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

July 28, 1993

Replaces IS-92-28

Working Paper Series
STERN IS-93-24

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 or a methodology to support the construction and maintenance of spreadsheet models. Using a dual logical/physical perspective, we identify four principal components that characterize any spreadsheet model: *schema*, *data*, *editorial*, and *binding*. We present a *factoring* algorithm for identifying and extracting these components from conventional spreadsheets with minimal user intervention, and a *synthesis* algorithm that assists users in the construction of executable spreadsheets from reusable model 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. Importantly, our approach to model management and the Windows-based prototype that we have developed are designed to *coexist* with, rather than *replace*, traditional spreadsheet programs. In other words, the users are not required to learn a new modeling language; instead, their logical models and data sets are extracted from their spreadsheets transparently, as a side-effect of using standard spreadsheet programs.

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

1 Introduction

Spreadsheet modeling represents one of the most pervasive and successful applications of personal computers. Since their introduction in the late 70's, spreadsheet programs transformed the notion of end-user computing, creating a new computational paradigm which offers 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 tool in modern business. Compared to their humble origins and limited objectives, today's spreadsheet programs are extremely powerful, versatile, and user-friendly. Yet in spite of this technological progress, 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 like [20], a *theory* of spreadsheet analysis and design is yet to emerge.

Viewed as model generators, spreadsheet programs have both pros and cons.[1] Their congenial user-interface and instant modeling power notwithstanding, 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-database era. Before data definition was elevated to the `DBMS` level, file structures were a fixed part of the programs' code. In a similar vein, the logic and documentation of spreadsheet models are often 'buried in the formulae,' and are largely inaccessible to people other then the spreadsheets' creators. In both cases, the implications were similar: redundant and inconsistent *file* and *model* structures, respectively. To complete the analogy, these problems arise because spreadsheet programs lack a *high level* means to support the design and maintenance of spreadsheet models.

With that in mind, we propose to treat spreadsheet models from two different perspectives:

[1] Throughout the paper, the term *spreadsheet programs* refers to spreadsheet modeling environments like `Lotus 123`, `QuattroPro`, and `Excell`. The terms *spreadsheet models*, or simply *spreadsheets*, refer to *specific* spreadsheets, e.g. P&L spreadsheets, inventory control spreadsheets, and the like.

logical and physical. The *logical* perspective consists of a formal and implementation-free description of the model's logic and data structures: the *physical* level concerns such details as storage, formatting, user interface, and other aspects that effect the model's implementation, 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 'untangled,' it will be difficult if not impossible to develop intelligent spreadsheet model management systems – systems that promote the construction of consistent and valid models across the organization.

Although a spreadsheet model management system would hardly matter for the casual user who builds several spreadsheets for personal use, the situation is quite different for organizations that depend on spreadsheets to support basic business functions, where model validity and consistency are critically important. In a recent field study, Cragg and King [4] have sampled spreadsheet examples from ten such organizations. After scrutinizing the various models that they have collected, they concluded that about 25% of them contained logical design errors, ranging from trivial cell misreferences to erroneous formulae that went undetected by their users. In two independent simulation studies, Brown and Gould [2] and Floyd and Pyun [7] had groups of subjects design spreadsheets to solve a variety of problems. In both cases, the authors reported that a significant error-rate characterized novice as well as expert users. Further, most of the subjects in these experiments have exhibited a great deal of confidence in their spreadsheets's validity, implying that spreadsheet design errors are not only prevalent, but also elusive. These results corroborate the observation that spreadsheet models suffer from weak accountability and face-validity [9].

Cragg and King categorized the errors that they have encountered in their field studies as follows: incorrect cell references, incorrect ranges, incorrect use of functions, erroneous formulae, data input errors (in particular – overriding formulae with constants), failure to incorporate key factors or variables in one's model, and a host of model manipulation mishaps that arise from using relative addressing instead of fixed addressing, and vice versa. Needless to say, such errors can have far reaching implications for the model users, and

the spreadsheet folklore contains several published anecdotes to that effect. For example, a U.S. contractor failed to include a major cost item in the range of a `@sum` formula that was supposed to calculate the total cost of a government project for which he has bid. After winning the undervalued bid and discovering the error, the contractor sued the spreadsheet program's vendor for selling a modeling tool that enabled such a mishap to go undetected. The contractor lost the suit, as well as the $254,000 that were inadvertently omitted from his bid [5]. In a similar case, a Dallas-based oil and gas company fired several executives for spreadsheet model oversights that cost the company millions of dollars [8].

We believe that many of these errors occur because the lines of logical design and physical implementation are blurred in the conventional setting of a spreadsheet program. To illustrate, consider the spreadsheet model in Figure 1. There are two ways to describe this spreadsheet. 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 $2,500,000; lease is fixed at $100,000 in the first two years and $500,000 thereafter; and tax is assumed to be 48% of gross income.

Put Figure 1 around here

As it turns out, however, the P&L spreadsheet has more to it than appears on the surface. 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 – a fact which is not at all evident from the spreadsheet's appearance. Likewise, 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

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

appearance of a spreadsheet can be deceiving, as it is not necessarily consistent with the logical structure that it suggests. One way to validate the integrity of a spreadsheet is to print out all 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.

Once built, the P&L spreadsheet is prone to many accidental maintenance 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 does not 'know' that the user is dealing with a profit and loss projection, there is no way to sense that such activities can corrupt the model's logical structure. For similar reasons, spreadsheet programs make it difficult to isolate the *data* element of a given spreadsheet. Although one can separate 'data 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 screen layout 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. Unlike modern languages, early programming languages did not have built-in features to support the writing of well-designed programs; instead, they permitted an unrestricted use of `GOTO` commands and undeclared variables, leading to long-term maintenance problems. Similarly, spreadsheet programs are totally unconstrained, allowing users to construct any spreadsheet that they desire, including, of course, poorly-designed and poorly-documented spreadsheets. One objective of this research is to preserve the tremendous design freedom that spreadsheet programs have to offer, and, at the same time, enable users to inspect their work from a logical perspective that promotes the construction of well-designed models.

The realization that unguided spreadsheet modeling is prone to design and maintenance errors has led to several recommendations to streamline the modeling process. For example, Mason & Keane [18] proposed to designate a human *model administrator* – akin to a DBA – to regulate and monitor spreadsheet modeling activities across the organization. In a similar vein, Williams [24] recommends to adopt an organizational standard to cover such aspects as spreadsheet specification, documentation, maintenance, and security (of the ten companies studied by Cragg & King, only one had a spreadsheet modeling standard). Others, e.g. McMickle [19] and Simkin [21], have advocated the use of spreadsheet audit software. Clearly, all these recommendations are consistent with our ultimate objective to develop a corporate-wide spreadsheet model management system (SMMS).

What kind of services should such a system provide? From a *model definition* perspective, an SMMS should support the construction of well-designed and well-documented spreadsheets that can communicate with other model and data resources in the organization. Yet ideally, an SMMS should accomplish these objectives behind the scene, without modifying the standard practice of spreadsheet modeling. In other words, we begin with the working assumption that the average user would like to continue to build models in his or her favorite spreadsheet program, objecting to the dictum of having to learn a new modeling language. From a *model manipulation* perspective, we observe that one of the major benefits of spreadsheet modeling is the ability to change assumptions and inspect the impact on some output criterion. Hence, an SMMS 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 SMMS should facilitate access to a repository of reusable models and model 'chunks,' or a model base. Ideally, the spreadsheet designer 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 designer to combine these models with other models and databases across the organization.

This paper presents the first step toward developing such a model management system.

The plan of the paper is as follows. Section 2 focuses on the interplay of the physical and the logical views of spreadsheet models. In particular, it identifies four principal components that characterize any spreadsheet: *schema, data, editorial,* and *binding.* Section 3 presents a top-down *factoring* algorithm that extracts these components from conventional spreadsheets with minimal user intervention. Section 4 takes the opposite route, describing a bottom-up *synthesis* algorithm that constructs executable spreadsheets and spreadsheet templates from a repository of reusable spreadsheet components. Both algorithms make use of a *functional relational language* (FRL), whose syntax and BNF are given in a separate appendix. The paper ends with a discussion section that comments on the implications of this research for the development of intelligent modeling assistants and spreadsheet model management systems.

2 The Physical and Logical Views of Spreadsheets

Our approach to spreadsheet analysis and design is based on the premise that spreadsheet models can be seen and operated on from two independent views: physical and logical. From a physical perspective, a spreadsheet model is a collection of addressable cells, arranged in a two-dimensional grid. Each cell has a row and column address, and a definition part 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 are changed.

In addition to this familiar physical perspective, every spreadsheet model embeds an implicit *logical view* which, in this research, we take to be a set of *functional-relations.* A functional-relation is similar to an ordinary relation in that both data structures consist of one or more attributes and of one or more tuples, the minimal practical case being a single-attribute/single-tuple relation. Unlike ordinary 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. The set of functional-relation definitions that are embedded in a particular spreadsheet is called hereafter the spreadsheet's *schema*, denoted $\mathcal{S}$ for brevity.

The schema provides the spreadsheet's skeleton, which is further augmented with several other features. Paraphrasing Wirth [25], we observe that every spreadsheet can be characterized by four principle properties, as follows: *spreadsheet = schema + data + editorial + binding.* The *schema* property ($\mathcal{S}$) stores a concise and formal definition of the spreadsheet's underlying logic. The *data* property ($\mathcal{D}$) is the structured collection of constants on which $\mathcal{S}$ operates. The *editorial* property ($\mathcal{E}$) can be defined as what is left over in the spreadsheet model after $\mathcal{S}$ and $\mathcal{D}$ have been carved out: titles, column and row headings, and documentation. Finally, the *binding* property ($\mathcal{B}$) is a logical-to-physical mapping that binds $\mathcal{S}$,$\mathcal{D}$, and $\mathcal{E}$ to the spreadsheet grid, using row and column addresses.

The *schema* and the *data* properties play a major role in our approach to modeling, and much of the paper evolves around the dual processes of (i) factoring these properties from existing spreadsheets and (ii) synthesizing them into new spreadsheets. The relationship between a conventional spreadsheet and its underlying $\mathcal{S}$ and $\mathcal{D}$ properties is depicted in figure 2, which is used throughout the paper as an illustrative example. In inspecting this figure, the reader is asked to ignore for now the details of the schema definition language and the functional-relations, focusing instead on the general relationship between the spreadsheet's physical view (top) and logical view (bottom). Note that the figure does not mention the editorial and binding properties. Theoretically as well as practically, $\mathcal{E}$ and $\mathcal{B}$ are not nearly as interesting and challenging to deal with as $\mathcal{S}$ and $\mathcal{D}$ – a point which is taken up later in the paper.

Put Figure 2 around here

The argument that models and data should be kept and managed separately is central in the model management literature [6, 22, 1]. Following this principle, we have designed

our $\mathcal{S}$ and $\mathcal{D}$ properties to be independent of the spreadsheet program, as well as independent of each other. Specifically, we take the spreadsheet schema to be a mathematical abstraction that can be described in terms of several different formalisms, of which the two-dimensional parlance of cells and formulae is only one representation. Likewise, we view the spreadsheet's data property to be a set of relations that can be manipulated by any relational DBMS, and the fact that the relations can be extracted from, or superimposed on, a spreadsheet grid is a useful but not mandatory property of their existence. Taken together, we use the symbolic sum $\mathcal{S} + \mathcal{D}$ to refer to the application of a schema to a fixed data set, similar to the notion of running a model on a particular scenario, or executing a program on a given input. Finally, the symbolic sum $\mathcal{S} + \mathcal{D} + \mathcal{E} + \mathcal{B}$ stands for the familiar notion of a conventional spreadsheet, where $\mathcal{B}$ maps $\mathcal{S} + \mathcal{D}$ on a two-dimensional grid that is further interspersed with textual labels drawn from $\mathcal{E}$.

Practically all the problems that were alluded to in §1 are related to the following observation: in conventional spreadsheet programs, users are encouraged to weave the four spreadsheet properties together and treat them as one entity from the outset, forming a prime example of how a modular system should *not* be constructed. As a result of this enmeshment, the two most important principles of software engineering– separating logical design from physical implementation, and separating algorithms from data – are violated by conventional spreadsheet programs almost by definition. The first step toward resolving these problems requires a precise understanding of the interplay of the *physical* and *logical* views of spreadsheet models, which we define as follows:

- The *physical view* of a spreadsheet is a list of entries of the form *(cell_address: definition, formatting_specifications)*, one entry for each active cell in the spreadsheet.
- The *logical view* of a spreadsheet is a pair $< \mathcal{S}, \mathcal{D} >$, consisting of the spreadsheet's underlying *schema* and *data* properties, respectively.

It is convenient to think of the spreadsheet's physical view as its underlying *map* – a linear representation of the spreadsheet's more familiar two-dimensional perspective. For example, the map of the P&L spreadsheet is depicted in Figure 3. Most spreadsheet programs

are capable of producing spreadsheet maps on demand as a standard service, available from the program's menu. The format of these maps varies from one program to another, but their substance is more or less the same. For the purpose of this research, we have written a special macro that produces an *annotated* version of a spreadsheet map, in which each cell entry is preceded by a relational label, as we treat later in the paper. Normally, spreadsheet maps are used "passively" for documentation and debugging purposes; Yet for us, the notion of an annotated spreadsheet map is central, as it serves as the primary input from which the spreadsheet's logical view is extracted.

The interplay of a spreadsheet's physical and logical views is illustrated in Figure 2. The top of the figure gives an outlined version of the P&L spreadsheet. According to this particular outline (which is not unique), 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 exact definitions of the two relations are given in the spreadsheet's *schema* (bottom left of Figure 2), whereas their data contents are stored in a separate data set (bottom right of Figure 2). The numeric values in the relations are user-supplied data, extracted from 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 ordinary 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.

It is important to note that even though they can be constructed from each other, the physical and the logical views of a spreadsheet model are independent entities. Specifically, the physical spreadsheet 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 spreadsheet schema. Thus, a

user may transpose the spatial 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 schema and data property intact.

The distinction between the logical and the physical views of spreadsheets has significant practical implications. Suppose that whenever a spreadsheet were loaded into a spreadsheet program, the program would also load a "behind the scene" image of its underlying $\mathcal{S}$, $\mathcal{D}$, $\mathcal{B}$, and $\mathcal{E}$ properties. By continuously comparing the user's activities at the physical spreadsheet grid to their implications for the four properties, the program could sense what he or she is trying to do not only in the way of manipulating physical cells and formulae, but also in the way of revising its logical and data building blocks. Such an extension would endow conventional spreadsheet programs with the ability to understand the *semantics* of spreadsheet models, something which is quite lacking in the present generation of spreadsheet modeling environments. As we argue later in the paper, this would enable the development of (i) spreadsheet model management systems, and (ii) interactive assistants for building consistent and valid spreadsheet models.

3 Factoring: from Physical to Logical

This section describes the process through which a conventional spreadsheet can be factored into its four principal properties: schema, data, editorial, and binding. The process consists of two key stages, as follows:

- *outlining* (interactive relations definition)
- *factoring* (automatic properties extraction)

In the preliminary *outlining* stage, which takes place within the host spreadsheet program, the user is asked to identify and name the functional-relations that make up the model. These specifications provide all the necessary inputs for the subsequent *factoring* stage – a

seven-step reduction algorithm that 'splits' the spreadsheet into its four principle properties with no additional human intervention.

3.1 Outlining a Spreadsheet

The notion of *relational outlining* is based on the observation that any spreadsheet can be viewed as a (non-unique) collection of functional relation *candidates*. A relation-candidate is a contiguous block of cells – a rectangle, a row, a column, or a single cell – that represents either a singular or a repetitive entity in the model's realm. In the P&L spreadsheet, for example, block `[B2..E2]` is a relation-candidate that contains a set of parametric assumptions. Technically speaking, each individual cell and contagious subsets thereof in the assumptions block are also relation-candidates, and yet it is reasonable to assume that the entire block will be manipulated as one unit, as in moving it around the screen 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 in a given model should be best left to the discretion of a human designer, as is normally done in constructing ordinary data models.

In the system that we have developed, the user outlines the spreadsheet through an interactive *outlining macro*, implemented in `Excell`'s macro language. The macro enables the user to define the relations directly from the spreadsheet program, using pop-up dialog boxes and screen-painting inputs. For each relation, the macro prompts the user to specify a `name`, an `alias`, a `scope`, and a spatial `orientation`. The `scope` refers to the relation's data boundaries; these are specified by anchoring the cursor in a certain cell and painting a rectangular area on the spreadsheet grid. The relation's `orientation` regards its spatial positioning, which is either *horizontal* (row tuples) or *vertical* (column tuples). If the relation's scope consists of a single row or a single column, the user is also asked to

specify whether the relation is designed to store a single tuple, in which case it is said to be a *vector relation.* Based on these inputs, the macro infers the screen coordinates of the relation's attributes, and, after highlighting them one by one, prompts the user to name them. Finally, if the relation is not of type `vector`, the user is also asked to designate a key attribute.

In sum, the outlining macro enables the user to superimpose a relational structure on a conventional spreadsheet grid. In addition, the outlining macro performs an elaborate 'behind the scenes' act: as the user provides the relations' specifications, the macro builds an annotated spreadsheet map, stored in a separate `ASCII` file. For each active cell in the spreadsheet, the macro constructs a map-entry that gives the cell's symbolic label (to be discussed shortly), address, definition, and formatting instructions. For example, when the outlining macro was applied to the P&L spreadsheet from figure 1, it produced the map shown in Figure 3.

Put Figure 3 around here

As the figure illustrates, the map is a list of *map-entries*, one for each active cell in the spreadsheet. Note that some entries are prefixed by a label of the form $r[i].x$, where r is a relation name, i is a tuple index, and x is an attribute name. These labels are obtained from the spreadsheet's outline through the following matching rule. If a cell falls inside the scope of a named relation (e.g. `C12`, which is inside p's outline – see Figure 2), it must sit in the intersection of a named attribute (`lease`), and a keyed tuple (`1993` – 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 does not fall inside the scope of any one of the user-defined relations, its map-entry is left unlabeled. The reader may wish to compare the spreadsheet's outline (Figure 2) and map (Figure 3) in order to track the labels generation rule.

The map that emerges from the outlining process 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 properties. Throughput the factoring process, the map is gradually reduced and rewritten, sheering away the properties $\mathcal{E}$, $\mathcal{B}$, $\mathcal{D}$, and $\mathcal{S}$, in that order.

3.2 Extracting the Editorial and Binding properties

The extraction of the $\mathcal{E}$ and $\mathcal{B}$ properties from the spreadsheet map is straightforward, and therefore it will be discussed here only in broad terms.

Recall that by the *editorial* property of a spreadsheet we refer to the collection of cells that carry auxiliary information such as titles, column and row headings, comments, and general documentation. Since the outlining process focuses only on the cells that make up the spreadsheet's relations, the cells that carry editorial information are left out of the relations' boundaries (see Figure 2), and thus they end up as non-labeled entries in the spreadsheet map. Therefore, the task of extracting and archiving the editorial property $\mathcal{E}$ is merely a matter of splitting the map into two sub-maps, consisting of non-labeled entries and labeled entries. The former map forms the spreadsheet's *editorial* property, which is stored in a separate file.

The latter map consists of entries of the form ($r[i].x$: *cell_address, cell_definition*). In each of these entries, the first two terms associate a relational data-item (the value of the x attribute in the ith tuple of relation r) with a physical spreadsheet cell. Taken together, the list of pairs ($r[i].x$: *cell_address*) can be used to superimpose, or "anchor," all the relations on the spreadsheet grid. This list, which forms the *binding* property of the spreadsheet, is copied from the map and stored separately.

3.3 Extracting the Data property

The labeled map-entries that remain in the map after $\mathcal{E}$ has been extracted fall into two categories:

constant entries: $(r[i].x : \ cell\_address, \ constant)$

formula entries: $(r[i].x : \ cell\_address, \ formula)$

The data extraction procedure begins by building a set of relational templates to accommodate all the values specified by the map-entries. Since the $r[i].x$ entry-labels provide all the necessary relation- and attribute-names (as defined by the user), the construction of the relational structures that they imply is carried out automatically. Once these relations have been constructed, the constants and formulae of each map-entry are pegged into their proper slots in the relations, using the $r[i].x$ entry-labels as pointers. The constants are copied verbatim, whereas the formulae definitions are replaced with the special symbol `C`, denoting a *calculated value*[3]. For example, the procedure uses the map-entry `(p[1].sales: 6000)` to set the `sales` attribute of the first tuple of the `p` relation to 6000, whereas the map-entry `(p[3].inc: +D9-D10-D11-D12)` causes the `inc` attribute of `p`'s third tuple to be set to the marker `C`.

The set of relations that are constructed and populated by this process forms the spreadsheet's *data* property. For example, the data property of the P&L spreadsheet consists of the relations `a` and `p`, depicted at the bottom right of Figure 2. Once "liberated" from the spreadsheet grid, the relations become stand-alone entities that can be (i) used by other spreadsheets, and (ii) maintained by a database management program.

3.4 Extracting the Spreadsheet Schema

The remainder of the factoring process is a series of steps that may be described as $\text{map}_i = \text{step}_i(\text{map}_{i-1}), i = 1, \ldots, 7$. The input of the process – map_0 – is the annotated spreadsheet map produced by the outlining macro. The output of the process – map_6 – is the spreadsheet's schema $\mathcal{S}$, written in the FRL language. The FRL syntax is self-explanatory, and is best described through examples. For a formal language description

[3]When a functional-relation contains many calculated values, a sparse matrix representation can be used to conserve disk-space.

and a BNF, the reader is referred to the paper's appendix.

Since it contains cell addresses, the spreadsheet map is an inherently physical entity. Hence, factoring begins with a preprocessing stage that transforms the spreadsheet map into a *logical map* which is independent of cell addresses. The preprocessing stage is given in figure 4 and described as follows.

Put Figure 4 around here

In step F1, the constants that were previously stored in $\mathcal{D}$ are replaced with their corresponding data-types. Next, F2 substitutes physical cell-addresses that appear in formulae with their corresponding entry-labels. For example, the physical map-entry (p[1].cogs: +B9*$D$2) 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 P&L spreadsheet map. When this substitution operation is completed, F3 takes another pass through the entire map, eliminating the cell-addresses from all the entries. The data structure that emerges from steps F1-F4 is denoted hereafter the spreadsheet's *logical map*, which, in the case of the P&L spreadsheet, is shown in Figure 5. The reader may want to compare this map to the physical map in Figure 3 in order to track the execution of steps F1-F4.

Put Figure 5 around here

Due to F4, the logical map becomes a list of *attribute-clusters*, each cluster being an ordered list of one or more map-entries whose labels are made 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 P&L's logical map consists of the attribute-clusters a.grate, a.ovhead, a.cogs, a.tax, p.year, p.sales, p.cogs, p.ovhead, p.lease, p.inc, p.tax, and p.net. The remaining steps of the factoring algorithm contract and transform these clusters into a formal spreadsheet schema, written in FRL. These steps are depicted in figure 6 and

illustrated in Figure 7, which shows how they transform the `p.sales`-cluster of the P&L spreadsheet map into an FRL attribute definition[4].

Put Figure 6 around here

Put Figure 7 around here

As Figure 7-b indicates, the `p.sales`-cluster is made up of three *generic sets* of entries – entries that convey exactly the same mathematical relationship, albeit with different indices, or tuple references. The goal of step `F5` is to transform generic entries into entries that have precisely the same right hand side definition. In Figure 7-b, all the right hand side attribute-references `p[`i`].sales` are *related*, because they have the same relation name as their left hand side entry-labels – in this case `p`. Hence, the i's in these references, which stand for absolute tuple numbers, are replaced by `F5` with relative tuple offsets. Next, the second part of `F5` rewrites every occurrence of `a[$1].grate` as `a[1].grate`, leading to Figure 7-c. Next, `F6` packs and rewrites each set of repetitive map-entries into a single, generic entry, leading to Figure 7-d. From here, the route to a formal spreadsheet schema (Figure 7-e) is straightforward, involving trivial syntactical conversions that are carried out by `F7`.

Implementation: The factoring algorithm was implemented in our system as a Pascal program that is launched automatically from within `Excell`, as a transparent side-effect of saving a spreadsheet. Using a lexical parser and analyzer whose rules follow steps `F1-F7`, the program converts the spreadsheet map (output of the outlining macro) into a formal schema which is stored on a separate `ASCII` file. When we applied the factoring program to the P&L spreadsheet, it produced the schema depicted at the bottom left of Figure 2.

Using the implementation, we can now generate and maintain logical views of spreadsheets

[4]Due to space limitations, the maps in Figures 3 and 5 correspond only to years 1992–1994 in the P&L spreadsheet. At the same time, the `p.sales`-cluster in Figure 7 is taken from the map of the entire spreadsheet, i.e. for years 1992-1997.

with minimal user intervention. It is important to emphasize that from the user's perspective, factoring is mostly a one-time operation. For example, let us assume that a particular spreadsheet has ben factored. From that point on, the spreadsheet's map, outline, schema, data, editorial, and binding elements become transparent properties that are continuously revised as a side-effect of using the spreadsheet. The revision process is as follows:

1. The user loads the spreadsheet into a host environment such as `Excell`, proceeding to use, maintain, or extend it, via standard spreadsheet commands.
2. When the user saves the spreadsheet, a transparent *housekeeping macro* is triggered. The macro generates a spreadsheet map and proceeds to compare it to the spreadsheet's old map.
3. If the two maps contain exactly the same set of cells (although *the contents of cells may well be different*), the housekeeping macro goes to step 5.
4. If the two maps contain different sets of cells, the housekeeping macro invokes the outlining macro, asking the user to outline the areas in the spreadsheet that were changed during the present `Excell` session.
5. The factoring program is transparently launched, creating updated versions of the spreadsheet's schema, data, editorial, and binding properties.

Stage 3 is of a particular interest here. If the user applies the spreadsheet to a certain scenario, as in changing some parameters or any other cell values, the refactoring process runs automatically, and the only impact from the user's standpoint is a slightly longer time to save the spreadsheet on the disk. As a matter of fact, even if the user *changes* the model in the way of modifying existing formulae, the refactoring process is once again carried out automatically, without any user intervention. The only case which requires human feedback occurs when the user adds or deletes one or more cells from the spreadsheet. And even then, the dialog with the user boils down to a partial outlining process that involves answering a few simple questions. In sum, the cost of factoring in terms of user's involvement is fairly minimal. Perhaps the most intriguing feature of the process is its low profile: the user builds and maintains spreadsheet models in his or her favorite spreadsheet program, without having to learn a new modeling language. The four properties of the spreadsheets are extracted and managed behind the scene, as a transparent side-effect of the user's activities at the spreadsheet program level.

4 Synthesis: from Logical to Physical

The previous section described a top-down factoring process that splits physical spreadsheets into their four principal properties. This section describes the reverse operation – *synthesis* – in which executable spreadsheets can be built bottom-up from reusable components[5]. The synthesis process has many practical benefits, not the least of them is model sharing. In its simplest form, synthesis enables different users to apply the same spreadsheet logic (schema) to different data sets. Although spreadsheet sharing and reuse is already done in practice on an informal basis, the synthesis process takes the practice one step further. For example, it ensures that changes made to a spreadsheet schema will propagate to all the spreadsheets that were synthesized from it (if the users so desire). Further, it enables the construction of a corporate model-base – a collection of spreadsheet templates – from which users can pick and choose generic models according to functionality and structural criteria.

The key player in the synthesis process is the spreadsheet schema $\mathcal{S}$. In the factoring algorithm, $\mathcal{S}$ was the final output; in synthesis, it is the major input, along with optional $\mathcal{D}$, $\mathcal{E}$, and $\mathcal{B}$ components. Note that in and by itself, the schema is not an executable entity. At the same time, it contains all the necessary *information* for constructing operational spreadsheets. Using this observation, we have designed a synthesis procedure that transforms a spreadsheet schema into a model that can be loaded directly into a target spreadsheet program. The synthesis process is highly flexible, enabling the user to mix the spreadsheet schema with different (but structurally compatible) data, editorial, and binding components. The resulting spreadsheets form families of related models whose relationships can be monitored and exploited by a spreadsheet model management system.

For example, let us assume that a certain spreadsheet has been previously factored into its four principle properties. The synthesis of all four properties, denoted $\mathcal{S} + \mathcal{D} + \mathcal{B} + E$, yields an executable spreadsheet that is completely identical to the original. The full

[5]Hereafter, the terms *component* and *property* will be used interchangeably.

reconstruction of a factored spreadsheet is hardly interesting on practical grounds, but it provides a convenient point of departure from which more interesting cases can be discussed. For example, if the data property is left out of the synthesis process, the combination $\mathcal{S}+\mathcal{B}+\mathcal{E}$ yields a *spreadsheet template* – a skeletal model structure that can be instantiated with a variety of different data sets, or modeling scenarios. Specifically, two spreadsheets of the form $\mathcal{S}+\mathcal{B}+\mathcal{E}+\mathcal{D}$ and $\mathcal{S}+\mathcal{B}+\mathcal{E}+\mathcal{D}'$ that differ only in their data property are said to be different *data instances* of the same generic spreadsheet. This will be the case, for example, when different divisions of the same company are required to use a standard spreadsheet template to produce their divisional P&L statements.

Other combinations of the four properties are equally instructive. To illustrate, consider the two spreadsheets $\mathcal{S}+\mathcal{D}+\mathcal{B}+\mathcal{E}$ and $\mathcal{S}+\mathcal{D}+\mathcal{B}'+\mathcal{E}'$, that differ only in their *binding* and *editorial* properties. Note that even though the two spreadsheets are physically different, they are invariant in terms of the logical view $<\mathcal{S},\mathcal{D}>$. This distinction could be useful if $\mathcal{E}$ and $\mathcal{E}'$ were the English and Spanish versions of the same spreadsheet, or if $\mathcal{B}$ and $\mathcal{B}'$ were alternative screen layouts of the same model, a variation that occurs whenever 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 practice, albeit in an informal and haphazard fashion. For example, consider Tom, a junior loan officer, who wants to analyze loan applications with a spreadsheet model created by his experienced colleague and savvy spreadsheet user, Jane. For Tom, the easiest way to adopt Jane's spreadsheet is to *clone* it. This is commonly done by copying Jane's spreadsheet, carefully erasing all its constant cells (implicit data property), and retaining all its formulae cells (implicit schema). Once the spreadsheet has been emptied from Jane's data, Tom can populate it with his own data, at which point Tom 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 spreadsheets as unrelated physical entities. Therefore, when Jane changes her spreadsheet to fix an error, or to accommodate a new credit rule, the change will not effect Tom's work in any way.

We see that when spreadsheets are shared and reused informally, maintenance and extension efforts must be duplicated. Had we had a intelligent framework for spreadsheet model management, this duplication could be minimized. For example, if Tom wants to clone Jane's spreadsheet, the safest way to do it is to (i) factor her spreadsheet into its four principal components, and (ii) synthesize her $\mathcal{S}$, $\mathcal{E}$, and $\mathcal{B}$ components with his $\mathcal{D}$ component, which could be entered interactively into the empty spreadsheet, or loaded batch-style from a file. If Jane were to change her spreadsheet's logic at a later point of time (an event which a model management system could sense by comparing her old and new $\mathcal{S}$ properties), the system could advise Tom that his spreadsheet is no longer logically identical to Jane's. If Tom wants his model to be completely in sync with Jane's, the discrepancy could be resolved by synthesizing his data component with Jane's modified schema.

It goes without saying that numerous technical ends must be met before such an integration can take place in practice. In the extreme case, Jane could modify her schema in such a way that would render it incompatible with Tom's existing data set. Yet in many situations the differences between two related spreadsheets are quite manageable, due to the fact that they represent evolutionary deviations from a single schema that the system already understands. For example, suppose that Jane uses a spreadsheet `copy` or `insert` command to add a new row or column to her model. Since the system already stores an image of her spreadsheet schema, it can automatically infer whether this spatial manipulation amounts to adding a new data *tuple*, which requires no further action at Tom's end, or adding a new *attribute*, in which case Tom's spreadsheet could be effected. In either case, the implications of Jane's actions on Tom's and other related spreadsheets could be inferred and acted upon by the system.

In order for a spreadsheet model management system to discern and manage such cases, the system must employ a taxonomy of schema modification scenarios, as well as a discrete metric that measures the structural proximity of different spreadsheet schemas. We are presently in the process of developing such tools, about which we intend to report in a future publication. So far, our research suggests that once the factoring/synthesis framework is in place, the problems that hindered the development of a spreadsheet model management

system become largely technical, not fundamental.

Going back to the subject of synthesis, note that the process is essentially the converse of factoring. Therefore, it traces the factoring steps backwards, beginning with a spreadsheet schema written in FRL and ending with an executable spreadsheet model. The process involves three main stages, as follows. First, the schema is converted into a logical map like the one depicted in Figure 5. At this point, the user has two options. If he or she wishes to 'instantiate' the schema with a stored data set, the logical map is merged with a given $\mathcal{D}$ component. If, alternatively, the user wishes to create a spreadsheet *template*, the logical map is merged with a generic data set that is consistent with the schema's structure. Next, the logical map is transformed into a physical spreadsheet by synthesizing it with binding and editorial components. These properties can be drawn from a documentation library, or added interactively by the user.

4.1 From a Spreadsheet Schema to a Logical Map

The synthesis algorithm begins with a preprocessing stage (Figure 9) that fuses a spreadsheet schema and a data component into a logical map. Since the algorithm processes the schema one relation definition at a time, it is sufficient to describe it for one relation only.

Put Figure 9 around here

Once again, we illustrate the algorithm in the context of the `p.sales` attribute. The primary input of the algorithm is the spreadsheet schema (Figure 7-e). It is important to recall that in spreadsheet schemas, attribute-references are abbreviated as much as possible, using FRL's syntax-default rules. For example, the expression `(cogs: sales*a.cogs)` is shorthand of `(p[n].cogs: p[n].sales*a[1].cogs)`. In steps `S1-S2`, all the abbreviated attribute-references are expanded to their fully-specified references. As a result, the schema definition of the `sales` attribute becomes the `p.sales` attribute-cluster listed in Figure 7-d.

In S3, each definition line is expanded, i.e. repeated for all the tuples that it covers, leading to the attribute-cluster listed in Figure 7-c. In order to carry out this expansion, the algorithm has to know how many tuples the relation presently contains in $\mathcal{D}$. If the user wants to synthesize the schema with a given data component, this entails a simple lookup operation. If the user wants to create a spreadsheet *template*, a dummy data component is cloned from $\mathcal{S}$, as we describe shortly.

Following S1-S3, the definition parts of the attribute-clusters typically contain attribute references with *relative tuple addressing*, e.g. `p[n-2].sales`. This kind of addressing is characterized by the presence of the special symbol `n`, as in r[n].x, r[n+j].x, or r[n−j].x, for some j. In S4, relative tuple references are substituted with their corresponding absolute values, leading to Figure 7-b. When applied to the entire P&L schema, the final output of steps S1-S4 is the P&L logical map (Figure 5).

4.2 Adding the Data Component

During synthesis, a spreadsheet schema $\mathcal{S}$ can be fused with a set of "real" data relations $\mathcal{D}$, or with a set of "dummy" relations – denoted $D_{\mathcal{S}}$ – that are constructed from the schema itself, as we describe below. In the former case, it is assumed that $\mathcal{D}$ is structurally compatible with $\mathcal{S}$ – a property that can be easily tested. In the latter case, $D_{\mathcal{S}}$ and $\mathcal{S}$ are compatible by construction.

The data component, be it real or dummy, is synthesized by posting the data items from the relations in $\mathcal{D}$ into the *constant entries* in the spreadsheet's logical map, i.e. the map-entries whose definition part is either `numeric`, `string`, `logical`, or `date` (see Figure 5). Specifically, each constant entry of the form (r[i].x: *data_type*) is rewritten as (r[i].x: *value*), where *value* is the value of the x attribute in the ith tuple of the relation $r \in \mathcal{D}$. Since the logical map has already been "stretched" by S1-S4 to accommodate all the data-items from all the relations, this operation completes the synthesis of the spreadsheet schema with the data component.

Spreadsheet templates: A spreadsheet template is a skeletal spreadsheet model – a model whose data-cells contain symbolic data-type markers rather than actual data. Spreadsheet templates are quite useful for documenting, distributing, and sharing, spreadsheet application software. Once loaded into a host spreadsheet program, a template can be populated with real data, a process that requires the user to fill in existing cells, and, occasionally, expand the template in the way of inserting new rows and columns and copying their formulae from the template's generic formulae. For example, a stock portfolio spreadsheet template can contain a single row with all the necessary formulae for calculating and presenting some statistics about a generic stock entity. Once put to actual use, the row can be expanded by the user to accommodate as many stocks as necessary, via the spreadsheet program's `copy` command.

Thus, a relation's template should contain just enough dummy tuples to reflect its generic structure and facilitate its later expansion by the user. In the above example, we implicitly assumed the existence of a `stocks` relation whose tuples follow the same definition. But in reality, a relation schema can contain *case structures*, in which case its dummy version must contain more than one generic tuple. Thus, the general rule for determining the number of filler tuples (m) in a dummy relation r is as follows. If r's schema contains no case structures of the form `condition` $\mapsto$ `definition`, set m to 1. If one or more of the relation's attribute definitions contains a `case` construct, set m to *one plus the highest tuple number referred to in the* `condition` *part of any one of these* `case` *constructs.* For example, in the P&L schema, the definitions of `assumptions` and `proforma` contain 0 and 1 `case` constructs, respectively. In the latter relation, the highest tuple number in the `case` construct is 5. Therefore, the `assumptions` and `proforma` dummy relations will contain one and six filler tuples, respectively. Note that the dummy `proforma` relation will contain six tuples irrespective of how many 'real' tuples 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 an attribute x in r's schema is of type `numeric`, `string`, `date`, or `logical`, the filler character `N`, `S`, `D`, or `L`, respectively,

is placed as the value of x in 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 $\mathcal{D}_\mathcal{S}$ component. Next, $\mathcal{D}_\mathcal{S}$ is synthesized with $\mathcal{S}$, as usual. That is to say, a separate synthesis algorithm for creating template spreadsheets is not necessary. Instead, a template spreadsheet for $\mathcal{S}$ can be always constructed by (i) cloning a dummy set of relations $D_\mathcal{S}$ from $\mathcal{S}$, as we have just described, and then (ii) synthesizing $\mathcal{S} + D_\mathcal{S}$ through the ordinary synthesis algorithm.

4.3 Adding the Editorial and the Binding Components

So far, we have focused on synthesizing a schema $\mathcal{S}$ and a data component $\mathcal{D}$ into a logical map, denoted $\mathcal{S} + \mathcal{D}$. In order to complete the transformation of the map into an executable spreadsheet, $\mathcal{S} + \mathcal{D}$ must be synthesized with the spreadsheet's *editorial* and *binding* components. This step involves many implementation details that are of little theoretical interest. We describe it here in broad terms, noting that the section can be skipped without losing the thread of the paper.

Note that if the structure of $\mathcal{S}$ and $\mathcal{D}$ has been altered after factoring, the original $\mathcal{E}$ and $\mathcal{B}$ components may no longer be compatible with $\mathcal{S} + \mathcal{D}$. As it turns out however, this is not a major problem. First, the $\mathcal{E}$ component can be modified by the user to match the structure of $\mathcal{S} + \mathcal{D}$. Second, a new (default) binding $\mathcal{E}$ can be generated for $\mathcal{S} + \mathcal{D}$ automatically. In general, a pre-processing stage can be used to ensure that $\mathcal{E}$ and $\mathcal{B}$ are compatible with $\mathcal{S} + \mathcal{D}$, and, if they are incompatible, highlight the discrepancies and allow the user to resolve them. Following this preprocessing, three additional steps are taken to complete the synthesis process. These steps are given in figure 10.

Put Figure 10 around here

The data structure that emerges from the synthesis process is a physical spreadsheet map that can be loaded into a host spreadsheet program. In the P&L example, steps `S1-S7` produce the map depicted in figure 3.

5 Discussion

The paper is based on four premises:

- First, in spite of their remarkable ability to create models quickly and effectively, conventional spreadsheet programs can lead to insidious problems such as duplication of modeling efforts, inconsistencies within and across models, and models that cannot interact with each other.
- Second, most of these problems are related to the lack of a unifying model management system – a system that treats spreadsheet models as a shared corporate resource, much like a database management system treats data.
- Third, spreadsheet model management systems will not be feasible until spreadsheets yield to a platform-independent representation that makes them accessible to other, non-spreadsheet, model- and data-management tools.
- Fourth, in order for such a representation to be practical, we must provide means to translate conventional spreadsheets into the representation, and vice versa.

This paper focused primarily on the last two premises. The key to our approach is a dual logical/physical perspective that identifies four principle properties in any given spreadsheet model: *schema*, *data*, *editorial*, and *binding*. The four properties and the algorithms that operate on them are summarized in Figure 11. In the figure, the area above the factoring/synthesis bubble corresponds to the physical realm of conventional 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 objects with distinct properties that can be constructed in different ways under the user's control. The top-down and bottom-up transitions between the physical and the logical views are made possible by the factoring and synthesis algorithms, respectively.

Put Figure 11 around here

Beginning with the physical realm, it is important to observe that our approach is completely unobtrusive to the standard practice of spreadsheet modeling. That is, unlike some software products, notably `Improv` [16] and `Javelin` [14], we do not expect people to change the normal way they build spreadsheets, nor do we propose a new spreadsheet modeling paradigm. Instead, we began with the working assumption that users would like to continue to build spreadsheets in such familiar environments as `123`, `Excell` and `Quattro`. Once implemented, though, we propose that some spreadsheets in the organization could be factored into their principle properties, either for documentation purposes or for the more ambitious objective of bridging a spreadsheet model management system.

In order to carry out this factoring operation, the only tools that the user needs are (i) an outlining macro, written in the host spreadsheet's macro language, and (ii) an implementation of the factoring algorithm described in section 3. For the purpose of this research, we have written the outlining macro in `Excell`, and the factoring process in a Pascal program that is launched from within `Excell`. We have used these tools to factor several spreadsheet models (including the P&L model described in the paper), and we are now in the process of refining and improving them. We are also completing work on a synthesis program that would allow users to construct executable spreadsheets from reusable objects. We view this as the first step toward developing a spreadsheet model management system.

Some of the benefits of our approach are depicted in figure 11. As the figure shows, once the four properties have been extracted from the spreadsheet's physical representation, they can be stored and managed in separate repositories which are *independent of spreadsheet programs*. Most importantly, spreadsheet *schemas* can be channeled to and managed by a system that supports model documentation, retrieval and reuse. Likewise, *data* properties 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 $\mathcal{B}$ properties, 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 one language to another (or, say, check its spelling) by operating directly

on its *editorial* property, which is essentially a list of textual labels implemented as an ASCII file. Similarly, a program could be written to manipulate the screen layouts of spreadsheets by operating exclusively on their *bindings* properties, without ever getting into their underlying data and formulae properties. Such a service would enable one to alter the spreadsheet's appearance without worrying about damaging its contents.

Hence, our dual logical/physical perspective has both 'micro' and 'macro' implications for spreadsheet modeling. At the micro level, the properties' modularity enables us to distinguish between different types of spreadsheet manipulations. *Neutral* manipulations, like transposing or moving relations around the screen, effect neither the $\mathcal{S}$ nor the $\mathcal{D}$ properties of the spreadsheet. *Data* manipulations, like adding or deleting rows or columns that correspond to repetitive tuples, effect only the spreadsheet's $\mathcal{D}$ property, leaving the $\mathcal{S}$ property intact. *Structural* manipulations, like adding or deleting rows and columns that correspond to attributes, effect both the $\mathcal{S}$ and the $\mathcal{D}$ properties of the spreadsheet. The key point here is as follows: once the property 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. 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.

In conclusion, the paper presents the conceptual framework, algorithms, and data definition language, necessary to take the standard practice of spreadsheet modeling one step beyond its present state of the art. To illustrate the feasibility of our approach, we have developed a Windows-based prototype that is capable of factoring Excell spreadsheets into FRL schemas. This work provides a foundation for developing (i) intelligent spreadsheet programs that 'understand' the model world of the user; and (ii) powerful spreadsheet model management systems that help manage and streamline 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.

Appendix: A Schema Definition Language

A schema definition language for spreadsheets 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 [10] 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 [17] 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-oriented relational systems, e.g. `Cactis` [12, 13] and `OZ+` [23]. In `Cactis`, functional attributes were implemented using attribute grammar techniques [15]. In `OZ+`, value dependencies were implemented through functions that operated on objects. Coming from a different direction, Ginzburg and Kurtzman [11] provided a relational view of spreadsheets through their *Spreadsheet History Schemes*, which once again contains a distinction between 'given' attributes and 'evaluated' attributes. Another relevant work is Campeoli and Lucchesi's [3], who proposed a spreadsheet-like interface for relational databases, called *Spreadview*. In a spreadview, each attribute is split into two components – a *head* and an *index* – which are used to map its values on a spreadsheet matrix. The resulting representation enables interesting user-interface manipulations such as rotating or transposing

relations, as well as relational operations such as project and join.

For the purpose of this research, we have sought a language that will allow us to carry out *two-way* transformations, from spreadsheets to relations, and vice versa, with similar ease. The resulting language, called FRL, is specifically suited for this purpose. In addition to standard relational features that can be found in other data definition languages, FRL offers both absolute and relative tuple addressing, in line with the addressing style of spreadsheet formulae. It is important to emphasize, though, that FRL is not intended to be a user- or even a programmer-oriented language. Instead, it should be viewed as an internal representations of spreadsheet schemas – a representation that lends itself to database and model-base management systems. The users need not know FRL because their model schemas are generated (through factoring) from the spreadsheets that they construct in the host spreadsheet program. Having said that, note that it is entirely possible to create spreadsheet schemas directly in FRL, and then have the synthesis process transform them into conventional spreadsheets.

The remainder of this appendix provides an overview of FRL, as it unfolds in the context of the P&L example (Figure 2).

Functional-relations: A functional-relation is a tabular data structure consisting of one or more attributes and one or more tuples. Each relation has a mandatory *name* and an optional *alias*, or abbreviated name. We distinguish between relations that normally contain many tuples, 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. The definitions of *functional* at-

tributes 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 a later BNF section.

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, each relation r must have at least one attribute x such that no two tuples in r have the same x value. If a certain relation does not have a natural or a user-supplied key candidate associated with it, a hidden *system key* which is essentially a tuple identifier is attached to the schema by default. 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 $k' < k$. 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 the following function from natural numbers to key values:

$$keyval(n) = \begin{cases} \min(key) & \text{if } n = 1 \\ \overbrace{succ(succ(\ldots succ}^{n}(\min(key))\ldots) & \text{if } n > 1 \end{cases}$$

Where n is a natural number, $\min(key)$ is the minimal value in the key's domain, and *succ* is the familiar successor function. Thus, if the domain of *key* is, say, the set {1992, 1993,1994, 1995,1996, 1997}, then $keyval(3) = 1994$.

Tuple addressing: Since a functional-relation r always has a totally ordered key, the relation's tuples can be indexed uniquely, either relatively or absolutely. In absolute indexing, the term $r[i]$ for some $i > 0$ refers to the tuple whose key value is $keyval(i)$. In relative indexing, the special term r[n] is used to refer to r's *current tuple*, and the term r[n + i] for some i (which may be either negative or positive) is used to refer to the tuple whose key value is i positions away from the key value of r's current tuple.

Attribute Addressing: As a rule, the value of an attribute x in the ith tuple (in the order of the key) in a relation r is denoted $r[i].x$. Thus, `p[3].sales` refers to the `sales`

value in the tuple of the p relation whose key value is 1994 (since $keyval(3) = 1994$). Two default rules are used to abbreviate these attribute-references. First, the value of x in the current tuple, i.e. $r[\mathtt{n}].x$, is abbreviated to $r.x$. 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.

To illustrate, consider the attribute definition `net: inc-tax` from Figure 2. This expression is a shorthand of `p[n].net: p[n].inc-p[n].tax`, meaning *this* year's net income equals *this* year's gross income minus *this* year's tax. Similarly, the attribute definition `cogs: sales*a.cogs` is a shorthand of the expression `p[n].cogs: p[n].sales * a[1].cogs`. The latter example illustrates another FRL syntax convention: when referring to the attributes of vector relations, there is no need to specify an index, because these relations contain only one tuple. Thus, expressions like `a.cogs` and `a.grate` are interpreted without ambiguity as `a[1].cogs` and `a[1].grate`, respectively.

Operators and Functions: The definition of functional attributes involves *operators* like + and −, *scalar functions* like SQRT and ABS, and *list functions* like SUM and AVG. The operators and the scalar functions operate on single-valued operands (attributes or constants), whereas the arguments of the list functions are lists of values, implemented in FRL as relational projections. Without getting into implementation details, suffice is to say that every spreadsheet operator and function has an equivalent definition in the FRL language with a slightly modified syntax.

Case structures: In addition to the standard spreadsheet operators and functions, FRL supports *case structures* that are reminiscent of inductive, or recursive, function definitions. In its most general form, the case construct has the following form:

$$\begin{array}{lll} \mathtt{attribute:} & i_1 \leq \mathtt{n} < i_2 & \mapsto \ exp_1 \\ & i_2 \leq \mathtt{n} < i_3 & \mapsto \ exp_2 \\ & \vdots & \\ & i_m \leq \mathtt{n} & \mapsto \ exp_m \end{array}$$

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 some 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[n].inc: p[n].sales-p[n].lease-p[n].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 tuple number i (in the order of the relation's key), one binds `inc` to the value $f(i)$. In a similar way, an expression like `lease: numeric` is in fact equivalent to the functional expression `p[n].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*.

It is instructive to compare the `Excell`-based P&L 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 model is expressed in a platform-independent language – FRL – yielding a clear and succinct description of the model's underlying structure.

The description of FRL was given in an appendix because the language proper plays a secondary role in the paper. The paper's main focus is the duality between the physical and logical views of spreadsheets, and the fact that they can be generated from each other using factoring and synthesis. To that end, any formal language that supports factoring and synthesis will do, and FRL is only one example which we found to be quite useful for our purposes. As it stands now, FRL is a "version-0" language that can benefit from many additional features, e.g. module definition capabilities and the ability to capture circular attribute definitions. We intend to improve the language in these directions as we continue

our research in the area of spreadsheet model management, where a more powerful language will be necessary.

FRL in BNF

```
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

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[key = Ref].Attr_name |
                    R_alias_name[Ref].Attr_name

Ref             ::= Num_expr | Attr_expr

Num_expr        ::= n | Num_expr + NAT | Num_expr − NAT

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      ::= < | ≤ | = | > | ≥

NUM             ::= numeric constants, (any rational number)
NAT             ::= natural number constants, 1,2,3,...
STRING          ::= string constants
DATE            ::= date constants
LOGICAL         ::= logical constants
```

Constraints:

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 r has a key, 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 $v_2 < v_1$. 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* t_i (denoted $t_{i+1} >>_r t_i$).

   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 $glb_r(v) = max(t.k | t \in r \text{ and } t.k < v)$); and of the tuple *closest from above*, i.e. the tuple with key $lub_r(v) = min(t.k | t \in r \text{ and } t.k > v)$.

   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.* There are three kinds of *ref*s, as follows:

   (a) *Absolute:* denoted by i, where i is a number. This is a reference to the ith tuple of r, in the order of the keys.

   (b) *Relative:* denoted by an expression of the form n, or n $\pm$ j for some number j. The interpretation of n is the *current tuple* of r, n$-j$ is the jth previous tuple, and n$+j$ is the jth next tuple, as defined above.

   (c) *Named Attribute:* denoted by an attribute name *att*, or an expression involving *att* and the functors *prev, next, lub, glb.* A reference r[key= *att*] points to the tuple in r whose key value equals the value of the attribute *att* 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.

References

[1] Robert H. Bonczek, Clyde W. Holsapple, and Andrew B. Winston. *Foundations of Decision Support Systems.* Academic Press, New York, 1981.

[2] Polly S. Brown and John D. Gould. An Experimental Study of People Creating Spreadsheets. *ACM Transactions on Office Information Systems*, 5(3):258–272, July 1987.

[3] Alessandro Campioli and Luciano Lucchesi. SPREADVIEWS. In D. Karagiannis, editor, *Proceedings of the International Conference on Databases and Expert Systems Applications (DEXA-91) Berlin, Germany, 21-23 August,*, pages 525–530. Springer-Verlag, 1991.

[4] Paul B. Crag and Malcolm King. Spradsheet Modeling Abuse: An Opportunity for OR? *J. Opl. res. Soc.*, (forthcoming), 1992.

[5] Steve Ditlea. Spreadsheets Can Be Hazardous to Your Health. *Personal Computing*, pages 60–93, January 1987.

[6] Daniel R. Dolk and Benn R. Konsynski. Model Management in Organizations. *Information & Management*, 9:35–47, 1985.

[7] Barry D. Floyd and Jisurk Pyun. Errors in Spreadsheet Use. Working paper 167, Center for Research in Information Systems, New York University, Information systems Department, New York, NY 10012, 1987.

[8] R. M. Freeman. A Slip of the Chip on Computer Spreadsheets Can Cost Millions. The Wall Street Journal, August 4 1986.

[9] Dennis F. Galletta, D. Abraham, M. E. Louadi, W. Leske, Y. A. Pollalis, and J. L. Sampler. An Empirical Study of Spreadsheet Error-Finding Performance. *Accounting, Management, and Information Technologies*, Forthcoming, 1993.

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

[11] 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.

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

[13] 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.

[14] Javelin Software Corporation. *Javelin Reference Manual*, 1985.

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

[16] Lotus Development Corporation. *Improv for Windows Release 2.0*, 1993.

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

[18] D. Mason and D. Keane. Spreadsheet modeling in practice: solution or problem. *Interface*, pages 82–84, 1989.

[19] P.L. Mcmickle. Troubleshooting spreadsheets. *Journal of Accounting and EDP*, 3(2):69–71.

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

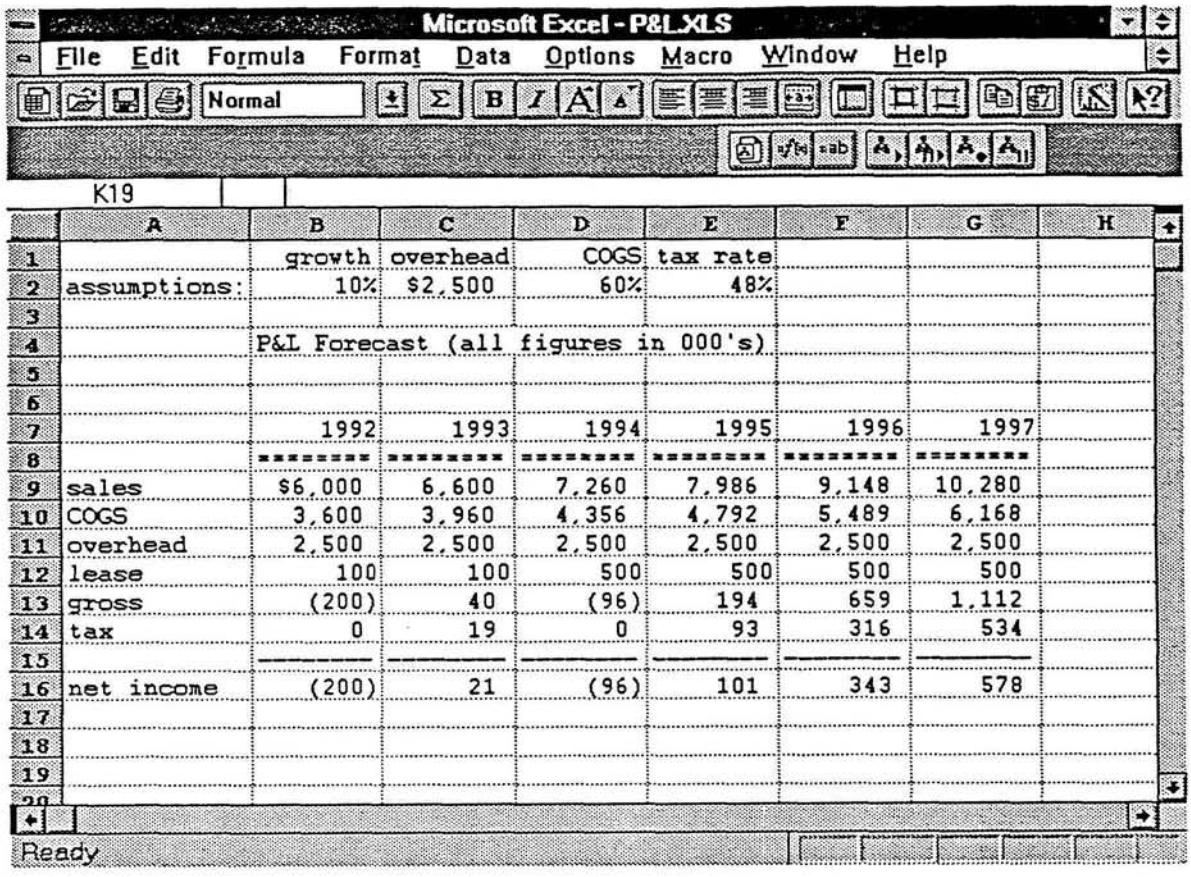
[21] M.G. Simkin. Micros in Accounting – how to validate spreadsheets. *Journal of Accountancy*, pages 130–138, August 1987.

[22] Ralph H. Sprague and Eric D. Carlson. A Framework For Decision Support Systems. *Database*, 4:1–13, 1980.

[23] 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.

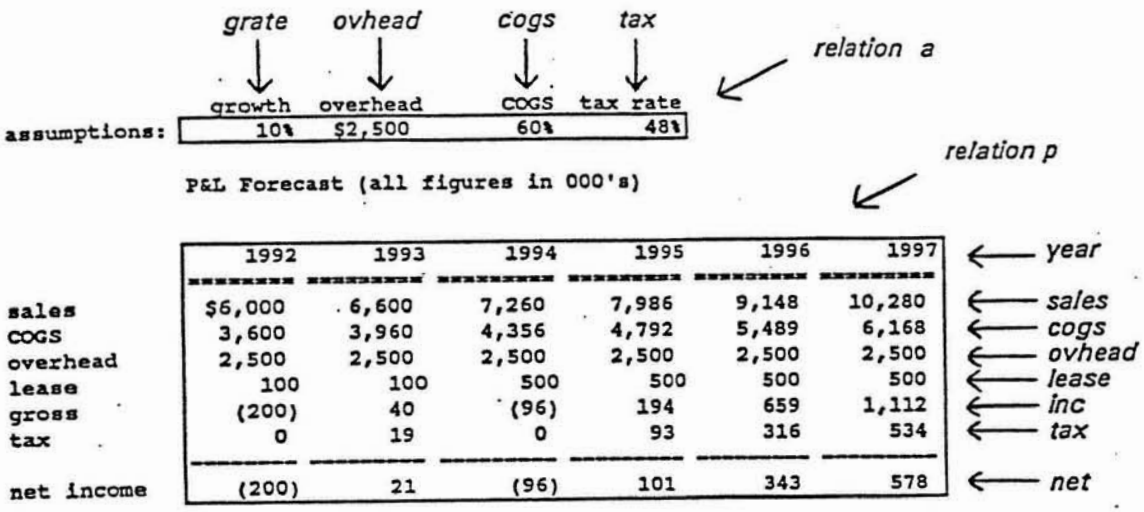
[24] T. Williams. Spreadsheet Standards. Technical report, Touche Ross & Co., 1987.

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



	A	B	C	D	E	F	G	H
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	
17								
18								
19								

Figure 1: The P&L Spreadsheet, in Excell



Schema ($\mathcal{S}$)

Data ($\mathcal{D}$)

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[n−1].sales*(1+ a.grate)
  $n \geq 5 \mapsto$ 0.5*(p[n−1].sales+p[n−2].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
1992	6000	C	C	100	C	C
1993	C	C	C	100	C	C
1994	C	C	C	500	C	C
1995	C	C	C	500	C	C
1996	C	C	C	500	C	C
1997	C	C	C	500	C	C

Figure 2: The outlined P&L spreadsheet (top) and its respective schema (left) and data property (right).

```
                B1:  'growth
                C1:  'overhead
                D1:  'COGS
                E1:  'tax rate
                A2:  'assumptions:
a[1].grate      B2:  0.1
a[1].ovhead     C2:  2500
a[1].cogs       D2:  0.6
a[1].tax        E2:  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:  6000
p[2].sales      C9:  +B9*(1+$B$2)
p[3].sales      D9:  +C9*(1+$B$2)
                A10: 'COGS
p[1].cogs       B10: +B9*$D$2
p[2].cogs       C10: +C9*$D$2
p[3].cogs       D10: +D9*$D$2
                A11: 'overhead
p[1].ovhead     B11: +$C$2
p[2].ovhead     C11: +$C$2
p[3].ovhead     D11: +$C$2
                A12: 'lease
p[1].lease      B12: 100
p[2].lease      C12: 100
p[3].lease      D12: 500
                A13: 'gross
p[1].inc        B13: +B9-B10-B11-B12
p[2].inc        C13: +C9-C10-C11-C12
p[3].inc        D13: +D9-D10-D11-D12
                A14: 'tax
p[1].tax        B14: @IF(B13>0,B13*$E$2,0)
p[2].tax        C14: @IF(C13>0,C13*$E$2,0)
p[3].tax        D14: @IF(D13>0,D13*$E$2,0)
                B15: \-
                C15: \-
                D15: \-
                A16: 'net income
p[1].net        B16: +B13-B14
p[2].net        C16: +C13-C14
p[3].net        D16: +D13-D14
```

Figure 3: The annotated map of the P&L spreadsheet, as produced by the *outlining macro*. Because of space limitations, the map covers only years 1992, 1993, and 1994, of the spreadsheet. To avoid clutter, the formatting-specifications of the cells are not depicted here.

F1: For each constant map-entry of the form ($r[i].x$: *cell_address, constant*), rewrite the entry as ($r[i].x$: *cell_address, data_type*), where *data_type* is the type of the *constant.*

F2: For each formula map-entry of the form ($r[i].x$: *cell_address, formula*), rewrite the entry as ($r[i].x$: *cell_address, formula'*), where *formula'* is the same as *formula*, except that all the physical cell-addresses that appear in *formula* are substituted with the labels of their corresponding map-entries, using a lookup operation. If a physical cell address is prefixed by a $ sign in *formula*, insert a $ sign before the tuple index of its respective label in *formula'* as well.

F3: Remove the *cell_address* terms from all the map-entries.

F4: Collate the map entries $r[i].x$ in clusters, so that each cluster contains entries that have identical relation names (r) and attribute names (x). Within each cluster, sort the map entries by the tuple index (i).

Figure 4: The first four steps of the factoring algorithm, which transform a physical spreadsheet map into a logical map.

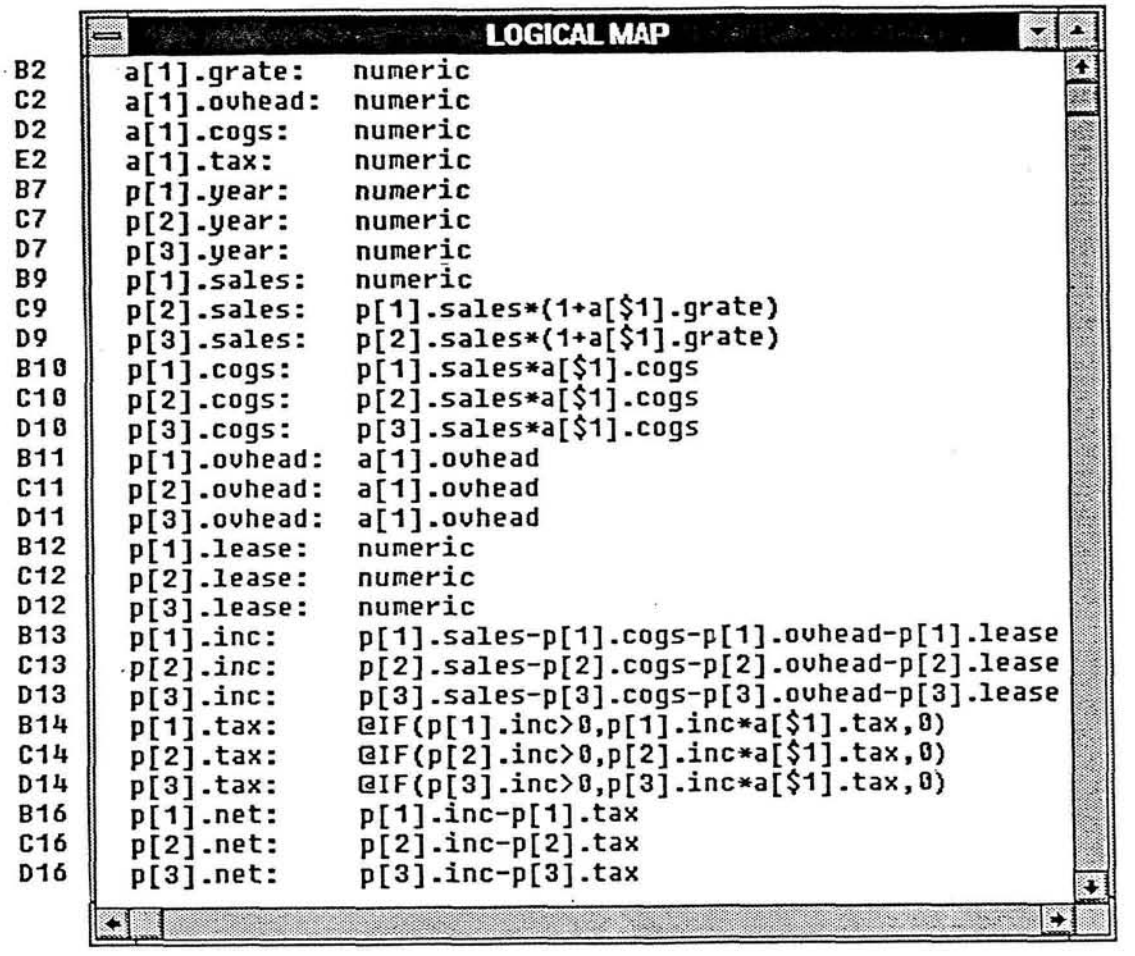


Figure 5: The P&L spreadsheet's logical map – the output of steps F1-F4 of the factoring algorithm. The cell addresses on the left margin are not part of the logical map, and are listed here only for reference purposes.

F5: *Convert absolute tuple references into relative tuple references.* For each map-entry whose label is $r[i].x$: if the entry's definition contains a related and *non-fixed* attribute-reference $r[j].y$ (an attribute-reference with the same relation prefix whose tuple index is not prefixed by a \$), let $d = j - i$. If $d > 0$, rewrite the related attribute-reference as: $r[\mathrm{n} + d].y$; if $d < 0$, rewrite it as $r[\mathrm{n} - d].y$; if $d = 0$, rewrite it as $r[\mathrm{n}].x$. Complete this operation throughout the map. †

Next, for each attribute-reference of the form $r[\$i].x$, rewrite the reference as $r[i].x$ (i.e. delete the \$ prefixes from all the attribute-references throughout the map).

F6: *Contract the map.* (Following F5, some $r.x$-clusters in the map contain one or more sets of repetitive map entries, i.e. entries that have exactly the same right hand side definition.) For each set of repetitive entries, eliminate all but the first entry in the set (the entry with the lowest index).

Compute tuple index ranges. (At this point, each $r.x$ cluster consists of one or more entries, each with a different definition.) Let the cluster's entry-labels be $r[k_1].x$, $[k_2].x$, ..., $r[k_m].x$. For each cluster, rewrite the entry-labels as $r[k_1 \leq \mathrm{n} < k_2].x$, $r[k2 \leq \mathrm{n} < k_3].x, \ldots$, $r[\mathrm{n} \geq k_m].x$. If a rewritten entry-label becomes $r[j \leq \mathrm{n} < j + 1].x$ for some j, rewrite it again as $r[\mathrm{n} = j].x$; If a rewritten entry-label becomes $r[1 \leq \mathrm{n} < j].x$, rewrite it again as $r[\mathrm{n} < j].x$. If the cluster consists of only *one* entry, rewrite its single label as $r[\mathrm{n}].x$.

F7: *Complete the schema.* Consult the user-defined spreadsheet's outline to obtain each relation's (i) *full name*; (ii) *cardinality* (*single* vs *multiple* tuples); and (iii) *key*. Use these specifications and FRL's syntax default rules to transform the map into a formal spreadsheet schema.

† Notational comment: throughout the algorithm, i, j, and d represent numbers, whereas n is a textual tag, i.e. the fixed character 'n'.

Figure 6: The last three steps of the factoring algorithm, which transform a spreadsheet's logical map into a spreadsheet schema.

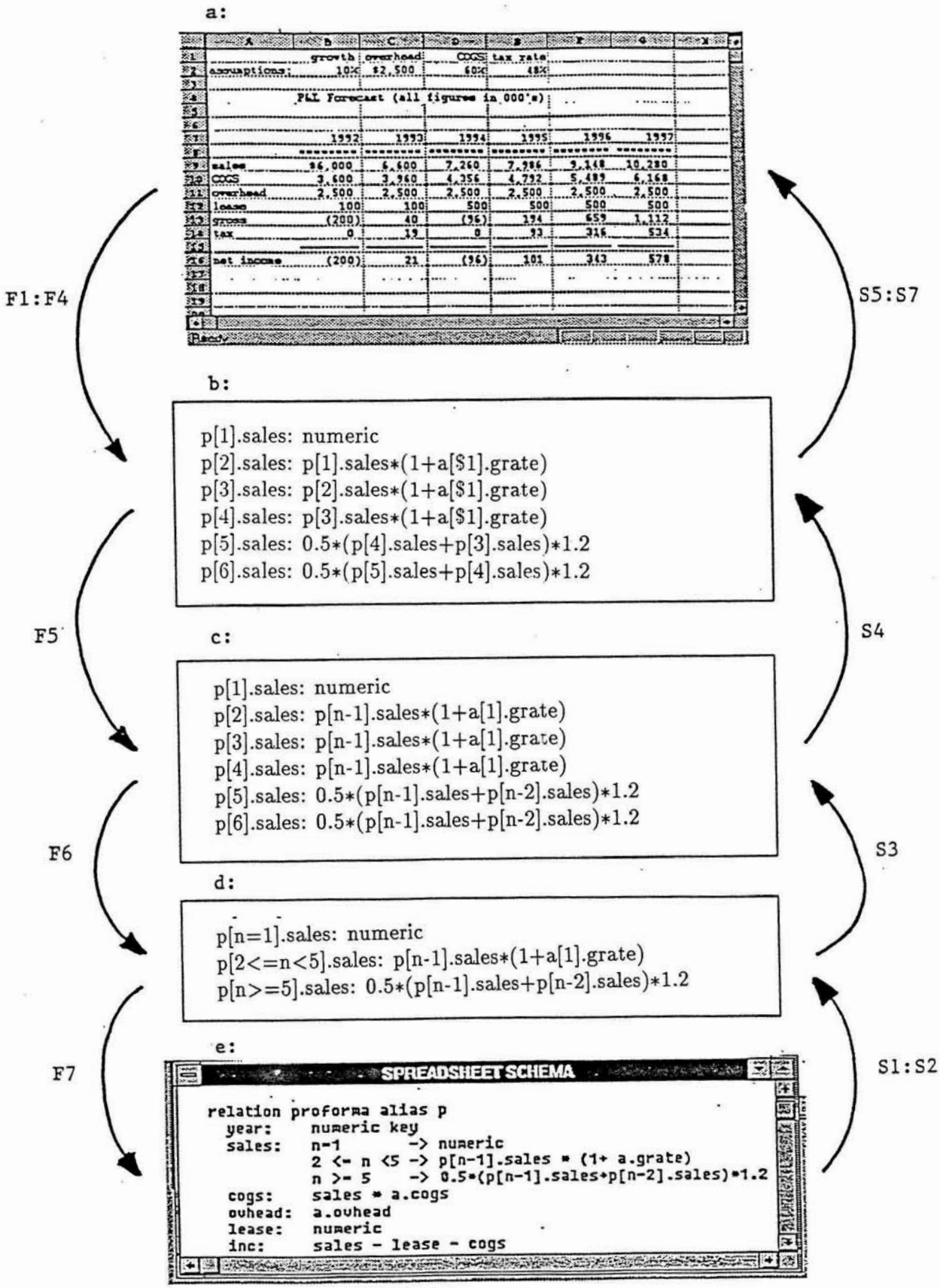


Figure 7: A step-wise illustration of the various stages of factoring and synthesis, as they operate on the `p.sales` attribute of the P&L spreadsheet.

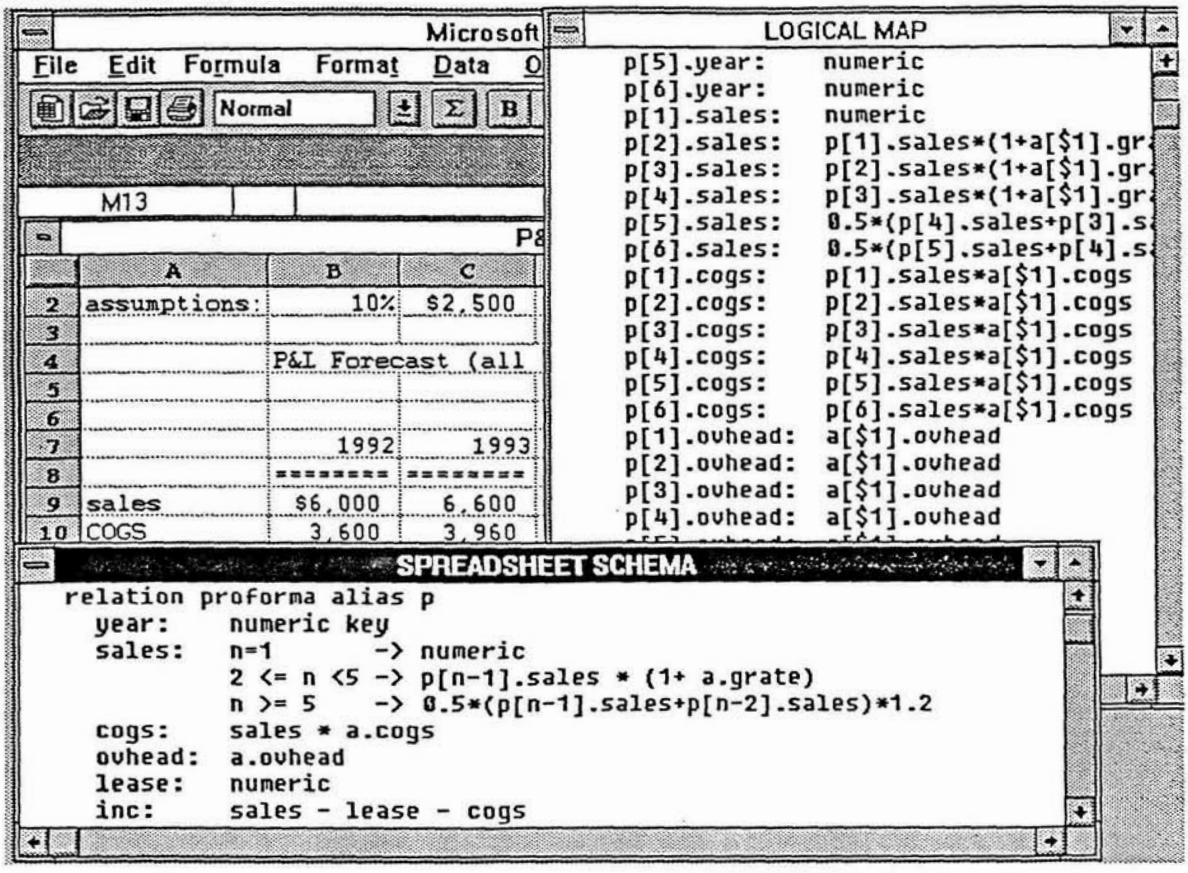


Figure 8: Three snapshots of the factoring process. The background window is the original P&L spreadsheet, in Excell. The foreground window is an excerpt of the spreadsheet's schema. The middle window – an interim result that the user does not normally see – is the spreadsheet's processed logical map.

S1: Let the schema of relation r consist of one or more lines of the form (x: *definition*), where x is an attribute name.

- **(a)** If x's *definition* contains no case structures, rewrite the line as (r[n].x: *definition*). †
- **(b)** If x's *definition* contains lines of the form $condition_j \mapsto definition_j$, rewrite the lines as ($r[condition_j].x$: $definition_j$).

S2: For each attribute-references that occurs in the *definition* part of each line, use FRL's syntax rules to rewrite the attribute-references in an extended, no-defaults syntax. If an attribute-reference becomes $r[j].x$ for some constant j, rewrite it further as $r[\$j].x$.

Next, eliminate the header of r from the schema.

S3: Let m be the cardinality (the number of tuples) of r.

- **(a)** Replace each line of the form (r[n].x: *definition*) with a series of m lines of the form ($r[1].x$: *definition*), ..., ($r[m].x$: *definition*).
- **(b)** Replace each line of the form ($r[k_1 \leq \text{n} < k_2].x$: *definition*) with a series of $k_2 - k_1 + 1$ lines of the form ($r[k_1].x$: *definition*), ..., ($r[k_2 - 1].x$: *definition*).
- **(c)** Replace each line of the form ($r[\text{n} \geq k].x$: *definition*) with a series of $m - k + 1$ lines of the form ($r[k].x$: *definition*),..., ($r[m].x$: *definition*).
- **(d)** Replace each line of the form ($r[\text{n} = i].x$: *definition*) for some i with a *single* line of the form ($r[i].x$: *definition*).

S4: If $r[i].x$ is the label of a line, and $r[\text{n} + j].y$ is a *related* attribute-reference in the line's *definition* (i.e. an attribute-reference whose relation prefix is also r), let $d = i + j$ (note: j may be either negative, zero, or positive). For each such line, rewrite its related attribute-references as $r[d].y$.

† Notational comment: throughout the algorithm, m, i, j, and d represent numbers, whereas n is a textual tag, i.e. the fixed character 'n'.

Figure 9: The first four steps of the synthesis algorithm, which transform a spreadsheet schema to a logical map.

S5: *Construct the physical spreadsheet map:* For each logical map-entry of the form ($r[i].x$: *definition*) and a matching binding-entry (element of the list $\mathcal{E}$) of the form ($r[i].x$: *cell_address*), create a physical map-entry of the form ($r[i].x$: *cell_address, definition*).

S6: *Convert relational references to cell addresses:* For each map-entry of the form ($r[i].x$: *cell_address, formula*), replace *formula* with *formula'*, where *formula'* is the same as *formula*, except that each attribute-reference $r[i].x$ that appears in *formula* is substituted with the *cell_address* of the map-entry whose entry-label is $r[i].x$.

S7: *Add the editorial entries:* Merge the list of map-entries produced S1-S7 with the list of editorial entries From $\mathcal{B}$.

Figure 10: The last three steps of the synthesis algorithm: adding the editorial and binding components.

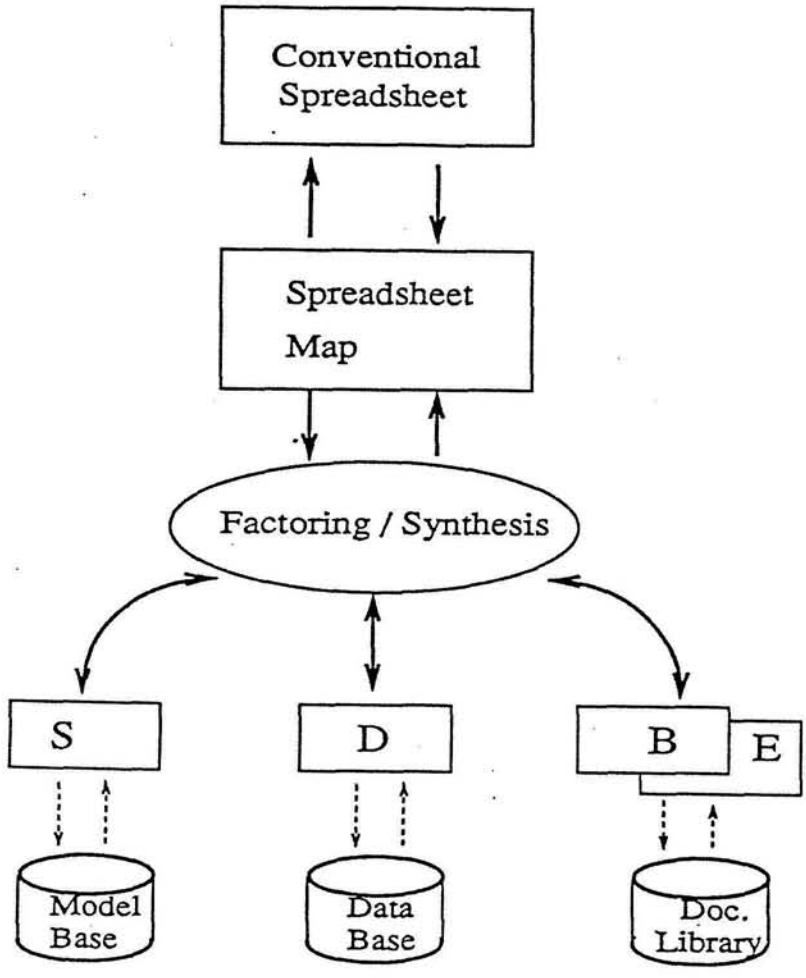


Figure 11: A spreadsheet model and its four components. Up arrows represent synthesis; down arrows factoring.