Timeline: Adding the Time Dimension to Spreadsheets

TOMAS PETRICEK, Charles University, Czech Republic
TOMÁŠ BOĎA, Charles University, Czech Republic

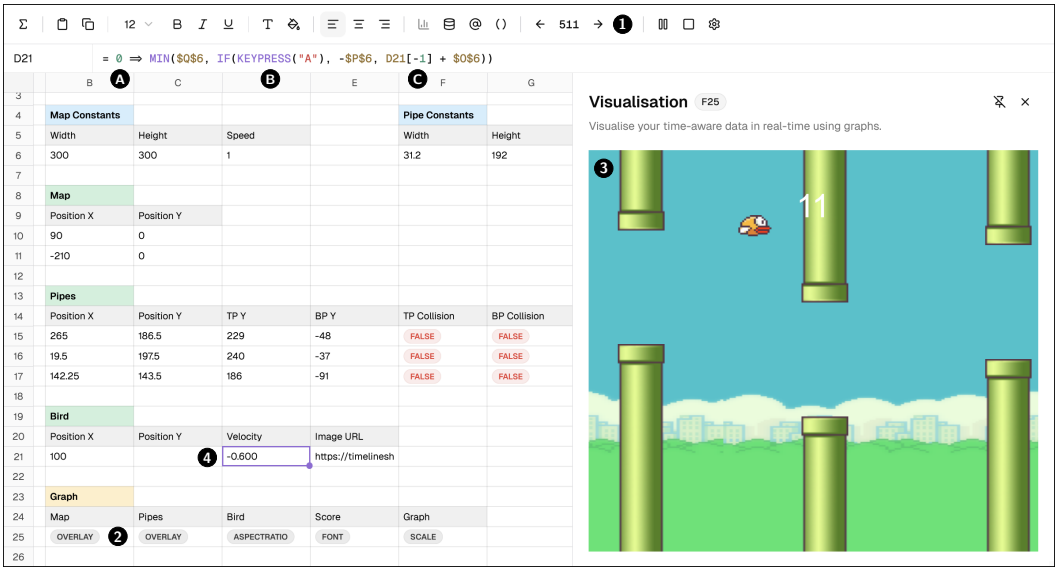


Fig. 1. A Flappy Bird game clone in Timeline. ❶ The system shows the current time step and lets the user advance step-by-step or play/stop time. ❷ Visualizations are built using a composable grammar and ❸ are displayed on the side. ❹ The selected formula computes the Y velocity in the current step. It Ⓐ specifies an initial value, Ⓑ reacts to user-interface events, and Ⓒ computes the current value using a previous one.

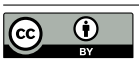
Spreadsheets make it easy to express computations over two-dimensional data, but two dimensions are not enough to express rich computations involving time such as physics simulations, agent-based modelling, financial data analyses, or simple interactive applications.

We present Timeline, a system that adds discrete time to spreadsheets. We motivate the design of Timeline through a small-scale formative study, present a *core calculus* that models system runtime, use it to formalise a static analysis to determine the required number of past values, and evaluate the system using a series of case studies from the four aforementioned areas.

The paper also shows that a large number of advanced programming language concepts can be directly applied in the context of spreadsheets with time. Timeline draws from *dataflow languages* to express computations over time, *coeffect systems* to ensure bounded memory usage, *functional reactive programming* to support interactivity, and *grammars of graphics* to support composable visualizations.

CCS Concepts: • **Software and its engineering → Very high level languages**.

Authors' Contact Information: Tomas Petricek, Charles University, Prague, Czech Republic, tomas@tomasp.net; Tomáš Boďa, Charles University, Prague, Czech Republic, tominoboda@gmail.com.



ACM 2475-1421/2026/10-ART370
https://doi.org/10.1145/3839502

Additional Key Words and Phrases: Spreadsheet Systems, Dataflow Languages, Functional Programming

ACM Reference Format:
Tomas Petricek and Tomáš Boďa. 2026. Timeline: Adding the Time Dimension to Spreadsheets. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 370 (October 2026), 30 pages. https://doi.org/10.1145/3839502

1 Introduction

Spreadsheets are sometimes referred to as "the most widely used programming systems in the world" [2]. Their uses range from personal productivity tools and decision-making support systems to scientific analytical instruments and financial reporting systems [29]. However, spreadsheets are limited to flat tabular data structures and lack abstraction mechanisms.

Many extensions to the core spreadsheet model have been proposed, generalising the tabular structure to non-tabular forms [8, 14], adding support for objects [47], or incorporating support for streaming data to use spreadsheets as the back-end for web development [18, 19]. Abstraction mechanisms developed for spreadsheets include sheet-defined functions [40, 61] and gridlets [39], while programming by example [30] simplifies data wrangling within spreadsheets.

The above extensions show that extending spreadsheets is a difficult balancing act. How can we make spreadsheets more expressive, while preserving their liveness, directness and usability [33]? The aim of Timeline is to extend spreadsheets with a built-in notion of discrete time, making it possible to use spreadsheets in a range of domains that are difficult to tackle with only two dimensions [44] such as physics simulations, agent-based modelling [1], financial data analysis, or simple interactive applications. As illustrated in Figure 1, Timeline adds an implicit discrete time to the spreadsheet programming model and extends the formula language to allow references to previous values and user interface events, but otherwise preserves the standard spreadsheet model.

Adapting programming language research. Timeline is an exploration of the design, expressive power and usability of spreadsheets with discrete time. We motivate the design through a small-scale formative study (§5) and reflect on the trade-offs it introduces (§7). The technical core of the paper presents Timeline from the programming language design perspective. Notably, we show that a large number of concepts and techniques developed in the context of programming language research are directly applicable to spreadsheets with time.

Timeline has a built-in notion of discrete time. Formulas can refer to other cells in the current time step (e.g. `G19`), but also values computed in earlier steps (e.g. `F19[-1]`). The design and implementation of Timeline draw inspiration from research on *synchronous dataflow languages* [9, 31]. To ensure that simulations and interactive applications execute using bounded memory whenever possible, we augment the runtime with a program analysis based on *coeffects* [54, 55]. The support for discrete time makes it possible to use the spreadsheet paradigm for building interactive applications. Here, we draw on the *functional reactive programming* paradigm [25, 66]. To support visualizing time-varying values, Timeline adopts a *composable data visualization* model [45, 52, 71].

Contributions. This paper presents the design and implementation of Timeline, a spreadsheet system with discrete time, and captures its essence through a core calculus.

- We provide an overview of the Timeline design through a simple physics simulation (§2) and examine the expressive power of the programming model through a number of case studies also covering agent-based modelling, financial data analysis, and interactive applications (§6).
- We discuss how the design and implementation of Timeline adapts advanced programming language concepts for the context of spreadsheets (§3), including synchronous languages, coeffects, functional reactive programming, and visualization grammars.

- We present a core calculus of the Timeline execution model (§ 4), which captures the way previous values are stored during the evaluation. We define a static checking mechanism for references to earlier time steps, formalize a space optimization and prove its soundness.
- We motivate the Timeline design through a small-scale formative study (§ 5) and discuss the resulting trade-offs using the cognitivie dimensions of notation framework (§ 7).

2 Overview

To demonstrate Timeline, we consider a simple numerical simulation of cannonball firing. In each step, the simulation computes the new cannonball position using the position in the previous step and the current velocity. The initial velocity is computed from the initial angle and firing speed and then updated using gravity and air drag, which itself depends on the velocity in the previous step.

There are three dimensions in the example: we have multiple cannonballs, with multiple attributes, updated over multiple time steps. Implementing the simulation in a conventional spreadsheet is possible, but challenging. As discussed later (§ 6.5), one option is having a table for each of the attributes. This makes it possible to add cannonballs and time steps using drag right and drag down operations (to copy formulas to further empty cells), but it results in a complex structure with computation spread across 8 tables.

Implementing Cannonball Simulation. In Timeline, we can implement the cannonball simulation using a single table. We use this example to introduce the key features of the system.

The calculation keeps track of attributes of individual cannonballs over time (Figure 2, ❶). The table includes the current position (`B3`, `C3`), velocity (`F3`, `G3`) and air drag (`H3`, `I3`) as well as the initial angle and speed (`D3`, `E3`). We discuss the calculation for the first cannonball (row 3). Other cannonballs are added by selecting the last row and using the drag down operation to copy its formulas, automatically updating the row references.

To specify the position of a cannonball, we need to give the initial position and then specify how to compute the position in the current time step in terms of the value in the previous time step. This illustrates two extensions that Timeline makes to the core spreadsheet model. First, the =>

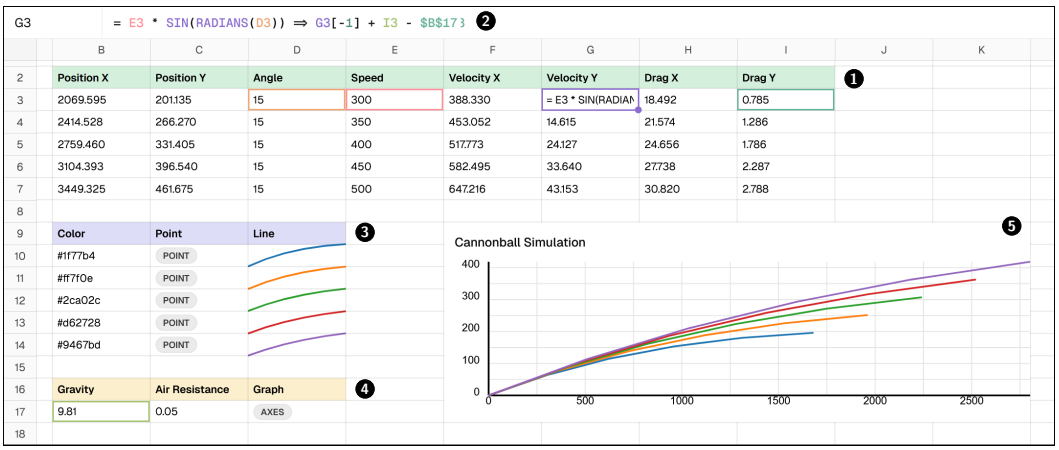


Fig. 2. Numerical cannonball firing simulation in Timeline. ❶ Single table records attributes of multiple cannonballs over time. ❷ The attributes are computed based on current and previous values of other attributes. ❸ A separate table plots past positions as lines. ❹ A table with configuration and composed final chart. ❺ The composed chart is then embedded in the sheet.

operator specifies an initial value for a cell. Second, the notation `C3[-1]` lets us access the value of the cell `C3` in a previous time step. For the cannonball, the initial X and Y positions are 0, the position in later steps is computed by adding the current X and Y velocities to the previous position and the computation stops if the cannonball hits the ground:

```
/* Position X */ B3 = 0 => IF(C3[-1] < 0, B3[-1], B3[-1] + F3)
/* Position Y */ C3 = 0 => IF(C3[-1] < 0, C3[-1], C3[-1] + G3)
```

The calculation of the cannonball velocity again uses =>. This time, the initial value is a formula using the firing speed and angle (`E3`, `D3`). The velocity is then computed from the value in the previous time step and air drag. For the Y velocity, we also subtract gravity, which is a fixed cell reference (`$B$17`) that is not updated during the drag down operation. Air drag is obtained by multiplying the previous X and Y velocities by the air resistance (`$C$17`):

```
/* Velocity X */ F3 = E3 * COS(RADIANS(D3)) => F3[-1] + H3
/* Velocity Y */ G3 = E3 * SIN(RADIANS(D3)) => G3[-1] + I3 - $B$17
/* Air Drag X */ H3 = F3[-1] * $C$17
/* Air Drag Y */ I3 = G3[-1] * $C$17
```

Note that the calculation of air drag does not require specifying an initial value, because there are no references to previous values of `H3` and `I3`. The references in the velocity computation refer to the current value. The air drag computation accesses velocity at the previous time step, which always exists because velocity defines an initial value using =>. At the first time step, `F3[-1]` in the definition of the X air drag refers to the initial velcotiy computed using trigonometric functions from the values of `E3` and `D3`. As in standard spreadsheet systems, references between current values of cells need to be acyclic. References to previous time steps do not introduce a cycle.

An interesting problem is how many past values need to be kept by the system. In the computation described so far, we access one previous value of cells `B3`, `C3` and `F3`, `G3` using the `[-1]` notation. It is thus sufficient to keep one past value for these four cells. For the chart (Figure 2, ❺) we need to store all previous values of the cells that define the line. We discuss this aspect in detail in §3.2.

Composable Data Visualization. Data visualizations in standard spreadsheet systems exist separately from the core spreadsheet model. They are created through a graphical user interface where the user specifies data sources and other chart attributes. With the additional time dimension, such user interface would grow more complex, but visualizations in Timeline also play a more central role. They make it possible to see past values that are not directly displayed in the grid. Timeline takes inspiration from research on spreadsheet-based and functional composable visualizations [45, 52] and allows construction of visualizations directly in the spreadsheet.

The cannonball firing chart (Figure 2, ❺) is obtained by constructing line charts that represent the movement of individual cannonballs over time and composing the individual lines (❸ and ❹). This is done in a separate table that contains one chart for each cannonball. The table row at index 10 (`C10`, `D10`) constructs a line chart for cannonball at row index 3 (`B3`, `C3`):

```
/* Point */ C10 = POINT(B3, C3)
/* Line */  D10 = GRAPH(STROKECOLOR(B10, LINE(C10[:-STEP()])))
/* Graph */ D17 = AXES("left bottom", OVERLAY(D10:D14))
```

The example uses a function `POINT` that constructs a value representing a two-dimensional point and a number of functions for constructing and transforming charts including `LINE`, `OVERLAY` and `AXES`. The latter are inspired by composable data visualizations [52]. `LINE` takes a range of points and constructs a primitive chart shape representing a line; `OVERLAY` takes a range of chart shapes and constructs a single chart shape consisting of the individual shapes; `AXES` attaches axes with labels to a given chart shape. This requires adding points and chart shapes as two additional primitive

values to the core spreadsheet model. In systems that support richer data structures [14, 21, 47, 73], a point could be a user-defined object or a record.

To construct the chart, the cell `C10` constructs a point consisting of the current X and Y positions of the cannonball. We then want to construct a line from all previous locations of the cannonball. This is done by specifying a time range `[:-STEP()]` in the cell reference. The syntax is a shorthand for `[-1:-STEP()]` and it selects all previous steps (`STEP` is a function that returns the number of the current step). The array of past values is then passed to `LINE`, which constructs a primitive chart shape. The `GRAPH` function used in cell `D10` instructs Timeline to display the chart inline in the cell as a sparkline [27] without modifying the shape value. Finally, the cell `D17` overlays the individual lines and adds axes and the underlying charting library automatically aligns the chart elements.

Extending the Spreadsheet Paradigm. The preceding example illustrated most of the Timeline extensions to the core model of spreadsheets. The only significant additional feature needed by the opening Flappy Bird clone (Figure 1) is the `KEYPRESS("A")` function, which detects whether a key has been pressed in the current time step. The game movements and collision detection are ordinary formulas and the game scene is constructed using the composable data visualization library, leveraging the `IMAGE` function to construct a primitive chart shape from an image URL.

The motivating example shows how the additions made in Timeline preserve the simplicity and directness of the core spreadsheet model. We add only a limited number of extensions to the formula language (the `=>` operator and references across time). Moreover, when creating a spreadsheet in Timeline, users still start with concrete values and then extend their formulas using the drag down operation to apply a formula across a range of inputs, preserving the *from concrete cases* mode of *abstraction construction* [37].

3 Research Themes

Syntactically, the only extensions that Timeline makes to the core spreadsheet model are the operator for specifying the initial value `=>` and a notation for accessing previous values `A1[-1]` or time ranges `A1[-1:-STEP()]`. However, the design and implementation of Timeline draw on a number of research directions on programming languages that have previously not been connected to spreadsheets. This section provides an overview discussing the major connections.

3.1 Synchronous Languages

The notion of discrete time, as implemented in Timeline, is based on the notion of time in declarative synchronous programming languages. Synchronous languages [9] are based on the idea that time proceeds in discrete logical ticks and components react concurrently to changes in each step. The approach has been successful in areas such as signal processing and embedded control systems.

In imperative synchronous languages such as Esterel [10], programs explicitly await and emit time signals. Our approach is akin to declarative synchronous languages like Lucid and Lustre [6, 17] where expressions denote streams of values and operators and functions operate on streams in a pointwise manner. The languages additionally include a number of operators for manipulating streams. Consider the following Lustre example adapted from [9]:

```
      nat = 0 -> pre(nat) + 1;
     edge = false -> (c and not pre(c));
edgecount = if edge then pre(edgecount) + 1 else pre(edgecount)
```

The definition of `nat` illustrates two operators for manipulating streams. The `e1 -> e2` operator creates a stream that has the value of the stream `e1` in the first time step and the value of `e2` in subsequent steps. The `pre(e)` operator constructs a stream that, in a time step t, returns the value of `e` in time step $t-1$. The Lustre examples map directly to Timeline:

```
      /* nat */ A1 = 0 => A1[-1] + 1
     /* edge */ C1 = FALSE => AND(B1, NOT(B1[-1]))
/* edgecount */ D1 = IF(C1, D1[-1] + 1, D1[-1])
```

Although we do not explain, or formally model, Timeline in terms of streams, in practice our => and `[-1]` operations provide the same functionality as -> and `pre` in Lustre. Of course, Lustre also provides higher-order abstraction capabilities that are not present in Timeline, but those could be supported through some of the proposed abstraction mechanisms for spreadsheets [40, 46, 61].

It is also worth noting that these examples could be easily encoded in an ordinary two-dimensional spreadsheet (using rows for time steps). As illustrated by our case study from the finance domain (§Case #3): encoding stream processing in Timeline using built-in time is practically useful when working with large data sets.

3.2 Coeffect Analysis

An interesting problem that Timeline shares with synchronous languages is how many past values need to be stored and whether the buffer size is bounded [15, 16]. In all the examples in §3.1, the memory is bounded—the runtime needs to store one past value for each cell accessed via `[-1]`.

In Timeline, we adopt a static analysis based on dataflow coeffects [13, 54, 55]. A type system based on coeffects annotates expressions with context requirements, such as variable liveness or, in the case of dataflow, the number of required past values. In dataflow coeffects, type judgements of the form $\Gamma\,@\,n \vdash e : \tau$ indicate that a given expression e can be evaluated in a variable context Γ with at least n past values for each of the variables. Structural coeffects [55] are more precise and track context requirements per-variable, by using a vector of annotations and explicit weakening, contraction and exchange rules. Figure 3 shows three of the coeffect typing rules.

The rule for variables states that accessing the current value of a variable does not require any past values. This is akin to a cell reference such as `B1`. The rule for prev e indicates that, if evaluating e requires n past values, then obtaining the previous value of e requires $n + 1$ past values. Function types in the coeffect calculus are annotated with how many past values the function requires (a function $\lambda x.\text{prev}\ x$ would be annotated with 1). The rule for function application thus combines context requirements of the expression that defines the function with the context requirements of the evaluation of the argument, followed by the evaluation of the function (if evaluating the argument requires n past values and the function needs k additional past values, then evaluating the function call requires $n + k$ past values).

In Timeline, we can only access previous values via a cell reference. Computations such as $\text{prev}(a + b)$ can be expressed by storing $a + b$ in a cell, say `A1`, and accessing `A1[-1]`. Time ranges in Timeline can be dynamic, e.g., `C10[:-STEP()]`, or even indirect, e.g., `A1[-B1]`. Such references cannot be statically checked and so the inferred number of required past values can be unbounded.

We describe the type system for checking previous value references formally in §4.4. The type system is used in two ways. First, it is used to determine the size of buffers required to store past values during spreadsheet execution. Second, it is used to check that a sufficient number of previous values is specified using the => operator (for cells with potentially unbounded number of past values, we assume that the first use of => specifies a default value for any previous time step; an alternative design would be to have a separate operator for this purpose).

$$\frac{x : \tau \in \Gamma}{\Gamma\,@\,0 \vdash x : \tau} \qquad \frac{\Gamma\,@\,n \vdash e : \tau}{\Gamma\,@\,n{+}1 \vdash \text{prev}\ e : \tau} \qquad \frac{\Gamma\,@\,m \vdash e_1 : \tau_1 \xrightarrow{k} \tau_2 \quad \Gamma\,@\,n \vdash e_2 : \tau_1}{\Gamma\,@\,\max(m, n + k) \vdash e_1\ e_2 : \tau_2}$$

Fig. 3. Coeffect typing rules for variables, previous value access and application.

A basic static checking mechanism simply ensures that a cell referenced via `A1[-n]` has at least n past values, but the coeffect calculus suggests a more flexible semantics. The following example shows how the same computation admits two evaluation strategies with different memory use:

```
            /* counter */ A1 = 0 => A1[-1] + 1
      /* prev(counter) */ B1 = 0 => A1[-1]
/* prev(prev(counter)) */ C1 = B1[-1]
```

We refer to cells that require storing past values as *stateful* and cells that are always computed instantaneously as *stateless*. Note that the same distinction is made in synchronous languages, where expressions may be stateless/combinatorial/memoryless or stateful/with memory.

In the above example, `A1` and `B1` are stateful and require 1 past value each. Correspondingly, we specify one previous value for each of the cells using `=>`. However, the same spreadsheet could be executed by making `B1` stateless and requiring two past values of `A1`:

```
            /* counter */ A1 = 0 => 1 => A1[-1] + 1
      /* prev(counter) */ B1 = A1[-1]
/* prev(prev(counter)) */ C1 = B1[-1]
```

This alternative corresponds to the application rule of the coeffect calculus where the context requirements of a function argument and the context requirements of a function are combined. If we make `B1` stateless, we can always recompute its single previous value using the second past value of `A1`. We discuss this coeffect-based generalisation of past value access checking in §4.5.

Our case studies (§6) are small enough that memory is not a practical constraint and Timeline makes it possible to keep all past values to enable stepping back through time for debugging. However, many Timeline applications can run efficiently with bounded buffer sizes. Unbounded buffers are needed in situations such as the cannonball visualization (§2) where we explicitly access all past data to draw a chart, but the Flappy Bird game (Figure 1) requires only one past value of several cells, rather than storing all previously encountered states.

The generalised coeffect-based execution makes it possible to trade space for time. This is theoretically interesting and it is useful in cases where, for example, a large table has a column of values that all depend on a single cell outside of the table. The optimization may reduce the memory footprint, although this would be practically useful only in limited execution environments.

3.3 Functional Reactive Programming

Functional reactive programming (FRP) [25, 66] is a programming paradigm for building reactive and interactive applications in functional programming languages. Given the close connection between spreadsheets and functional programming [14], it is not surprising that Timeline can draw on functional reactive programming ideas.

In FRP, programs are structured in terms of two abstractions: *behaviors* model time-varying values and *events* represent discrete event occurrences. Programs are constructed using a number of combinators that relate the two abstractions. In Fran [25] the type `Behavior a` represents streams of values of type `a`, while `Event a` represents discrete events emitting values of type `a`. The two abstractions are composed using combinators such as `untilB`:

```
-- Behave as the initial behavior, changing to the one emitted by the event
untilB :: Behavior a -> Event (Behavior a) -> Behavior a
```

The behavior `untilB start evt` has the value of `start` initially. When `evt` occurs, the value becomes that of the time-varying value emitted by the event. In Yampa [23, 35], behaviors are represented using arrows `SF a b` from input `a` to output `b`, which can be composed using a family of switch combinators such as:

```
-- Switch to a new signal function 'SF a b' whenever the specified event occurs
switch :: SF a (b, Event c) -> (c -> SF a b) -> SF a b
```

Conceptually, FRP systems are based on a continuous notion of time. Together with the rich higher-order primitives, this has historically been difficult to implement efficiently [7, 42] and some implementations adopted a model with clocks [67] more akin to synchronous languages. Timeline is based on discrete time and so it eschews such implementation difficulties.

Timeline also does not use switching functions to dynamically reconfigure the dataflow graph [22]. When the projectile hits the ground in the cannonball simulation, the simulation continues running but the value is no longer updated. (The Timeline user interface allows specifying a reset condition outside of the formula language, which is used in the Flappy Bird clone.) Despite these limitations, it is still useful to view Timeline formulas as specifying time-varying behaviors.

Our main inspiration from functional reactive programming is for the integration of events into the programming model of Timeline. For example, in the Flappy Bird clone (Figure 1), the key press event controls the velocity of the bird. In the game, the cells `$O$6`, `$P$6`, and `$Q$6` contain constants that specify the gravity, jump speed and maximal velocity. The new velocity is:

```
/* Velocity */ D21 = 0 => MIN($Q$6, IF(KEYPRESS("A"), -$P$6, D21[-1] + $O$6))
```

In Timeline, we can think of `IF(KEYPRESS("A"), .., ..)` as a switching operator akin to `untilB` or `switch` that changes the behavior from one time-varying value to another when the event occurs. However, because of the discrete time model, the behavior changes once and the state is stored implicitly in the history of the cell, rather than explicitly using accumulator combinators as in FRP.

3.4 Composable Data Visualizations

In traditional spreadsheets, visualizations are created outside of the formula language and data is selected through a user interface. Timeline provides a more flexible, formula-based mechanism that makes it easy to display not just the current but also past values that are not directly visible in the grid. It makes it possible to build rich visualizations across multiple dimensions, and it also enables users to learn how to construct visualizations by examining existing spreadsheets [58].

Research that originates from grammar of graphics [71] offers a more declarative and compositional alternative [59, 68]. In systems based on the grammar of graphics, a chart is created by putting together data, scales that transform data to visual values, coordinate system, geometric marks that specify how to display data, data transformations and other aspects of visualization. For example, using the ggplot2 library [68] we can compose a chart as:

```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point(alpha = 0.5) + geom_smooth() + scale_y_log10()
```

The snippet separately specifies the data source diamonds, aesthetics that map properties carat and price from the data source to X and Y coordinates, adds two geometric marks (points and smooth line) and makes the Y scale logarithmic. Grammar of graphics is very effective for scientific visualizations, but it assumes a two-dimensional data source, requires a large number of primitive constructs and its range of charts is still limited.

A more flexible approach, which fits well with the spreadsheet paradigm, is based on the composition of graphical shapes. This has been developed in the context of functional programming [52], but is conceptually similar to a design prototype envisioned for spreadsheets [45].

In composable visualizations, graphical shapes are specified using domain values such as exchange rate, number of votes, country names, or population. When composed, the system infers composite scales, checks that all shapes have a compatible scale, and arranges individual shapes. This makes it possible to compose visualizations that combine different visual elements generated from data. For example, the following uses Compost.js [51] to overlay fruit images over a column chart:

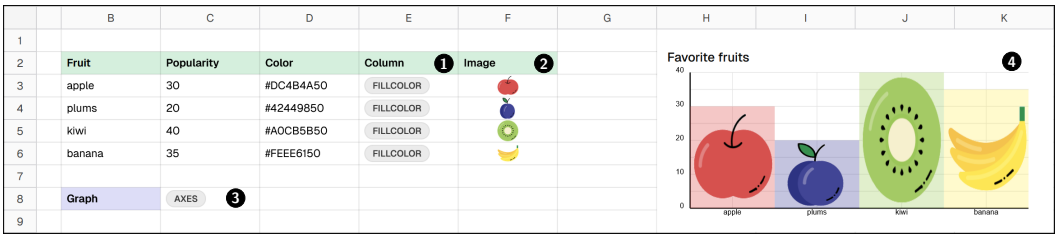


Fig. 4. Composing rich data visualization in Timeline. ❶ Columns are constructed using the fruit name and popularity. ❷ Image for each fruit is loaded, positioned and also displayed inside the table. ❸ Columns and images are composed in a single cell, ❹ which is displayed as a chart in the spreadsheet.

```
let fruitBars =
  c.preserveAspectRatio("none", c.overlay(fruits.map(f =>
    c.overlay([
      c.fillColor(f.color,
        c.column(f.kind, f.value)),
      c.image("img/" + f.kind + ".png",
        [[f.kind, 0], 0], [[f.kind, 1], f.value]) ])
    )))
```

The snippet iterates over the data using `fruits.map`. For each data point, it creates a column and an image and overlays the two. The X scale is categorical and the image position is specified as `[f.kind,0]` and `[f.kind,1]`, which defines the left and the right end of the area allocated for the category `f.kind`. The Y scale is continuous and the values are numbers. The individual images are combined using `overlay` and the snippet uses `preserveAspectRatio` to enable scaling.

As shown in Figure 4, the same chart can be constructed in Timeline. We specify input data in a table and add two columns representing the column and image. Those are then overlaid in cell `C8`:

```
/* Construct column and image graphical shapes for each data point */
E3 = FILLCOLOR(COLUMN(B3, C3))
F3 = GRAPH(IMAGE(APPEND(B3, ".png"), POINT(COORD(B3,0),0), POINT(COORD(B3,1), 1)))

/* Overlay chart elements and specify additional chart properties */
C8 = AXES("left bottom", ASPECTRATIO("none", OVERLAY(E3:E6,F3:F6)))
```

The column is constructed using `COLUMN` which takes a categorical value (fruit name) and a numerical value (fruit popularity). It creates a rectangle that spans the whole area allocated for the category in the X dimension and ranges from 0 to the specified value in the Y dimension. When loading an image, we construct the image URL (simplified for presentation) and specify its location using `COORD`, which creates a categorical coordinate and `POINT`, which combines X and Y coordinates. Note that `COLUMN` is built on top of a primitive for filling a polygon that takes a list of points.

The programming model offered by composable visualizations in Timeline offers many of the user experiences prototyped in Cuscus [45]. As in Cuscus, the graphical shape of a visualization can be specified using formulas in a table that supports the drag down operation, enabling non-programmers to create visualizations using tools they are already familiar with. Timeline improves on Cuscus in that it specifies shape dimensions in domain values and allows more flexible composition, such as overlaying a specified range. Visual shapes in Timeline cannot be, however, modified using direct manipulation. The encouraging results of user studies performed for Cuscus thus likely also apply to visualizations in Timeline.

Column name	N	$=$	A | B | C | ...	*Value*	V	$=$	$c \mid \{V_{i,j}\}$
Row index	m	$\in$	$\mathbb{N}_1$	*Formula*	F	$=$	$V \mid R \mid f(F_1, \ldots, F_n)$
Address	a	$=$	Nm	*Sheet*	σ	$\in$	$Address \to Formula$
Range	R	$=$	$a_1 : a_2$	*Grid*	γ	$\in$	$Address \to Value$

Fig. 5. Syntax of the core Timeline calculus.

4 Calculus

In this section, we present a core calculus of Timeline. We take the *core calculus for spreadsheets* [72] as our starting point (§4.1, §4.2) and extend it with the notion of time (§4.3). We then define a system for checking references to earlier time steps (§4.4), formalize our coeffect-based optimization (§4.5) and show their soundness (§4.6).

4.1 Syntax

Figure 5 defines the syntax of the calculus. Addresses Nm consist of a column name (such as A, B, or AB) and a row index (starting from 1) and ranges are pairs of addresses. Values can be constants c or two-dimensional arrays $\{V_{i,j}\}$ with indices $i \in 1..n$, $j \in 1..m$.

In contrast to prior work [12, 72], we omit empty values, error values and implicit error propagation. We treat errors as constants and assume that built-in functions return error when called with the error value as argument. Although we do not support spilling [72], we include arrays of values and later extend those to support ranges over time. Note also that a singleton array $\{V\}$ is distinct from a primitive value V. Formulas can be primitive values V, range references R or built-in function calls. For simplicity, cells can only be referenced through ranges so a reference `B5` is modelled as a singleton range reference `B5:B5`.

Following the terminology of Williams et al. [72], we use the term *sheet* to refer to the source code and *grid* for the result of evaluating a sheet. A sheet σ is a partial function from addresses to formulas, while a grid γ is a partial function from addresses to values.

4.2 Semantics

The evaluation rules are shown in Figure 6. We follow the style of Williams et al. [72] and define evaluation with respect to an unevaluated sheet σ. This means that dereferencing an address, defined as an auxiliary relation $\sigma \vdash a\,!\,V$ obtains the formula of the target cell and evaluates it recursively. Consequently, evaluation of cyclic references is not defined. An alternative approach [12] defines evaluation with respect to (other) evaluated values as $\gamma \vdash F \Downarrow V$ and imposes consistency requirements on the recalculation process. This rules out cyclic references for which there is no consistent value assignment, such as A1 = A1 + 1, but permits self-references such as A1 = A1.

Formula evaluation $\sigma \vdash F \Downarrow V$ is then a relation that evaluates a formula F to a value V given the sheet definition σ. A value V evaluates to itself. A function call $f(F_1, \ldots, F_n)$ evaluates its arguments and then obtains the result by applying $[\![f]\!]$, which defines the semantics of the built-in function, to the arguments. Evaluating a singleton range $a:a$ obtains a single value at the given address using dereferencing. Evaluating a range $a_1 : a_2$ iterates over all addresses in the range. The helper size returns the size of the range and $a + (i, j)$ returns an address moved by 1-based offsets i and j.

Sheet evaluation $\sigma \Downarrow \gamma$ evaluates the value at each address $a \in \mathrm{dom}(\sigma)$ defined in the sheet σ. We do not specify evaluation order, but the evaluation is deterministic (§4.6). Our Timeline implementation builds a dependency graph between cells and evaluates them in a topological order. As discussed earlier, sheet evaluation is not defined for sheets containing a cyclic reference. To preserve determinacy when we add time, we will need to require an additional condition (§4.5).

Formula evaluation: $\sigma \vdash F \Downarrow V$

$$\mathsf{size}(N_1 m_1 : N_2 m_2) = (N_2 - N_1 + 1, m_2 - m_1 + 1)$$
$$N_1 m_1 + (i, j) = (N_1 + i - 1)(m_1 + j - 1)$$

$$\frac{}{\sigma \vdash V \Downarrow V} \qquad \frac{\sigma \vdash F_i \Downarrow V_i \quad [\![f]\!](V_1, \ldots, V_n) = V}{\sigma \vdash f(F_1, \ldots, F_n) \Downarrow V} \qquad \frac{\sigma \vdash a \,!\, V}{\sigma \vdash a : a \Downarrow V}$$

$$\frac{a_1 \neq a_2 \quad \mathsf{size}(a_1 : a_2) = (n, m) \quad \forall i \in 1..n, j \in 1..m \;.\; \sigma \vdash (a_1 + (i, j)) \,!\, V_{i,j}}{\sigma \vdash a_1 : a_2 \Downarrow \{V_{i,j}\}}$$

Address dereferencing: $\sigma \vdash a \,!\, V$

$$\frac{\sigma(a) = F \quad \sigma \vdash F \Downarrow V}{\sigma \vdash a \,!\, V}$$

Sheet evaluation: $\sigma \Downarrow \gamma$

$$\frac{\forall a \in \mathsf{dom}(\sigma) \;.\; \gamma(a) = V \wedge \sigma \vdash \sigma(a) \Downarrow V}{\sigma \Downarrow \gamma}$$

Fig. 6. Reduction rules that define formula evaluation of the core Timeline calculus.

4.3 Adding the Time Dimension

To model Timeline, we extend the core calculus with a notion of discrete time. The extended syntax is shown in Figure 7. Time t is modelled using positive integers and, following the Timeline implementation, we refer to history using negative values $-t$. We treat time as a kind of value V. A time interval T is defined by a pair of expressions that evaluate to time values. A range R can now refer to a three-dimensional range over a sheet area and a time interval. A reference to a single past value such as `F19[-1]` is modelled as a singleton range `F19:F19[-1:-1]`. Evaluating a reference to a three-dimensional range produces a three-dimensional array of values, represented as a new kind of value written as $\{V_{t,i,j}\}$. Singleton ranges $a : a[E : E]$ evaluate to a single value.

Note that intervals can only range over historical values and so we cannot write, for example, `SUM(F19[0:-10])` to add the current and the previous value of `F19`. This simplifies the model as previous and current values are stored differently. Corresponding computations can be expressed explicitly using, for example, `F19+SUM(F19[-1:-10])`.

To model the *followed by* operator $\Rightarrow$, we separate formulas F and expressions E. A formula can be an expression, or a formula prefixed with an initial value (i.e., a formula has zero or more prefixed values). In addition to the previously defined grid γ, we maintain the history of previously evaluated values. A history ω is a partial function mapping addresses to timelines. A timeline stores a continuous range of t previous values of a cell, indexed by a time interval from 1 to t. We refer to a cell as *stateful* when its history is retained, when $a \in \mathsf{dom}(\omega)$; other cells are *stateless*.

Time	t	$\in$	$\mathbb{N}_1$	*Expression*	E	$=$	$V \mid R \mid f(E_1, \ldots, E_n)$
Interval	T	$=$	$E_1 : E_2$	*Formula*	F	$=$	$E \mid E \Rightarrow F$
Range	R	$=$	$a_1 : a_2 \mid a_1 : a_2[T]$	*Timeline*	ι	$\in$	$\{1 \ldots t\} \rightarrow \mathit{Value}$
Value	V	$=$	$\ldots \mid t \mid \{V_{t,i,j}\}$	*History*	ω	$\in$	$\mathit{Address} \rightarrow \mathit{Timeline}$

Fig. 7. Syntax of the Timeline calculus with time.

Formula evaluation: $\sigma, \omega \vdash F \Downarrow V$ History access: $\omega \vdash a[-t] \mathbin{¡} V$

$$\frac{\sigma, \omega \vdash F \Downarrow V}{\sigma, \omega \vdash E \Rightarrow F \Downarrow V} \qquad \frac{\sigma, \omega \vdash E \Downarrow -t \qquad \omega \vdash a[-t] \mathbin{¡} V}{\sigma, \omega \vdash a{:}a[E{:}E] \Downarrow V} \qquad \frac{\omega(a)(t) = V}{\omega \vdash a[-t] \mathbin{¡} V}$$

$$\frac{\begin{array}{c}\sigma, \omega \vdash E_1 \Downarrow -t_1 \qquad \sigma, \omega \vdash E_2 \Downarrow -t_2 \\ a_1 \neq a_2 \vee E_1 \neq E_2 \qquad \mathsf{size}(a_1{:}a_2) = (n, m) \qquad l = t_2 - t_1 + 1 \\ \forall i \in 1..n, j \in 1..m, t \in 1..l \;.\; \omega \vdash a_1[-t_1] + (t, i, j) \mathbin{¡} V_{t,i,j}\end{array}}{\sigma, \omega \vdash a_1{:}a_2[-t_1{:}-t_2] \Downarrow \{V_{t,i,j}\}}$$

$$N_1 m_1[-t_1] + (t, i, j) = (N_1 + i - 1)(m_1 + j - 1)[-(t_1 + t - 1)]$$

Fig. 8. Reduction rules that define formula evaluation of the Timeline calculus with time.

Formula evaluation. Figure 8 defines the extended semantics. Formula evaluation is now defined with respect to the sheet σ but also previously evaluated values represented using a history ω. The additional context can be easily added to the rules for evaluating function calls and accessing current values of cells (in Figure 6), but we define new rules for three-dimensional ranges and formulas prefixed with initial values. The value of $E \Rightarrow F$ is the value of F. The initial value will be used later, when initializing the history ω.

To access past values, we define an auxiliary relation $\omega \vdash a[-t] \mathbin{¡} V$ akin to address dereferencing. The relation is defined with respect to the previously evaluated values and it accesses the past value from the timeline, rather than evaluating it.

In general, time ranges are specified by expressions that need to be evaluated before accessing past values. The evaluation again distinguishes the case when a range (syntactically) refers to a single value. If so, the value is obtained from the history and returned. For other ranges, we iterate over all the values and build a three-dimensional array $\{V_{t,i,j}\}$. We extend our earlier auxiliary definition of + to also support computing an offset for an address at a specified time.

Note also that some built-in Timeline functions are time-dependent. In particular, `STEP()` returns the number of the current time step. To model this, we would need to supply additional information to the semantic function $[\![-]\!]$. We omit the details for simplicity.

Sheet evaluation. Ordinary spreadsheets are evaluated to a grid of values using the $\sigma \Downarrow \gamma$ relation (Figure 6). In Timeline, the evaluation proceeds iteratively in discrete time steps for as long as the user desires (Figure 9). In each step, the sheet is evaluated using the sheet evaluation $\sigma, \omega \Downarrow \gamma$, relation which computes the current values of the formulas, with respect to a given history. This relation remains unchanged from the ordinary version, except that it now provides the current history ω to the formula evaluation.

The iterated evaluation $\sigma \Downarrow_n \gamma$ models the first n steps of the evaluation of a Timeline spreadsheet, following an approach akin to step-indexed semantics [5]. The first time step is 1. At step 1, a reference such as `A1[-1]` resolves to the first default value specified by the formula in `A1`. For instance, if `A1=42=>0`, then then `A1[-1]` evaluates to 42 at step 1. Also note that Timeline allows users to write `A1=0=>A1[-10]+1` to define a counter that increments every 10th step, and so the value 0 serves not just as the value at time step -1, but also as the default for any further historical accesses. This is reflected in the definition of the initial history ω_0, which is defined for each cell that has some number of default values specified using the `E =>` notation. If there are n such defaults, the history maps the first n past time steps to corresponding values, but it also returns the first

Sheet evaluation: $\sigma, \omega \Downarrow \gamma$

$$\frac{\forall a \in \mathrm{dom}(\sigma)\ .\ \gamma(a) = V \wedge \sigma, \omega \vdash \sigma(a) \Downarrow V}{\sigma, \omega \Downarrow \gamma}$$

Iterated evaluation: $\sigma \Downarrow_n \gamma$

$$\omega_0(a)(i) = V_i \quad (\text{where } i \in 1..n,\ \sigma(a) = E_n \Rightarrow \ldots \Rightarrow E_1 \Rightarrow E \text{ and } \emptyset, \emptyset \vdash E_i \Downarrow V_i)$$
$$\omega_0(a)(i) = V_n \quad (\text{where } i > n,\ \sigma(a) = E_n \Rightarrow \ldots \Rightarrow E_1 \Rightarrow E \text{ and } \emptyset, \emptyset \vdash E_n \Downarrow V_n)$$

$$\sigma \Downarrow_n \gamma_n \iff \sigma, \omega_{n-1} \Downarrow \gamma_n \text{ where } \forall k \in 0..n-2\text{:}$$
$$\omega_{k+1}(a)(1) = \gamma_{k+1}(a) \quad (\text{where } \sigma, \omega_k \Downarrow \gamma_{k+1})$$
$$\omega_{k+1}(a)(1+t) = \omega_k(a)(t)$$

Fig. 9. Sheet evaluation and iterated sheet evaluation for Timeline.

value as the default for any further historical accesses. As we show later (§4.5), we do not need this for cells with bounded buffer sizes. Also note that the default values are evaluated with an empty sheet and an empty history as the context. That is, an expression that defines a default value cannot itself reference other cells or their historical values (the semantics could be extended to allow referencing of other cells with greater or equal number of default values).

After constructing the initial history ω_0, the iterated evaluation constructs a sequence of histories $\omega_1, \ldots, \omega_{n-1}$ such that the previous value of a cell in step $k+1$ becomes the value in the evaluated grid in step k and all earlier past values are shifted by 1.

4.4 Checking Past Value Accesses

Existing work on type checking spreadsheets [3, 4] could be directly applied to Timeline, but a problem unique to Timeline is ensuring bounded memory usage during spreadsheet evaluation. As discussed earlier (§3.2), we adopt an analysis based on coeffects. First, we formalize the simple mechanism ensuring that, whenever there is a reference such as `A1[-1]`, the referenced cell `A1` has at least one past value at runtime. In the next section, we revisit the model, allowing `A1` to be stateless if its value at a previous time step can be computed from other available values.

Figure 10 defines the rules for checking past value accesses in terms of judgement $\Omega \vdash F$ ok. The context Ω is a mapping from cell addresses to required buffer sizes from the set $\mathbb{N}_1 \cup \{\infty\}$, representing positive natural numbers with an additional value ∞ for an unbounded buffer.

The rules for values, cell references, function calls and formulas with default values do not impose any requirements. For past value accesses, we distinguish between the case $a[-t_1 : -t_2]$ where the time index is known statically and the case $a[E_1 : E_2]$ where the time index is evaluated dynamically. In the former, we ensure that there is a sufficient number of values required by $\Omega(a)$. In the latter, we require an unbounded buffer size, i.e., $\Omega(a) = \infty$. For accesses to past values of a range of cells, we impose the requirements on all cells in the range.

Note that Timeline does not include a mechanism for dynamically computed cell references (only dynamically computed time indices). In Excel, this can be done by using `INDEX(A1:D10,E1,F1)`, which returns a cell from the range `A1:D10`, specified by the row and column indices `E1`, `F1`. A dynamic time index reference to a dynamically computed cell would have to impose an unbounded buffer requirement on all cells in the range `A1:D10`. (Even more flexible dynamic cell references such as `INDIRECT`, and `OFFSET` cannot be statically checked, but Timeline could require an explicit range restriction, for example by using intersection ranges.)

$$\frac{}{\Omega \vdash V \text{ ok}} \qquad \frac{}{\Omega \vdash a_1 : a_2 \text{ ok}} \qquad \frac{\Omega \vdash E_i \text{ ok}}{\Omega \vdash f(E_1, \ldots, E_n) \text{ ok}} \qquad \frac{\Omega \vdash F \text{ ok}}{\Omega \vdash E \Rightarrow F \text{ ok}}$$

$$\frac{\Omega(a) \geq t \quad t = \max(t_1, t_2)}{\Omega \vdash a[-t_1 : -t_2] \text{ ok}} \qquad \frac{\forall a_i \in \text{expand}(a_1 : a_2).\ \Omega \vdash a_i[-t_1 : -t_2] \text{ ok}}{\Omega \vdash a_1 : a_2[-t_1 : -t_2] \text{ ok}}$$

$$\frac{\Omega(a) = \infty}{\Omega \vdash a[E_1 : E_2] \text{ ok}} \qquad \frac{\forall a_i \in \text{expand}(a_1 : a_2).\ \Omega \vdash a_i[E_1 : E_2] \text{ ok} \quad \nexists t_1, t_2.(E_1 = t_1) \wedge (E_2 = t_2)}{\Omega \vdash a_1 : a_2[E_1 : E_2] \text{ ok}}$$

$$\text{expand}(a_1 : a_2) = \{a_1 + (i, j), \forall i \in 1..n, j \in 1..m\} \text{ where } (n, m) = \text{size}(a_1 : a_2)$$

Fig. 10. Simple system for checking past value accesses. The judgement $\Omega \vdash F$ ok states that formula F is well-formed given the buffer sizes specified by Ω.

4.5 Generalised Coeffect-based Semantics

We now extend our evaluation semantics and checking of past value accesses to allow a more general mechanism discussed earlier (§3.2). The system is inspired by the coeffect dataflow calculus [13, 55] and allows a space optimization where past values of a cell can be obtained either from a buffer (for stateful cells) or recomputed (for stateless cells). We first discuss the checking mechanism and then the corresponding evaluation semantics.

Coeffect-based checking. As before, we use a context $\Omega : \mathit{Address} \to \mathbb{N}_1 \cup \{\infty\}$, representing the required buffer size for a given cell address. We say that a cell at an address a is *stateful* if $a \in \text{dom}(\Omega)$ and *stateless* if $a \notin \text{dom}(\Omega)$. Whether a cell is stateful is therefore not an intrinsic property but a consequence of the context Ω. The same spreadsheet can be evaluated with a cell stateful (storing its history) or stateless (recomputing it from other cells), as illustrated earlier (§3.2).

Checking is now written in terms of a judgement $\Omega \mathbin{@} t \vdash F$ ok, which states that the value of a formula F can be obtained for a given (current or past) time step $t \in \mathbb{N}_1 \cup \{\infty\}$ with buffer sizes Ω. In contrast, in the simple system discussed earlier (Figure 10), we only needed to check formulas at the current time step $t = 0$. Notation-wise, we follow the style of flat coeffect calculi [55].

The generalized coeffect-based checking rules for past cell references are shown in Figure 11. When accessing a time interval specified by statically known time indices $a[t_1 : t_2]$, we distinguish between two cases. In case the cell at the address a is stateful, the access is correct if the context Ω guarantees a sufficient number of past values, computed by adding the current or past time index t_0 with the greater of the two indices specifying the time interval $-t_1 : -t_2$. If the cell at a is stateless, the value needs to be recomputed. This can be done if the formula $\sigma(a)$ can be evaluated for a time obtained by shifting the current time t_0 to the past by the greater of the two indices of the interval. This rule is the key addition, inspired by the handling of function calls in the coeffect calculus.

The rules for access to time intervals that are not statically known, i.e., $a[E_1 : E_2]$, are similar. If the cell at a is stateful, it is required to have unbounded buffer for past values. If the cell at a is stateless, it needs to be possible to evaluate the formula $\sigma(a)$ at an arbitrary past time step, specified by setting the coeffect to ∞.

Finally, a new rule is also needed for checking access to a cell range via a reference. When evaluated at a previous time index t_0, the referenced cells need to be accessible at the same time index. Rules for other kinds of expressions are omitted and remain the same as before, except that they pass the unmodified context $\Omega \mathbin{@} t$ to their premises.

Coeffect-based formula evaluation. In the standard formula evaluation, defined before (§4.3), access to past values of a cell is defined using the $\omega \vdash a[-t] \mathbin{¡} V$ rule, which finds the previous value in the

Coeffect-based checking: $\Omega \mathbin{@} t \vdash_\sigma F$ ok

$$\frac{\Omega(a) \geq t_0 + t \quad t = \max(t_1, t_2)}{\Omega \mathbin{@} t_0 \vdash_\sigma a[-t_1 : -t_2] \text{ ok}} \qquad \frac{\Omega(a) = \infty}{\Omega \mathbin{@} t_0 \vdash_\sigma a[E_1 : E_2] \text{ ok}}$$

$$\frac{a \notin \text{dom}(\Omega) \quad \Omega \mathbin{@} t_0 + t \vdash_\sigma \sigma(a) \text{ ok} \quad t = \max(t_1, t_2)}{\Omega \mathbin{@} t_0 \vdash_\sigma a[-t_1 : -t_2] \text{ ok}} \qquad \frac{a \notin \text{dom}(\Omega) \quad \Omega \mathbin{@} \infty \vdash_\sigma \sigma(a) \text{ ok}}{\Omega \mathbin{@} t_0 \vdash_\sigma a[E_1 : E_2] \text{ ok}}$$

$$\frac{\forall a_i \in \text{expand}(a_1 : a_2).\ \Omega \mathbin{@} t_0 \vdash_\sigma \sigma(a_i) \text{ ok}}{\Omega \mathbin{@} t_0 \vdash_\sigma a_1 : a_2 \text{ ok}}$$

Fig. 11. Coeffect-based checking of references and past value accesses. The judgement $\Omega \mathbin{@} t \vdash_\sigma F$ ok states that formula F is well-formed when evaluated t steps in the past, given the sheet σ and the buffer sizes Ω.

Coeffect-based access: $\sigma, \omega \vdash a[-t] \mathbin{¡} V$

$$\frac{\omega(a)(t) = V}{\sigma, \omega \vdash a[-t] \mathbin{¡} V} \qquad \frac{a \notin \text{dom}(\omega) \quad \sigma, \text{shift}_t(\omega) \vdash \sigma(a) \Downarrow V}{\sigma, \omega \vdash a[-t] \mathbin{¡} V}$$

$$\text{where } \text{shift}_t(\omega)(a)(t') = \omega(a)(t + t')$$

Fig. 12. Coeffect-based past value access. The judgement $\sigma, \omega \vdash a[-t] \mathbin{¡} V$ retrieves the value of cell a at t steps in the past. If a is stateful ($a \in \text{dom}(\omega)$) the cached value is used, otherwise the cell is evaluated against a shifted history $\text{shift}_t(\omega)$.

Coeffect-based evaluation: $\sigma \Downarrow_n^\Omega \gamma$

$$\begin{aligned}
&\omega|\Omega = \omega' \quad \text{such that} \\
&\quad \omega'(a)(t) = \omega(a)(t) \quad (\text{when } a \in \text{dom}(\omega) \wedge t \leq \Omega(a)) \\
&\\
&\sigma \Downarrow_n^\Omega \gamma_n \iff \sigma, \omega'_{n-1} \Downarrow \gamma_n \text{ where } \forall k \in 0..n-2: \\
&\quad \omega_{k+1}(a)(1) = \gamma_{k+1}(a) \quad (\text{where } \sigma, \omega'_k \Downarrow \gamma_{k+1}) \\
&\quad \omega_{k+1}(a)(1 + t) = \omega'_k(a)(t) \\
&\quad \text{and } \omega'_{k+1} = \omega_{k+1}|\Omega
\end{aligned}$$

Fig. 13. Coeffect-based sheet evaluation and iterated sheet evaluation for Timeline.

timeline for the given cell using $\omega(a)(t)$. In the generalized coeffect-based evaluation, the value accessed using $a[-t]$ can be obtained from the timeline $\omega(a)$ when the cell a is stateful. If the cell is stateless, $\omega(a)$ is not defined and the value has to be recomputed using the formula $\sigma(a) = F$.

The rules modelling the coeffect-based past value access are shown in Figure 12. The rules for formula evaluation (Figure 6, Figure 8) remain the same, except that they now need to provide the sheet definition σ to past value access $\sigma, \omega \vdash a[-t] \mathbin{¡} V$. The operation is defined using two disjoint rules for stateful cells ($\omega(a)$ is defined) and stateless cells ($\omega(a)$ is not defined). The former case remains as before. In the latter case, we obtain the formula of the referenced cell $\sigma(a)$. To evaluate it, we obtain a view of the history ω at the previous point in history. This is done using $\text{shift}_t(\omega)$ which adds the offset t to any past time step t' that is accessed during the evaluation.

Coeffect-based sheet evaluation. The evaluation mechanism discussed in the previous section is defined with respect to the history ω. The standard way of constructing the history, shown in Figure 9, keeps full history for all cells in the spreadsheet. As discussed in §3.2, this is wasteful. In a typical case study (§2, §6) many cells can be stateless or use bounded memory.

To model the optimized coeffect-based sheet evaluation, we define an operation $\omega|\Omega$ (Figure 13), which restricts the cached past values to those required by the static context Ω. The history $\omega|\Omega$ stores timelines only for stateful cells specified in Ω and stores only the required number of values for each cell. The size may still be unbounded if $\Omega(a) = \infty$. The revised iterated evaluation $\sigma \Downarrow_n^\Omega \gamma$ (Figure 13) revisits the original definition (Figure 9) and constructs a series of histories ω'_k, obtained by restricting the history to a statically required range of past values.

4.6 Properties and Coeffect Soundness

We now use the formalization discussed in the preceding sections to show two key properties of Timeline. Both are standard properties of programming language calculi that are equally applicable and important for spreadsheets.

First, the evaluation is deterministic, i.e., if a spreadsheet can be evaluated in two ways, the results are the same. Second, the coeffect-based optimization is sound, i.e., if a spreadsheet can be checked with respect to Ω, then it can also be evaluated using the coeffect-based evaluation that restricts the stored value history as specified by Ω.

Determinacy. There are three sources of non-determinism in Timeline sheet evaluation. First, the sheet evaluation does not define the order in which the cells evaluate. Second, Timeline includes non-deterministic functions such as `RAND`. We omit those in the formal model. Third, the decision which cells to treat as stateful can also affect the evaluation. We focus on this last issue first.

The generalised coeffect-based semantics models an optimization where only a limited number of past values is stored for stateful cells. As illustrated in §3.2, the evaluation can proceed differently depending on which cells are treated as stateful. The evaluation can yield different results if the sheet contains inconsistent default values. For example, consider the sheet:

```
/* First default */      /* Second default */      /* Tricky cell */
B1 = 1                   A1 = 0 => B1              C1 = A1[-10]
```

If we make `A1` stateful, the value of `C1` will be 0 in the first 10 steps, but if we make `A1` stateless, its value will be computed as 1. However, if a sheet has consistent default values, and we pick two different sets of stateful cells, the results (if defined) will be the same. More formally:

Definition 1 (Consistent default values). A sheet σ has *consistent default values* when for any context Ω that determines which cells are stateful, and for all cells a, if $\sigma, \omega_0|\Omega \vdash a[-t] \downarrow V$ and also $\omega_0(a)(t) = V'$ (where ω_0 is defined as shown in Figure 9) then $V = V'$.

In other words, if we initialize the history and pick any restriction that allows the value of any given cell to be evaluated, the evaluated value is the same as the one obtained from the explicitly specified default values. The above example violates this condition. Before considering the main determinacy theorem, we consider the evaluation order of cells:

Lemma 2 (Sheet evaluation determinacy). *For any σ, ω, if $\sigma, \omega \Downarrow \gamma$ and $\sigma, \omega \Downarrow \gamma'$ then $\gamma = \gamma'$.*

Proof. Our definition follows Williams et al. [72], who show that it is deterministic. Briefly, the $\Downarrow$ relation is not defined for sheets with cycles. If the sheet is acyclic, the evaluation of each individual cell is deterministic and so the overall sheet evaluation is also deterministic. □

We now turn attention to the determinacy of the coeffect-based optimization. We want to show that the evaluation is deterministic for any two contexts Ω, Ω'. Note that here, they only determine

which cells are stateful. (Soundness ensures that a sufficient number of values are stored, but here we assume that evaluation is defined.) The key difficulty is that, in the past value access rule $\sigma, \omega \vdash a[-t] \nmid V$, the different contexts Ω and Ω' may treat the cell at a differently, making it stateless in one case and stateful in the other case.

Theorem 3 (Coeffect-based evaluation determinacy). *For a sheet σ with consistent default values, it is the case that for any Ω, Ω', n, if $\sigma \Downarrow_n^{\Omega} \gamma$ and $\sigma \Downarrow_n^{\Omega'} \gamma'$ then $\gamma = \gamma'$.*

Proof. The proof proceeds by induction on n. For $n = 1$, the values are obtained or computed from ω_0. The result is deterministic because the sheet is required to have *consistent default values*.

In the induction step $n = k+1$, the interesting case is when a cell a is stateless in Ω ($*$) but stateful in Ω' (†). The value of a is obtained using $\omega_k(a)(t) = V$ ($*$) or using $\sigma, \mathrm{shift}_t(\omega_k) \vdash \sigma(a) \Downarrow V'$ (†). In the former case ($*$), the value stored in the timeline was computed before $t-1$ steps using:

$$\omega_k(a)(t) = \omega_{k-1}(a)(t-1) = \ldots = \omega_{k-t+1}(a)(1) = \gamma_{k-t+1}(a)$$

The grid γ_{k-t+1} was obtained using $\sigma, \omega'_{k-t} \Downarrow \gamma_{k-t+1}$ where $\sigma, \omega'_{k-t} \vdash \sigma(a) \Downarrow V$ and $\gamma_{k-t+1}(a) = V$. The history $\mathrm{shift}_t(\omega_k)$ (†) coincides with ω'_{k-t} for all a and t on which they are both defined and so the two ways of evaluating the cell will yield the same results. □

Soundness. The second important property that we want to show is that the coeffect-based optimization is sound. As defined in Figure 13, the key aspect of the optimization is that the history ω is restricted according to a context Ω that defines which cells are stateful and how many past values are needed. If a sheet can be checked with respect to Ω and it has default values for all the stateful cells, the evaluation will not get stuck due to an access to an undefined past value. The equivalent of a well-typed sheet is defined as follows:

Definition 4 (Well-timed and well-initialized sheet). A sheet σ is:

- *well-timed* with respect to Ω if $\forall a \in \mathrm{dom}(\sigma).\ \Omega \mathbin{@} 0 \vdash_\sigma \sigma(a)$ ok.
- *well-initialized* with respect to Ω if $\forall a \in \mathrm{dom}(\Omega).\ \exists E, F.\ \sigma(a) = E \Rightarrow F$.

We write $\sigma : \Omega$ if a sheet σ is well-timed and well-initialized with respect to Ω.

We need to show that, as the spreadsheet evaluates, the history will always contain the required number of values for all stateful cells and that, if this is the case, the evaluation will complete successfully. We define the condition as:

Definition 5 (Adequacy). We say that a history ω is adequate with respect to the context Ω, written as $\omega \vDash \Omega$ when $\forall a \in dom(\Omega),\ \forall t \leq \Omega(a).\ \omega(a)(t)$ is defined.

Now, we can show that, if we have a well-timed and well-initialized sheet σ and some formula F from the sheet then, if the formula can be checked at a past time step t_0, then the formula can also be evaluated at that past time step. This is the key auxiliary lemma, necessary to show that coeffect-based optimization is sound later.

Lemma 6 (Formula evaluation progress). *If $\sigma : \Omega$, $\omega \vDash \Omega$, and $\Omega \mathbin{@} t_0 \vdash_\sigma F$ ok, then it is the case that $\sigma, shift_{t_0}(\omega) \vdash F \Downarrow V$ for some V.*

Proof. By induction on the derivation of $\Omega \mathbin{@} t_0 \vdash_\sigma F$ ok (well-founded since σ is acyclic).

Values and functions directly follow from the induction hypothesis. For current range reference $a_1 : a_2$, the rule in Figure 11 requires coeffect t_0 for all the cells in the range and so they can also be evaluated. For stateless past value accesses, the induction hypothesis applies with the coeffect $t_0 + t$. Adequacy of ω ensures that the shifted history has sufficient number of past values. For stateful past value accesses, the value is available by adequacy. The dynamic access cases are analogous. □

Lemma 7 (Adequacy preservation). *If $\sigma : \Omega$ then $\omega'_k \vDash \Omega$ for all $k \geq 0$.*

Proof. By induction on k. For $k = 0$, $\omega'_0 = \omega_0|\Omega$. The adequacy follows from the fact that the sheet is well-initialized and ω_0 is defined for all $t \geq 1$ and the restriction retains the required entries.

For the inductive step, the sheet can evaluate by Lemma 6. The new history $\omega'_{k+1} = \omega_{k+1}|\Omega$, constructed from the previous adequate history, is defined for all required cells and time indices (using γ_{k+1} for $t = 1$ and ω'_k for subsequent time indices). □

Theorem 8 (Soundness). *For any σ, Ω such that $\sigma : \Omega$, for any n, $\exists \gamma.\ \sigma \Downarrow^{\Omega}_{n} \gamma$.*

Proof. By induction on n. Lemma 7 gives $\omega'_k \vDash \Omega$ at every step k. Lemma 6 with $t_0 = 0$ then gives $\sigma, \omega'_k \Downarrow \gamma_{k+1}$ for each k, establishing $\sigma \Downarrow^{\Omega}_{n} \gamma_n$. □

Checking the preconditions. The soundness and determinacy results rely on preconditions that a Timeline programmer must satsify. Well-timedness is defined by the coeffect judgment (Figure 11) and can be checked statically. Consistent default values (Definition 1) cannot in general be checked statically, since the condition compares the results of evaluating two different formulas. A stricter system would require the user to supply default values that uniquely determine the possible evaluation strategy. Our current prototype does not implement those checks.

5 Formative Study

The design of Timeline is motivated by two observations introduced earlier (§ 2), namely that modelling tasks involving discrete time often have three-dimensional structure, and that this structure is poorly served by traditional two-dimensional spreadsheets.

Analysing discrete time models. To support the first observation, we examined a number of models published at the NetLogo Models Library [63]. NetLogo is a modelling language with discrete time that is used in a variety of domains including social sciences, natural sciences, computer science and mathematics. For example, agent-based models in economics [1] often involve multiple agents with multiple attributes that evolve over time. Physics simulations that require numerical solution also require tracking multiple attributes of multiple entities over time. Financial analyses of historical data have the same structure.

Formative user study. Prior to implementing Timeline, we conducted a small-scale formative study with four participants that supports the second observation. We asked participants to complete a four-part modelling task in Excel (30-45 minutes), and concluded with a semi-structured interview (30-45 minutes). To see different approaches to the problem, we recruited two expert spreadsheet users and two students with a computer science background. Although the small scale of the study limits generalisability, we provide a brief summary of the task and the findings.

Modelling task and findings. Participants were asked to model a race, where a number of runners run from a starting point and the first runner reaching the finish line wins. We started with a simple version of the task and gradually introduced further requirements:

1) Each runner runs independently at a constant randomly generated speed. There is no finish line.
2) We add a finish line and runners now run in a track with separate lanes. A runner stops when it crosses the finish line, or if its direct neighbour on the left or on the right stops.
3) The speed of the runners now changes by a small randomly generated number in each step.
4) Runners now stop whenever any agent crosses the finish line, not just a direct neighbour.

Three of the study participants completed all four parts. The key findings derived from session observations and follow-up interviews were:

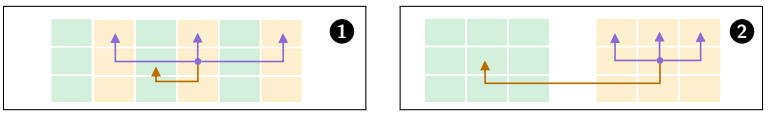


Fig. 14. Two ways of storing multiple attributes per agent. When using a column per attribute ❶ references to neighbouring agents need to target the correct column (and using a range becomes hard). When using table per attribute ❷ references to other attributes become challenging.

a) An expert spreadsheet user attempted to use an analytical solution to avoid "losing" a dimension for modelling time. For (1) and (2), they stored a time step in a single cell and calculated locations analytically. For (3) and (4), they resorted to using rows for time.
b) Part (3) required storing multiple attributes per agent. Two approaches used by participants involved using a table per attribute and keeping attributes as columns in a single table (Figure 14).
c) In (2), inter-agent dependencies posed no difficulty, but (4) required more complex addressing and some solutions used an auxiliary table to check for race completion.
d) Notably, none of the participants attempted to use a recursive formula. In Excel, this can be done by enabling *iterative calculation* and manually recomputing a value for each time step.

Conclusions. Although conceptually simple, modelling problems involving time in the traditional spreadsheet paradigm is difficult. Using one dimension for time forces users to seek workarounds for encoding the remaining two dimensions, for example by splitting attributes of individual agents across separate tables. In practice, analysts are reluctant to use one of the two spreadsheet dimensions for time. This has been confirmed by one participant, who noted that *"once you set the rows to time steps, you have lost a dimension of your spreadsheet"* and *"if you use one [dimension] for time, everything else becomes miserable."*

The following section shows how the above challenges are avoided in Timeline. It removes the need for dimension reduction (b) and simplifies addressing (c), although some of the case studies we discuss still use auxiliary tables.

6 Case Studies

To showcase the expressive power of Timeline, we present four case studies from four different application domains. In addition to the interactive game example discussed in the opening, the examples include a physics simulation of the solar system, an agent-based model of flocking adapted from Netlogo [70], and a backtesting simulation of a moving average crossover trading strategy [74]. For each case study, we give an overview of the Timeline spreadsheet structure, provide a reference to a full version available online, and briefly assess the solution.

Case #1: Planetary Prbit Simulation

The planetary orbit simulation (Figure 15)[1] models the orbit of planets in the solar system. It uses real astronomical units and measurements such as planet masses, distances from the Sun and orbital velocities, and models the planet movement by computing the gravitational force that the Sun applies to the planets. For simplicity, we only include the first four planets.

Spreadsheet structure. The spreadsheet consists of four tables. The first one contains constants, such as the astronomical unit and the gravitational constant. The second and third tables contain information about the Sun and the planets and the fourth table composes the final visualization by overlaying a light blue background and visual representations of the Sun and the planets.

[1]Available at: https://timelinesheets.com/spreadsheet/1fdc6e97-b1fd-4f49-9623-d885dd541199. To run the simulation, click *View* → *Graph* and then click the *Play* or *Next step* button.

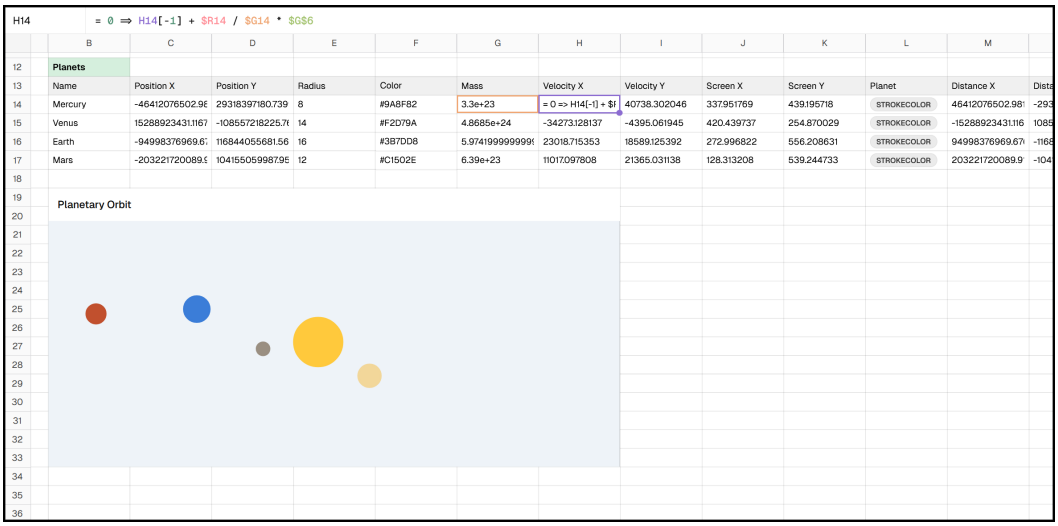


Fig. 15. Solar system simulation. The displayed table stores attributes of individual planets.

The table with planets (shown in Figure 15) records constant information about each planet (name, radius, mass, color) and information related to the planet's movement. This is the current distance from the Sun, force applied by the Sun, velocity and position, kept separately for X and Y coordinates. For the planet Earth, the distance from the Sun and the applied force are computed as follows (cells `$M16` and `$N16` store the X and Y distances from the Sun, `$H16` and `$I16` store the X and Y velocity, `$C$6` is the gravitational constant, `$G16` is the Earth's mass, and `$G$10` is the Sun's mass; the initial X position is -1 AU while the initial Y position is 0):

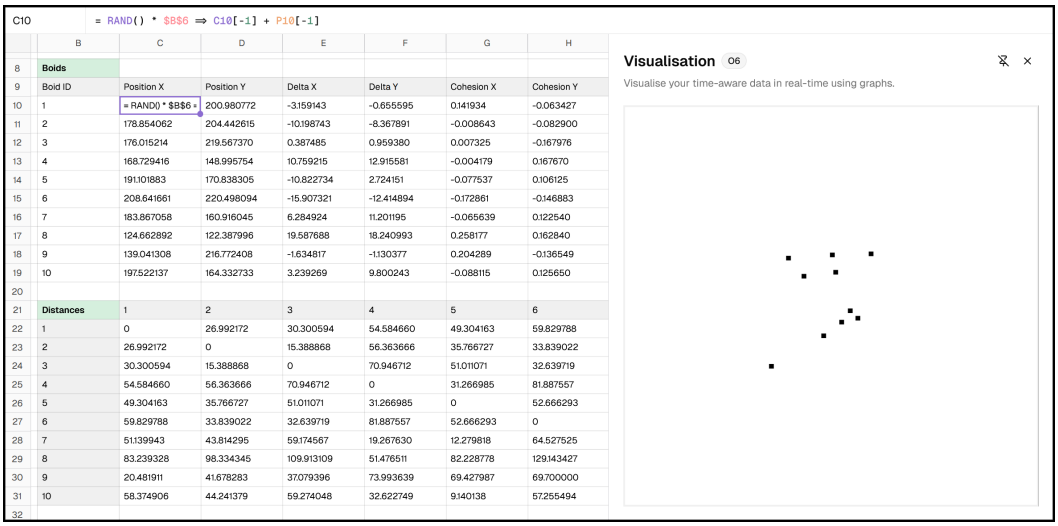
```
/* Distance */    O16 = SQRT($M16 * $M16 + $N16 * $N16)
/* Force */       P16 = $C$6 * $G16 * $G$10 / POWER($O16, 2)

/* X Position */ C16 = -1 * $B$6 => C16[-1] + $H16[-1] * $G$6
/* Y Position */ D16 = 0 => D16[-1] + $I16[-1] * $G$6
```

Case study assessment. The case study is similar to the cannonball simulation (§2). It uses absolute references (dollar sign) to enable the drag down operation for adding further planets; and it uses `[-1]` to access one past value when updating the position and velocity. What it adds to the cannonball example is scale and richer structure. We keep 18 attributes for each planet and the gravitational computation links every planet to the Sun (we ignore the gravitational forces of other planets).

Representing the simulation in a two dimensions would require one of the workarounds we observed in the formative study (§5). We could use one wide table and repeat the same 18 columns for each planet, or 18 distinct tables (Figure 14). Both approaches make cross-attribute references and adding more planets awkward. The difficulty is structurally the same as the one participants encountered when part (3) of our study required storing multiple attributes per agent. In Timeline, the model fits a single table, and the directness of the paradigm allows formulas to be checked against concrete values as they are written.

Case #2: Agent-based Flocking Simulation

Our second case study implements the Boids flocking algorithm [57] and is inspired by a model developed in Netlogo [70]. It is an example of a broad class of agent-based models that are widely used in biology, epidemiology, behavioral economics and social sciences.



	B	C	D	E	F	G	H
8	**Boids**						
9	Boid ID	Position X	Position Y	Delta X	Delta Y	Cohesion X	Cohesion Y
10	1	= RAND() * B6	200.980772	-3.159143	-0.655595	0.141934	-0.063427
11	2	178.854062	204.442615	-10.198743	-8.367891	-0.008643	-0.082900
12	3	176.015214	219.567370	0.387485	0.959380	0.007325	-0.167976
13	4	168.729416	148.995754	10.759215	12.915581	-0.004179	0.167670
14	5	191.101883	170.838305	-10.822734	2.724151	-0.077537	0.106125
15	6	208.641661	220.498094	-15.907321	-12.414894	-0.172861	-0.146883
16	7	183.867058	160.916045	6.284924	11.201195	-0.065639	0.122540
17	8	124.662892	122.387996	19.587688	18.240993	0.258177	0.162840
18	9	139.041308	216.772408	-1.634817	-1.130377	0.204289	-0.136549
19	10	197.522137	164.332733	3.239269	9.800243	-0.088115	0.125650
20							
21	**Distances**	1	2	3	4	5	6
22	1	0	26.992172	30.300594	54.584660	49.304163	59.829788
23	2	26.992172	0	15.388868	56.363666	35.766727	33.839022
24	3	30.300594	15.388868	0	70.946712	51.011071	32.639719
25	4	54.584660	56.363666	70.946712	0	31.266985	81.887557
26	5	49.304163	35.766727	51.011071	31.266985	0	52.666293
27	6	59.829788	33.839022	32.639719	81.887557	52.666293	0
28	7	51.139943	43.814295	59.174567	19.267630	12.279818	64.527525
29	8	83.239328	98.334345	109.913109	51.476511	82.228778	129.143427
30	9	20.481911	41.678283	37.079396	73.993639	69.427987	69.700000
31	10	58.374906	44.241379	59.274048	32.622749	9.140138	57.255494
32							

Fig. 16. Agent-based flocking simulation. The main birds table and an interaction table.

The simulation[2] models the flocking behavior of birds that emerges from the interaction between individual birds. The birds follow three simple rules: (i) each bird steers away from other birds in its close proximity, (ii) each bird flies towards the average position of flockmates in its visual range, and (iii) each bird tries to match the direction and velocity of birds in its visual range.

Spreadsheet structure. The model consists of a table with constants, a table with attributes of the birds and 9 square tables ($N \times N$ where N is the number of birds) that model interactions between birds. The 9 tables calculate pairwise distances and various properties based on the distances. The computation relies on the fact that the tables are aligned and so the drag down operation can be used to extend a computation for one pair of birds across the entire $N \times N$ table.

Figure 16 shows the main table and an interaction table calculating the distance between each pair of birds. The flocking logic is defined in terms of three properties. Separation is computed by multiplying the total number of birds in close proximity by a constant *avoid factor*. Cohesion is calculated as the average location of birds within its visual range, multiplied by the *centering factor*. Finally, alignment is computed as the average X and Y velocity of birds in the visual range, multiplied by the *matching factor*. The individual components (`G10`, `I10`, `K10`, `U10` for X and `H10`, `J10`, `L10`, `V10` for Y) are then added to compute the new speed:

```
/* Delta X */  E10 = RAND()*10-5 => E10[-1] + G10[-1] + I10[-1] + K10[-1] + U10[-1]
/* Delta Y */  F10 = RAND()*10-5 => F10[-1] + H10[-1] + J10[-1] + L10[-1] + V10[-1]
/* Speed */    O10 = SQRT(E10 * E10 + F10 * F10)
```

The interaction between birds is computed in square tables where a cell at relative offset i and j computes the interaction between a bird at index i and a bird at index j. For example, in the *Distances* table, cell `C23` calculates the distance between the first bird (position `$C$10`, `$D$10`) and the second bird (position `$C11`, `$D11`). The cell `C35` in the *Visual Range* table then checks if the birds are within the visual range of each other (`DIST` is a user-defined helper function):

```
/* Distances */     C23 = DIST($C11, $D11, $C$10, $D$10)
/* Visual Range */  C35 = IF(AND(C23 != 0, C23 < $D$6), 1, 0)
```

[2]Available at: https://timelinesheets.com/spreadsheet/c9b53ee9-0f1b-4d5d-a73e-19f0d201e204

Case study assessment. Unlike the previous two case studies, the flocking model uses *both* spreadsheet dimensions for its core computation. The $N \times N$ tables model interactions between each pair of birds and there is no dimension left for time. Using a flat table with N^2 columns would make indexing across tables challenging. The flocking model thus best shows what Timeline enables.

The per-bird logic again uses only simple formulas and the `[-1]` construct, but the interaction table tracking distances requires more care to construct. We can use drag down to update formulas in one direction but not in both, hence `$C11` has one flexible dimension, but `$C$10` does not. In our experience, similar interaction tables are a common design pattern for agent-based models implemented in Timeline. Their size grows quadratically with the number of agents, which bounds the scale of models that remain practical, but this is a fundamental limitation of spreadsheets.

Case #3: Moving Average Crossover

Moving average crossover is a trading strategy that relies on the interaction of a short-term and a long-term moving average [74]. When the short-term average crosses over the long-term average, it suggests a buying opportunity, while the other direction suggests a possible exit point. Our third case study implements a backtesting algorithm that replays historical price data and calculates the gains or losses of the strategy over the history.

Spreadsheet structure. The spreadsheet (Figure 17)[3] loads data from an uploaded CSV file using the `DATASET` function. This is a time-varying function that returns a value from a row determined by the current time step. In our case, it replays the history of the price. The computation of the fast and slow moving average looks at the last 10 and 20 values using a range interval. To detect a crossover, we compare the relationship in the current and the previous step:

```
B5 = NUMBER(DATASET("btc", "price"))
B9 = AVERAGE(B5[:-10])
C9 = AVERAGE(B5[:-20])
D9 = IF(AND(B9 > C9, B9[-1] <= C9[-1]),1,IF(AND(B9 < C9, B9[-1] >= C9[-1]),-1,0))
```

Case study assessment. Unlike the accumulation-based simulations of the earlier case studies, this example uses the `[-1]` construct for edge detection, which is a pattern common in synchronous dataflow languages (§3.1). The example compares the relationship between the two averages in the current and previous step. The example also uses larger history buffers (10 values for cell `B9`, 20 for `C9`). These are specified inline and so our analysis can compute the buffer sizes statically. If the floating window sizes were parameters in the sheet, we would need a more powerful analysis in order to determine that the value of the cell is constant. The example also illustrates the power of composable data visualizations, creating sparklines [27] to show the price changes inline.

Case #4: Flappy Bird Game Clone

Flappy Bird is a simple side-scroller game where the player aims to jump through a series of pipes without hitting them. The bird is dragged down by gravity and jumps up when the user hits a key.

Spreadsheet structure. Our implementation (Figure 1)[4] contains four tables that define a number of constants (images, map, pipe and bird properties). The main logic is implemented in two tables that track the positions of the bird and the pipes. The bird moves down with gravity or up when the keypress is detected using `KEYPRESS("A")`. The pipes move to the left. A new size and position are chosen when the pipe moves out of view and so the table only has a fixed size of 3.

[3] Available at: https://timelinesheets.com/spreadsheet/e653e57b-9ba9-463d-9f2b-44ddd5a426d3

[4] Available at: https://timelinesheets.com/spreadsheet/1187c0bd-fe59-4efd-9541-622f35587031. To run the game, click *View* → *Graph*, click *Play* and then press `A` to jump.

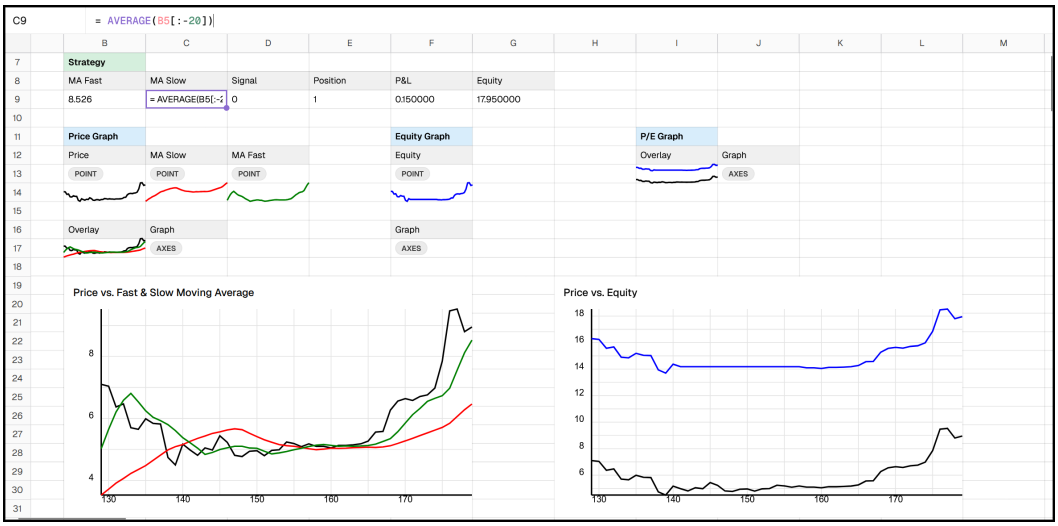


Fig. 17. Moving average crossover. Trading strategy and live charts over a floating window.

Case study assessment. The Flappy Bird game illustrates the somewhat unexpected capabilities of the Timeline programming model, extended with a simple function to detect key presses. Discrete time serves as the main game loop, and composable visualizations can be used to construct the game scene from individual images (background, bird and pipe sprites).

Two aspects point at the limits of the model. First, new rows cannot be added to a table dynamically, so the three pipes are reused as they scroll off-screen. Second, Timeline lacks the higher-order switching combinators of functional reactive programming (§ 3.3), and so reset is handled by a condition specified outside of the formula language.

6.5 Working with Two Dimensions

The formative study (§ 5) identified embedding of three-dimensional data into a grid as one of the challenges when using conventional spreadsheets. We now revisit the four case studies (§ 6), consider how they use the grid and whether (and how) they could be implemented using an ordinary two-dimensional spreadsheet system:

- Planetary orbit (Case #1) and cannonball (§ 2) simulations use a table *entities* × *attributes*. To use an ordinary spreadsheet, we would need to embed entities and attributes in a single dimension, facing the problem discussed earlier (Figure 14).
- Flocking (Case #2) uses both an *entities* × *attributes* table of size 10×17, but it also uses multiple *entities* × *entities* tables for pairwise comparison and other interactions. Doing so in an ordinary spreadsheet would require a table with $entities^2$ columns and dynamic index calculation.
- Moving average crossover (Case #3) does not use two-dimensional tables and could be encoded in an ordinary spreadsheet (using rows for individual values and columns for the averages and other computed attributes). The benefit of Timeline is that it can provide a live animated chart and could support streaming data. Streaming can be also supported in two-dimensions [19].
- Flappy bird (Case #4) also does not use two-dimensional tables, but it leverages the fact that the discrete time in Timeline is unbounded (the game can continue indefinitely) and that keyboard events can be used as inputs. This could also be achieved in a two-dimensional model [18, 19].

Implementing the first three case studies in an ordinary spreadsheet system is possible, but it would require large amount of space, especially for Case #2. This is a situation encountered in part 3 of our formative study that two of the participants judged as *miserable* and *painful*, respectively.

7 Usability Analysis

Spreadsheets are a canonical example of end-user programming tool [41, 48], but the question why and how exactly spreadsheets achieve their usability does not have an established answer. We review existing analyses and discuss what they suggest about the usability of Timeline.

Evaluating spreadsheet systems. Many analyses of spreadsheets focus on documenting their errors and engineering limitations [34, 50]. A few analyses attempt to explain what makes spreadsheets work. Nardi [48] argues that spreadsheets succeed because they closely fit the domain of tabular data analysis, while not requiring explicit control flow and abstraction. Spreadsheets also achieve a high degree of liveness, due to the edit-triggered recomputation [62].

A more structured analysis of spreadsheets [32] has been done using the cognitive dimensions of notations framework [28]. It identified several notable characteristics. Spreadsheets support *closeness of mapping*. If the problem naturally has a two-dimensional strucutre, the problem domain fits with the program domain. They do not suffer from high *viscosity*, meaning that changes are easy to make (even if this requires copying the modified formula). However, spreadsheets suffer from poor *visibility* of code and *hidden dependencies*, because logic is hidden inside formulas that have to be explicitly viewed. The analysis also attributes the success of spreadsheets to their low *abstraction level* and support for *progressive evaluation*, i.e., live immediate feedback. The analysis provides a limited baseline for heuristic evaluation of spreadsheet extensions.

Adding the time dimension. Applying Nardi's analysis to Timeline suggests that spreadsheets with discrete time fit a broader range of application domains. In addition to tabular data analysis, these include the four domains discussed above (§ 6) and so Timeline can achieve better *closeness of mapping* by not requiring an auxiliary tables (Figure 14) to store values. The system also preserves a high degree of liveness through edit-triggered recomputation.

Reference to values in earlier steps, such as `A1[-1]`, are less direct because the value of `A1[-1]` is not readily displayed in the grid and cannot be referenced by clicking on it when writing a formula. This property is not identified by aformentioned analyses, but relates to *concrete programming* [24]. Programming in Timeline is thus less concrete than in ordinary spreadsheets.

Timeline has a higher *viscosity* than conventional spreadsheets when the user wants to modify a value at certain time steps. When encoding time as rows, this can be done by selecting the affected range and modifying the value. In Timeline, such change can only be done by using `IF` function with a condition based on the current `STEP()`. Modifying the logic for all time steps is easier in Timeline, where it exists in a single formula, as opposed to multiple copies of the same formula.

The *visibility* of code remains unchanged, but *visibility* of computed values is decreased because only values for a single time step are displayed. (When representing time as rows, multiple time steps are visible.) To alleviate this limitation, Timeline makes it easy to construct sparklines as shown in the financial analysis (Case #3), which make previous values visible in a concise form.

The limits of discrete time. Timeline uses discrete time, which limits what kind of models it can express. The grid is fixed and entities cannot be created over time. A flocking simulation with constant number of birds (Case #2) is expressible, but a predator-prey model in which individuals are born and die can only be modelled by pre-allocating a table of entities. Moreover, Timeline has no analogue of the clock calculi of synchronous languages [15], which would allow cells to progress at different rates. Supporting a more general notion of discrete time would be useful in domains

such as finance where users often work with data indexed by time. Generalising the notion of time would require adapting not just the evaluation model, but also the indexing logic and notation.

8 Related Work

Timeline shows that a large number of programming language research concepts are directly applicable to spreadsheets. Earlier discussion (§ 3) covers work on coeffects, functional reactive programming and visualization. This section reviews the remaining main related research areas.

Extending the Spreadsheet Model. Forms/3 [14] is a pioneering extension of spreadsheets to non-tabular format, that inspired later work [8]; spreadsheets have been extended to support object-orientation [21, 47] and programming by demonstration [30]. An early attempt to use spreadsheets for interactive graphics is by Wilde et al. [69]. Websheets [18] and streaming extensions [19] integrate spreadsheets with live data (but without adding a third dimension). Spreadsheets served as an inspiration for multiple other languages [53, 73]. Cuscus is a design prototype for describing rich visualizations in a spreadsheet [45] and our work has also been inspired by sparklines [27].

Abstractions and Types for Spreadsheets. Sheet-defined functions are introduced in [40] with formal semantics developed later [11, 12, 61]. Later work allows variable-size input [46], while gridlets [39] serve as reusable sheet blocks and have been implemented using spilled arrays [72], which directly inspired our semantics. Multiple systems provide static type systems for spreadsheets [3, 4], perform static analysis of formulas [20] or apply other software engineering techniques [33].

Dataflow and Functional Reactive Programming. The Timeline execution draws inspiration from synchronous dataflow languages [6, 9, 10, 17, 31]. Our programming model is most closely related to Lustre, whose clock calculus [15] resembles our checking of past accesses. Functional reactive programming systems [23, 25, 66] do not use discrete time; event-based functional reactive programming is more similar to our approach [67].

Semantics of Reactive and Dataflow Programs. Our semantics has been inspired by operational approaches [11, 12, 72]. More dynamic reactive systems often use graph-based evaluation [49, 60]. The semantics of dataflow languages has been given in both monadic [65] and comonadic style [64]. The semantics of functional reactive programming languages has been defined denotationally [26] using a range of categorical models [38, 43], but recently also in an operational style [36].

9 Conclusions

Spreadsheets are the most widely used programming systems in the world, but their expressive power is limited. Attempts to extend spreadsheets face a challenge—how to extend the power of the programming model, without losing their liveness, directness and usability? Timeline shows that adding a time dimension to spreadsheets keeps their desirable properties, but makes them more powerful. Spreadsheets with time can be used to implement agent-based models, physics simulations, financial analyses and even interactive games.

Technically, the design and implementation of a spreadsheet system with time can draw on a rich body of programming languages research. Our execution model takes inspiration from synchronous dataflow languages and functional reactive programming, we guarantee memory-bounded execution using coeffects, and we can support rich charts by using ideas rooted in grammar of graphics. There is more to spreadsheets than meets the (programming language researcher's) eye!

Acknowledgments

The authors thank Clemens Nylandsted Klokmose for fruitful discussions that shaped the project, Mickaël Laurent for pointing out the link to synchronous languages, Vít Ungermann for contributions to the composable visualization support, Raphael Wimmer, Jakob Haimerl and Tobias Wittl for feedback on the system design and anonymous reviewers for their feedback, suggestions and encouragement to include qualitative evaluation.

The work has been supported by the Charles University grant PRIMUS/24/SCI/021 and by the Czech Ministry of Education, Youth and Sports grant ERC-CZ LL2325.

Data Availability Statement

An implementation of Timeline is available as an associated artifact [56], consisting of:

- Stand-alone package that can be used to run Timeline locally (Docker image)
- Easily accessible implementation of the four case studies discussed in the paper (Flappy Bird, Moving average crossover, Flocking simulation, Solar system simulation) and the introductory example (Cannonball simulation).

The artifact differs from the fully general theoretical account provided in the paper in that:

- The implementation of cell evaluation is based on topological ordering of dependencies and update propagation. This aspect is left unspecified in this paper.
- All the examples and case studies work exactly as described in the paper. Some may use advanced features for readability (such as named ranges and user-defined functions).
- The system implements only the simple mechanism for checking the required number of past values (without the coeffect-based optimization). Coeffect-based model is theoretically interesting, but not necessary for efficiency in practice.
- The execution using bounded memory can be enabled or disabled (it saves memory, but makes it difficult to go to the previous step, which is helpful for debugging)
- Formula editing based on holes is not implemented. Currently, Timeline inserts a cell reference on click when `Ctrl` is pressed. Implementing the theory described in the paper is future work.

References

[1] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory M.P. O'Hare. 2017. Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Computer Science Review* 24 (2017), 13–33. doi:10.1016/j.cosrev.2017.03.001

[2] Robin Abraham, Margaret Burnett, and Martin Erwig. 2009. *Spreadsheet Programming.* John Wiley & Sons, Ltd, 2804–2810. doi:10.1002/9780470050118.ecse415

[3] Robin Abraham and Martin Erwig. 2006. Type inference for spreadsheets. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Venice, Italy) *(PPDP '06)*. Association for Computing Machinery, New York, NY, USA, 73–84. doi:10.1145/1140335.1140346

[4] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. 2003. A type system for statically detecting spreadsheet errors. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* 174–183. doi:10.1109/ASE.2003.1240305

[5] Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 69–83. doi:10.1007/11693024_6

[6] Edward A. Ashcroft and William W. Wadge. 1977. Lucid, a Nonprocedural Language with Iteration. *Commun. ACM* 20, 7 (1977), 519–526. doi:10.1145/359636.359715

[7] Patrick Bahr. 2022. Modal FRP for all: Functional reactive programming without space leaks in Haskell. *Journal of Functional Programming* 32 (2022), e15. doi:10.1017/S0956796822000132

[8] Pavel Bažant and Michaela Maršálková. 2018. A Non-Tabular Spreadsheet with Broad Applicability. In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming (Programming '18).* Association for Computing Machinery, New York, NY, USA, 161–165. doi:10.1145/3191697.3214343

[9] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003), 64–83. doi:10.1109/JPROC.2002.805826

[10] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. doi:10.1016/0167-6423(92)90005-V

[11] Alexander Asp Bock, Thomas Bøgholm, Peter Sestoft, Bent Thomsen, and Lone Leth Thomsen. 2022. On the cost semantics for spreadsheets with sheet-defined functions. *J. Comput. Lang.* 69 (2022), 101103. doi:10.1016/J.COLA.2022.101103

[12] Alexander Asp Bock, Thomas Bøgholm, Peter Sestoft, Bent Thomsen, and Lone Leth Thomsen. 2020. On the semantics for spreadsheets with sheet-defined functions. *Journal of Computer Languages* 57 (2020), 100960. doi:10.1016/j.cola.2020.100960

[13] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–370. doi:10.1007/978-3-642-54833-8_19

[14] Margaret Burnett, John Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer Gottfried, and Sherry Yang. 2001. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming* 11, 2 (2001), 155–206. doi:10.1017/S0956796800003828

[15] Paul Caspi. 1992. Clocks in dataflow languages. *Theoretical Computer Science* 94, 1 (1992), 125–140. doi:10.1016/0304-3975(92)90326-B

[16] Paul Caspi, Grégoire Hamon, and Marc Pouzet. 2008. Synchronous functional programming: The Lucid Synchrone experiment. *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes* (2008), 28–41.

[17] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 178–188. doi:10.1145/41625.41641

[18] Kerry Shih-Ping Chang and Brad A. Myers. 2014. Creating Interactive Web Data Applications with Spreadsheets. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. Association for Computing Machinery, New York, NY, USA, 87–96. doi:10.1145/2642918.2647371

[19] Kerry Shih-Ping Chang and Brad A. Myers. 2015. A Spreadsheet Model for Handling Streaming Data. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. Association for Computing Machinery, New York, NY, USA, 3399–3402. doi:10.1145/2702123.2702587

[20] Tie Cheng and Xavier Rival. 2015. Static Analysis of Spreadsheet Applications for Type-Unsafe Operations Detection. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 26–52. doi:10.1007/978-3-662-46669-8_2

[21] Chris Clack and Lee Braine. 1997. Object-oriented functional spreadsheets. In *Proceedings of the 10th Glasgow Workshop on Functional Programming (GlaFP'97)*.

[22] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 294–308. doi:10.1007/11693024_20

[23] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell* (Uppsala, Sweden) *(Haskell '03)*. Association for Computing Machinery, New York, NY, USA, 7–18. doi:10.1145/871895.871897

[24] Jonathan Edwards. 2004. Example centric programming. (2004), 124. doi:10.1145/1028664.1028713

[25] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands) *(ICFP '97)*. Association for Computing Machinery, New York, NY, USA, 263–273. doi:10.1145/258948.258973

[26] Conal M. Elliott. 2009. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (Edinburgh, Scotland) *(Haskell '09)*. Association for Computing Machinery, New York, NY, USA, 25–36. doi:10.1145/1596638.1596643

[27] Leo D. Frishberg. 2011. Interactive sparklines: a dynamic display of quantitative information. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI EA '11)*. Association for Computing Machinery, New York, NY, USA, 589–604. doi:10.1145/1979742.1979655

[28] Thomas R. G. Green and Marian Petre. 1996. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *Journal of Visual Languages and Computing* 7, 2 (1996), 131–174. doi:10.1006/jvlc.1996.0009

[29] Thomas A Grossman and Vijay Mehrotra. 2023. A Use Case-Engineering Resources Taxonomy for Analytical Spreadsheet Models. In *21st EuSpRIG Annual Conference "The Spreadsheet Crisis: Regaining Control"*. doi:10.48550/ARXIV.2309.00104

[30] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (Aug. 2012), 97–105. doi:10.1145/2240236.2240260

[31] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language Lustre. *Proc. IEEE* 79, 9 (1991), 1305–1320. doi:10.1109/5.97300

[32] D.G. Hendry and T.R.G. Green. 1994. Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model. *International Journal of Human-Computer Studies* 40, 6 (1994), 1033–1065. doi:10.1006/ijhc.1994.1047

[33] Felienne Hermans, Bas Jansen, Sohon Roy, Efthimia Aivaloglou, Alaaeddin Swidan, and David Hoepelman. 2016. Spreadsheets are Code: An Overview of Software Engineering Approaches Applied to Spreadsheets. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. 56–65. doi:10.1109/SANER.2016.86

[34] Felienne Hermans, Martin Pinzger, and Arie van Deursen. 2015. Detecting and Refactoring Code Smells in Spreadsheet Formulas. *Empirical Software Engineering* 20, 2 (2015), 549–575. doi:10.1007/s10664-013-9296-2

[35] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. *Arrows, Robots, and Functional Reactive Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 159–187. doi:10.1007/978-3-540-44833-4_6

[36] Jordan Ischard, Frédéric Dabrowski, Jules Chouquet, and Frédéric Loulergue. 2025. A Mechanized Formalization of an FRP Language with Effects. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing* (Catania International Airport, Catania, Italy) *(SAC '25)*. Association for Computing Machinery, New York, NY, USA, 1431–1439. doi:10.1145/3672608.3707907

[37] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (2023), 13:1–13:59. doi:10.22152/programming-journal.org/2023/7/13

[38] Wolfgang Jeltsch. 2014. An abstract categorical semantics for functional reactive programming with processes. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages Meets Program Verification* (San Diego, California, USA) *(PLPV '14)*. Association for Computing Machinery, New York, NY, USA, 47–58. doi:10.1145/2541568.2541573

[39] Nima Joharizadeh, Advait Sarkar, Andrew D. Gordon, and Jack Williams. 2020. Gridlets: Reusing Spreadsheet Grids. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3334480.3382806

[40] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. 2003. A user-centred approach to functions in Excel. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden) *(ICFP '03)*. Association for Computing Machinery, New York, NY, USA, 165–176. doi:10.1145/944705.944721

[41] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages. doi:10.1145/1922649.1922658

[42] Neelakantan R. Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 221–232. doi:10.1145/2500365.2500588

[43] Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. 257–266. doi:10.1109/LICS.2011.38

[44] Steve Litt. 2017. Mitigating Spreadsheet Risk in Complex Multi-Dimensional Models in Excel. In *Proceedings of the EuSpRIG 2017 Conference "Spreadsheet Risk Management"*. European Spreadsheet Risks Interest Group (EuSpRIG), London, UK.

[45] Mariana Marasoiu, Detlef Nauck, and Alan F. Blackwell. 2019. Cuscus: An End User Programming Tool for Data Visualisation. In *End-User Development*, Alessio Malizia, Stefano Valtolina, Anders Morch, Alan Serrano, and Andrew Stratton (Eds.). Springer International Publishing, Cham, 115–131. doi:10.1007/978-3-030-24781-2_8

[46] Matt McCutchen, Judith Borghouts, Andrew D. Gordon, Simon Peyton Jones, and Advait Sarkar. 2020. Elastic sheet-defined functions: Generalising spreadsheet functions to variable-size input arrays. *Journal of Functional Programming* 30 (2020), e26. doi:10.1017/S0956796820000234

[47] Matt McCutchen, Shachar Itzhaky, and Daniel Jackson. 2016. Object Spreadsheets: A New Computational Model for End-User Development of Data-Centric Web Applications. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*. Association for Computing Machinery, New York, NY, USA, 112–127. doi:10.1145/2986012.2986018

[48] Bonnie A. Nardi. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, Cambridge, MA. 162 pages.

[49] Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter. 2023. A Graph-Based Formal Semantics of Reactive Programming from First Principles. In *Proceedings of the 24th ACM International Workshop on Formal Techniques for Java-like Programs* (Berlin, Germany) *(FTfJP '22)*. Association for Computing Machinery, New York, NY, USA, 18–25. doi:10.1145/3611096.3611101

[50] Raymond R. Panko. 1998. What We Know About Spreadsheet Errors. *Journal of Organizational and End User Computing* 10, 2 (1998), 15–21. doi:10.4018/joeuc.1998040102

[51] Tomas Petricek. 2017. Compost.js: Composable Data Visualization Library. https://github.com/compostjs/compost. Accessed 10 March 2026.

[52] Tomas Petricek. 2021. Composable data visualizations. *Journal of Functional Programming* 31 (2021), e13. doi:10.1017/S0956796821000046

[53] Tomas Petricek. 2022. The Gamma: Programmatic Data Exploration for Non-programmers. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–7. doi:10.1109/VL/HCC53370.2022.9833134

[54] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2013. Coeffects: Unified Static Analysis of Context-Dependence. In *Automata, Languages, and Programming*, Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg (Eds.). Springer Berlin Heidelberg, 385–397. doi:10.1007/978-3-642-39212-2_35

[55] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 123–135. doi:10.1145/2628136.2628160

[56] Tomas Petricek and Boďa Tomáš. 2026. *Timeline: Adding the Time Dimension to Spreadsheets (Artifact)*. doi:10.5281/zenodo.21971165

[57] Craig W. Reynolds. 1998. *Flocks, herds, and schools: a distributed behavioral model.* Association for Computing Machinery, New York, NY, USA, 273–282. doi:10.1145/280811.281008

[58] Advait Sarkar and Andrew D. Gordon. 2018. How do people learn to use spreadsheets? (Work in progress). In *Proceedings of the 29th Annual Conference of the Psychology of Programming Interest Group (PPIG 2018)*. PPIG, 28–35. https://advait.org/publications-web/sarkar-2018-spreadsheet-learning

[59] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 341–350. doi:10.1109/TVCG.2016.2599030

[60] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 659–668. doi:10.1109/TVCG.2015.2467091

[61] Peter Sestoft and Jens Zeilund Sørensen. 2013. Sheet-Defined Functions: Implementation and Initial Evaluation. In *End-User Development*, Yvonne Dittrich, Margaret Burnett, Anders Mørch, and David Redmiles (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 88–103. doi:10.1007/978-3-642-38706-7_8

[62] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. doi:10.1109/LIVE.2013.6617346

[63] The Center for Connected Learning and Computer-Based Modeling. 2025. *NetLogo Models Library.* https://ccl.northwestern.edu/netlogo/models/ Accessed: 25 February 2026.

[64] Tarmo Uustalu and Varmo Vene. 2006. The Essence of Dataflow Programming. In *Central European Functional Programming School*, Zoltán Horváth (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 135–167. doi:10.1007/11894100_5

[65] William Wadge. 1995. Monads and intensionality. In *International Symposium on Lucid and Intensional Programming*, Vol. 95.

[66] Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 242–252. doi:10.1145/349299.349331

[67] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-Driven FRP. In *Practical Aspects of Declarative Languages*, Shriram Krishnamurthi and C. R. Ramakrishnan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–172. doi:10.5555/645772.667941

[68] Hadley Wickham. 2010. A Layered Grammar of Graphics. *Journal of Computational and Graphical Statistics* 19, 1 (2010), 3–28. doi:10.1198/jcgs.2009.07098

[69] Nicholas Wilde and Clayton Lewis. 1990. Spreadsheet-based interactive graphics: from prototype to tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Seattle, Washington, USA) *(CHI '90)*. Association for Computing Machinery, New York, NY, USA, 153–160. doi:10.1145/97243.97268

[70] Uri Wilensky. 1998. NetLogo Flocking model. http://ccl.northwestern.edu/netlogo/models/Flocking.

[71] Leland Wilkinson. 2012. *The Grammar of Graphics.* Springer Berlin Heidelberg, 375–414. doi:10.1007/978-3-642-21551-3_13

[72] Jack Williams, Nima Joharizadeh, Andrew D. Gordon, and Advait Sarkar. 2020. Higher-Order Spreadsheets with Spilled Arrays. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020 (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 743–769. doi:10.1007/978-3-030-44914-8_27

[73] A.G. Yoder and D.L. Cohn. 1994. Real spreadsheets for real programmers. In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*. 20–30. doi:10.1109/ICCL.1994.288396

[74] Valeriy Zakamulin and Javier Giner. 2025. *The Ultimate Moving Average Handbook: Bringing Science into the Art of Trend Following*. Palgrave Macmillan, Cham, Switzerland. doi:10.1007/978-3-031-90907-8

Received 2026-03-17; accepted 2026-08-06