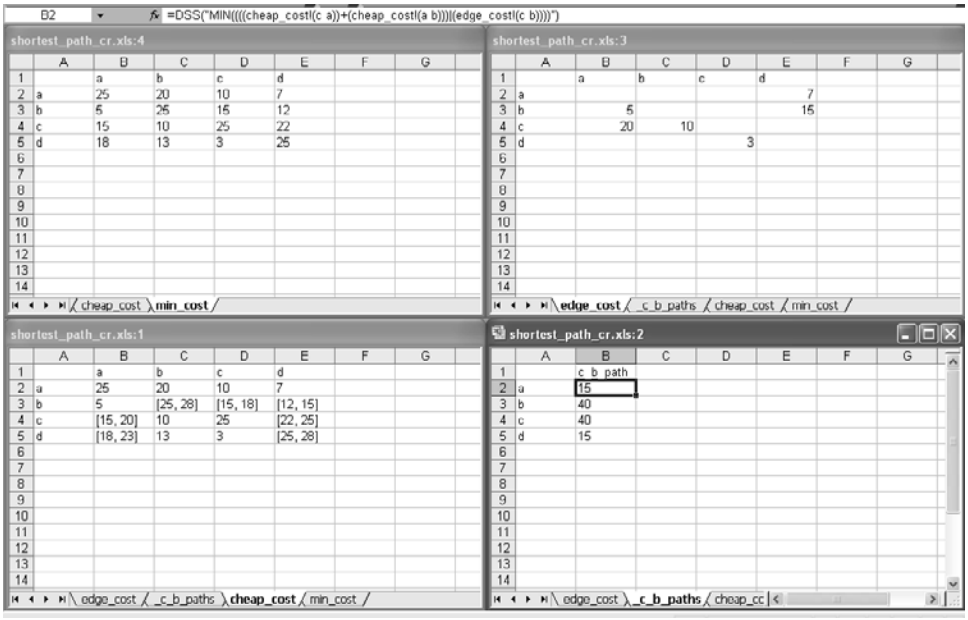


Figure 4 Example of a shortest path problem in graphs

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/she types in all the entries into the edge column. These are the direct edges between nodes (see the edge column in Figure 3(b)). Second, he/she 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/she 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) The Shortest Path Problem in Graphs. We sketch the encoding in XcelLog of the problem of computing the length of the shortest path between any two nodes in a directed graph, where the edges are labeled with non-negative numbers indicating their length. The encodings to this problem is shown in Figure 4.

The edges in a given graph are represented in DSS by the sheet 'edge_cost'. In the figure, the example graph has four veritices (labeled a, b, c, and d). The edges in the graph are represented

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press

in the form of an adjacency matrix: a number in column c and row r indicates the length of the edge from vertex c to vertex r; and if a cell is empty, then there is no edge between the corresponding vertices. Thus, in the example there is an edge from a to b of length 5, and there is no edge from b to a.

We construct the shortest path using two intermediate tables. Sheet 'cheap_cost' specifies a set of short paths between all pairs of vertices: paths that are not necessarily the shortest, but include the shortest path among them. The costs of short paths between two vertices is in turn defined using intermediate table '_c_b_paths', which specifies the set of short paths from c to b. A short path from c to b via a is obtained by taking a short path from c to a, and then a short path from a to b. The edge from c to b is also a short path from c to b. Observe in the example that the DSS formula for cell (c_b_path a) in sheet '_c_b_path' captures exactly this set. The set of short paths from c to b is then the set of all values in column c_b_path of sheet '_c_b_paths'. This is specified by copying the values in this column and filling the cell (c b) of 'cheap_costs' sheet.

The final sheet 'min_cost' is defined by simply filling each cell in the table with the minimum of values in the corresponding cell of 'cheap_cost'. Note that, in this formulation, the sheets 'cheap_cost' and '_c_b_costs' are mutually dependent (i.e. recursively defined); and the sheet 'min_cost' is not defined recursively, but based on the sheet 'cheap_cost'. The recursive formulation in DSS and its evalution corresponds to the dynamic programming algorithm for computing all-pair shortest paths.

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 values were entered explicitly and not by filling were the cells in the edge_cost sheet. In addition, we explicitly entered the expression for short path in row a and column _c_b_path. All the other expressions were obtained through copy-and-paste operations. Note that the abstractions generated by copy-and-paste operations 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, the shortest path problem encoded above is one such example. 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 an edge cost, 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: for example, 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 the recent proposals that are most similar to ours.

Knowledgesheet (Gupta & Akhter, 2000) and *PrediCalc* (Kassoff *et al.*, 2005) 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)

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press

program, executes it, and returns the solution as cell values. In *PrediCalc*, the constraint-solving engine is more integrated into spreadsheet interaction, and the 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 do not 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 (Schwartz *et al*., 1986; Jayaraman & Moon, 2000). Our DSS might be viewed as a visual interface to a language like that of (Jayaraman & Moon, 2000); 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 that 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 (Augusti *et al*., 1996), 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 which integrate *user-defined (non-recursive) functions* into the spreadsheet grid, rather than treating them as a 'bolt-on' (Jones *et al*., 2003). 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 the 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 the explanations for deduction (Roychoudhury *et al*., 2000;

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press

Dong *et al.*, 2003) 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 (Saha & Ramakrishnan, 2003) seems to be well suited for this purpose. Progress on these fronts will accelerate the acceptance of DSS as a mainstream technology.

7 Acknowledgements

This work was done at XSB, INC (`http://www.xsb.com`), and was supported by the United States Defense Advanced Research Projects Agency (DARPA) under Contract No. W31P4Q-05-C-R034. We thank Chris Rued for implementing the Excel-XSB interface.

References

Augusti, J., Puigsegur, J., Robertson, D. and Schorlemer, W. 1996 Visual logic programming through set inclusion and chaining. In *CADE 13 Workshop on Visual Reasoning*.

Couch, A. L. and Gilfix, M. 1999 It's elementary, dear watson: Applying logic programming to convergent system management processes. In *Proceedings of the 13th USENIX Conference on Systems Administration* (LISA), pp. 123–138.

Das, S. K. *Deductive Databases and Logic programming*. Addidon-Wesley, Wokingham, England, 1992.

Dong, Y., Ramakrishnan, C. R. and Smolka, S. A. 2003 Evidence explorer: a tool for exploring model-checking proofs. In *Fifteenth International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science 2725, pp. 215–218, Springer.

Gupta, G. and Akhter, S. F. 2000 Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs. In *Practical Aspects of Declarative Languages (PADL)*, Vol. 1753 of *Lecture Notes in Computer Science*, pp. 308–323, Springer.

Guttman, J. D., Herzog, A. L. and Ramsdell, J. D. 2003 Information flow in operating systems: Eager formal methods. In *Workshop on Issues in the Theory of Security* (*WITS*).

Jaeger, T., Sailer, R. and Zhang, X. 2003 Analyzing integrity protection in the SELinux example policy. In *USENIX Security Symposium*.

Jayaraman, B. and Moonm, K. 2000 Subset logic programs and their implementation. *Journal of Logical Programming*, **42**(2): 71–110.

Jones, S. P., Blackwell, A. and Burnett, M. 2003 A user-centered approach to function in excel. In *ICFP*.

Kassoff, M., Zen, L.-M., Garg, A. and Genesereth, M. 2005 PrediCalc: A logical spreadsheet management system. In *31st International Conference on Very Large Databases (VLDB)*.

Li, N., Grosof, B. and Feigenbaum, J. 2000 A practically implementable and tractable delegation logic. In *IEEE Symposium on Security and Privacy*, pp. 27–42.

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

Li, N., Winsborough, W. H. and Mitchell, J. C. 2003 Distributed credential chain discovery in trust management. *Journal of Computer Security*, **11**(1): 35–86.

Lloyd, J. W. *Foundations of Logic Programming*. 2nd Edn. Springer Verlag, New York, 1987.

Loscocco, P. and Smalley, S. 2001a Integrating flexible support for security policies into the Linux operating system. In *Proceedings of FREENIX track of the 2001 Usenix Annua; Technical Conference*. Available at `http://www.nsa.gov/selinx/`.

Loscocco, P. and Smalley, S. 2001b Meeting critical security objectives with security-enhanced Linux. In *Proceedings of 2001 Ottawa Linux Symposium*, 2001. Available at `http://www.nsa.gov/selinx/`.

Maier, D. and Warren, D. S. 1988 *Computing with Logic: Logic Programming and Prolog*. Menlo Park, CA: Benjamin/Cummings Publishers, ISBN 0-8053-6681-4, 535 pp.

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

Ramakrishnan, C. R. and Sekar, R. 2002 Model-based analysis of configuration vulnerabilities. *Journal of Computer Security*, **10**(1/2): 189–209.

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press

Reeves, D. M., Wellman, M. P. and Grosof, B. N. 2001 Automated negotiation from declarative contract descriptions. In J. P. Müller, E. Andre, S. Sen, and C. Frasson (eds.), *Proceedings of the Fifth International Conference on Autonomous Agents*, pp. 51–58, Montreal, Canada: ACM Press.

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

Sagonas, K., Swift, T., Warren, D. S., Freirre, J. and Rao, P. 2001 *XSB Programmers Manual*. Available at `http://xsb.sourceforge.net/`.

Saha, D. and Ramakrishnan, C. R. 2003 Incremental evaluation of tabled logic programs. In *Proceedings of International Conference on Logic Programming*.

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

Schwartz, J. T., Dewar, R. B., Schonberg, E. and Dubinsky, E. 1986 *Programming with Sets; An Introduction to SETL*. New York, NY: Springer-Verlag.

https://doi.org/10.1017/S026988890700118X Published online by Cambridge University Press