Object Spreadsheets: A New Computational Model for End-User Development of Data-Centric Web Applications *

Matt McCutchen
Massachusetts Institute of Technology, USA
rmccutch@mit.edu

Shachar Itzhaky
Massachusetts Institute of Technology, USA
shachari@mit.edu

Daniel Jackson
Massachusetts Institute of Technology, USA
dnj@mit.edu

Abstract

Spreadsheets offer many advantages as the computational and data-storage engine for applications that are authored by end users. Paradoxically, however, their main failing in this regard is their computational model. Despite being used in almost all cases to represent data that is essentially relational (with some hierarchical structuring), the spreadsheet model treats the two-dimensional grid as largely unstructured, with formulas linking cells in an ad hoc way.

This paper reports on a quest to rethink the spreadsheet model. The model we propose supports not only conventional flat tables, but also nested variable-size lists and object references. It includes a formula language suited to the data model and procedures to specify updates.

The model has been implemented in a tool called Object Spreadsheets, which is intended for the development of data-centric web applications. We describe several example applications we built using the tool to demonstrate its applicability.

Categories and Subject Descriptors H.4.1 [*Office Automation*]: Spreadsheets; D.1.7 [*Visual Programming*]; D.2.6 [*Programming Environments*]: Interactive environments; H.2.1 [*Logical Design*]: Data models; H.2.3 [*Languages*]: Database programming languages, Query languages

General Terms Design, Human Factors, Languages

Keywords End-user development

* This project was supported by a grant from Wistron Corporation as part of a collaboration between Wistron and MIT's Computer Science and Artificial Intelligence Laboratory. This project also received partial support from the National Science Foundation under Grant No. CCF-1438982.

Onward!'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4076-2/16/11...
http://dx.doi.org/10.1145/2986012.2986018

1. Introduction

1.1 Languages and Users

The most widely used programming language is not Python or Java or C++, but the formula language of Microsoft Excel. A 2005 study [33] estimated that 12 million people in the US call themselves programmers, but 50 million use spreadsheets and databases, more than half of the 90 million who use computers at work. So it's not surprising that developers turn to spreadsheets when they want to provide a way for users to download or upload application data, or to interact with persistent data (as in Google Forms). The spreadsheet is becoming a kind of universal data structure with "add row to spreadsheet" (as it's called in IFTTT) being the fundamental operation.

1.2 A Fundamental Mismatch

But there's an irony at the heart of the popularity of the spreadsheet. In most applications, a spreadsheet is a table: a homogeneous structure in which each row contains the same types of data item in the same order. And yet the spreadsheet computational model isn't designed for tables, and "add row" is a UI operation that is not part of the formula language. On the contrary, the spreadsheet model treats the two-dimensional grid of cells as unstructured, with formulas expressed as algebraic expressions defining one cell in terms of some other cells, either at fixed locations or at particular relative displacements. The only nod to the table structure is found in the notation for horizontal and vertical ranges. Consequently, spreadsheets often hold relational data—that is, data structured as it would be in the tables of a relational database—but they have no relational operations with which to manipulate it.

This mismatch, between the spreadsheet's computational model and its most common use, leads to friction and complexity, and makes it hard to build powerful applications on top of spreadsheets. Google Forms, for example, uses a spreadsheet to present completed form inputs, with one row per response. And yet Google's own code to display summaries seems to rely not on this spreadsheet, but instead on some hidden data structure, which is presumably more

amenable to the needed queries. (To observe this, just edit the data in the spreadsheet, say by deleting rows or columns, or hiding columns, and notice that the summaries do not change accordingly. This means that if you include a question in your form that solicits optional personally identifying data, you will not be able to exclude it from summaries you might otherwise have wanted to share.)

In one respect, though, spreadsheet data is often not purely relational: it has hierarchical structure. Database programmers aren't troubled by hierarchical data because they never need to look directly at the content of the database tables. If a table representing a company's org chart maps divisions to departments, it matters little that the rows of the table repeat the division identifier for each department in that division. When the COO makes a spreadsheet with the same data, however, she will likely want to see the org chart displayed as a horizontal tree, with each division having a list of its departments next to it. In fact, spreadsheet users commonly start with the visual layout, and then add formulas later. The resulting formulas are brittle, and will generally have to be rewritten if the data changes size (say when removing a row or a cell).

1.3 A Programming Core for Simple Apps

Computer scientists are often faced with explaining technical issues to friends. Perhaps the most frustrating question to answer is why building a simple web app should be so hard, often requiring months or years of work by an experienced team. "All I want is an app that lets my students enter their team preferences, checks to make sure each person is on exactly one team, and then notifies them of the assignment. How hard can that be?"

The answer, sadly, is "very hard". We find ourselves in a conundrum: we know it would be rash to suggest a foray into Rails or Meteor, and we're not confident that a graphical app builder will make the job that much easier, or will even be able to express the required constraints[1]. So we swallow our professional pride and tell our friend what we would do ourselves: give all the students access to a Google spreadsheet and hope for the best (or create a form and resolve inconsistencies by hand later).

If we were to build a new development environment with a simple but expressive structure at its core, what would that core be? As the most successful end-user development tool in computing history, the spreadsheet is the obvious candidate. What makes the spreadsheet attractive, though, is not its computational model. On the contrary, spreadsheets are great for representing relational data not because of this model but in spite of it. Rather, it's the visual layout of the data (especially the flexibility to arrange it hierarchically, with some cells occupying multiple rows), the ability to enter data and formulas at once and get immediate feedback, and the fact that the data and its schema occupy the same structure (more on this later).

[1] The reasons that app builders such as QuickBase haven't replaced programmer tools such as Meteor for simple websites are not entirely clear, but we discuss some of their key limitations later in the paper.

1.4 A Quest for a New Spreadsheet Model

In our view, then, there is a single major obstacle to exploiting spreadsheets for building web apps. Although spreadsheets are ideally suited to representing and displaying relational (and hierarchical) data, their formula languages are not up to the task. Our challenge is to find a new computational model, with a new formula language, that preserves the simplicity of the traditional spreadsheet model, but which treats the spreadsheet as a table (or a collection of tables)—not as a two-dimensional array of arbitrarily related data items. This paper reports on our effort to meet this challenge.

Due to their popularity and known deficiencies in their computational model, spreadsheets have been the subject of many research efforts. Most of these, however, have focused on extending them with capabilities such as more powerful programming languages[11, 31], user-defined functions [6, 30], and stream processing [36]. In contrast, our project aims to reconsider the most fundamental assumptions of the spreadsheet data model.

In short, our model replaces computation over cells with computation over columns, with a formula per column that defines the value of each of its cells. Hierarchy is not just a visual representation of a relational redundancy, but is fundamental to the structuring of the data. As the simplest example, the summary cell of a table in a traditional spreadsheet (representing, for example, a total of the costs of items in a shopping cart) becomes a cell in a new column (giving the total of each cart, now itself represented explicitly as an *object* containing its items). The formula language is adapted so that, unlike a standard spreadsheet language, its terms refer not to individual cells and their values, but rather to sets of cells and sets of values. These cells are addressed not by their coordinates on the two-dimensional grid, but by their location in the data structure. An extension of the language with mutating operations provides the ability (amongst other things) to programmatically add rows to a table; these operations are packaged into "transactions" that can be bound to buttons in the user interface.

1.5 Data-Centric Web Applications

While a new computational model is the primary contribution of this paper, our ultimate aim is to build a new end-user application development environment based on the spreadsheet paradigm. We have therefore selected a class of application that is increasingly important and an ideal target for this approach.

Data-centric applications are interactive web applications that involve routine but non-trivial manipulations of data, to support sharing of information, small-scale social interactions, and business processes. They are unlike the kinds

of application that can be easily built with content management systems, because they typically require complex domain-specific queries and updates; and on the other end of the spectrum, they are unlike scientific applications because they do not require resource-intensive computations. An example is the kind of application a school would use to arrange parent-teacher conferences, which lets the family of each student schedule a meeting with each of the student's teachers, avoiding problems such as double-booking (Fig. 1).

Organizations with such a need face a dilemma: to adopt an off-the-shelf solution (which may be a less than perfect match to the requirements); to engage a developer (which is usually too expensive); or, as is most commonly done, to cobble together tools such as email, spreadsheets and online forms using form builders like Google Forms [16] and Wufoo [37] (leaving a considerable burden of manual work and possibly undesirable risks to data confidentiality and integrity).

Ideally, an organization's administrators would build a application themselves to their exact requirements (*end-user development*), but this approach is not as easy or as widely used as it could be. General-purpose web application frameworks continue to demand a high level of technical understanding from their users, even as design advances over time, such as scaffolding scripts and object-relational mapping, reduce the amount of code that has to be written. Existing *application builders* (such as App2You [22] and Intuit QuickBase [32]) make it easier for people with little or no programming experience to build data-centric applications of low to medium complexity, offering menu-driven, WYSIWYG, or other visual interfaces to specify the structure of the information stored and the ways in which users may view and update it. Such tools are more general than form builders, which allow unprivileged users to add records but offer very limited options (if any) for them to view and edit records other than their own.

Many organizations use application builders to great effect, but others continue to use piecemeal solutions, likely because the existing application builders are still too clunky and intimidating. A major source of the clunkiness is that the schema, data, and formulas are often spread across many screens, so developers must go back and forth to find the part of the application that needs to be edited and see the consequences of the edit. And advanced developers often prefer to use general-purpose frameworks because they scale better to moderately complex application logic, even though they lack some of the conveniences of application builders.

We propose that the use of the spreadsheet paradigm in an application builder can help to overcome both of these problems. Let's review the essential characteristics that give spreadsheets their appeal:

- A simple and flexible visual structure for organizing data;

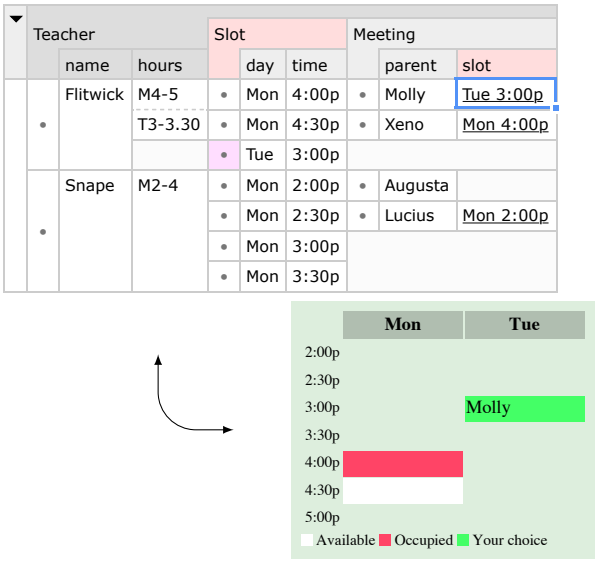


Figure 1. A simple parent-teacher conference application, with the Object Spreadsheet on the top and a Web UI below.

- The use of the very same structure and interface not only for data, but also for the schema underlying the data and the formulas defining queries on it;
- A continual computation strategy, which allows the impact on example data to be viewed as formulas are constructed and modified;
- A simple and declarative formula language that provides a smooth learning curve from simple data transformations, which can be selected visually, to more complex transformations written with the help of integrated language documentation and immediate feedback on subexpression results.

One can hope that a spreadsheet-based application builder would:

- Provide a smooth path for end-user developers who are comfortable with data entry and formulas in a traditional spreadsheet to specify additional structure on their data (tables, hierarchy, references, etc.) and build formulas that traverse that structure to achieve the desired application semantics.
- Avoid the clunkiness of going back and forth between different screens to work with the schema, data, and formulas.
- Feel powerful enough to win the respect of advanced developers, even as it offers significant guidance to novice developers.

However, adapting the spreadsheet paradigm to the development of data-centric web applications is not trivial, and in Section 2 we outline particular challenges. In fact, Quilt [5] is an example of a web application builder backed

by a traditional spreadsheet in the style we propose, but it does not address most of these challenges and consequently supports only the very simplest applications. Our aim in this project, therefore, has been to retain the essential appeal of the spreadsheet paradigm while providing good solutions to these challenges.

1.6 Object Spreadsheets

We have implemented a prototype tool, called Object Spreadsheets, that embodies our new computational model, and which is sufficient to build all the logic for a data-centric web application. The spreadsheet serves as the standard UI for development and administrative access to the application data; the developer would design a separate, customized *application UI* for regular use by unprivileged users. (In the rest of this paper, we consistently use the term "developer" for a person who defines the schema and logic of an application or spreadsheet, however simple, and "user" for a person who merely reads and writes data.)

In addition to the editable sheet with formulas, Object Spreadsheets supports stored procedures to define the updates that users of an application can make, and exposes an API for the application UI to display data and invoke procedures. We envision combining the tool with a suitable UI builder to provide a complete solution for application development that requires no prior knowledge of web technologies. We have implemented a prototype of the spreadsheet tool in the Meteor web application framework and demonstrated it on a collection of example applications, building the application UIs directly in Meteor using the exposed API.

1.7 Paper Outline

The rest of the paper describes in more detail how the logic of data-centric web applications can be built using Object Spreadsheets. Our contributions include:

- An analysis of the challenges of extending the spreadsheet paradigm to support data-centric application development;
- A data model (Section 3) and spreadsheet interface designed to handle web application data in a way that is natural to end-user developers;
- A simple formula language that supports the queries needed by our target applications (Section 4);
- A prototype implementation of the tool (Section 6);
- A suite of example applications that demonstrate common difficulties presented by this application class, and their implementations in our tool (also in Section 6).

Demos. Interactive demos of the example applications and a video demonstrating how to build an application with Object Spreadsheets are available on the project web site at `http://sdg.csail.mit.edu/projects/objsheets/`. These materials may help the reader quickly get a sense of what Object Spreadsheets does before reading further in the paper.

	A	B	C	D	E	F
1	room	sq. footage	occupant	role	alloc.	free
2	Dungeon Five	480	Sirius	Grad. student	12	**436**
3			James	Post-doc	20	
4			Wormtail	Grad. student	12	
5	Greenhouse Two	561	Bellatrix	Visiting Prof.	45	**476**
6			Lily	Post-doc	20	
7			Remus	Post-doc	20	

8	role	alloc. space
9	Grad. student	12
10	Post-doc	20
11	Visiting Prof.	45

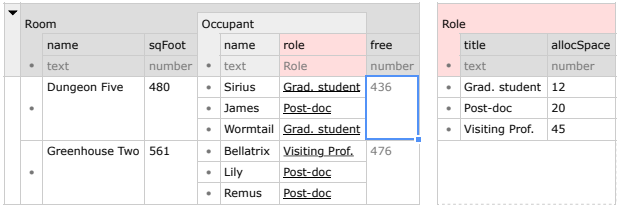
```
=VLOOKUP(D5, A$9:B$11, 2)
=B2−sum(E2:E4)
=B5−sum(E5:E7)
```

Figure 2. Tracking the used space in each room at a university department based on space allocation amounts for each role. Adding an occupant or room requires careful adjustment of the formulas. A real implementation would have a separate table of people like the table of roles above, but we omit this detail to simplify the example.

2. Challenges and Approach

In this section we outline the key challenges of extending a spreadsheet to support web application development and how they are addressed in our design. We point out a few alternatives and explain why we believe they are inferior. Further discussion of alternatives and similar systems is given in Section 7.

Nested variable-size sets. All but the simplest web applications contain one or more sets of objects and allow users to add objects to and remove objects from these sets. Many even include two or more nested levels of such sets. For example, consider an application used by an administrator to manage the space allocation for a university department, shown in spreadsheet form in Fig. 2. Each person is allocated an amount of space depending on their role, and people must be assigned to rooms in such a way that each room is large enough for the people assigned to it. The administrator is constantly facing the problem of finding a room with enough free space to accommodate the next person, so he wrote formulas that subtract the total allocated space from the square footage of each room. Unfortunately, this requires hard-wiring the cell ranges corresponding to the occupants of each room (`E2:E4` and `E5:E7`). So when he adds a new occupant to a room or adds a room to the list, he has to manually adjust the formulas to refer to the new cell ranges.




```
Room.free ≙ sqFoot − sum[o : Occupant](o.role.allocSpace)
```

Figure 3. An Object Spreadsheet for the space allocation example.

This example illustrates the fundamental challenges of handling variable-size sets in a spreadsheet. Applications require both per-item computations, such as the lookup of the allocated space for each person based on their role, and computations over sets, such as summing the allocated space for the occupants of a room. Actually, the latter computation is also a per-item computation at the room level. To support variable-size sets, a spreadsheet must be able to:

1. Fit as many items as are needed;
2. Automatically apply per-item computations to added items;
3. Maintain enough information to locate sets and their enclosing items as sizes change.

These capabilities are difficult to achieve in a traditional spreadsheet, in which data items and formulas are bound to individual cells in the grid and there is no paradigm for adapting the structure to programmatic changes in data size. One strategy to handle two levels of sets is to lay out one level (e.g. the rooms) along one axis, and another level (e.g. the occupants) along the other. This limits nesting to two levels, and is also hard to maintain when the inner items are composed of several fields, as in the example. Another strategy is to move all the inner items to a separate table with references to the outer items, as in a relational database, and use functions such as `SUMIF` to query all inner items that belong to a given outer item. But end-user developers may not know to try this transformation, and even if they do, it may present an ongoing burden to understanding of the application implementation.

In Object Spreadsheets, we solve the problem by abandoning the two-dimensional grid of cells as the fundamental data model in favor of a richer model that is merely viewed in a two-dimensional layout. The model is never dependent on the spreadsheet view for its correct functioning, and the view adapts to arbitrary changes in the size of the model; there are no issues of "running out of room in the grid". Some commands in the spreadsheet UI depend on the state of the view at the time of invocation, but they ultimately result in a change to the model that is expressed in view-independent terms.

The model we choose is based on what is historically known as the "hierarchical data model with virtual records"; it directly supports nested variable-size sets of objects. Every formula defines a computed field of an object type and is automatically evaluated on each object of that type in the sheet, ensuring uniformity. A root object is available to hold global values and formulas. The spreadsheet view uses the "nested table" layout with each object type occupying a range of columns and objects of the same type occupying vertically stacked rectangles, which is common in other tools [2, 3, 17]. The result for the space allocation example is shown in Fig. 3.

Object references. Web applications include relationships between objects, not all of which are well captured via hierarchy, which leaves a need for object references of some form. In the space allocation example (Fig. 2), this can be seen by the "role" of each occupant, which refers to a role listed in the table at the bottom. The administrator then wants to retrieve the allocated space for the role from this table. This can be done with the `VLOOKUP` function, but it becomes tedious and error-prone to specify the target cell range for each such lookup in an application. This approach is the analogue of a join or subquery on a foreign key in a relational database.

Another approach is to have the occupant's role cell store a cell reference in string form, such as "A11" in the case of Bellatrix, and use a formula like `=OFFSET(INDIRECT(D5),0,1)` to look up the allocated space. This approach avoids specifying the cell range of the role table but still requires significant boilerplate, and it fails if the role table must be moved to allow the room table to grow. Furthermore, to enter the role of an occupant into the sheet, the administrator has to manually look up the correct cell reference.

Object Spreadsheets provides object references that are analogous to the "A11" mentioned above, but since the data model supports objects directly, these references do not break when the layout changes. So if the developer defines an *object type* named Role corresponding to the role table, then references to individual Role objects can be stored in the "role" column of the occupant table and manipulated like any other data type. A dot notation is used to access fields of the target object, so the lookup of the allocated space might be expressed as "`=role.allocSpace`". The same notation is used to access ancestor or child objects in the hierarchy, for example, to retrieve all occupants of a room to compute the free space. We call these dot expressions *navigations*. Finally, Object Spreadsheets lets the developer designate one field of each object type (defaulting to the first field) as its *display field*, which is used as a string representation to display and input references to objects of that type in the spreadsheet. So, by default, a reference to a role from the occupant table would display the role title, underlined to remind the developer that it is a reference.

Binding the application UI to data. Any web application builder has to give the developer a way to specify what data should appear in a given page of the (user-facing) application UI. In existing tools such as QuickBase and App2You, each page is associated with a particular object type, and a form building interface is used to specify what fields of the object type and what tables of related objects to display. The amount of logic inherent in such UI building becomes nontrivial if related objects are nested or are filtered, potentially depending on parameters chosen by the user on the page.

We propose to harness the benefits of spreadsheets for this task by making each page merely a stylized view of a dedicated region of the spreadsheet that contains the data

for display. As described earlier in this section, our spreadsheet model has a hierarchical structure, which aligns well with how user interfaces are normally built, plus plenty of expressive power to select and assemble data. We discuss further details in Section 5.

Mutations. Finally, the developer must specify the kinds of mutations that users may make to the application's state. For example, the administrator of the space allocation application may want to allow other users to assign occupants, but not add rooms or alter their square footage. In the simplest case, if a web application builder has a flexible means to bind mutable state directly to a page, the developer could choose to allow edits to some of the displayed values and creation and deletion of objects that meet the criteria to appear in tables on the page, perhaps subject to conditions expressed as formulas. If the page is bound to a view defined by formulas on the source data, then the natural way to achieve equivalent functionality is to provide default view-update semantics for formulas with appropriate syntactic forms, as (for example) PostgreSQL does.

However, some of our target applications, such as Got Milk (see Section 6), have composite mutations that cannot be expressed in this form without complicated tricks. The most basic, general way to support such mutations is to use stored procedures and allow users to call certain procedures with arguments of their choice; the procedures would also receive some built-in parameters such as the identity of the calling user and the time. This is the approach we take. We designed a small procedural language as an extension of the formula language (thus, we hope, making it easy for end-users to understand). A room assignment procedure might look like this:

```
assign    room: Room, name: text, role: Role
          let a = new room.Occupant
          a.name = name
          a.role = role
          check room.free >= 0
```

3. Data Model

The design of the Object Spreadsheets data model reflects a compromise among the following goals:

1. Provide sufficient expressive power for moderately complex data-centric applications.
2. Stay close to an end-user developer's mental model of a data-centric application, which we assume is well-described as an entity-relationship model [9].
3. Support a spreadsheet-like interface that upholds the properties that make spreadsheets easy to use and understand.
4. Keep the design simple so we could produce a reasonably complete prototype demonstrating the essential characteristics of the system with modest development effort. We describe several future extensions that would be important in order to achieve the best possible usability.

Our data model is based on what is historically known as the "hierarchical data model with virtual records" and incorporates the following essential features of this model:

- Object attributes, or *fields*, as the basic unit of data, rather than relational tuples.
- Object references rather than explicit foreign keys.
- An ownership hierarchy of objects.

Our system also shares much of the philosophy of DAPLEX [34] and some of its design, including support for object references as a first-class data type and set-valued fields analogous to the "multivalued functions" of DAPLEX; we mention further similarities to DAPLEX in Section 4. We give an informal exposition of the design of Object Spreadsheets in this paper; for a mathematically rigorous specification, see [25].

A sheet in Object Spreadsheets consists of a *schema*, a *data instance* conforming to the schema, and a *program* of formulas and procedures. (Formulas are described in Section 4. We do not describe procedures in detail in this paper, but a few examples may be seen in Section 6.) The schema takes the form of a tree whose nodes are *object types* that describe the objects in the sheet and *fields* (which must be leaves), each of which gives the name and type of values that may exist in each object of the parent object type. The schema of (a representative part of) the space allocation sheet of Fig. 3 is shown at the top of Fig. 4.

The sheet's *data instance* consists of a tree of *objects* and *values* that parallel the object types and fields of the schema, respectively, in the sense that each data node d is described by a schema node that is a child of the schema node describing the parent of d. This can be illustrated by the data tree for the space allocation sheet, shown at the bottom of Fig. 4; the correspondence to the schema nodes is indicated by vertical dotted lines. In our model, an object may have more than one value in a field (though this does not occur in the space allocation sheet); we discuss the reason for this decision in Section 4.4. Note that every sheet has a *root object*, which is the unique object of the *root object type*. Its fields can be used to store global values.

Objects in Object Spreadsheets are much like database records, but we avoid the use of the term "record" because it has the connotation of being flat and rigid. Objects do not currently support many of the features of object-oriented programming, such as methods; such features may be reasonable extensions if they are consistent with the spirit of the system.

The schema and data trees are currently displayed in a layout similar to the "nested table" layout of [4]:

- The schema nodes (taken in preorder) become the columns of the grid, with each object type drawn as spanning its descendants in the schema.
- Each data node becomes a cell in the corresponding column. An object cell has a bullet icon, while a value cell displays the actual value.
- The data subtree rooted at an object o of type T lies in a rectangle spanning the columns representing the schema subtree rooted at T. The object cell spans the height of this rectangle, as do value cells in single-valued fields.[2]

This presentation is simple and unambiguous but may not be the easiest to understand or use for a given task. Indeed, the ability for users to lay out data arbitrarily is a major element of the appeal of spreadsheets, so increased flexibility in layout without compromising the ability to automatically adapt the layout to arbitrary changes in data size would be an important extension to our system. A template structure similar to ViTSL [1] but with unlimited nesting may provide enough generality.

The data types supported by our system include common primitive types (boolean, number, text, date) and "reference to T" for any object type T in the sheet. In the space allocation sheet, the role field of Occupant has type "reference to Role", and individual Occupant objects contain references to Role objects, shown as dotted arrows in Fig. 4. As mentioned in Section 2, each object type has a *display field* that provides a string representation for display and input of references to objects of that type.

A field of an object type T may be *computed*, meaning that its content is given by a formula that is automatically evaluated once for each object of type T and may read any other data on the sheet. Fields that are not computed are called *state fields* and represent independently mutable parts of the application state. Entire object types may also be computed; we explain in Section 4.4 how this works, but the end result is that arbitrary custom views of data can be constructed. We plan to add support for *updatable* views that can accept manual and programmatic writes and make corresponding writes to underlying data, in general by executing an arbitrary procedure, though we may offer simplified options for common cases. We do not expect novice developers to build complex updatable views manually. Instead, updatable views provide a clean way to define the semantics of add-on data modeling and transformation features (which may be included with the system or built by advanced third-party developers) for incorporation into sheets by novice developers, who may gradually learn how to debug and modify the views using the tools built into the spreadsheet.

[2] We envision fields being specified as single-valued in the schema, but our prototype does not yet support this and simply stretches every cell that is currently the only one in its field.

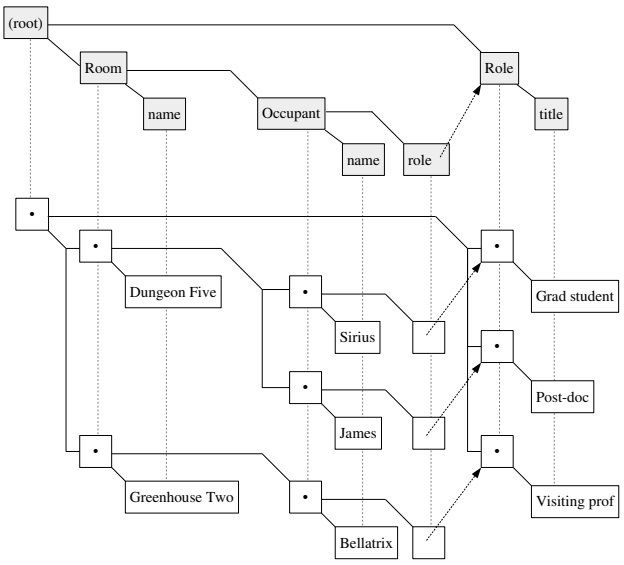


Figure 4. Schema tree (gray nodes) and data tree (white nodes) for part of the space allocation sheet. Capitalized schema nodes are object types; lowercase schema nodes are fields. Bullet nodes are objects; other data nodes are values. The dotted arrows are references.

3.1 Rationale

We designed Object Spreadsheets to present data hierarchically because hierarchy is widely accepted as an understandable and efficient approach to presenting structured data, as in [4]. However, this does not fully determine our choice of data model because a hierarchical view can be implemented on top of any data model that can represent relationships between objects, including the relational and network models. We specifically choose a hierarchical model with *ownership*, i.e., an object owns all of its descendants in the data tree (but not objects that it merely references), and the only object deletion operation we provide is one that deletes the descendants. This concept is easy to illustrate in the spreadsheet: when the developer selects an object cell, the rectangle it owns is highlighted. Ownership provides a rudimentary form of encapsulation: data of any kind can be added to an object's representation and it will be deleted with the object. No one seems to question this functionality for fields, and we argue it should be offered for nested objects as well. In practical terms, ownership captures the nature of a significant fraction of entity relationships in real applications; for example, if an application includes invoices that contain line items, one would expect deleting an invoice to delete the line items.

Of course, many binary relationships, such as the one between Occupant and Role in the space allocation sheet, cannot be captured by ownership. In our model, they are represented by reference-valued fields, which are equivalent to the "virtual records" of the hierarchical data model. We believe the usability benefits of references over explicit foreign keys are clear. Furthermore, we can support references as a first-class data type without adding any complexity to the

functioning of the data model or the spreadsheet interface, provided that we refrain from imposing restrictions to prevent references from becoming broken by deletion of their target objects. Other approaches such as a separate "relationship" concept would add complexity, and it's unclear if they would permit the examples of Section 4.3 to be expressed in a comparably natural way.

In the space allocation sheet, each occupant has one role, so the Occupant object type has a single-valued role field referring to the role. In other cases—for example, if an application has User and Newsletter object types, and each user is subscribed to multiple newsletters—a multi-valued field User.subscriptions would be used. If the application needs to traverse the relationship in the other direction, the developer can add a computed field Newsletter.subscribers (updatable once this is supported by our system) that would contain the references to the corresponding users; we will likely offer this as a predefined option.

Although our basic data model does not prevent references from becoming broken, many applications will want to impose referential integrity constraints. We envision offering predefined updatable views that provide the equivalent of SQL's "`ON DELETE RESTRICT`" (disallow deletion of referenced objects) and "`ON DELETE CASCADE`" (automatically delete all referencing objects). In general, we prefer to build data modeling features on top of existing features of our system rather than add complexity to the basic data model.

When applications require ternary or higher-arity relationships, they can be represented by dedicated object types as in other database systems.

Particular tasks may call for hierarchical views that traverse relationships that are represented via references rather than ownership. Such views can be constructed using computed fields and objects and will ultimately be updatable, and our system could be extended with view-building commands like those of SIEUFERD [2] that automate this construction. Once the desired hierarchical structure is assembled, customizing its on-screen layout for the best usability remains a separate issue, as mentioned earlier.

4. Formulas and Computation

4.1 Overview of Computation

In a traditional spreadsheet, each cell can have its own formula, and unintended inconsistencies in formulas are a notorious source of errors. In contrast, in Object Spreadsheets, a formula is assigned to an entire non-root schema node, i.e., a field or a nested object type, corresponding to a column in the nested table layout. The formula is written in terms of a *context object* of the parent object type T and is evaluated once for each object of type T to produce the data for that object. For example, in the space allocation sheet, the computed field free of Room has the formula `sqFoot − sum[o : Occupant](o.role.allocSpace)`. The free field of Dungeon Five is computed by evaluating the formula with Dungeon Five as the context object, so sqFoot and Occupant refer to Dungeon Five's square footage and set of occupants (according to semantics we explain in Section 4.2), and the result is 436.

Since the same formula is used for all objects of the same type, if different behavior is desired for some objects, the developer must write an explicit conditional. This design both ensures that no unintended inconsistencies are introduced and provides predictable behavior for objects that are subsequently created, either manually or programmatically. Of course, this does not mean the behavior is what the developer wants for a new object that is different from the existing ones in some way! The developer may need to add or modify conditionals in the face of new requirements; special interaction techniques may be helpful to streamline this process.

In our computational model, the basic unit of data in a sheet is the *family*. A family is identified by a pair $\langle s, o \rangle$, where s is a non-root schema node and o is an object of the parent object type of s, and consists of all child data nodes of o described by s; it is called *state* or *computed* depending on whether s is state or computed. A family $\langle s, o \rangle$ represents either the complete content of a field of o or o's entire set of child objects of a certain type. In Fig. 4, families containing multiple nodes are indicated by forked edges; in this example, every value node is in a family by itself. The *content* of a family is the set of values of its nodes, where the "value" of an object node is a reference to the object itself. The content is indeed a set: the nodes in a family are unordered, and we do not allow duplicate values within a field of a single object (duplication does not arise with nested objects, which always have distinct references).

The data instance of a sheet is uniquely determined by its set of families and their content, and indeed, this is the most useful form of the data instance for computation. The content of each computed family $\langle s, o \rangle$ results from one evaluation of the formula of s with o as the context object, hence if a runtime error occurs in the evaluation, the entire family is erroneous. The family (state or computed) is also the unit of data that can be read by a formula, thus it is the granularity at which dependencies would be tracked for incremental recomputation (not yet implemented in our prototype). In a traditional spreadsheet, this granularity would be the cell. The computational model of DAPLEX is not explicitly stated in [34], but it would likely be very similar to ours.

4.2 Formula Language

The most important feature of formulas in a spreadsheet is access to other data in the sheet. Excel and other traditional spreadsheets use a row-column coordinate notation that is either absolute or relative. Since our data model is hierarchical, data access must follow this hierarchy using an operation we refer to as a *navigation*. As in many other systems, navigations are written using a dot notation: *start*.*targetName*, where *start* evaluates to a starting object and *targetName* is the name of a schema node to navigate to. An unqualified

expr	::=	*var-name*	// *local variable*	literal	::=	`$` // *root cell literal*
	|	(expr`.`)? *targetName*	// *navigation*		|	*number*
	|	literal			|	`"`*string*`"`
	|	`{` expr* `}`	// *union*		|	`true` | `false`
	|	*op* expr	// *unary operator*			// *(all literals are singleton sets)*
	|	expr *op* expr	// *binary operator*			
	|	*function-name* `(` expr* `)`	// *built-in function invocation*			
	|	`{` *var-name* `:` expr `|` expr `}`	// *filter comprehension*			
	|	`sum[`*var-name* `:` expr`](`expr`)`	// *sum comprehension*			

Table 1. Formula syntax.

targetName denotes a navigation starting from the context object of the formula, and $*targetName* starts from the root object. (In the formula for Room.free in the space allocation sheet, sqFoot and Occupant are examples of unqualified navigations, while o is a bound variable.)

We allow two types of navigation:

- Up navigation—following the parent relationship to go to one of the object's ancestors (or itself as a special case; we believe that using the object's type name is a helpful reminder compared to a special variable with a fixed name such as this).
- Down navigation—reading the content of a child family of the starting object corresponding to a particular field or nested object type, which in general results in a set of values. As mentioned in Section 4.1, the "value" of a nested object is a reference to it, so down navigation to either a reference-valued field (e.g., o.role in the space allocation sheet) or a nested object family (e.g., $Role) yields a set of references, which can be the starting point for a further navigation. Since developers like to use short names that describe the meaning of a schema node with respect to its immediate parent (e.g. "name"), we allow each down navigation to go down only one level so that its meaning is clear. To go down more than one level in the hierarchy, one can chain navigations.

Like families in the sheet, all formulas are set-valued, with scalars represented by singleton sets. A navigation from a set of starting objects returns the union of the navigation results for the individual objects in the set. (In these respects, Object Spreadsheets works the same way as Alloy [19] and DAPLEX [34].)

The target name of a navigation is resolved to a schema node when the formula is entered (the formula cannot be saved if this resolution fails), and the formula is automatically updated if the schema node is later renamed by the developer, much as a cell reference in a traditional spreadsheet formula updates if the target cell moves. To perform this resolution, we must know the type of the starting object, so formulas are type-checked when they are entered. Type-checking ensures that every subexpression evaluates to a set of elements of the same type (which may be a primitive type or a reference type).

Formula expressions are drawn from a language whose syntax is described in Table 1. It includes set equality (`=`) and inclusion (`in`) operators, which can also be used for scalar equality and scalar membership in a set; a set comprehension notation {*var* : *set* | *pred*} that filters an existing set using a predicate; and various numeric, boolean, date, and set-related operators and functions (`+`, `-`, `<`, `>`, `&&`, `count`, etc.).

4.3 Computing with Sets

Here are a few representative examples of set computations that occur in data-centric applications and how they are handled in Object Spreadsheets:

1. Given a set `members` of members of a research group (as references to `Person` objects), compute the set of their offices (assumed to be a field `office` of each person). This can be done with a navigation `members.office`, which takes the union of the result for each object in the starting set.
2. Compute the set of cars (`Car` objects) currently available in a car sharing service, assuming that each car has an `available` field. This could be achieved with a navigation `$Car.selfIfAvailable` given an auxiliary computed field

   `Car.selfIfAvailable` $\widehat{=}$ `if(available, Car, {})`

   but such filtering (like "`WHERE`" in SQL) is so common that we believe it is well worth providing a filtering construct. The current syntax is `{c : $Car | c.available}`; it can probably be improved to make it easier for new developers to understand, perhaps drawing inspiration from LINQ [23], which is a good fit for our set-based system.
3. When a user (represented by a `User` object) visits the application, show the distance from their current location (given by `User.location`) to each of their favorite restaurants (given by `User.favorites`, a set of references to `Restaurant` objects which also have a `location` field). The distance depends on both the user and the restaurant, so it can't be defined simply as a computed field on one or the other. One way we can represent the distances is to change the representation of each user's favorites from a plain set

of references to a set of nested `Favorite` objects, each of which contains a `restaurant` reference. (Indeed, Object Spreadsheets provides a command to wrap the existing values of a field in objects in this manner.) When a user adds or removes a favorite, we create or delete a `Favorite` object rather than just adding or removing a restaurant reference. We can then define:

```
Favorite.distance ≙
      dist(User.location, restaurant.location)
```

(imagining for the purpose of this example that `dist` is a built-in function on locations).

4.4 Computed Objects

The last example started to push the boundaries of what can be achieved with the features covered so far. Consider a slightly more complex problem: given a user of the car sharing service of example 2, with a `User.location` field, show the distances to all available cars (which now have a `Car.location` field). If we had `UserCarInfo` objects nested in each `User` object for the available cars, we could define `UserCarInfo.distance` as in example 3. However, it would be awkward to manually create or delete `UserCarInfo` objects for all users when a car changes availability. Instead, we can define `UserCarInfo` as a *computed object type* by assigning it a *key formula* that returns the set of available cars. The formula is evaluated once in the context of each `User` object u, and each returned car c generates a `UserCarInfo` object $x_{u,c}$ nested in u. The car reference c becomes the *key* of $x_{u,c}$ and is placed in a designated *key field*, which we would name `UserCarInfo.car`.

In our example so far, the key formula returns the same set of cars for each user, resulting in parallel sets of nested `UserCarInfo` objects, but this need not be the case in general; for example, we could filter the set of cars further based on the user's preferences. Keys may be of any primitive or reference type. A computed object type is the analogue of a use of the "`COMPOUND OF`" function in DAPLEX [34].

Further computed fields (such as `UserCarInfo.distance`) and nested object types may be defined on a computed object type in terms of its key and any data accessible via its parent. This process can be repeated for any number of levels of nesting, with formulas at each level able to access the data of all ancestor objects, to transform data into arbitrary structures. In this way, Object Spreadsheets offers expressive power similar to SQL but requires—and allows—queries to be broken down into steps that return one set at a time, maintaining the local nature of computation of the spreadsheet paradigm. One must keep in mind that while a key formula produces values that appear in the key field of a computed object type T, it is associated with T and its context is the parent of T, not T itself, which can only be populated after the set of keys has been generated by the formula. Key fields may be regarded as computed but do not have formulas of their own.

A computed object may go in and out of existence at any time as the result set of the key formula on its parent changes to include or exclude its key. Therefore, we do not allow a computed object to contain state data because it's unclear what should happen to the state data if its owner goes out of existence and then comes back; however, a computed object may contain a view (in the future, updatable) of state data stored elsewhere. Semantically, a reference to a computed object consists of its parent and its key (though the reference is still shown using a display field like any other reference). That is, we are considering two computed objects that exist at two points in time t_1, t_2 with the same parent and equal keys to be the same object, and a reference stored at time t_1 (even in state data) will work at time t_2.

We give one final example that uses computed objects:

4. Given a set `members` of members of a university department, with a field `city` giving the city in which each lives, compute the set who live in each city for further analysis. In SQL, this would be done with "`GROUP BY`". In Object Spreadsheets, one would define a top-level computed object type `CityGroup` with key formula `members.city`, which would generate one `CityGroup` for each city with at least one resident. One would define a computed field for the set of people in each city as follows:

   ```
   CityGroup.residents ≙ {m : members | m.city = city}
   ```

   Further computed fields can then be added to `CityGroup`. This boilerplate is tolerable for now; it can be further automated with a special "group by" feature.

Sets and beyond. We currently choose the unordered set as the structure for families and subexpression results because it is the simplest choice that provides the power to express arbitrary data transformations in combination with our design for computed objects. We can still represent scalars as singleton sets, and in some contexts the language semantics requires a singleton value (e.g., a Boolean condition for `if`).

Our choice of sets has its limitations. In the space allocation example (Fig. 3), we would like to use the simpler formula

```
Room.free ≙ sqFoot − sum(Occupant.role.allocSpace)
```

but to get correct results, `Occupant.role` must able to return a multiset, rather than a set. We could add support for multisets as well as lists and ordered sets, which would make certain application logic easier to express but would make the computation semantics more complicated; in particular, for each new structure, we would need to decide what happens if it is returned by the key formula for a computed object type (or if this should even be allowed). For now, we support a special `sum` construct that binds a variable, similar to the mathematical Σ notation, and other aggregation operations could be designed in the same way.

ParentView		ChildView		MeetingView			
	parent		student	enrollment	teacher	sel. slot	avail. slots
	Person		Person	Enrollment	Person	Slot	Slot
•	Molly	•	Fred	• Fred @ Potions	Snape	Snape @ 1pm	
				• Fred @ Divination	Trelawney		Trelawney @ 4pm
							Trelawney @ 5pm
							Trelawney @ 6pm
		•	George	• George @ Charms	Flitwick		Flitwick @ 10am
							Flitwick @ 11am

Hello, Molly.

Meetings for Fred — **scheduled**: Snape @ 1pm [×] — **pick one:** [○] Trelawney @ 4pm; [○] Trelawney @ 5pm; [○] Trelawney @ 6pm

Meetings for George — **pick one:** [○] Flitwick @ 10am; [○] Flitwick @ 11am

Figure 5. A view model and one instance for scheduling parent-teacher meetings, with an example rendering.

5. Application Views

Application views follow MVC guidelines. The design of HTML views is hierarchical by nature, so it seems desirable to have a hierarchical model backing it. We define a *view model* to be a designated sub-tree of the data model, by picking an object type T and including all its descendant data. The view model is crafted through formulas to contain exactly those data items that are to be displayed. If a view needs some parameters, such as the currently logged-in user, selected class, etc., then these parameters are placed in state fields of a common *view instance* object of type T. This allows several instances of the same view to exist simultaneously.

The view instance is then mapped onto an HTML template using standard templating techniques. Notice that at this point the template does not have to contain any logic such as conditional statements; such logic can be pushed to the formulas populating the view. This makes the binding straightforward, following the nested structure of the view model.

An example from the parent-teacher conference application described in Section 6 is shown in Fig. 5. Under the object type `ParentView`, `parent` is a state field that is filled with a reference to the user requesting the view. Formulas then pull out the relevant data from the other columns in the data model. Notice that `ParentView` contains `ChildView` and `ChildView` contains `MeetingView` in the schema, matching the nesting structure of the rendered HTML.

To understand how the control aspect works, notice the buttons [×] and [○] in the figure; clicking a button fires a transaction that mutates the data in order to schedule or cancel a meeting. The button is contained in a UI element, which is in turn associated with an object in the spreadsheet, simplifying the task of associating the click with the relevant data item(s) that need to be updated.

6. Experiments and Evaluation

We have built a prototype of the Object Spreadsheets execution engine and developer interface on top of the Meteor web framework [26]; all our applications thus inherit the reactivity of Meteor. The developer UI is rendered via a Handsontable [18] widget with cell merging managed by our code, and supports editing the schema (that is, the overall structure) and its contents. Formulas and values are type-checked to ensure conformance to the schema. Transaction procedures are executed by the engine, but they cannot yet be edited in the developer interface (so they must be provided in a file).

To assess the applicability of our model, we collected scenarios in which our colleagues faced a need for a collaborative data-centric web application for a specific task. We noted a few of the most interesting features of each application and considered how best to implement them in an object spreadsheet. We then built the essential parts of these applications, and hand-coded UIs for them using Meteor templates (eventually, UI building will be integrated in the developer interface).

The applications are:

- ***PTC***—the parent-teacher conference application mentioned in the the introduction. Teachers, students, and parents are stored as Person objects. A reference field links students to their parents. Teachers own Slot objects that represent potential meeting times. Classes are stored using another top-level object type, and each class owns Section objects, which in turn have references to teachers teaching those sections and nested Enrollment objects that link to enrolled students. Parents can only schedule one meeting per Enrollment of each of their children, in a slot of the correct teacher (i.e., the teacher of the section in which the student is enrolled); and slots cannot be double booked.
- ***Dear Beta***, a site for students working on a system architecture assignment to share advice on correcting particular test failure modes. Students can vote on questions and answers as on Stack Overflow. The questions are organized in a tree structure matching the structure of the test cases.
- ***Hack-q***, a system for participants in a hackathon to request help from mentors in particular areas of expertise. This case study is discussed in more detail below.
- ***Got Milk***, a management application for a group of people who share a pool of fresh milk for coffee. Teams of two members take turns buying the milk for the entire group. The application sends email notifications for members when it is their turn to buy the milk, and alerts when milk supply is low.

To give an idea of the size of the applications, Table 2 shows the sizes of the spreadsheets that were used to back them. The numbers under "Data" and "Formulas" indicate the number of schema nodes of the respective kind. The

numbers under "Procedures" indicate the number of lines of procedure code that were written for mutations.

Case study. We present the data model, formulas, and transaction procedures constructed for the "Hack-q" example and explain their function in finer detail. In this application, participants of an organized hackathon access a web form where they fill in their name, the programming area in which they require assistance, and their current location. Meanwhile, designated mentors have been classified according to their area of expertise—each mentor has been assigned one or more "skills". The submitted request then shows up in the relevant mentors' queues as a "call". A mentor can then "pick" the call, in which case it disappears from the queues of other mentors. After talking to the participant, the mentor may close the call (discarding it from the queue), or forfeit the call, putting it back so that it reappears in all other queues and can subsequently be picked up again by another mentor.

Fig. 6 shows a sample sheet containing some concrete data. Column names, their types, and their hierarchy are shown by the header of the spreadsheet. The formula for the column "inbox" (under "Staff") computes a mentor's incoming queue. Every mentor is assigned a set of "Call" objects on subjects relevant to the mentor's skills as listed in the "expertise" column. The calls are sorted according to the "time" column. Calls assigned to other mentors, and calls that have been forfeited by that mentor, are subtracted from their queue. The transactions are used to insert and remove elements from the queue, and are quite straightforward.

For comparison, we built as much of Hack-q in QuickBase as we could. We created Skills and Calls tables as in Fig. 6, but we stored the expertise information in reverse by adding a QuickBase user-list field, "Experts", to the Skills table to hold the set of mentors with the skill. With this representation (which we found slightly unnatural), we were able to define an Inbox report on the Calls table that tested whether the current user was in the Experts list of the skill record associated with each call. However, if we wanted to enhance the application so that a call could require multiple skills, this would require only a small change in Object Spreadsheets but we are not aware of a way to express such logic in QuickBase; this illustrates the risk that developers take by investing in a tool with limited expressive power. Also, QuickBase does not support application-specific mutation patterns (such as assigning a call to the current user) in the core application builder; instead, the developer has to write a formula to concatenate strings into a URL that will make the desired change via the QuickBase API and then add a link to this URL to the page.

	Data		Formulas	Procedures
	# fields	# object types	# computed schema nodes	LOC
PTC	28	12	9	17
Dear Beta	6	7	1	7
Hack-q	10	3	1	9
Got Milk	11	5	1	16

Table 2. Sizes of sample applications.

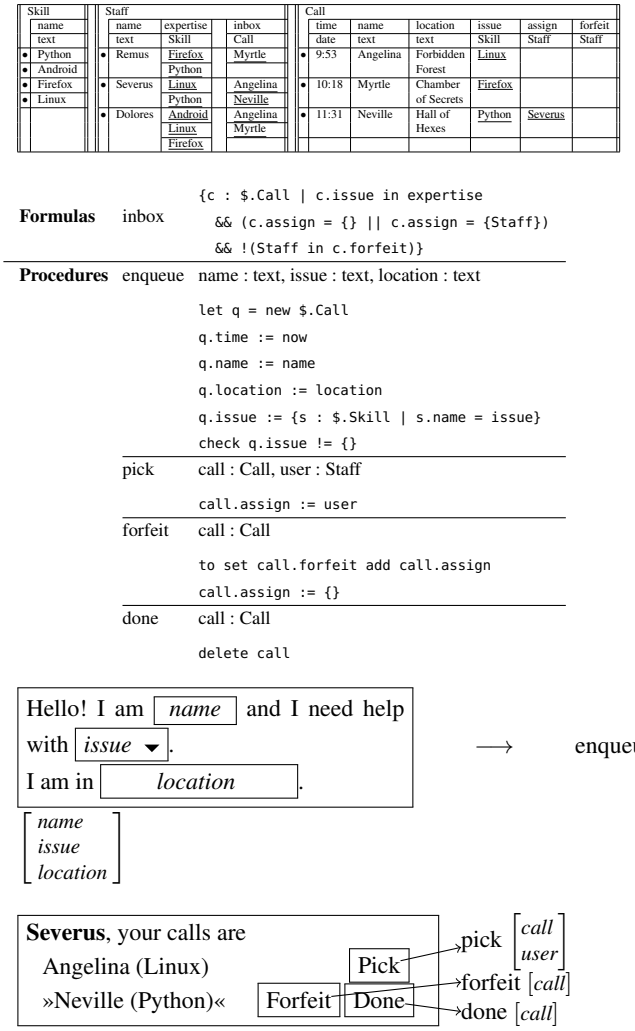


Figure 6. Data model, formulas, transaction procedure code, and HTML forms associated with the simple queuing example "Hack-q" in the case study.

7. Related Work

Spreadsheet-backed application builders. The only spreadsheet-backed application builder designed to allow persistent state in the spreadsheet to be mutated via the application UI is Quilt [5]. It uses an unmodified Google Spreadsheet and does not attempt to overcome the limitations of the traditional spreadsheet model, so it only supports flat tables containing one record per row. The developer creates an HTML page and specifies an element to be repeated to display each record, subelements of which may be bound to fields of the record using column names or may be hidden

conditionally based on fields. In addition, controls can be designated to add and delete records.

Spreadsheet design features have been pursued more extensively for development of "mashup" applications that combine, transform, and query data from multiple sources but do not maintain their own persistent state. Gneiss [7, 8] and SpreadMash [21] both retain the two-dimensional grid but allow cells to contain nested data structures retrieved from external sources. They can extract and filter items from these structures, and SpreadMash can even define computed fields on them, but neither tool can generate or mutate such structures on its own.

Other data-centric application builders. Subtext with Two-Way Dataflow [13] is analogous to our work in offering a continuously visible rich data model with computed data and application UI binding support, and it has an intriguing design for batching view updates using a total order on the data model. However, its developer UI is less familiar than a spreadsheet, and it's unclear to us how the UI will scale to development of our target applications.

App2You [22] and AppForge [38] both let the developer build hierarchical forms, constructing the schema automatically, and offer a menu of access control policies. However, neither has demonstrated how end-user developers would build arbitrary logic. AppForge supports only filters of the form "field operator constant", and [22] does not state what filters are supported by App2You, though it mentions a formula language as a future extension.

Finally, the mainstream application builders QuickBase [32], FileMaker [15], and Knack [20] all support computed fields that are a function of fields of the same object or aggregations of related objects, but none has the ability to repeat a computation on each related object, as Object Spreadsheets can by introducing a computed object type.

Naked Objects. Our work may recall the Naked Objects approach to application design [29]. The essential principles of this approach are (1) a commitment to encapsulating all logic in the objects it affects and (2) automatic generation of the application UI from the schema and programmatic interfaces. The use of Object Spreadsheets as a data model and development tool for the application logic appears to be orthogonal to both of these principles. (Object Spreadsheets currently does not provide any features to enforce encapsulation, but a developer can still choose to respect it.) Furthermore, the spreadsheet UI on the application state could play the role of the automatically generated application UI, except for the need for read access control. It does not yet provide a way to invoke procedures, but this is planned.

Nested table interfaces to relational data. Related Worksheets [3] is a spreadsheet-like tool that lets a developer construct a schema for a set of related tables and join them into editable nested-table views. The original vision was to provide most of the features of spreadsheets, but formula support was never added. Instead, the authors went on to develop SIEUFERD [2], a tool for exploring existing relational databases. SIEUFERD lets a developer construct nested-table views using menu commands for joins, filtering, and sorting, but does not support modifying the data or the schema. SIEUFERD also supports computed fields, with a formula language that supports navigations both up and down the hierarchy, though many data transformations are only achievable via the menu commands. The semantics of a SIEUFERD view are given by translation to SQL and do not determine the value of a cell in terms of a small number of other cells, and indeed, some changes to the view definition have non-local effects that surprised us. We believe there is a subset of SIEUFERD's functionality that (with the use of some boilerplate) is equivalent to the computational functionality of Object Spreadsheets, though we have not verified this. Object Spreadsheets differs in its support for schema and data editing, stored procedures, and abstract object references and its demonstration as a backend for web applications.

SheetMusiq [24] is similar in computational capabilities to SIEUFERD, but its UI differs superficially from a nested table layout: values in columns at outer nesting levels are repeated for each row at the innermost level.

Mashroom [17] uses a typed, nested data model and a nested table layout like ours and has a formula language similar in spirit to ours with support for hierarchical navigation (Mashroom does not have object references). However, its computational model is based on a script of transformations starting from the source data, such as "insert a column containing a snapshot of the result of this formula", which can then be replayed on new source data. One could achieve a development cycle similar to ours by modifying formulas in the script and replaying it, with the limitation that dependencies must be acyclic at the column level rather than the family level. Also, Mashroom does not consider mutations to a permanent state, as contrasted with edits recorded in the script.

Structured spreadsheets. Several existing spreadsheet tools are able to maintain varying degrees of structure within a two-dimensional grid. In Microsoft Excel, if a formula is entered in a cell in a range designated as a "table", then the column of the range becomes "calculated" and is automatically filled with the same formula (including rows added to the table later) and Excel warns if the formula is overridden in individual cells.

MDSheet [12] is somewhat more general. It lets the developer define the structure and formulas of a spreadsheet using the ClassSheets [14] modeling format, which supports repeating row and column groups and a dot notation for navigation, and it maintains the structure and formulas as the developer adds and removes instances of the repeating groups. There appears to be no obstacle in principle to supporting programmatic addition and removal of such instances. How-

ever, MDSheet supports only one level of repetition along each axis of the grid, which limits its expressive power compared to Object Spreadsheets.

Sumwise [27, 28] lets the developer annotate rows and columns with arbitrary tags (which could be used to mark object types or fields) and then declaratively bind formulas to all cells with certain tags. However, the available documentation is insufficient for us to evaluate its expressive power compared to Object Spreadsheets, and it does not explicitly mark the repeating row and column groups, which would be necessary to support programmatic addition and removal of instances.

Other spreadsheet extensions. We have reviewed several systems that extend spreadsheets with new capabilities to see if they might meet our requirement to be able to define a formula in the context of a member of a nested variable-size set, even if their original motivation differed from ours. Mini-SP [39] meets this test, since it allows sheets to be nested in cells and has a rich programming language that includes the ability to instantiate a nested sheet for each cell in an input array, but the code required is more complex than in Object Spreadsheets. Forms/3 [6] is capable of mapping auxiliary sheets containing per-item formulas across a variable-size input array but does not support nested data. The Analytic Spreadsheet Package [31] and the spreadsheet of Clack and Braine [11] combine the two-dimensional grid with formulas in more powerful programming languages that can manipulate nested data structures, so such structures can be stored in a single cell, but items in these structures are not first-class entities in the system as they are in Object Spreadsheets.

Other tools that bridge databases and spreadsheets. Senbazuru [10] automatically recognizes and extracts relational data from existing spreadsheets and provides a UI for developers to perform certain types of queries, but it does not allow queries to be defined persistently, does not match the expressiveness of Object Spreadsheets, and does not have an approach to handle programmatic mutations.

Sroka et al. [35] give a construction to store relational tables in a traditional spreadsheet and execute SQL queries reactively by translating them to intricate spreadsheet formulas. This may be a convenient environment to work with data, but it does not improve upon SQL in terms of end-user development of queries.

8. Conclusion

Our experience with the data model and the computational model shows that they yield compact programs that are easy to understand and easy to change, making them suitable for rapid prototyping without up-front design and for situations where the requirement specifications tend to change frequently. We have shown that spreadsheets can greatly benefit from having more structure built into them, without making it harder to write formulas, and while keeping the data model coherent with the visual representation. Programming with formulas can be made even easier with additional point-and-click support, which reduces the need to type identifier names and is very intuitive for spreadsheet users. Having a statically typed language aids in early detection of mistakes, but does not burden the developer with annotations, since most of the types can be inferred. Tighter integration with a UI builder can help filling in the types of transaction parameters by selecting them from the view. Most importantly, the application is always "live", with the view reflecting the current value of the state and the formulas that have been entered, thus breaking the code-build-debug cycle that slows down and obscures conventional programming.

Acknowledgments

We would like to thank Edward Doong for his help developing example applications; David Karger and Eirik Bakke for their advice on the design of the system, positioning our work and the user study; Jonathan Edwards for early discussions; and various other colleagues at MIT and the anonymous reviewers who provided feedback on our work and drafts of our papers. We are grateful to Wistron Corporation and the National Science Foundation for their support of this work.

References

[1] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual specifications of correct spreadsheets. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 189–196, Sept 2005.

[2] Eirik Bakke and David R. Karger. Expressive query construction through direct manipulation of nested relational results. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1377–1392, New York, NY, USA, 2016. ACM.

[3] Eirik Bakke, David R. Karger, and Rob Miller. A spreadsheet-based user interface for managing plural relationships in structured data. In *Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Vancouver, BC, Canada, May 7-12, 2011*, pages 2541–2550, 2011.

[4] Eirik Bakke, David R. Karger, and Robert C. Miller. Automatic layout of structured hierarchical reports. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2586–2595, 2013.

[5] Edward Benson, Amy X. Zhang, and David R. Karger. Spreadsheet driven web applications. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 97–106, New York, NY, USA, 2014. ACM.

[6] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Funct. Program.*, 11(2):155–206, March 2001.

[7] Kerry Shih-Ping Chang and Brad A. Myers. Creating interactive web data applications with spreadsheets. In *Proceedings*

of the 27th Annual ACM Symposium on User Interface Software and Technology, UIST '14, pages 87–96, New York, NY, USA, 2014. ACM.

[8] Kerry Shih-Ping Chang and Brad A. Myers. A spreadsheet model for using web service data. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, pages 169–176, July 2014.

[9] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

[10] Zhe Chen, Michael Cafarella, Jun Chen, Daniel Prevo, and Junfeng Zhuang. Senbazuru: A prototype spreadsheet database management system. *Proc. VLDB Endow.*, 6(12):1202–1205, August 2013.

[11] Chris Clack and Lee Braine. Object-oriented functional spreadsheets. In *Proc. 10th Glasgow Workshop on Functional Programming*, GlaFP '97, 1997.

[12] Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva. MDSheet: A framework for model-driven spreadsheet engineering. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1395–1398, Piscataway, NJ, USA, 2012. IEEE Press.

[13] Jonathan Edwards. Two-way dataflow. In *Future of Programming Workshop 2014*. https://vimeo.com/106073134.

[14] Gregor Engels and Martin Erwig. ClassSheets: Automatic generation of spreadsheet applications from object-oriented specifications. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 124–133, New York, NY, USA, 2005. ACM.

[15] Create custom solutions: FileMaker. http://www.filemaker.com/.

[16] Google Forms - create and analyze surveys, for free. http://www.google.com/forms/about.

[17] Yanbo Han, Guiling Wang, Guang Ji, and Peng Zhang. Situational data integration with data services and nested table. *Serv. Oriented Comput. Appl.*, 7(2):129–150, June 2013.

[18] A minimalist Excel-like data grid editor for HTML & JavaScript. www.handsontable.com.

[19] Daniel Jackson. Alloy: A new technology for software modelling. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, page 20, 2002.

[20] Knack - easy online database and business apps. https://www.knackhq.com/.

[21] Woralak Kongdenfha, Boualem Benatallah, Régis Saint-Paul, and Fabio Casati. SpreadMash: A spreadsheet-based interactive browsing and analysis tool for data services. In *Proceedings of the 20th International Conference on Advanced Information Systems Engineering*, CAiSE '08, pages 343–358, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] Keith Kowalczykowski, Kian Win Ong, Kevin Keliang Zhao, Alin Deutsch, Yannis Papakonstantinou, and Michalis Petropoulos. Do-it-yourself custom forms-driven workflow applications. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[23] LINQ (Language-Integrated Query). https://msdn.microsoft.com/en-us/library/bb397926.aspx.

[24] Bin Liu and H. V. Jagadish. A spreadsheet algebra for a direct data manipulation query interface. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 417–428, Washington, DC, USA, 2009. IEEE Computer Society.

[25] Richard Matthew McCutchen. Object Spreadsheets: an end-user development tool for web applications backed by entity-relationship data. Master's thesis, Massachusetts Institute of Technology, May 2016.

[26] An open source platform for building web applications. www.meteor.com.

[27] Darren Miller, Gary Miller, and Luis M. Parrondo. Sumwise: A smarter spreadsheet. In *EuSpRiG*, 2010.

[28] Gary Miller. The spreadsheet paradigm: A basis for powerful and accessible programming. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 33–35, New York, NY, USA, 2015. ACM.

[29] Richard Pawson. *Naked objects*. PhD thesis, Trinity College, June 2004.

[30] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in Excel. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 165–176, New York, NY, USA, 2003. ACM.

[31] Kurt W. Piersol. Object-oriented spreadsheets: The analytic spreadsheet package. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 385–390, New York, NY, USA, 1986. ACM.

[32] Business apps, online databases & custom software: Intuit QuickBase. http://quickbase.intuit.com/.

[33] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 207–214, Sept 2005.

[34] David W. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.*, 6(1):140–173, March 1981.

[35] Jacek Sroka, Adrian Panasiuk, Krzysztof Stencel, and Jerzy Tyszkiewicz. Translating relational queries into spreadsheets. *IEEE Transactions on Knowledge and Data Engineering*, 27(8):2291–2303, August 2015.

[36] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 360–384, 2014.

[37] Online form builder with cloud storage database: Wufoo. http://www.wufoo.com/.

[38] Fan Yang, Nitin Gupta, Chavdar Botev, Elizabeth F Churchill, George Levchenko, and Jayavel Shanmugasundaram. WYSIWYG development of data driven web applications. *Proc. VLDB Endow.*, 1(1):163–175, August 2008.

[39] A.G. Yoder and D.L. Cohn. Real spreadsheets for real programmers. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 20–30, May 1994.