TOWARD A LOGICAL/PHYSICAL THEORY OF SPREADSHEET MODELING

Tomás Isakowitz

Shimon Schocken

Henry C. Lucas, Jr.

Department of Information Systems
Leonard N. Stern School of Business
New York University

October 23, 1992

Replaced by IS-93-24

Working Paper Series
STERN IS-92-28

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

Toward a Logical/Physical Theory of Spreadsheet Modeling

In spite of the increasing sophistication and power of commercial spreadsheet packages, we still lack a formal theory of spreadsheets and a methodology that aids their construction and maintenance. Using a new functional relational language, this paper identifies four principal components that characterize any spreadsheet model: *Model*, *Data*, *Editorial*, and *Binding*. We present a *factoring* algorithm for identifying and extracting these components from conventional spreadsheets automatically, and a *synthesis* algorithm that assists users in the construction of executable spreadsheets from reusable components. This approach opens new possibilities for applying object oriented and model management techniques to support the construction, sharing, and reuse, of spreadsheet models in organizations.

CR Categories and Subject Descriptors: H.4.1 [Information Systems Applications]: Office Automation - Spreadsheets; H.4.2 [Information Systems Applications]: Types of Systems - Decision Support; I.6.4 [Simulation and Modeling]: Model Validation and Analysis; I.6.5 [Simulation and Modeling]: Model Development; K.8.1 [Personal Computing]: Application Packages - Spreadsheets.

General terms: Theory, Design, Languages

Additional Key Words and Phrases: Model Management

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

Introduction

Spreadsheet modeling represents one of the most successful applications of information technology. The original Visicalc program transformed the notion of end-user computing radically, creating a new computational paradigm which offered a unique combination of ease of use, on the one hand, and unprecedented modeling power, on the other. As a result, spreadsheet programs became the most widely used decision support tools in modern business. Compared to their humble origins in the late 70's, today's spreadsheet programs are extremely powerful, versatile, and user-friendly. Yet the basic practice of building a spreadsheet model remains the same as it was a decade ago. Further, with the exception of a few scattered efforts (e.g. Ronen, Palley and Lucas, [12]), a *theory* of spreadsheet analysis and design is yet to emerge.

The central theme of this paper is that spreadsheet models should be analyzed, if not constructed, at two separate levels: logical and physical. The *logical* level consists of a formal and implementation-free description of the model's intrinsic structure. The *physical* level concerns such details as storage, formatting, user interface, and other aspects that effect the model's appearance, but not its underlying structure. This distinction is nonexistent in the common practice of spreadsheet modeling, where logical, physical, and data elements are intermingled and treated as one entity. We believe that until this built-in dependency is resolved, it will be difficult if not impossible to develop intelligent model management systems for spreadsheet models.

This paper identifies four principal components that characterize any spreadsheet: *Model, Data, Editorial*, and *Binding*. Of the four components, the most important and interesting

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

one is the *Model*, which represents the spreadsheet's logical structure in a formal *functional relational language*. We present a top-down *factoring* algorithm that extracts the four components from conventional spreadsheets automatically, and a bottom-up *synthesis* algorithm that constructs executable spreadsheets and spreadsheet templates from a repository of reusable spreadsheet components. Further, the paper defines clearly what is meant by the *logical* and *physical* views of spreadsheet models, leading to a dual framework for spreadsheet analysis and design. The tools and techniques that the framework has yielded make it possible to apply a new object-oriented approach to building and maintaining spreadsheet models. In addition, they transform the conceptual notion of a model management system to a feasible undertaking, i.e. to a system that can be actually implemented in the field.

The remainder of this section of the paper discusses software engineering and model management topics that pertain to this research. The next section defines and illustrates the distinction between the physical and the logical views of spreadsheet models. The functional relational language for representing spreadsheet models, whose formal syntax is covered in a separate `BNF` appendix, is illustrated in the third section. The two sections following it present the factoring and the synthesis algorithms, respectively. Lastly, we comment on the implications of this research for the common practice of spreadsheet modeling, and points at future research directions.

Software engineering and spreadsheets: Viewed as model generators, spreadsheet programs have both pros and cons.[1] In spite of their considerable ease of use and instant mod-

[1]Throughout the paper, the term *spreadsheet programs* refers to spreadsheet modeling environments like `Lotus`, `Quattro`, and `Excell`. The terms *spreadsheet models*, or simply *spreadsheets*, refer to *specific* spreadsheets, e.g. P&L spreadsheets, inventory control spreadsheets, and the like.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

eling power, they suffer from several limitations which typically go unnoticed by novice users: implicit logic, inaccessible model structure, data dependency, and lack of a unifying model base. In many ways, the present state of spreadsheet modeling is reminiscent of the state of data management in the pre-DBMS era. Just as application programs of the past were permitted to define redundant and inconsistent file structures, today's spreadsheets often contain overlapping and inconsistent models. Before data definition was elevated to the DBMS level, file structures were fixed in the program's code. Likewise, the logic and documentation of spreadsheet models are often 'buried in the formulae,' and are largely inaccessible to people other then the spreadsheet's creator. To complete the analogy, most of these problems arise because spreadsheet programs lack *high level* means to support the practice of building and maintaining models.

To illustrate some of these points, consider the spreadsheet example in Figure 1. There are two ways to describe this example. Viewed from a logical, or a functional, perspective, the spreadsheet represents a parameterized profit and loss projection model. Viewed from a physical perspective, though, the spreadsheet amounts to two 'blocks' of cells (`B2..E2` and `B9..G16`) that are interrelated through a set of formulae[2]. Although the cell formulae are not presented here, one can consult the appearance of the spreadsheet and common sense to guess the following relationships: Sales are expected to grow 10% annually; Cost of goods sold is assumed to be 60% of sales; Overhead is assumed to be fixed at $2,500; Lease is fixed at $100 in the first two years and $500 thereafter; and tax is assumed to be 48% of gross income.

[2]Following common practice, continuous blocks of cells are denoted by their top-left and bottom-right coordinates.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

Put Figure 1 around here

As it turns out, however, spreadsheet models have more to them than appears on the surface. For example, it is true that sales grow 10% annually, but only in the first four years. In 1996 and 1997, sales are 20% greater than the average sales in the previous two years. Similarly, although it is reasonable to assume that net income equals gross minus tax, there is absolutely no reason to believe that this is actually the case in this particular spreadsheet. The lesson is clear: the physical appearance of a spreadsheet can be deceiving, as it is not necessarily consistent with the logical structure that it suggests. One obvious way to validate the integrity of the spreadsheet is to print out the cell formulae and inspect their definitions. However, even at this intimate layer of representation, the model's logic is anything but readily available. For example, the tax of 1992 is computed through the formula `B14: @IF(B13>0,B13*$E$2,0)`. The logical equivalence of this expression is `IF net>0 THEN tax=net*taxrate ELSE tax=0`, but this useful documentation is external to the spreadsheet model, and may not be available.

Currently, spreadsheet programs represent spreadsheets in a strictly physical fashion, treating them as collections of cell addresses and formulae with no underlying semantics. For this reason, users of spreadsheet models are prone to many accidental mishaps. For example, one can delete cells that impact other cells (which may be out of sight), override generic formulae with fixed values, add a new 'cost' item without modifying the 'total cost' formula, and the like. Since the spreadsheet program 'doesn't know' what the user is trying to do *in the model's realm*, there is no way to sense that such activities can corrupt the logical structure of the spreadsheet. In a similar vein, spreadsheet programs make it difficult to isolate the *Data* element of a given spreadsheet. Although one can separate 'data

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

cells' from 'model cells' by focusing only on the cells that contain constant values, the data will have no supporting structure. For example, what is the meaning of a cell definition like `C12: 100`? An inspection of the spreadsheet image suggests that `100` is the value of the `lease` item in the year 1992, but this interpretation is strictly in the eye of the user, and is not a formal part of the spreadsheet model.

To a large extent, many of these problems resemble the kinds of problems that preceded the development of structured programming techniques. The first generation of high-level languages permitted certain programming practices that later led to long-term maintenance problems. For example, no restriction was put on the use of `GOTO` commands, resulting in the infamous phenomenon of 'spaghetti code.' In a similar vein, spreadsheet programs do not restrict the use of any cell and formulae pattern, allowing users to construct any spreadsheet that they desire, including, of course, poorly-designed spreadsheets. One objective of this paper is to present a systematic approach to modeling that promotes the construction of *structured models*, i.e. models that are easy to extend and maintain.

Model management and spreadsheets: According to Blanning et al. [1], model management systems (`MMS`) began with the realization that there is a need to insulate the users of decision support systems from the physical aspects of the organization and processing of decision models. One of the most comprehensive efforts to develop an `MMS` was undertaken by Geoffrion [3], who argued for a generalized *Structured Modeling* (`SM`) framework for representing management science and operations research models. For Geoffrion, a model is specified independently of its data and of the programs that are used to execute it. This modularity promotes a flexible model base that encourages consistency and reuse. Recently, Geoffrion augmented his framework with a special modeling language – `SML` –

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

and a syntax-directed editor for developing structured models [4, 5].

Closer to the world of spreadsheets one finds `IFPS` [7]– a model generation package with a special emphasis on financial applications. Like `SM`, `IFPS` supports reuse of different model components by separating the model schema from the data on which it operates. Although `SM` and `IFPS` are capable of producing spreadsheet-like outputs, they require the user to learn a new modeling language in order to construct the initial model. This limitation was addressed, to a certain extent, by advanced spreadsheet programs like `Javelin` and `Excell`. In both packages, one can build a conventional spreadsheet and then assign symbolic names to selected rows and columns. This structure endows the spreadsheet with a certain degree of logical independence and, in the case of `Javelin`, a clear separation between data cells and model cells.

What kind of services should an idealized model management system provide in the context of spreadsheets? One of the major benefits of spreadsheet modeling is the ability to change assumptions and inspect the impact on some output criterion. Hence, a spreadsheet `MMS` should facilitate the storage and retrieval of different data sets associated with different sensitivity and 'what-if' analyses. In addition, the system should facilitate transparent access to remote databases so that data can be piped to and from spreadsheets without human intervention. Similarly, a spreadsheet `MMS` should facilitate access to a repository of reusable models and model 'chunks,' or a model base. Ideally, the user should be able to retrieve models according to a variety of search criteria such as functional purpose, generic structure, and relationship to other models. Once retrieved, the system should allow the user to combine these models with other models and databases across the organization.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

Today's spreadsheet programs lag far behind this idealized notion of a spreadsheet model management system. Contrary to the objectives of model reuse and data independence, present spreadsheets tend to be 'owned' by their creators, and the data that they operate on are embedded in their underlying logic. Hence, before we begin to articulate the notion of a spreadsheet `MMS`, we first have to demonstrate that data and structure *can* be extracted from conventional spreadsheets and then treated formally as separate entities. This modularity will also enable the reverse operation, in which data and structure modules are synthesized into executable spreadsheets under the user's control. In order for any of these ideas to be practically feasible, a new dual perspective on spreadsheet models is needed.

The Physical and Logical Views of Spreadsheets

A conventional spreadsheet is a collection of addressable *cells*, arranged in a 2-dimensional matrix. Each cell has a *definition* that binds it to either a *constant* value or to a calculated value, obtained through a *formula.* Taken as a whole, these definitions determine the user's view of the spreadsheet, which is automatically updated whenever one or more of the cell definitions is changed. In addition to this familiar 2-dimensional perspective, every spreadsheet has a linear representation, denoted hereafter the *spreadsheet map*, which is normally used for documentation and debugging purposes. For example, the map of the P&L spreadsheet of Figure 1 is depicted in Figure 3. For each active cell in the spreadsheet, the map contains an entry that gives the cell's address, formatting specifications (if any), and definition. Spreadsheet maps can be printed out or stored in `ASCII` files on demand by all spreadsheet programs. For the purpose of this article, the *physical view* of a spreadsheet is taken to be its respective map, as produced by the host spreadsheet program.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

What then is the *logical view* of a spreadsheet? We observe that each spreadsheet can be characterized by four principal components. Paraphrasing Wirth [14], this observation can be summarized as: *Spreadsheet = Model + Data + Editorial + Binding.* The *Model* ($\mathcal{M}$) component is the logical structure of the spreadsheet. The *Data* ($\mathcal{D}$) component is the structured collection of constants on which $\mathcal{M}$ operates. The *Editorial* ($\mathcal{E}$) component can be defined as what is left over in the spreadsheet after $\mathcal{M}$ and $\mathcal{D}$ have been carved out: titles, column and row headings, and documentation. Finally, the *Binding* ($\mathcal{B}$) component is the physical mapping that binds $\mathcal{M}$,$\mathcal{D}$, and $\mathcal{E}$ to each other, and to the spreadsheet grid. For example, the $\mathcal{M}$ and $\mathcal{D}$ components of the P&L spreadsheet are depicted in Figure 2, which is discussed extensively in the next section.

Put Figure 2 around here

Although all four components are equally important on practical grounds, $\mathcal{M}$ and $\mathcal{D}$ are far more interesting and challenging to deal with from a theoretical perspective. Ideally, $\mathcal{M}$ and $\mathcal{D}$ should be (i) independent of the spreadsheet program; and (ii) independent of each other. The *Model* component of the spreadsheet should be viewed as a mathematical abstraction that can be described in terms of several different formalisms, of which cells and formulae is only one, and certainly not the most effective, representation. Likewise, the *Data* component should be treated as stand-alone entity that might be a subset, or a view, of a remote database. Taken together, the 'sum' of the four components $\mathcal{M} + \mathcal{D} + \mathcal{E} + \mathcal{B}$ forms the familiar notion of a conventional spreadsheet.

Practically all the problems that were alluded to earlier in the paper are related to the fact that, in a conventional spreadsheet program, users are encouraged to weave these compo-

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

nents together and treat them as one entity, forming a prime example of how a modular system should *not* be constructed. As a result, the two most important principles of software engineering– separation of logical design and physical implementation, and separation of algorithms and data – are widely violated by conventional spreadsheet programs. The first step to resolving these problems requires a precise definition of what is meant by the *logical view* of a spreadsheet model.

In this paper we define the logical view of a spreadsheet to be its underlying *Model* component, which is further implemented as a collection of *functional relation schemas.* A functional relation is similar to a regular relation in that both data structures consist of one or more attributes and of one or more tuples, the minimal case being a single-attribute/single-tuple relation. Unlike regular relations, though, functional relations have two types of attributes: *data attributes* and *functional attributes.* Data attributes define slots that store constants, whereas functional attributes are bound to functions that are calculated 'when needed,' to borrow a term from object oriented programming.

We take the position that any spreadsheet, no matter how complex, can be viewed as a (non-unique) collection of functional relations. To illustrate, consider the top part of Figure 2, which depicts an outlined version of the P&L spreadsheet. According to this figure, the spreadsheet can be seen as involving two functional relations, named `assumptions` and `proforma`, or `a` and `p` for brevity. In each relation, some attributes (e.g. `year` and `lease`) contain constant values, whereas other attributes (e.g. `sales` and `cogs`) are bound to *functions* that relate them to attributes in the same relation as well as to attributes in other relations. The *union* of all the functional relation schemas that make up a particular spreadsheet is denoted the *spreadsheet's schema* or the $\mathcal{M}$ component of the spreadsheet.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

The schema of the P&L spreadsheet is depicted at the bottom left of Figure 2. A complete discussion of the *syntax* of spreadsheet schemas and the process through which they are constructed is given later in the paper.

The *Data* component ($\mathcal{D}$) of the P&L spreadsheet is depicted at the bottom right of Figure 2. The numeric values in the relations are user-supplied data, extracted from corresponding cells in the spreadsheet. The special `C` symbols denote calculated values that correspond to functional attributes in the spreadsheet schema. When these functions are 'evaluated,' the `C` values become constant values, and the functional relations become data relations, i.e. relations that contain only constant values. We see that each functional relation induces an ordinary data relation in the database sense of the word.

The physical and the logical views of spreadsheets are independent of each other; the physical characteristics of each functional relation, e.g. its location, column/row headings, and spatial orientation, are external to, and independent of, the relation's schema. Likewise, the physical arrangement of the relations on the spreadsheet grid (side-by-side, top-bottom, etc.) is independent of the schema. Thus, a user may transpose the spreadsheet image of a functional relation from a row-wise orientation to a column-wise orientation, and vice versa, or simply move it to another area in the grid, leaving the spreadsheet's $\mathcal{M}$ component intact.

The distinction between the logical and the physical views of spreadsheets has significant practical implications. Suppose that spreadsheet programs were capable of recognizing this modularity explicitly. That is, whenever a spreadsheet is loaded into such an extended spreadsheet program, the program would also load a transparent image of its underlying $\mathcal{M}$, $\mathcal{D}$, $\mathcal{B}$, and $\mathcal{E}$ components. By continuously comparing the user's activities on the

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

physical spreadsheet grid to their implications for the four components, the program could sense what the user is trying to do not only in the way of manipulating physical cells and formulae, but also in the way of building logical models. This extension would endow spreadsheet programs with the ability to understand the *semantics* of spreadsheet models, something which is lacking in the present generation of spreadsheet modeling environments.

The Functional Relational Language

A spreadsheet data definition language must address two important aspects of spreadsheet models. First, many spreadsheets have one or more *repetitive patterns*, e.g. the `years` entity in the P&L example. Second, many spreadsheets are characterized by *functional interdependencies*, e.g. the sales of *this* year are based on the sales of the *previous* year. The first requirement – repetition – prompted us to base our language on the relational approach to data definition. The second requirement – functional interdependencies – led us to consider a functional extension of the relational model.

There have been several proposals to extend the standard relational model with functional and object oriented capabilities. For example, Gehani [2] described a financial database in which monetary values were expressed in terms of several international currencies. Using currency conversion functions and the prevailing exchange rates, the system could automatically revise monetary attributes to reflect their real values in terms of a given currency. Taking a more fundamental approach, Maier [11] presented a general *computed relation* formalism in which attributes could be expressed as functions of other attributes within the same relation. The notion of computed attributes played a key role in several object ori-

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

ented relational systems, e.g. `Cactis` (Hudson and King, [8, 9]), and `OZ+` (weiser-lochovsky, [13]). In `Cactis`, functional attributes were implemented using attribute grammar techniques [10]. In `OZ+`, value dependencies were implemented through functions that operated on objects. Coming from a different direction, Ginzburg and Kurtzman [6] provided a relational view of spreadsheets through their *Spreadsheet History Schemes*, which once again contained the distinction between 'given' attributes and 'evaluated' attributes.

The functional relational language that is described in this paper (denoted hereafter `FRL`) differs from the above formalisms in several ways. First, it allows the functional attributes of a certain relation to refer to attributes in *other* relations. Second, it offers both absolute and relative tuple addressing, in line with the addressing style of spreadsheet formulae. Finally, the language was designed in such a way that it will not require spreadsheet users to change the way they normally construct spreadsheets. That is, we sought a data definition language that will enable automatic conversions of conventional spreadsheets to spreadsheet schemas, and vice versa.

The remainder of this section provides an overview of `FRL` as it unfolds in the context of the P&L example. A complete description of the language syntax is given in a separate `BNF` appendix.

Functional relations: A functional relation is a tabular data structure consisting of one or more attributes and one or more tuples. For example, the `assumptions` (or `a`) relation in Figure 2 consists of 4 attributes and one tuple, whereas the `proforma` (or `p`) relation consists of 8 attributes and 6 tuples. Each relation has a mandatory *name* and an optional *alias*, or abbreviated name. We distinguish between relations that normally contain many tuples,

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

and relations that are designed to contain one tuple only. The latter data structures, denoted *vector relations*, are uncommon in relational databases but occur frequently in spreadsheet modeling. In the P&L spreadsheet, `a` is a vector relation designed to store a single tuple of model parameters.

The attributes of a functional relation fall into two categories: *data* and *functional.* For example, all the attributes of the `a` relation are of type 'data.' The `p` relation has two data attributes – `year` and `lease` – and six functional attributes: `sales`, `cogs`, `ovhead`, `inc`, `tax`, and `net`. For each data attribute, the relation *schema* specifies a data type which is either `numeric`, `string`, `date` or `logical`, consistent with the standard data types of spreadsheet constants. For example, the definition of the `lease` attribute is `lease: numeric`, indicating that `lease` is a slot designed to store user-supplied data of type `numeric`. The definitions of *functional* attributes, e.g. `tax: if(inc>0,inc*a.tax,0)`, are more involved, making use of such constructs as *functions*, *operators*, and *case structures.* We now describe each of these constructs in broad terms, leaving their precise definitions to the appendix.

Keys and orderings: With the exception of vector relations, each functional relation must have a *key* in the database sense of the term. That is, for each relation schema s, there is an attribute x such that all tuples in every relation r whose schema is s have different x values. For example, the key of the `p` relation in Figure 2 is `year`, and the values of that key are `1992, 1993, 1994, 1995, 1996`, and `1997`. For the sake of homogeneity, FRL requires that *all* relation schemas have a designated key. If a certain relation schema doesn't have a natural key candidate associated with it, a hidden *surrogate key* which is essentially a tuple identifier is attached to the schema by default.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

The *domains* of the key attributes (the sets of values that the key attributes can attain) are assumed to be totally ordered. That is, for each two key values k and k', either $k < k'$ or $k' < k$, as is obviously the case with the `year` key of the `p` relation. When a total ordering among key values is not natural, an arbitrary ordering is imposed, based on tuple identifiers. The total order implies the existence of a minimal key value and a maximal key value (within a particular relation), denoted *min(key)* and *max(key)*, respectively. Given any key value k, the function *prev(k)* returns the key value immediately preceding k, the function *succ(k)* returns the key value immediately succeeding k, whereas *prev(min(key))* and *succ(max(key))* return the special value *null*. It's important to observe that the values of *prev(k)* and *succ(k)* (as well as *min(key)* and *max(key)*) are relation-dependant. For example, given the present contents of the `p` relation in Figure 2, we have `succ(1996)=1997`; however, if the last two years in that relation were `1996` and `2000` instead of `1996` and `1997`, we would have had `succ(1996)=2000`.

Tuple addressing: Since a functional relation always has a key, the relation's tuples can be indexed, or referred to, by either absolute or relative key values. For example, let r be a relation whose key attribute is named *key*. In that relation, the tuple whose key value is k (i.e. $key = k$) is referred to by the notation $r[k]$, whereas the symbolic notation $r[key]$ refers to the *current tuple* – the tuple that is presently being processed or defined. For example, the absolute notation `p[1994]` refers to the `p` tuple whose key value is `1994`; `p[year]` refers to `p`'s current tuple; `p[prev(year)]` refers to the tuple that precedes `p`'s current tuple; and `p[min(year)]` and `p[max(year)]` refer the first and last tuples in the relation, respectively.

References to individual tuples can also be implemented via tuple numbers. Since the tuples

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

of every functional relation r are always totally ordered by their respective key values, a reference like "r's third tuple," denoted $r[\#3]$, can be interpreted without ambiguity as $r[succ(succ(min(key)))]$. In general, then, the ith tuple in the relation is denoted $r[\#i]$, whereas the tuple whose key value is k is denoted $r[k]$. The two types of references are interchangeable through the following mapping:

$$r[\#i] = \begin{cases} r[min(key)] & \text{if } i = 1 \\ r[\overbrace{succ(succ(\ldots succ}^{i-1}(min(key))\ldots) & \text{if } i > 1 \end{cases}$$

The practice of referencing tuples by their tuple numbers is normally not used in building *new* spreadsheet schemas in FRL. At the same time, it is useful in some situations when schemas are *extracted* from existing spreadsheets, as discussed later in the paper.

Attribute Addressing: Spreadsheet formulae operate on physical operands. For example, consider the formula `B3+@SQRT($A$15)/2`, which operates on the cells `B3` and `A15`. Since one of the objectives of the functional relational model is to convert physical formulae to logical expressions that are independent of the host spreadsheet environment, FRL contains several features to address operands logically, as opposed to physically.

In general, the value of an attribute x in the tuple whose key value is k in the relation r is denoted $r[k].x$. Thus, `p[1992].sales` refers to the `sales` value in the tuple of the `p` relation whose key value is `1992`. Similarly, $r[\#3].x$ refers to the value of the x attribute of r's 3rd tuple (in the order of the relation's key). Two default rules are used to abbreviate these attribute references. First, the value of x in the current tuple, i.e. $r[key].x$, is denoted $r.x$. Second, when an attribute x is referred to within the schema of its own relation, the relation prefix can also be dropped and one is left with the reduced attribute reference x.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

$$\texttt{attribute}: \begin{array}{ll} i_1 \leq \texttt{n} < i_2 & \mapsto \ exp_1 \\ i_2 \leq \texttt{n} < i_3 & \mapsto \ exp_2 \\ \vdots & \\ i_{m-1} \leq \texttt{n} < i_m & \mapsto \ exp_m \end{array}$$

This construct reads: "for tuples $i_1, i_1+1, \ldots, i_2-1$, bind the attribute to exp_1; for tuples $i_2, i_2+1, \ldots, i_3-1$, bind the attribute to exp_2;" and so on. Case structures are implicitly used in spreadsheet modeling, where it is quite common to specify a model by providing *base* values for the first few tuples and defining the formulae that control subsequent tuples in an iterative fashion. In Figure 2, for example, this construction by cases is used to define the `p.sales` attribute.

We note in passing that *all* attribute definitions in FRL are in fact functions of key values. To illustrate, recall that an attribute definition like `inc: sales-lease-cogs` is actually a shorthand of `p[year].inc: p[year].sales-p[year].lease-p[year].cogs`. From a functional standpoint, this is equivalent to the expression $f(x) = p[x].sales - p[x].lease - p[x].cogs$. Thus, to obtain the value of the `inc` attribute of a certain tuple whose key value is k, the function f is applied to the tuple's key, and `inc` is bound to the value $f(k)$. In a similar way, an expression like `lease: numeric` is in fact equivalent to the functional expression `p[year].lease = I(numeric)`, where $I(x)$ is the identity function and `numeric` is whatever number the user chooses to enter for that year. We see that *all* the attributes in FRL are bound to functions, thus the name *functional relations.*

To summarize this section, we revisit the schema of the `p` relation in Figure 2. The first attribute, `year`, is the key of the relation, which is of type `numeric`. The `sales` attribute is bound to a case structure. For the first tuple in the relation, the value of `sales` is set to the

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

constant 6000. For the second, third, and fourth tuples, it is set to the sales value of the previous tuple – p.[prev(year)].sales – multiplied by the growth factor 1+ a.grate, where a.grate is the value of the grate attribute in the single tuple of the a relation. For the fifth tuple and thereafter, the value of sales is set to the average sales values in the previous two tuples, multiplied by the constant growth factor 1.2. The next attribute – lease – is a data attribute of type numeric. Cost of goods is computed by multiplying the COGS rate assumption – a.cogs – by the sales value of the current tuple of p. Gross income is obtained by subtracting the values of lease and cogs from the value of sales, all attributes values taken from p's current tuple. The tax payable amount is set to the inc value of p's current tuple times the tax rate assumption a.tax, but only if p.inc is positive. Finally, net income is obtained by subtracting the tax value from the inc value in p's current tuple.

It is instructive to compare the original spreadsheet model at the top of Figure 2 with its respective schema at the figure's bottom left. In the former representation, the spreadsheet's data, physical layout, and logical structure are intermingled in one format. In the latter representation, the user's model is expressed in a platform-independent language, resulting in a clear and succinct description of the model's underlying structure. It turns out that there are two ways to construct such spreadsheet schemas in FRL. First, one can define a new schema directly, using a text editor. Alternatively, one can extract a schema from an existing spreadsheet through a *factoring algorithm*, which is the subject of the next section.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

The Factoring Process: from Physical to Logical

This section describes the process through which a conventional spreadsheet can be factored into its four principal components: *Model*, *Data*, *Editorial*, and *Binding*. The process is based on a minimal set of user-supplied specifications regarding the characteristics of the one or more functional relations that make up the spreadsheet model. These specifications set the stage for a nine-step reduction algorithm that requires no additional human intervention.

The outlining process: Any spreadsheet can be viewed as a collection of functional relation *candidates*. A relation-candidate is a continuous block of cells that represents either a singular or a repetitive entity in the model's realm. The block may be a rectangle, a row, a column, or a single cell. In the P&L spreadsheet, for example, block `[B2..E2]` is a relation-candidate that records all the model's assumptions. Although each individual assumption cell and subsets thereof are also relation candidates, it is reasonable to assume that the entire assumptions block will be manipulated as one unit, as in moving it around the spreadsheet or changing its spatial orientation from a row vector to a column vector. In a similar vein, block `[B9..G16]` is also a relation-candidate, representing sales and expense figures for several years – a repeating pattern in the model's realm. Clearly, the task of identifying a 'good' set of relation candidates is semi-structured. Although several rules may be used to guide the process, and even automate it to a certain extent, the final decision as to which relations to employ should be left to the discretion of a human designer, as is normally done in designing ordinary data models.

For each relation that has been identified, the user has to specify a name, a spatial orienta-

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

tion (horizontal or vertical), and a physical scope. Since each relation occupies a rectangular subset of spreadsheet cells, the act of scope specification is essentially the same as defining a block (range) in a conventional spreadsheet program, i.e. anchoring the cursor at the block's origin and painting a rectangular area on the screen. If the relation's scope consists of a single row or a single column, the user is asked to specify whether the relation is deigned to store a single tuple, in which case it is denoted a *vector relation.* For each of the relations thus specified, the user is asked to name the relation's attributes and designate a key attribute. In the case of a vector relation, a key attribute is not necessary.

This process of superimposing a relational structure on the spreadsheet grid is denoted hereafter *outlining.* As it turns out, the outlining process completes the human's role in the spreadsheet factoring task. That is, once a spreadsheet has been outlined, its four principal components can be extracted automatically by successive manipulations of its respective map. Recall that the spreadsheet map is a linear list that gives the addresses and definitions of all the active cells in the spreadsheet. For example, Figure 3 depicts the map of the *outlined* P&L spreadsheet. The right hand side of the figure contains the standard spreadsheet map, produced in this particular case by the `Lotus` program. The left hand side of the figure contains *entry labels*, obtained from the spreadsheet's outline through the following matching rule. If a cell falls inside the scope of a named relation (e.g. `C11`, which is inside `p`'s outline – see Figure 2), it must sit in the intersection of a named attribute (`lease`), and a keyed tuple (`1992` – or tuple number 2 in `p`). In that case, the respective map entry of the cell is labeled `p[2].lease`. If a spreadsheet cell doesn't fall inside the scope of any relation outline, it's map entry is left unlabeled. A comparison of the outlined spreadsheet from Figure 2 and its respective map in Figure 3 might help in tracking the labels generation rule.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

Put Figure 3 around here

In general, then, the labels that identify the spreadsheet map entries have the form $r[i].x$, where r is a relation name, i is a tuple index, and x is an attribute name. The map that emerges from this labeling procedure conveys two types of information. First, it subsumes all the information contained in the original spreadsheet. Second, it offers all the meta-information necessary to factor the spreadsheet into its four principal components: model ($\mathcal{M}$), *Data* ($\mathcal{D}$), *Editorial* ($\mathcal{E}$), and *Binding* ($\mathcal{B}$). This partitioning is done through a series of nine steps that may be described as follows: $\texttt{map}_i = \texttt{step}_i(\texttt{map}_{i-1}), i = 1, \ldots, 9$. The input of the process – $\texttt{map}_0$ – is the physical spreadsheet map, as produced by the host spreadsheet program. The output of the process – $\texttt{map}_9$ – is the spreadsheet's FRL schema. In each step of this transformation, the map is reduced and rewritten in a gradual fashion, extracting the components $\mathcal{E}$, $\mathcal{B}$, $\mathcal{D}$, and $\mathcal{M}$, in that order, along the way.

Extracting the Editorial and the Binding Components: The extraction of the $\mathcal{E}$ and $\mathcal{B}$ components is straightforward. First, all the non-labeled entries are extracted from the map and archived together under the name $\mathcal{E}$. This list of cell definitions, which constitutes the *Editorial* component of the spreadsheet, contains such information as titles, attribute headings, and general documentation. The labeled entries that *remain* in the map after $\mathcal{E}$ has been extracted have the following general form:

```
r[i].x  cell-address:  [formatting-specs]  cell-definition
```

The `formatting-specs` are optional. In order to extract the *Binding* component from the map, each of these map entries is split into two types of entries, as follows:

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

'Binding entry:' `r[i].x: cell-address [formatting-specs]`

'Definition entry:' `r[i].x: cell-definition cell-address`

Taken together, the binding entries form the *Binding* component of the model, which is essentially a list of instructions regarding the physical placement and formatting specifications of each of the model's attributes. This list is archived under the name $\mathcal{B}$. The remaining 'definition entries' are then passed on to the next stage in the factoring process.

Extracting the Data Component: The extraction of the *Data* component of the spreadsheet is also a simple procedure, and therefore it will be discussed here only in broad terms. The procedure involves building a set of relations to accommodate all the constant values from the spreadsheet map. Note that at this point of processing, the map consists of two types of entries: `r[i].x: constant cell-address` and `r[i].x: formula cell-address`. Since the entry labels $r[i].x$ provide all the relation names and attribute names (as defined by the user), the construction of the relational structures that they imply is a straightforward task. Once these relations have been constructed, the constants and formulae that correspond to each $r[i].x$ label are pegged into their proper slots in the relations, using the labels as pointers. The constants are copied verbatim, and the formulae definitions are replaced with the special symbol `C`, denoting *calculated values.* The set of relations thus constructed is then archived under the aggregate name $\mathcal{D}$. If this procedure were applied to the spreadsheet map in Figure 3, it would yield the two relations depicted at the bottom right of Figure 2.

Extracting the Model Component: After the $\mathcal{E}$, $\mathcal{B}$, and $\mathcal{D}$ components of the spread-

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

sheet have been extracted, several steps are taken to prepare the map for further processing. First, the constants that were previously stored in $\mathcal{D}$ are removed from the map. Next, physical cell addresses are substituted with their respective attribute labels. For example, the map entry `p[1].cogs B10: +B9*$D$2` (Figure 3) is rewritten as `p[1].cogs: p[1].sales*a[1].cogs`, because `p[1].sales` and `a[1].cogs` are the entry labels of the cells `B9` and `D2`, respectively, in the spreadsheet map. Formally, we have the following steps:

F1: Rewrite all entries of the form

```
r[i].x: constant cell-address    as:
r[i].x: data-type cell-address
```

where `data-type` is the type of the `constant`.

F2: Rewrite all entries of the form

```
r[i].x:  formula  cell-address    as:
r[i].x:  formula' cell-address
```

where `formula'` is the same as `formula`, except that all physical cell addresses are replaced with their respective entry labels from the map. If a physical cell address is fixed (preceded by a $ prefix), fix the tuple index in its respective label as well.

F3: Rewrite all the entries that emerge from `F1-F2`:

```
r[i].x:  definition  cell-address  as:
r[i].x:  definition
```

i.e. eliminate the `cell-address` from all the map entries.

F4: Sort the map by the entry labels `r[i].x`, as follows. Primary sort key: relation name (`r`). Secondary sort key: attribute name (`x`). Ternary sort key: tuple index (`i`).

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

The data structure that emerges from F4 is denoted the spreadsheet's *logical map*. The logical map of the P&L spreadsheet is shown in Figure 4, and the reader may want to compare it to the physical map in Figure 3 in order to track the execution of steps F1-F4.

Put Figure 4 around here

Due to the sorting operation (F4), the logical map becomes a sequence of attribute clusters, each cluster being an ordered set of entries whose labels consist of the same relation prefix r and the same attribute name x. In what follows, these sets of entries are denoted $r.x$ – *clusters*. For example, the a.tax-cluster of the P&L spreadsheet map consists of one entry, whereas the p.sales-cluster consists of 6 entries, as follows:[4]

```
p[1].sales:  numeric
p[2].sales:  p[1].sales*(1+a[$1].grate)
p[3].sales:  p[2].sales*(1+a[$1].grate)
p[4].sales:  p[3].sales*(1+a[$1].grate)
p[5].sales:  0.5*(p[3].sales+p[4].sales)*1.2
p[6].sales:  0.5*(p[4].sales+p[5].sales)*1.2
```

Note that the cluster is made up of three sets of *isomorphic* entries – entries that convey exactly the same mathematical relationship, albeit with different indices. The goal of the next step in the algorithm is to discover and tag such isomorphic entries, and, in the process, replace absolute tuple references with relative references[5].

[4]Due to space limitations, the maps in Figures 3 and 4 correspond only to years 1992–1994 in the spreadsheet. At the same time, the p.sales-cluster described above is taken from the map of the entire spreadsheet, i.e. for years 1992-1997.

[5]Notational comment: in what follows, i, j, and d represent numbers, whereas n is a textual tag, i.e. the character 'n'.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

F6: Each *r.x-cluster* in the map contains 0 or more sets of repetitive map entries, i.e. entries that have exactly the same right hand side definition. In each set of repetitive entries, eliminate all but the first entry in the set (the entry with the lowest index).

When step `F6` is applied to the `p.sales-cluster`, the third, fourth, and sixth entries of the cluster are erased, leading to the following cluster:

```
p[1].sales:  numeric
p[2].sales:  p[n-1].sales*(1+a[1].grate)
p[5].sales:  0.5*(p[n-2].sales+p[n-1].sales)*1.2
```

This cluster conveys the following information: In the first tuple of the `p` relation, the attribute `sales` is a numeric constant. In tuple numbers $n = 2$, $n = 3$, and $n = 4$, it should be bound to the expression `p[n-1].sales*(1+a[1].grate)`. In tuple number $n = 5$ and thereafter, it should be bound to the expression `0.5*(p[n-2].sales+p[n-1].sales)*1.2`. The next step in the algorithm makes this binding explicit through a series of cluster rewriting rules.

F7: Comment: at this stage of processing, all the $r.x$ clusters in the map contain only unique entry definitions. Let the entry labels of a cluster be $r[k_1].x, r[k_2].x, \ldots, r[k_m].x$. Rewrite these entry labels as $r[k_1 \leq n < k_2].x, r[k2 \leq n < k_3].x, \ldots, r[n \geq k_m].x$. If a rewritten entry label becomes $r[i \leq n < i+1].x$ for some i, rewrite it again as $r[n = i].x$. Finally, if the cluster consists of only *one* entry, rewrite it again as $r[n].x$

When step `F7` is applied to the `p.sales-cluster`, the cluster changes to:

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

```
p[n=1].sales:       numeric
p[2<=n<5].sales:    p[n-1].sales*(1+a[1].grate)
p[n>=5].sales:      0.5*(p[n-2].sales+p[n-1].sales)*1.2
```

If steps F6 and F7 were applied to *all* the attribute clusters in the *entire* P&L spreadsheet map (Figure 4), they would yield the map in Figure 5. In the next and final two steps of the algorithm, this map is transformed into a formal spreadsheet schema, consistent with FRL's syntax:

F8: Use FRL's syntax conventions and default rules to abbreviate the attribute references generated by F6-F7 as much as possible.

F9: Consult the original outline of the spreadsheet to obtain the following specifications for each relation: (i) the relation's *full name*; (ii) the relation's *cardinality* (single tuple vs multiple tuples); and (iii) the relation's *key*. Use these specifications and the syntax rules of FRL to transform the map obtained from F7 into a formal spreadsheet schema.

Put Figure 5 around here

When step F8 is applied to the map in Figure 5, the map entry `p[n].cogs: p[n].sales* a[1].cogs` is transformed into the attribute definition `cogs: sales*a.cogs`. Likewise, the map entry `p[n].tax: @IF(p[n].inc>0,p[n].inc*a[1].tax,0)` becomes the attribute definition `tax: IF(inc>0,inc*a.tax,0)` (these are just two representative examples). Taken together, steps F8-F9 convert the map from Figure 5 to the schema in the bottom left of Figure 2, which constitutes the $\mathcal{M}$ component of the P&L spreadsheet.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

The Synthesis Process: from Logical to Physical

The previous section described a top-down factoring process that splits physical spreadsheets into their four principal components. This section describes a the reverse operation – *synthesis* – in which executable spreadsheets are built bottom-up from reusable components. The key player in both processes is the schema, or the $\mathcal{M}$ component, of the underlying spreadsheet. In the factoring algorithm, $\mathcal{M}$ is the major output of the process; in the synthesis algorithm, it is the major input.

It is important to note that even though the schema is not an executable entity, it contains all the necessary *instructions* for constructing operational spreadsheets. This construction occurs through a synthesis process that converts a given schema into a physical spreadsheet that can be executed on a target spreadsheet program. More precisely, the synthesis process is designed to 'mix' different model components, $\mathcal{M}$, $\mathcal{D}$, $\mathcal{B}$, and $\mathcal{E}$, in order to produce different variants of the same generic spreadsheet.

To illustrate, suppose that a certain spreadsheet has been factored into its four principle components $\mathcal{M}$, $\mathcal{D}$, $\mathcal{B}$, and $\mathcal{E}$. The synthesis of all four components, denoted $\mathcal{M}+\mathcal{D}+\mathcal{B}+E$, yields a fully operational and executable spreadsheet that is identical to the original, unfactored, spreadsheet. If the *Data* component is left out of the synthesis, the combination $\mathcal{M}+\mathcal{B}+\mathcal{E}$ yields a *spreadsheet template* – a data independent model structure that can be instantiated with a variety of different data sets, or modeling scenarios. Specifically, two spreadsheets of the form $\mathcal{M}+\mathcal{B}+\mathcal{E}+\mathcal{D}$ and $\mathcal{M}+\mathcal{B}+\mathcal{E}+\mathcal{D}'$ that differ only in their *Data* component are said to be different *instances* of the same generic spreadsheet. This will be the case, for example, when different divisions are required to use the same

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

spreadsheet template to produce P&L statements that conform to a certain organiztional reporting standard.

Other combinations of the four components are equally instructive. For example, consider the two spreadsheet $\mathcal{M}+\mathcal{D}+\mathcal{B}+\mathcal{E}$ and $\mathcal{M}+\mathcal{D}+\mathcal{B}'+\mathcal{E}'$, that differ only in their *Binding* and *Editorial* components. Note that even though the two spreadsheets are physically different, they are completely isomorphic in terms of logical structure and data contents. For example, $\mathcal{E}$ and $\mathcal{E}'$ can be the English and Spanish versions of the same spreadsheet. In a similar vein, $\mathcal{B}$ and $\mathcal{B}'$ might be alternative screen layouts of the same model, a useful distinction when two users wish to present or print the same spreadsheet in two different ways.

The type of component manipulation that was described above already occurs in industry, albeit in an informal and haphazard fashion. For example, suppose that a junior loan officer (Joe) wants to analyze loan applications with a spreadsheet model created by an experienced colleague (Jane). For Joe, the easiest way to adopt Jane's spreadsheet is to *clone* it. This is commonly done by copying Jane's spreadsheet, erasing all its constant cells (implicit *Data* component), and retaining all its formulae cells (implicit *Model* component). Once the spreadsheet has been emptied from Jane's data, it can be loaded with Joe's data, at which point Joe and Jane apply the same model to two different data sets. Yet in spite of this logical proximity, a conventional spreadsheet program will treat the two spreadsheet as unrelated physical entities. Therefore, when Jane changes her spreadsheet to fix an error or accommodate a new credit rule, the change will not propagate to Joe's spreadsheet.

We see that when spreadsheets are shared and reused in an informal manner, maintenance and extension efforts must be duplicated. Had a formal framework existed for spreadsheet

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

models management, this duplication could be minimized. For example, if Joe wants to clone Jane's spreadsheet, the safest way to do it is to (i) factor Jane's spreadsheet into its four principal components, and (ii) synthesize Jane's $\mathcal{M}$, $\mathcal{E}$, and $\mathcal{B}$ components with Joe's $\mathcal{D}$ component. If and when Jane changes her spreadsheet (or, more accurately, the $\mathcal{M}$ component of her spreadsheet), the revised spreadsheet of Jane can be refactored and resynthesized with Joe's data. The new data can be either taken from a file, or added interactively to the spreadsheet template.

Since synthesis is the converse of factoring, it traces the factoring steps backwards. The input of the algorithm is an $\mathcal{M}$ component, i.e. a spreadsheet schema written in FRL, and optional $\mathcal{D}$, $\mathcal{E}$, and $\mathcal{B}$ components. The output of the process is an operational spreadsheet that can be executed on a host spreadsheet program. Synthesis involves three main stages. In the first stage, the schema is converted into a logical map like the one depicted in Figure 4. At this point, the user has two options. If the goal is to load the spreadsheet with stored data, the map can be merged with a given $\mathcal{D}$ component. If the goal is to create a spreadsheet *template*, the map can be merged with a 'cloned' data set that is consistent with the map's structure. Next, the logical map is transformed into a physical spreadsheet by synthesizing it with *Binding* and *Editorial* components. These components can be taken from a documentation library or added interactively by the user.

Transforming the schema to a logical map: Recall that all the attribute references that appear in the schema were abbreviated as much as possible, using FRL's default rules. For example, the attribute definition `cogs: sales*a.cogs` is shorthand of `p[year].cogs: p[year].sales*a[1].cogs`. In the first step of the synthesis algorithm, all the defaulted attribute references are expanded to their fully specified references:

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

S1: Using FRL's syntax rules and default conventions, rewrite all the attribute references that appear in the schema in an extended (no defaults) FRL syntax.

Step S1 generates extended attribute references of the form $r[key].x$, $r[prev(\ldots prev(key)\ldots)].x$, or $r[next(\ldots next(key)\ldots)].x$. The next step in the synthesis algorithm converts all key-based tuple references to index-based tuple addresses:

S2: Rewrite each attribute reference of the form $r[key].x$ as $r[\mathrm{n}].x$, each attribute reference of the form $r[prev(\ldots prev(key)\ldots)].x$ were *prev* appears i times as $r[\mathrm{n}-i].x$, and each attribute reference of the form $r[succ(\ldots succ(key)\ldots)].x$ were succ appears i times as $r[\mathrm{n}+i].x$.

When steps S1-S2 are applied to the schema from Figure 2, they yield the map shown in Figure 5. Focusing once again on the p.sales-cluster, the result will be as follows:

```
p[n=1].sales:     numeric
p[2<=n<5].sales:  p[n-1].sales*(1+a[1].grate)
p[n>=5].sales:    0.5*(p[n-2].sales+p[n-1].sales)*1.2
```

In the next step, each map entry is expanded, i.e. repeated for all the tuples that it covers. In order to carry out this expansion, we have to know the cardinality (number of tuples) of each relation in $\mathcal{D}$. If the user wants to synthesize $\mathcal{M}$ with a given $\mathcal{D}$, this is a simple lookup. Alternatively, if a $\mathcal{D}$ component is not available, a *cloned* (generic) version of $\mathcal{D}$ can be constructed from $\mathcal{M}$. In the latter case, the outcome of the synthesis process will be a spreadsheet template to which the user can add data interactively. The cloning process, which is straightforward, is described later in this section.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

S3: Let n_r be the cardinality of r, i.e. the number of tuples that are presently stored in the relation r.

Replace each map entry of the form `r[n].x: definition`,
with a series of n_r map entries of the form
`r[1].x: definition` ,..., `r[`n_r`].x: definition`.

Replace each map entry of the form `r[`$k_1 \leq$`n<` k_2`].x: definition`,
with a series of $k_2 - k_1 + 1$ map entries of the form
`r[`k_1`].x: definition` ,..., `r[`$k_2 - 1$`].x: definition`.

Replace each map entry of the form `r[n`$\geq k$`].x: definition`,
with a series of $n_r - k + 1$ map entries of the form
`r[`k`].x: definition` ,..., `r[`n_r`].x: definition`.

Replace each map entry of the form `r[n=`i`].x: definition`,
with a *single* map entry of the form
`r[`i`].x: definition`.

To illustrate, step S3 expands the `p.sales`-cluster into the following cluster:

```
p[1].sales:  numeric
p[2].sales:  p[n-1].sales*(1+a[1].grate)
p[3].sales:  p[n-1].sales*(1+a[1].grate)
p[4].sales:  p[n-1].sales*(1+a[1].grate)
p[5].sales:  0.5*(p[n-2].sales+p[n-1].sales)*1.2
p[6].sales:  0.5*(p[n-2].sales+p[n-1].sales)*1.2
```

At this point of processing, the right hand side definitions of each cluster contain two kinds of tuple addressing: relative and absolute. Relative addressing is characterized by the presence of the symbol n, as in r[n].x, r[n + j].x, or r[n − j].x, for some j. Absolute addressing consists of constant tuple references, as in $r[j].x$ for some j. The next step in

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

the synthesis algorithm marks absolute tuple references by prefixing them with the special character \$.

S4: Throughout the map, rewrite all attribute labels of the form $r[j].x$ as $r[\$j].x$.

Next, relative tuple references (those that are not prefixed by \$) are converted to their corresponding tuple numbers:

S5: Let $r[i].x$ be the label of a map entry, let $r[\mathrm{n}+j].y$ be a *related* attribute label in the entry's *definition* (i.e. a label with the same relation prefix), and let $d = i+j$ (note: j may be either negative, zero, or positive). Rewrite the related attribute label as $r[d].y$. Repeat this operation for each map entry whose *definition* part contains attribute labels that are related to the entry's label.

When applied to the schema of the P&L spreadsheet, steps `S4-S5` will yield the logical map shown in Figure 4.

Adding the Data Component: The data element of a synthesized spreadsheet can come from three alternative sources:

- a *stored* $\mathcal{D}$ component;
- a *generic* $\mathcal{D}$ component;
- a *user-supplied* $\mathcal{D}$ component

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

In the first alternative, $\mathcal{M}$ is synthesized with a given $\mathcal{D}$ component taken from a database. It is assumes that $\mathcal{D}$ is either the originally (but possibly modified) factored *Data* component of the spreadsheet, or another set of relations that passed an applicability test indicating that they are compatible with $\mathcal{M}$'s structure. In the second and third alternatives, $\mathcal{M}$ is synthesized with a *dummy Data* component $\mathcal{D}_{\mathcal{M}}$ which is generated from the schema $\mathcal{M}$. The synthesis $\mathcal{M} + \mathcal{D}_{\mathcal{M}}$ produces a spreadsheet template that can be either left as is, or populated with data that is entered by the user at the spreadsheet program level (following synthesis).

Instead of providing a separate algorithm for template generation, we note that a *dummy Data* component $\mathcal{D}_{\mathcal{M}}$ can be easily generated for each given schema $\mathcal{M}$. The $\mathcal{D}_{\mathcal{M}}$ component is a collection of dummy relations consisting of *filler* values. A dummy relation $r \in \mathcal{D}_{\mathcal{M}}$ is constructed from a relation schema $s \in \mathcal{M}$ through the following straightforward process.

First, the number of filler tuples in r must be determined. Recall that the right hand side of the `case` construct of `FRL` consists of expressions of the form `condition` $\mapsto$ `definition`. Further, the `condition` parts always make references to tuple numbers. Now, if the relation schema s contains no `case` constructs, the number of dummy tuples in r is set to one. If s contains one or more `case` constructs, the number of dummy tuples in r is set to *one plus the highest tuple number referred to in the* `condition` *part of any one of the* `case` *constructs in* s. For example, consider Figure 2, where the schemas of `assumptions` and `proforma` consist of 0 and 1 `case` constructs, respectively. In the latter relation schema, the highest tuple number in the `case` construct is 5. Therefore, the dummy relations corresponding to `assumptions` and to `proforma` will contain 1 and 6 filler tuples, respectively. Note that the dummy `proforma` relation will contain 6 tuples irrespective of how many 'real' tuples

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

the data relation `proforma` actually contains in $\mathcal{D}$.

Once the number of dummy tuples has been determined, the dummy relations are 'populated' with filler data through the following straightforward process. If a data attribute x in s is of type `numeric`, `string`, `date`, or `logical`, the filler character `N`, `S`, `D`, or `L`, respectively, is placed as the attribute value of x in the dummy relation r. Functional attributes are represented through the special character `C`, standing for *calculated value.* The collection of all the dummy relations thus constructed forms the dummy $\mathcal{D}_{\mathcal{M}}$ component.

Next, the $\mathcal{D}$ component, be it 'real' or 'dummy,' is merged with the 'data entries' in the spreadsheet's logical map. The data entries can be easily identified by focusing on the map entries of the form `r[i].x data-type` where `data-type` is either `numeric`, `string`, `logical`, or `date` (see Figure 4). In order to populate these entries with data, the `data-type` of each entry is substituted with a constant value which is retrieved from $\mathcal{D}$ according to the pointer $r[i].x$. Note that this pointer specifies the relation name, tuple index, and attribute slot, where the constant value resides in $\mathcal{D}$.

Adding the Editorial and the Binding Components: The synthesis of $\mathcal{M}+\mathcal{D}$ with $\mathcal{E}$ and $\mathcal{B}$ is mainly an implementation issue which is of little theoretical interest. We describe it here in broad terms, noting that the reader can skip this section without losing the thread of the paper.

In what follows, it is assumed that $\mathcal{D}$, $\mathcal{M}$, $\mathcal{E}$, and $\mathcal{B}$ are the original components that were factored from the spreadsheet. If the structure of any of these components has been modified after factoring, $\mathcal{E}$ and $\mathcal{B}$ may not be compatible with $\mathcal{M}+\mathcal{D}$. As it turns out however, this is not a major problem. First, the $\mathcal{E}$ component can be modified to match the

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

modified $\mathcal{M}+\mathcal{D}$. Second, a new (default) binding $\mathcal{E}$ can be easily generated for $\mathcal{M}+\mathcal{D}$. For the sake of brevity, we will not describe the implementation details of these adjustments.

In the next two steps of the synthesis process, the logical map is 'joined' with the *Binding* component (using the map entries as the matching criterion). The result of the join is a physical spreadsheet map.

S6: For each logical map entry of the form

`r[i].x  definition`

and a corresponding binding entry (element of $\mathcal{E}$) of the form

`r[i].x:  cell-address  [formatting-specs]`

create a physical map entry of the form

`r[i].x:  cell-address  [formatting-specs]  definition`

S7: The `definition` part of each physical map entry is either a constant value, or a `formula`. If it is the latter, replace the `formula` part with `formula'`, where `formula'` is the same as `formula`, except that each attribute label $r[i].x$ that appears in `formula` is substituted with its corresponding `cell-address`, as obtained from the physical map entry whose entry label is $r[i].x$.
Following these substitutions, the labels of the map entries $r[i].x$ are no longer necessary. Rewrite each map entry of the form

```
r[i].x:  cell-address  [formatting-specs]  definition
as:
         cell-address  [formatting-specs]  definition
```

The next and final step of the synthesis process merges $\mathcal{M}+\mathcal{D}+\mathcal{B}$ with $\mathcal{E}$. Recall that the *Editorial* $\mathcal{E}$ component is simply a list of entries of the form `cell-address string`.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

components in any given spreadsheet: *Model*, *Data*, *Editorial*, and *Binding*. The four components and the algorithms that operate on them are presented in Figure 6. In the figure, the area above the factoring/synthesis bubble corresponds to the physical realm of commercial spreadsheet programs, along with their appealing and intuitive user interfaces. The area below the bubble corresponds to a logical realm in which spreadsheet models are viewed as modular amalgamations of generic components that can be constructed in different ways under the user's control. The two-way transition between the physical and the logical views is made possible by the factoring and synthesis algorithms.

Put Figure 6 around here

Beginning with the physical realm, note that our approach is complementary to the standard practice of spreadsheet modeling. That is, we assume that users will continue to build spreadsheets via conventional spreadsheet programs like `Lotus`, `Excell` and `Quattro`. Once implemented, though, such spreadsheets can be outlined and then translated into spreadsheet maps by the host spreadsheet program. Next, the factoring algorithm can be invoked to decompose the maps into their respective principle components. The reverse direction, from generic components to conventional spreadsheets, is handled by a symmetric synthesis algorithm which assists users in the construction of executable spreadsheets from reusable objects. Both algorithms make extensive use of `FRL` – a specialized data definition formalism for spreadsheet models.

What are the benefits of this dual perspective on spreadsheet modeling? As Figure 6 illustrates, once the four components have been extracted from the spreadsheet's physical representation, they can be stored and managed in separate repositories which are *inde-*

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

pendent of spreadsheet programs. Most importantly, *Model* components can be channeled to and managed by a model management system that supports model documentation, retrieval and reuse. Likewise, *Data* components can be archived and accessed via a database management system that offers all the flexibility and power of a general-purpose DBMS.

The $\mathcal{E}$ and $\mathcal{B}$ components, which are of lesser theoretical importance, are placed in a separate documentation library. This way, a user with no spreadsheet experience can translate a spreadsheet from English to Spanish (or, say, check its spelling) by operating directly on its *Editorial* component, which is essentially a list of textual labels implemented as an ASCII file. Similarly, the screen layout of a spreadsheet can be manipulated by operating on its underlying *Binding* component, for example if one wants to protect the spreadsheet's $\mathcal{M}$ and $\mathcal{D}$ components from operations that pertain to appearance only.

Hence, the dual perspective has both 'micro' and 'macro' implications for the standard practice of spreadsheet modeling. At the micro level, for example, the components' modularity enables us to distinguish between different types of spreadsheet manipulations. *Neutral* manipulations, like transposing or moving relations around the screen, effect neither the $\mathcal{M}$ nor the $\mathcal{D}$ components of the spreadsheet. *Data* manipulations, like adding or deleting rows or columns that correspond to repetitive tuples, effect only the spreadsheet's $\mathcal{D}$ component, leaving the $\mathcal{M}$ component intact. *Structural* manipulations, like adding or deleting rows and columns that correspond to attributes, effect both the $\mathcal{M}$ and the $\mathcal{D}$ components of the spreadsheet. The key point is that once the component modularity of spreadsheets is explicitly recognized by the host modeling environment, an intelligent modeling 'assistant' could be designed to sense from the physical spreadsheet what the user is trying to do in the way of building *logical* models that interact with corporate repositories of models and data.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

At the 'macro' level, the dual perspective redefines the conventional notion of spreadsheets in such a way that makes them accessible to other, non-spreadsheet software environments. This opens new and exciting possibilities for integrating spreadsheet-, data-, and model-management systems in novel ways that were previously unfeasible.

Conclusion This paper presents the conceptual framework, data definition language, and factoring and synthesis algorithms, necessary to take spreadsheet modeling one step further beyond the present 'state of the art.' Specifically, we provide a foundation for two important developments: (i) building intelligent spreadsheet programs that 'understand' the model world of the user; and (ii) building powerful spreadsheet model management systems that help manage and streamline huge repositories of spreadsheets as well-organized corporate resources. Our objective is to use this foundation as a point of departure for future research in these directions.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

Appendix: FRL - The language in BNF form

The FRL data definition language for functional-relational schemas is described next in BNF form. The applicable constraints are given after the BNF description.

Syntax

```
Model_Schema      ::= R_schema |
                      R_schema Model_Schema

R_schema          ::= R_def Key_Attr_descr |
                      R_def Key_Attr_descr Rest_Attr_descr

R_def             ::= relation R_Name alias R_alias_name |
                      R_Name alias R_alias_name (type vector)

R_Name            ::= Name

Name              ::= String

R_alias_name      ::= Letter

Key_Attr_descr    ::= Data_Attr_descr key

Rest_Attr_descr   ::= Attr_descr |
                      Attr_descr Rest_Attr_descr

Attr_descr        ::= Data_Attr_descr |
                      Func_Attr_descr

Data_Attr_descr   ::= Attr_name :  Type

Type              ::= number | string | date | logical
```


Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

```
Attr_name        ::= Name

Func_Attr_descr  ::= Attr_name :  Expr

Expr             ::= Simple_Expr
Expr             ::= Case_Expr

Case_Expr        ::= Boolean_Cond ↦ Simple_Expr |
                     Boolean_Cond ↦ Simple_Expr Case_Expr

Boolean_Cond     ::= n Comparator NUM |
                     NUM < n ≤ NUM

Simple_Expr      ::= Type |
                     Constant |
                     Reference |
                     If_Expr

Constant         ::= NUM | STRING | DATE | LOGICAL

Reference        ::= R_alias_name[Ref].Attr_name

Ref              ::= Num_expr | Attr_expr

Num_expr         ::= #n | Num_expr + 1 | Num_expr − 1

Attr_expr        ::= Attr_name | FUNC(Attr_exp)

FUNC             ::= next | prev | glb | lub

If_Expr          ::= IF( Bool_Cond, Simple_Expr,Simple_Expr)

Bool_Cond        ::= Reference Comparator Reference
Comparator       ::= < | ≤ | = | > | ≥
```


Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

```
NUM             ::= numeric constants
STRING          ::= string constants
DATE            ::= date constants
LOGICAL         ::= logical constants
```

Constraints

In the main body of this article we explained how to synthesize a functional relation into a logical map of a spreadsheet. This, in a sense provides an *operational semantics* for functional relations. Although we do not define a precise declarative semantics for the functional relational model, we provide the following explanations on a semi-formal level.

1. *Types.* Although the language is not typed, it is simple to obtain a strongly typed language by assigning types to the different spreadsheet functions and enforcing typing at the language definition level. We have chosen the untyped version of the language for the sake of brevity.

2. *Keys and orderings.* We assume that each relation has a key in the database sense, i.e., there is an attribute `x` of `r` such that no two tuples of `r` have the same value for `x`.

   In addition, we require that the domains of key attributes (the sets of values that the attributes can attain) be totally ordered. That is, there is a relation $<$ defined on the domain D_k of a key `k`, such that $<$ is asymmetric and transitive, and that for any two elements v_1, v_2 of D_k, either $v_1 < v_2$ or $v_2 < v_1$.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

The ordering among the keys of r induces an ordering on the tuples of r as follows. Let k be the key of relation r, and let t_1, t_2 be tuples of r, then

$$t_1 < t_2 \quad \text{iff} \quad t_1.k < t_2.k$$

3. *Successors and predecessors.* Since each relation contains only finitely many tuples, we can define the notions of *immediate predecessor* and *immediate successor* as follows.

   Let $t_1, \ldots, t_n$ be all the tuples in a relation r, ordered by their keys. Then for $1 \leq i < n$, t_i *immediately precedes* t_{i+1} (denoted $t_i <<_r t_{i+1}$,) and t_{i+1} *immediately succeeds* (denoted $t_i\ t_{i+1} >>_r t_i$.)

   Hence, it makes sense to talk about the *next* or *previous tuple*, and about the tuple *closest from below* to a certain value v in the domain D_k (i.e., the tuple with key $glb_r(v) = max(t.k | t \in \mathtt{r} \text{ and } t.k < v)$); and of the tuple *closest from above*, i.e. the tuple with key $lub_r(v) = min(t.k | t \in \mathtt{r} \text{ and } t.k > v)$.

   Note that the notion of immediacy depends on the relation r. If r and r' are two relations with the same schema but different data, it might happen that a tuple t immediately precedes a tuple t' in r, but not in r'.

4. *References.* These are of the general form r[ref].x. Each reference appears in the definition of an attribute y in a relation q. The attribute y is called "the owning attribute" and q the "the owning relation". Intuitively, r[ref].x is a reference to the value of attribute x in the tuple of relation r whose key value is ref. (Note that x has to be an attribute of r.)

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

There are three kinds of `refs`:

(a) *Absolute:* denoted by `#i`, where `i` is a number. This is a reference to the $\mathtt{i}^{th}$ tuple of `r`, in the order of the keys.

(b) *Relative:* denoted by an expression of the form `#n`, or `#n` $\overset{+}{_{-j}}$. The interpretation of `#n` is the *current tuple* of `r`, `#n`$-j$ is the j^{th} previous tuple, and `#n`$+j$ is the jt^{th} next tuple, as defined above.

(c) *Named Attribute:* denoted by an attribute name `z`, or an expression involving `z` and the functors `prev, next, lub, glb`. A reference `r[z]` points to the tuple in `r` whose key value equals the value of the `z` attribute in the current tuple. References with `prev, next, lub, glb` are interpreted by the *immediate predecessor*, *immediate successor*, $glb_r(v)$ and $lub_r(v)$ functions described above.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

References

[1] Robert Blanning, Andrew Whinston, Vasant Dhar, Clyde Holsapple, Mathias Jarke, Stephen Kimbrough, Javier Lerch, and Michael Prietula. Precis of Model Management and the Language of Thought Hypothesis. In Edward A. Stohr, editor, *Proceedings ISDP-89*, 1989.

[2] Narain H. Gehani. Databases and Units of Measure. *IEEE Transactions on Software Engineering*, SE-8(6):605–611, November 1982.

[3] Arthur M. Geoffrion. An Introduction To Structured Modeling. *Management Science*, 33(5):547–588, May 1987.

[4] Arthur M. Geoffrion. The SML Language For Structured Modeling: Levels 1 and 2. Western management science institute working paper, UCLA School of Management, Western Management Science Institute, School of Management, University of California, Los Angeles, CA 90024, April 1991.

[5] Arthur M. Geoffrion. The SML Language For Structured Modeling: Levels 3 and 4. Western management science institute working paper, UCLA School of Management, Western Management Science Institute, School of Management, University of California, Los Angeles, CA 90024, April 1991.

[6] Seymour Ginzburg and Stephen Kurtzman. Spreadsheet Histories, Object-Histories and Projection Simulation. In *ICDT - Proceedings of the 2nd International Conference on Database Theory - Lecture Notes in Computer Science no. 326*, Berlin, 1988. Springer-Verlag.

[7] Paul Gray. *Guide to IFPS/Personal.* McGraw-Hill Book Company, 1988.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

[8] Scott Hudson and Roger King. The Cactis Project: Database Support for Software Engineering. *IEEE Transactions on Software Engineering*, June 1988.

[9] Scott Hudson and Roger King. Cactis Project: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. *ACM Transactions on Database Systems*, 14(3):291–321, September 1989.

[10] D. Knuth. Semantics of Context Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[11] David Maier. *The Theory of Relational Databases*, chapter 14, pages 533–549. Computer Science Press, 1988.

[12] Boaz Ronen, Michael Palley, and Henry C. Lucas Jr. Spreasdheet Analysis and Design. *Communications of the ACM*, 32(1):84–93, January 1989.

[13] Steven P. Wesier and Frederick H. Lochovsky. Object-Oriented Concepts, Databases and Applications. In Won Kim and Frederick H. Lochovsky, editors, *OZ+: An Object-Oriented Database System*, chapter 13, pages 309–340. ACM Press, 1989.

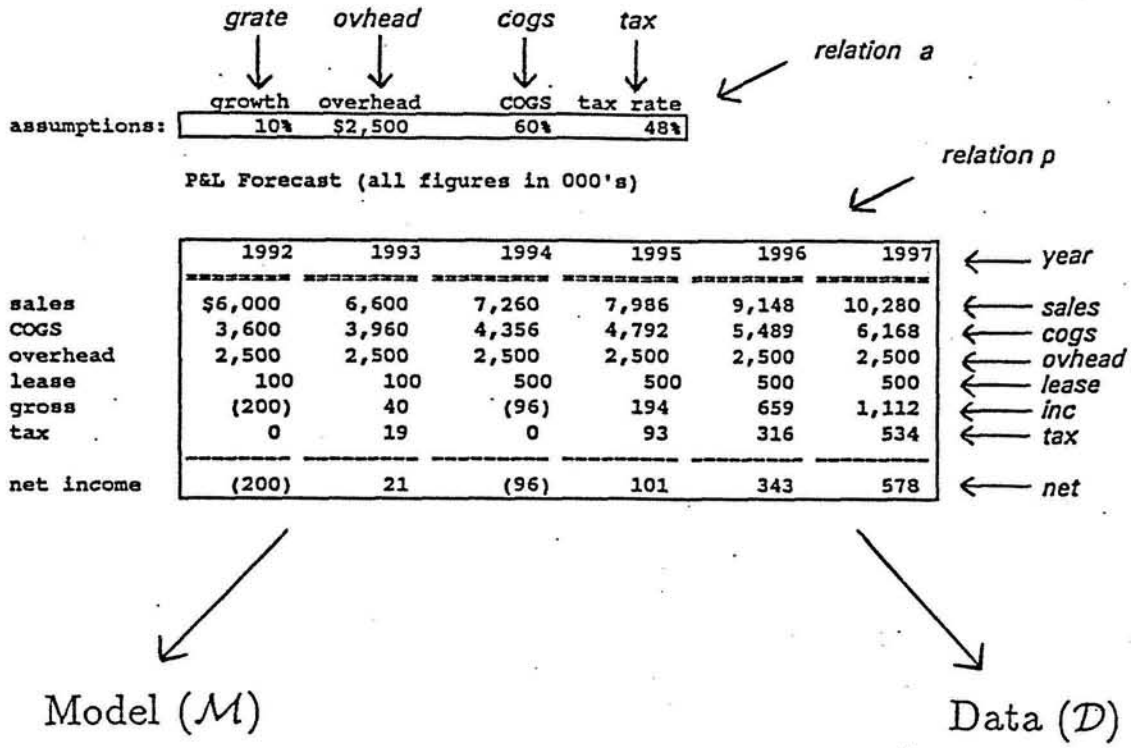
[14] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Series in Automatic Computing. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1976.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

	A	B	C	D	E	F	G
1		growth	overhead	COGS	tax rate		
2	assumptions:	10%	$2,500	60%	48%		
3							
4		P&L Forecast (all figures in 000's)					
5							
6							
7		1992	1993	1994	1995	1996	1997
8		========	========	========	========	========	========
9	sales	$6,000	6,600	7,260	7,986	9,148	10,280
10	COGS	3,600	3,960	4,356	4,792	5,489	6,168
11	overhead	2,500	2,500	2,500	2,500	2,500	2,500
12	lease	100	100	500	500	500	500
13	gross	(200)	40	(96)	194	659	1,112
14	tax	0	19	0	93	316	534
15		--------	--------	--------	--------	--------	--------
16	net income	(200)	21	(96)	101	343	578

Figure 1: The P&L Spreadsheet

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28



relation assumptions **alias** a **type** vector
- grate: numeric
- ovhead: numeric
- cogs: numeric
- tax: numeric

relation proforma **alias** p
- year: numeric key
- sales: $n = 1 \mapsto$ numeric
  $2 \leq n < 5 \mapsto p[prev(year)].sales * (1 + a.grate)$
  $n \geq 5 \mapsto 0.5 * (p[prev(year)].sales + p[prev(prev(year))].sales * 1.2$
- cogs: $sales * a.cogs$
- ovhead: $a.ovhead$
- lease: numeric
- inc: $sales - lease - cogs$
- tax: if(inc > 0, $inc * a.tax$, 0)
- net: $inc - tax$

relation a:

grate	ovhead	cogs	tax
0.1	2500	0.6	0.48

relation p:

year	sales	cogs	ovhead	lease	inc	tax	net
1992	6000	C	C	100	C	C	C
1993	C	C	C	100	C	C	C
1994	C	C	C	500	C	C	C
1995	C	C	C	500	C	C	C
1996	C	C	C	500	C	C	C
1997	C	C	C	500	C	C	C

Figure 2: The outlined P&L spreadsheet (top) and its respective Model ($\mathcal{M}$) component (left) and Data ($\mathcal{D}$) component (right).

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

```
                B1: 'growth
                C1: 'overhead
                D1: 'COGS
                E1: 'tax rate
                A2:  'assumptions:
a[1].grate      B2: (P0) 0.1
a[1].ovhead     C2: (C0) 2500
a[1].cogs       D2: (P0) 0.6
a[1].tax        E2: (P0) 0.48
                B4: 'P&L Forecast (all figures in 000's)
                B5: '---------------------------------
p[1].year       B7: 1992
p[2].year       C7: 1993
p[3].year       D7: 1994
                B8: \=
                C8: \=
                D8: \=
                A9: 'sales
p[1].sales      B9: (C0) 6000
p[2].sales      C9: (,0) +B9*(1+$B$2)
p[3].sales      D9: (,0) +C9*(1+$B$2)
                A10: 'COGS
p[1].cogs       B10: (,0) +B9*$D$2
p[2].cogs       C10: (,0) +C9*$D$2
p[3].cogs       D10: (,0) +D9*$D$2
                A11: 'overhead
p[1].ovhead     B11: (,0) +$C$2
p[2].ovhead     C11: (,0) +$C$2
p[3].ovhead     D11: (,0) +$C$2
                A12: 'lease
p[1].lease      B12: 100
p[2].lease      C12: 100
p[3].lease      D12: 500
                A13: 'gross
p[1].inc        B13: (,0) +B9-B10-B11-B12
p[2].inc        C13: (,0) +C9-C10-C11-C12
p[3].inc        D13: (,0) +D9-D10-D11-D12
                A14: 'tax
p[1].tax        B14: (,0) @IF(B13>0,B13*$E$2,0)
p[2].tax        C14: (,0) @IF(C13>0,C13*$E$2,0)
p[3].tax        D14: (,0) @IF(D13>0,D13*$E$2,0)
                B15: (,0) \-
                C15: (,0) \-
                D15: (,0) \-
                A16: 'net income
p[1].net        B16: (,0) +B13-B14
p[2].net        C16: (,0) +C13-C14
p[3].net        D16: (,0) +D13-D14
```

Figure 3: The map of the P&L spreadsheet (right hand side), annotated by user-defined attribute labels (left hand side), obtained from the spreadsheet's outline. E limitations, the map covers only years 1992, 1993, and 1994, of the origina

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

a[1].grate:	numeric	B2
a[1].ovhead:	numeric	C2
a[1].cogs:	numeric	D2
a[1].tax:	numeric	E2
p[1].year:	string	B7
p[2].year:	string	C7
p[3].year:	string	D7
p[1].sales:	numeric	B9
p[2].sales:	p[1].sales*(1+a[$1].grate)	C9
p[3].sales:	p[2].sales*(1+a[$1].grate)	D9
p[1].cogs:	p[1].sales*a[$1].cogs	B10
p[2].cogs:	p[2].sales*a[$1].cogs	C10
p[3].cogs:	p[3].sales*a[$1].cogs	D10
p[1].ovhead:	a[1].ovhead	B11
p[2].ovhead:	a[1].ovhead	C11
p[3].ovhead:	a[1].ovhead	D11
p[1].lease:	numeric	B12
p[2].lease:	numeric	C12
p[3].lease:	numeric	D12
p[1].inc:	p[1].sales-p[1].cogs-p[1].ovhead-p[1].lease	B13
p[2].inc:	p[2].sales-p[2].cogs-p[2].ovhead-p[2].lease	C13
p[3].inc:	p[3].sales-p[3].cogs-p[3].ovhead-p[3].lease	D13
p[1].tax:	@IF(p[1].inc>0,p[1].inc*a[$1].tax,0)	B14
p[2].tax:	@IF(p[2].inc>0,p[2].inc*a[$1].tax,0)	C14
p[3].tax:	@IF(p[3].inc>0,p[3].inc*a[$1].tax,0)	D14
p[1].net:	p[1].inc-p[1].tax	B16
p[2].net:	p[2].inc-p[2].tax	C16
p[3].net:	p[3].inc-p[3].tax	D16

Figure 4: The logical map of the forecasting spreadsheet– the output of steps F1-F4 of the factoring algorithm. The cell addresses on the right are not part of the logical map, and are given here only for reference purposes.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

```
a[n].grate:       numeric
a[n].ovhead:      numeric
a[n].cogs:        numeric
a[n].tax:         numeric
p[n].year:        string
p[n=1].sales:     numeric
p[2<=n<5].sales: p[n-1].sales*(1+a[1].grate)
p[n>=5].sales:    0.5*(p[n-2].sales+p[n-1].sales)*1.2
p[n].cogs:        p[n].sales*a[1].cogs
p[n].ovhead:      a[1].ovhead
p[n].lease:       numeric
p[n].inc:         p[n].sales-p[n].cogs-p[n].ovhead-p[n].lease
p[n].tax:         @IF(p[n].inc>0,p[n].inc*a[1].tax,0)
p[n].net:         p[n].inc-p[n].tax
```

Figure 5: The spreadsheet map after step F7 of the factoring algorithm.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28

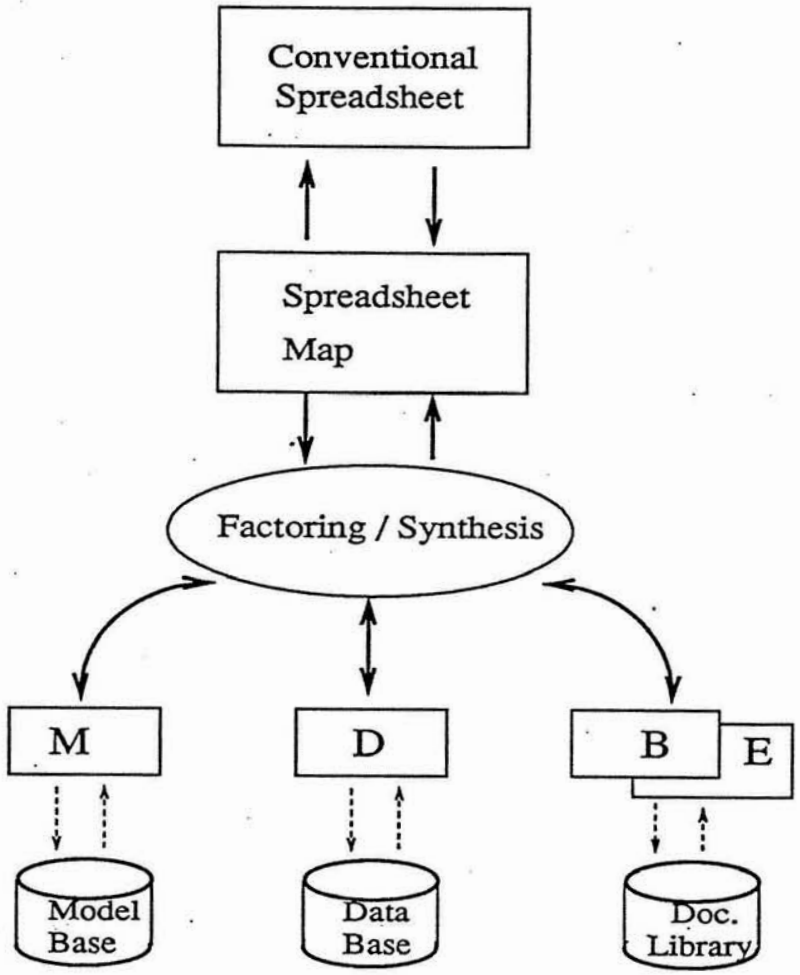


Figure 6: A spreadsheet model and its four components. Up arrows represent synthesis; down arrows factoring.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-92-28