

USER-FACING SPREADSHEET PROGRAMMING LANGUAGE

TECHNICAL FIELD

[0001] The present invention generally relates to computer spreadsheet systems, and more particularly relates to methods and systems for a user-facing spreadsheet programming language.

BACKGROUND OF THE DISCLOSURE

[0002] Many current mainstream languages used by professional programmers, such as C, C++, Java, and Python, are based on a linear flow of plain text. A central goal of these languages is to enable application building in a Lego®-like manner where functionalities are not built from scratch, but rather assembled from existing libraries, usually in the form of functions and classes. Different libraries can coexist and cooperate and the programmers' main job is simply to customize and orchestrate these logic blocks. Today's languages have proven to be extremely versatile and have largely attained this goal. Unfortunately, they've also proven to be too abstract to a majority of non-programmers, resulting in a huge gulf between programmers and non-programmers, and leading to non-programmers devolving into mere passive consumers of technology.

[0003] On the other side of the chasm are spreadsheets, which is based around a two-dimensional (2D) grid structure rather than plain text. Spreadsheet systems such as Microsoft Excel and Google Sheet are some of the only practically accessible systems where non-programmer users can express non-trivial logic for execution by a computer. However, when viewed as a development environment, spreadsheets lack basic facilities considered essential for mainstream programming languages, such as

the ability to re-use code in the form of functions and classes, the ability to unit test, and the ability to import prebuilt third-party spreadsheets. Therefore, spreadsheets in practice are often found with copious amounts of duplicate code and bugs, leading to an environment where spreadsheets quickly become unmanageable as requirements get more complex. Furthermore, there's no upgrade path from a spreadsheet to a more technically sound solution. Users either turn to Visual Basics for Applications (VBA) programmers to build more complex functionalities (where VBA is a conventional plain text-based language) or, very often, simply throw away spreadsheets and completely reimplement the logic in a mainstream language.

[0004] Thus, there is a need for a robust user-facing spreadsheet programming language which overcomes the drawbacks of prior spreadsheet enabled programming and provides an easy, flexible and versatile backward-compatible spreadsheet-friendly solution. Furthermore, other desirable features and characteristics will become apparent from the subsequent detailed description and the appended claims, taken in conjunction with the accompanying drawings and this background of the disclosure.

SUMMARY

[0005] According to at least one aspect of the present embodiments, a method for spreadsheet programming is provided. The method includes assigning labels to cells of a spreadsheet, each of the labels assigned to one of a plurality of adjoining variable value cells for data corresponding to a plurality of values of a variable. The plurality of adjoining variable value cells is user defined by a pair of labels indicating a start and an end of a cell range of the plurality of adjoining variable value cells. The method further includes defining a spill area into which data can flow from a cell having an expression defined therein in response to the expression including a

variable range defined by the plurality of values of the variable. The spill area is automatically sized to a plurality of expression solutions corresponding to the plurality of values of the variable.

[0006] According to another aspect of the present embodiments, an additional method for spreadsheet programming is provided. The method includes substituting another item into a spreadsheet in response to cell values of a cell in the spreadsheet comprising a function call and/or a remote callback.

[0007] According to a further aspect of the present embodiments, an additional method for spreadsheet programming is provided. The method includes substituting dynamic webpage request query parameters into a spreadsheet in response to a dynamic webpage request and rendering the spreadsheet as a webpage in response thereto.

[0008] According to an additional aspect of the present embodiments, a spreadsheet is provided. The spreadsheet includes a plurality of cells arranged in a two-dimensional array. One or more of the plurality of cells include cell values for substituting another item into the spreadsheet, the cell values including a function call and/or a remote callback.

[0009] According to yet another aspect of the present embodiments, a spreadsheet is provided. The spreadsheet includes a plurality of cells arranged in a two-dimensional array. One or more of the plurality of cells include cell values for substituting dynamic webpage request parameters in response to a dynamic webpage request. The spreadsheet is rendered into a webpage in response to the dynamic webpage request parameters.

[0010] According to a further aspect of the present embodiments, a spreadsheet is provided. The spreadsheet includes a plurality of cells arranged in a two-

dimensional array. One or more of the plurality of cells include a plurality of adjoining variable value cells for data corresponding to a plurality of values of a variable. The plurality of adjoining variable value cells are user defined by a pair of cell values indicating a start and an end of a cell range of the plurality of adjoining variable value cells. The plurality of cells also include a spill area into which data can flow from a cell having an expression defined therein in response to the expression including a variable range defined by the plurality of values of the variable. The spill area is automatically sized to a plurality of expression solutions corresponding to the plurality of values of the variable.

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] The accompanying figures, where like reference numerals refer to identical or functionally similar elements throughout the separate views and which together with the detailed description below are incorporated in and form part of the specification, serve to illustrate various embodiments and to explain various principles and advantages in accordance with present embodiments.

[0012] FIG. 1 is an illustration of an exemplary spreadsheet spill in accordance with present embodiments.

[0013] FIG. 2, comprising FIGs. 2A to 2D, depicts illustrations of spreadsheet oscillating spill areas in accordance with present embodiments, wherein FIG. 2A depicts a potential oscillating spill area, FIG. 2B depicts a first state of the oscillating spill area, FIG. 2C depicts a second state of the oscillating spill area, and FIG. 2D depicts a steady final state of the oscillating spill area.

[0014] FIG. 3, comprising FIGs. 3A, 3B and 3C, depicts reuse logic by substitution in accordance with the present embodiments, wherein FIG. 3A depicts an exemplary tax computation sheet, FIG. 3B depicts the exemplary tax computation sheet of FIG.

3A with substitutable cells defined, and FIG. 3C depicts an exemplary sheet tax after substitution of values in the exemplary tax computation sheet of FIG. 3B.

[0015] FIG. 4, comprising FIGs. 4A, 4B and 4C, depicts sheets exemplifying rules for simple substitutions and nested substitutions in accordance with the present embodiments, wherein FIG. 4A depicts a first sheet used in the examples of substitutions in FIGs. 4B and 4C, FIG. 4B is a simple substitution, and FIG. 4C is a nested substitution.

[0016] FIG. 5, comprising FIGs. 5A to 5D, depicts sheets exemplifying callback rules in accordance with the present embodiments, wherein FIG. 5A depicts a labelled sheet, FIG. 5B depicts a callback to the sheet of FIG. 5A, FIG. 5C depicts a labelled sheet, and FIG. 5D depicts a thunk callback to the sheet of FIG. 5C.

[0017] FIG. 6, comprising FIGs. 6A and 6B, depicts operation of a rule for spill barriers in accordance with the present embodiments, wherein FIG. 6A is a sheet with a spill barrier defined and FIG. 6B is a substituted sheet where the spill barrier automatically creates cells in an elastic row in order to accommodate additional values.

[0018] FIG. 7 depicts use of label position expressions in accordance with the present embodiments

[0019] FIG. 8, comprising FIGs. 8A, 8B and 8C, depicts operation of a rule for style regions in accordance with the present embodiments, wherein FIG. 8A depicts a style table, FIG. 8B depicts a sheet having a style region defined by the style table of FIG. 8A, and FIG. 8C depicts a table a style region programmed for a three row/two column portion of a spreadsheet.

[0020] FIG. 9, comprising FIGs. 9A and 9B, depicts operation of a rule for table-within-tables (TWITs) in accordance with the present embodiments, wherein FIG. 9A

depicts a sheet with unpleasantly wide cells below a title cell and FIG. 9B depicts the contents of the sheet of FIG. 9A with a defined TWIT to provide a better visual appearance.

[0021] FIG. 10, comprising FIGs. 10A and 10B, depicts operation of classes as directories in accordance with the present embodiments, wherein FIG. 10A depicts creation of a table of the rectangle class and FIG. 10B depicts computation of an area of a rectangle.

[0022] And FIG. 11 depicts a spreadsheet importing the document described by FIGs. 10A and 10B at a certain node in accordance with the present embodiments.

[0023] Skilled artisans will appreciate that elements in the figures are illustrated for simplicity and clarity and have not necessarily been depicted to scale.

DETAILED DESCRIPTION

[0024] The following detailed description is merely exemplary in nature and is not intended to limit the invention or the application and uses of the invention. Furthermore, there is no intention to be bound by any theory presented in the preceding background of the invention or the following detailed description. It is the intent of present embodiments to present a user-facing spreadsheet-based programming language that allows a user to build, test, and reuse logic blocks, to build dynamic web sites, and to build large-scale shared spreadsheets, all while staying completely within the constraints of a two-dimensional (2D) spreadsheet. The intent is further to put 2D spreadsheets on an equal footing with today's general-purpose programming languages.

[0025] Many visual languages were invented in pursuit of bringing general-purpose programming to the masses, but dragging little shapes across screens proved to be too

cumbersome with very limited expressiveness. Some visual languages survive in niches that map well to a graphical representation, such as MATLAB Simulink for signal processing and Unreal game engine's Blueprints for scripting gameplay elements, but none is considered viable as a general-purpose programming environment.

[0026] Referring back to the visual environment of 2D spreadsheets having a two-dimensional array of cells, in today's conventional spreadsheets defining an expression for a single cell is fairly straightforward. Cells can be referenced either by their coordinates, like *A1*, or by assigning labels. In accordance with the present embodiments, the method of assigning labels is exclusively utilized for referencing cells.

[0027] In conventional spreadsheets, filling an area relies essentially on copying and pasting expressions. How the cells should behave when a row is inserted in the middle or when one of the expressions is changed by the user is difficult to determine. Should the new row automatically be filled with the current pattern? Should the same expression change apply to cells that were pasted from it? The ambiguity is dangerous by itself, but more importantly, the ambiguity thwarts the effort to reuse the spreadsheet's logic with any certainty.

[0028] Some current spreadsheet systems have array expressions to mitigate the problem. However, the user must explicitly specify a range of cells for application of the expression and what happens when the input range changes size is still not clear. In recognition of this fragility of copied logic and other conventional spreadsheet issues, a plurality of spreadsheet programming rules in accordance with the present embodiments are defined to overcome the drawbacks of current conventional spreadsheet programs.

[0029] In accordance with a first aspect of the present embodiments, cell spill areas are defined for robust and advantageous auto-filling of a range of cells in a spreadsheet. Referring to FIG. 1, an illustration 100 depicts a spreadsheet spill in an exemplary spreadsheet 105 in accordance with the present embodiments. The user only defines a single expression 110 in the top-left cell 120, of an area 130, without having to specify a size. The head 120 evaluates the expression 110 not necessarily to a single value, but in accordance with the present embodiments can evaluate the expression 110 to a rectangular table of values. The values then spill to cells on the right and to the bottom of the cell with the expression as necessary. If any cell in the spill area already has a value set, or is also spilled into from another cell, then a spill conflict is reported as discussed hereinbelow.

[0030] In the expression 110, *#billing_rate* is a shorthand variable for *billing_rate:Zbilling_rate*, which is a user defined pair of labels which denotes a rectangular area of adjoining variable value cells from the cell with label *billing_rate* to the cell with label *Zbilling_rate*, both inclusive, which contain data *100 \ 150 \ 60* corresponding to a plurality of values of the variable *#billing_rate*. Similarly, *#hours* denotes the rectangular area of adjoining variable value cells from *hours* to *Zhours*. In the exemplary spreadsheet 105, cell 120's expression 110 *#billing_rate * #hours* multiplies the two tables *100 \ 150 \ 60* for areas 132 and *10 \ 2 \ 10* for area 134 together to produce a table of 3 rows and 1 column *1000 \ 300 \ 600*. Since this table cannot be contained in cell 120 itself, it spills into the two cells below, making cell 120's spill area 130. Subsequently, in expression 150, without an explicit label *Zamount* defined in sheet 105, *#amount* denotes this spill area 130 of cell 120 *amount*, including the cell *amount* itself. The entire expression *Sum(#amount)* 150 in the cell

140 thus computes the sum of the spill area 130 of cell 120 *amount*, $1000 + 300 + 600 = 1900$.

[0031] Of particular note is the problem of oscillating spill areas. Referring next to FIGs. 2A, 2B, 2C and 2D, illustrations 200, 220, 240, 260 depict spreadsheets exemplifying the problem in accordance with the present embodiments. In the illustration 200, a function *Zeros(a, a)* 205 generates a table 210 of size *a* by *a* filled with zeros in response to *a* in cell 215a set equal to "2".

[0032] If the user were to change the value *a* to "3" as shown in the cell 215b of the spreadsheet 220, the spreadsheet will then oscillate between two states: State One shown in the illustration 220 (FIG. 2B) where we have enlarged the spill area of the top left cell 225a to 3x3, causing the cell 215b to report a spill conflict, thereby causing a pending re-evaluation of the cell 225a; and State Two shown in the illustration 240 (FIG. 2C) where the top left cell 225b has an error value and therefore no spill area beyond itself, and the cell 215c has the value of "3" with no spill conflict.

[0033] In order to address the issue of oscillating spill areas and in accordance with an aspect of the present embodiments, a spill area rule is defined that spill areas are non-shrinking, that is, they can only enlarge before subsequent user inputs. Following this spill area rule, the spill area of the top-left cell remains a 3 by 3 spill area 265 even in State Two, resulting in the steady final state shown in the illustration 260 (FIG. 2D).

[0034] In accordance with the present embodiments, this spill area rule will advantageously always result in a steady final state in the absence of new user input. Intuitively, the number of pairs of spill areas conflicting each other is capped by the pairs of expressions defined. If spill areas cannot shrink, then the number of such relationships must converge to a steady value. Once the spill area conflict

relationships have stabilized, a cell value can either be computed unambiguously; otherwise it's part of a dependency loop that can easily be detected and reported as such.

[0035] Another issue with conventional spreadsheets is that it is challenging to reuse logic defined in another spreadsheet. Referring to FIG. 3A, an illustration 300 depicts a sheet 310 which is defined to compute the hypothetical tax due for an individual. Even though the logic is clearly written out, conventional spreadsheets do not provide an adequate way to apply the logic defined in the sheet 310 to another individual's taxes. Either the input values of the income, the number of children (i.e., *nb_children*), etc are redefined, therefore destroying the currently computed values, or the logic must be duplicated into another sheet and modified in place there, even though duplicating code is usually strongly discouraged in programming.

[0036] In order to overcome this deficiency in conventional spreadsheets, a substitution rule is defined in accordance with the present embodiments to support logic reuse by substitutions. Referring to FIG. 3B, an illustration 330 depicts a sheet 335 with substitutable cells 342, 344, 346, 348 (those labels that start with X) defined. With sheet 335, cell 340 can be defined to be the *Return* value of *tax*, the logic to compute the tax. For example, the code can be written *tax(income=106000, nb_children=2, donations=1000 \ 2000 \ 3000)* to compute the tax for another hypothetical person. This code represents a substituted sheet of *tax* with a few amendments as depicted in FIG. 3C, and takes the value of *#Return* in the substituted sheet as its value. Positional inputs in the sheet 335 are supported if labels such as *X0*, *X1* are defined.

[0037] Referring to FIG. 3C, an illustration 360 depicts a sheet 365 represented by the code *tax(income=106000, nb_children=2, donations=1000 \ 2000 \ 3000)*. The

sheet 365 is copied from the sheet 335 *tax*, with the cell *Xincome* 370 replaced with the code *106000* 372, the cell *Xnb_children* 374 is replaced with the code *2* 376, and the cell *Xdonations* 378 is replaced with the code *1000 \ 20000 \ 3000* 380. The code set for other cells in the range *Xdonations:ZXdonations* are also removed, that is *ZXdonations=200* in sheet 335 is removed. By removing the label *ZXdonations*, the value of *Xdonations* 378 defined by the code *1000 \ 20000 \ 3000* 380 is advantageously allowed to successfully spill into the cells below it, while the meaning of *#Xdonations* remains well defined and conforms to a user's expectations.

[0038] Positional inputs are supported using labels of the pattern *X0*, *X1*. For example, a code *tax(106000, 2, donations=1000 \ 2000)* would constitute substitutions of *X0* with code *106000*, *X1* with code *2*, and named input *Xdonations* substituted as described previously.

[0039] An issue with substitutions as defined above is nested substitutions. Referring to FIGs. 4A and 4B, illustrations 400, 430 depict sheets 410 *add*, and 440 *square_add*. An ambiguity arises when we wish to evaluate *square_add(20)*. It is unambiguous that *square_add.X0* should be substituted with *20*, which in turns causes *square_add.Return* to request *add(400)*. Further, in the sheet 410 *add*, it is unambiguous that *X0* should be substituted with *400*, but in *add.Return*, it's not clear what *square_add.X0* should be. There are two possible interpretations. The first interpretation is that *square_add.X0* should be *10* because in the substitution *add(400)*, *square_add.X0* is not substituted. This first interpretation is termed a simple substitution and is more performant because there are fewer substitutions, but it could be somewhat surprising. The second interpretation is that *square_add.X0* should be *20*, because *add(400)* is nested in the substitution *square_add(20)* where

square_add.X0 is substituted as *20*. This second interpretation is termed a nested substitution and is more costly to compute.

[0040] As both interpretations have their uses, the second interpretation is denoted using double parentheses to distinguishably designate it differently from the first interpretation. So, referring to a sheet 470 in an illustration 460 of FIG. 4C, if the second interpretation is desired, the Return cell 475 of sheet *square_add* is written with double parentheses.

[0041] It has now been shown how a spreadsheet in accordance with the present embodiments can act analogously to functions in conventional programming languages. Similarly, spreadsheets can be partially bounded with parameters, like conventional functions. A sheet name, together with a possibly empty set of substitutions of labels to values, is called a thunk in accordance with the present embodiments. A cell can be assigned a thunk as its value. In accordance with a further aspect of the present embodiments, thunks can be passed in as function arguments, enabling the use of callbacks and/or higher-order functions. Referring to FIGs. 5A and 5B, illustrations 500, 520 depict a sheet 505 named *add1*, and a sheet 525 exemplifying a rule for callbacks in accordance with the present embodiments. The sheet 525 passes the callback *add1* to the call to *Map*, a built-in function.

[0042] Further to the example above, we note that it's impractical to have to define an *addN* for every *N*. Referring to FIGs. 5C and 5D, illustrations 540, 560 depict sheets 545, 565 exemplifying a rule for partially bound thunk callbacks in accordance with the present embodiments. Where the sheet 545 is named *add*, an expression *add{1}* 570 represents a partial binding of the sheet 545 *add*, producing a thunk that's functionally equivalent to *add1* 505. A thunk can also take more substitutions,

creating further thunks. As an example, the following are all equivalent: (a) *add*(1,2), (b) *#add*{1,2}.*Return*, (c) *add*{1,2}(), (d) *add*{1}(2), and (e) *add*{1}{2}().

[0043] Similar to the distinction of simple substitution versus nested substitution () and (()), discussed hereinabove, thunk substitutions can be either simple or nested as denoted by {} and {{}} respectively.

[0044] Since range substitutions are allowed, no matter how far blocks of cells are placed apart, all possible spill conflicts cannot be avoided. Referring to the spreadsheet 360 (FIG.3C), if someone made more than four donations, the *Return* cell would get a spill conflict. Thus, a rule for spill barriers is defined in accordance with the present embodiments. Referring to FIGs. 6A and 6B, illustrations 600, 650 depict operation of the spill barrier rule in accordance with the present embodiments. The illustration 600 depicts the *tax* sheet 640 with a spill barrier 620 defined to establish an elastic limit to a spill area in accordance with the present embodiments. Spills cannot cross barriers; instead, the barrier creates elastic spill areas as needed to contain the spill values. Cells in an elastic spill area cannot have user defined labels or expressions. In a substituted *tax* sheet 660 in the illustration 650, the spill barrier 620 enables automatic creation of cells in a row 670 in an elastic spill area in order to accommodate an additional donation value that would otherwise cause a spill conflict.

[0045] Referring to FIG.7, an illustration 700 depicts a spreadsheet 710. Suppose a user wishes to express a range denoting all the content of the sheet. Noting that *Xbilling_rate* and *Xhours* are both substitutable and hence can have any height, the user can use *label position expressions* to precisely describe the logic required. Label position expressions are expressions that operate on the coordinates of labels, instead of a value referred to by the labels. Examples are: (a) *Xbilling_rate* ^ 1 refers to the coordinate of the cell one row above the label *Xbilling_rate*; (b) *Z(Xbilling_rate)*

refers to the bottom-right corner of the spill area of the cell *Xbilling_rate*, i.e. the cell with value 60; and (c) *BottomRight(Z(Xbilling_rate), last_remark)* refers to the top-left-most cell that's to the bottom, or to the right, or to the bottom-right of both *Z(Xbilling_rate)* and *last_remark*, i.e. the empty cell below *last_remark*. With these definitions, the user can elegantly express the concept "all the content on the sheet" with the expression *Xbilling_rate*¹ : *BottomRight(Z(Xbilling_rate), last_remark)*, regardless of what values *Xbilling_rate* and *Xhours* are substituted with.

[0046] The usual way to style a spreadsheet is to manually select an area, then select a style to apply. This breaks down in the face of spills and substitutions. Thus, in accordance with the present embodiments a user can define multiple style regions for each sheet, ranked in priority. Unlike spill areas, style regions can overlap. Cells in overlapping regions accumulate the style specifiers from all covering regions. If the styles specified are conflicting, then that with a higher priority is effective. Each style specifier is itself a normal expression that evaluates to a 2D table, and is tiled to fill the whole region. For example, we specify for sheet 835 in FIG.8B the following style regions, ranked from high priority to low: (a) from *billing_rate*¹ to *amount*¹, style "*font-weight*" & "*bold*", (b) from *amount* to *Z(amount)*, style "*font-size*" & *Where(#amount > 500, "large", "normal")*, and (c) from *billing_rate*¹ to *Z(amount)*, style "*background-color*" & ("*lightgray*" \ \ "*white*") \ \ "*font-size*" & "*small*". The style definition evaluates to a table 800 in FIG. 8A, where a first row 805 and a second row 810 define *background-color* to alternate between *lightgray* and *white*, and a third row 815 defines font size as *small*.

[0047] Referring to FIG. 8B, an illustration 830 depicts a sheet 835 incorporating the style definitions (a) to (c) as defined previously. Note that the font-size specifier

of the second rule (b) overrides the font color specifier of the third row 815 of the table 800.

[0048] Alternatively, the user can also leave the *to* label empty, and specify "*span*" & 3 & 2 in the style to apply the style to a region of three rows and two columns starting from the *from* label. Once defined, the style of a region is accessible programmatically. For example, *Style(a, b)* retrieves the styles of the range from cells *a* to *b*. Referring to FIG. 8C, a table 860 depicts how *Style(hours, Z(amount))* would evaluate for the sheet 835. Note that in the table 860, the *Style()* call automatically generates the row 865 "*span*" & 3 & 2. This allows one to easily copy one region's styles to another region by assigning *Style(#source)* to the destination cell.

[0049] When a cell becomes very wide, it forces semantically unrelated cells below to be unpleasantly wide. Referring to FIG. 9A, an illustration 900 depicts a table 905 having unpleasantly wide cells below a title cell. Conventional spreadsheets allow cells to be merged to alleviate this problem. However, cell merges violate the strict 2D grid and is difficult to reconcile with other features, such as spills and style regions. Accordingly, a rule for table-within-tables (TWITs) is defined in accordance with the present embodiments to isolate wide rows and tall columns.

[0050] A user can specify a TWIT, where a region is rendered as a mini-table inside a larger table, usually that of a whole sheet. The number of rows, the number of columns, and the total area occupied are preserved, but the individual row heights and column widths can otherwise be freely adapted to the content of the cells. In accordance with the present embodiment, a TWIT from *title* to *title* >> 2 can be defined in the sheet 905. An illustration 910 of FIG. 9B depicts a sheet 915 incorporating the above defined TWIT, thereby advantageously providing a better visual appearance.

[0051] TWITs can be nested, but they cannot intersect. Intersecting TWIT definitions are ignored.

[0052] A row of sheets is hard to organize: conventional spreadsheets have no hierarchy as all sheets are arranged in a row. In order to overcome this deficiency in conventional spreadsheet programs, spreadsheets are made hierarchical in accordance with another aspect of the present embodiments. Each sheet can have child sheets, much like a folder can contain files and also more folders, except that the sheet acts as both a folder and a file. There are several benefits to making sheets hierarchical in accordance with the present embodiments other than a hierarchy being more logical. For example, the hierarchy maps easily to URLs when exporting the spreadsheet document as a website. In addition, it is easier to assign meta-data, such as read/write permissions, to a group of sheets rather than assigning meta-data individually to sheets one by one. Also, the hierarchy forms the basis of defining classes, as discussed hereinbelow.

[0053] One can presently save a spreadsheet as HTML pages to be displayed as a static website. But the same cannot be said of dynamic websites whose contents are generated on the fly by querying data sources and doing computation, even though the logic and layout have been defined in the spreadsheet. Currently such a website must be reimplemented again from scratch, with the spreadsheet as a reference, but no actual code reuse is possible.

[0054] In accordance with an additional aspect of the present embodiments, a web page can also be a substitution to enable using spreadsheets as dynamic websites. When a URL is requested, the server may walk the path from the *Root* sheet in a spreadsheet document following the request URL. For example, in response to a request:

http://doc.abc.com/demo/ticket_manager/ticket?id=123

the server might look in the document root and find spreadsheet file *demo/ticket_manager*. It opens the file and locates the sheet at *Root.ticket* in accordance with the present embodiment. It does the substitution for *Xid* to *123* and renders the sheet *Root.ticket* into HTML. The substitution process is fundamentally the same as sheet substitutions in function calls, including the *X* prefix, except the whole substituted sheet is rendered as the result.

[0055] Most conventional mainstream programming languages support classes in some form, which to a first approximation are a bundle of data and associated functions. For example, the call *a.area()* executes different code depending on what class the value *a* belongs to. In accordance with a further aspect of the present embodiments, a dynamically specified spreadsheet reference can be substituted into the spreadsheet in response to cell values of a cell in the spreadsheet, where the dynamically specified spreadsheet reference includes a dynamically specified directory reference and a spreadsheet name of another spreadsheet. Thus, the behavior of classes is emulated by dynamic directory references that is understandable in the context of a spreadsheet.

[0056] If the first row of a table is "*Classs*" & "*reference-to-directory*", then the table is considered to be of the class of the said directory. The keyword *Classs* is intentionally misspelled to minimize the chance of a random table having the correct keyword, although the choice of the actual keyword is not essential in the context of this invention. The special syntax *a->area()* is then translated to the call *directory.area(a)*. The expression *directory.area* is a dynamic sheet reference, comprising of a dynamic directory reference and a sheet name. As an example, FIG. 10A and 10B depict illustrations 1000, 1050 of how the *rectangle* class works. A

create sheet 1010 in the illustration 1000 acts as a function that creates a new table of the *rectangle* class given the width and height, while an area sheet 1060 in the illustration 1050 computes the area of a rectangle. These definitions allow code such as the following to be well defined: *rectangle.create(width=3, height=3)->area()*. Note how *circle.create(radius=3)->area()* would call the sheet *circle.area* instead. The syntax *#X0["width"]* means to search for the string "width" in the first column of the given table *#X0*, and to take the value to right of it as the value of the expression.

[0057] Importing third-party spreadsheets would allow one to share functions and classes with other users. Therefore, our invention allows a spreadsheet document to be imported as a node in a sheet tree. Referring to FIG. 11, an illustration 1100 depicts a spreadsheet importing the document described by the sheets 1010, 1060 at a certain node. The directory structure of the imported document is preserved.

[0058] As described above, the complete source code of the third-party spreadsheet document is imported and is then executed on the same computer as the rest of the document. Sometimes it is instead desired that the imported spreadsheet remains on the third-party computer and execute on that computer and the user is only be informed of the results. There could be a few reasons for this. For example, the third-party sheet is proprietary and the author does not wish to distribute the source code. Or, the execution is resource intensive or requires specialized hardware and is best executed by the third party.

[0059] In accordance with the present embodiment, we allow multiple sheets residing on different computers to form one logical distributed document. The major challenge in this type of system is cycle detection, where some cell *a* residing on computer *A* depends on some cell *b* residing on computer *B*, which depends on some cell *c* residing on computer *C*, which in turn depends on cell *a*. In a fully general

system where any cell can reference any possibly remote cell like in the present example, dependency detection is a hard problem. The present invention therefore proposes a limited star-shaped form that is easier to work with while remaining useful.

[0060] Only one computer is considered the hub for the document, and can reference any remote sheets as it wishes. Other computers, called libraries, cannot reference back to the hub or any other computers, except via remote thinks passed in by the hub as parameters. For example, someone develops a ticket management system as a collection of sheets that they don't want to release the source code for, and hence the ticket management system runs on the developers' computer as a library. A user can then incorporate this library as remote sheet. However, the library does not know by itself which users can log in or can have access to which tickets, this is essentially a policy that the user has to set. So, as configurations the user passes an *authenticated* think and an *authorized* think, to define rules for password validation and access control, respectively. These thinks stay on the user's computer and only the think IDs are passed to the computer running the library. When the library needs to execute the think and realizes that the think ID is not available locally, the third-party system will send a subscribe request back to a user's computer a.k.a. the hub.

[0061] Dependency loop detection in accordance with the present embodiments is made simpler by the fact that a star-shaped dependency is maintained. Two libraries referenced by the hub cannot talk to each other directly; they can only talk to the hub of the document. When libraries publish cell value updates to the hub, they also list all the values on the hub used to compute this value. This way, whenever there's a dependency loop, the hub can easily detect it and set the relevant cells to have a dependency loop error. Note that a library is free to act as the hub of further libraries,

as long as those further libraries are invisible to its hub. So, the ticket management library can itself import other libraries, but it'll still appear as a single monolith to the user. The loop detection algorithm in accordance with the present embodiments is agnostic to the internal implementation of the ticket management library.

[0062] In accordance with the present embodiments, collaborative sheets are provided. In the simplest case, collaborative sheets are merely multiple UIs connected to the same backend, each sending commands like set a cell's code and set a horizontal barrier, and receiving cell updates from the backend.

[0063] Things get more interesting when access control is enabled. Conventional collaborative spreadsheet systems have very coarse access control. For example, Google sheets have several permission bits per user for the whole document, such as whether the user can edit the document, whether the user can read the document, or whether the user can comment on the document. Such a simplistic model, however, is often not adequate.

[0064] For example, a company may wish to put all its administrative and operational data into one huge document. The HR data live in one subtree, including sensitive information such as everyone's salary, which must be visible only to HR staff. Customer support data live in another subtree. In this subtree, the customer support team plans to build a ticket management system. They wish to use HR's record of who's in which department, which is not sensitive information, to assign roles in their ticket management system. As they should not have direct access to the HR data, someone in HR needs to extract the departmental information and specifically grant permission so that the customer support team can read that data.

[0065] Clearly the coarse access controls in conventional systems are inadequate in such a scenario. However, a naive approach that simply defines who can read and

write to each sheet is easily circumvented. For example, suppose sheet *secret* is only accessible to the HR, and sheet *public* is readable and writable by everyone. Any user can easily set a cell *X* in sheet *public* to have code *secret.everyones_salary* and read the cell *X* in sheet *public*.

[0066] Operation in accordance with the present embodiments defines per-sheet owner and readers. Per-sheet owners and readers include concrete users which are actual users and abstract users which are common denominators of some set of users. Together concrete users and abstract users are the users. Each user can act-as some set of other users. For example, a hypothetical company has the following users: (a) anyone: any user in the system (this is always present and anyone can act-as anyone), (b) hr-person (e.g., someone who works in HR and can act-as anyone), (c) customer-support-person (e.g., someone who works in customer support), (d) manager (e.g., any manager), (e) hr-manager (e.g., can act-as hr-person and manager), (f) chess-club-person, (g) Tyrion (e.g. can act-as hr-manager and chess-club-person), and (h) Samwell (can act-as customer-support-person and chess-club-person).

[0067] Act-as relationships are transitive, so, for example, Tyrion can act-as a hr-person, too.

[0068] In accordance with the present embodiments, every sheet has one user as its owner, which must be assigned statically, i.e. not computed. All expressions in the sheet are always executed with the permission of the owner, regardless of who is looking at the sheet. This way, everyone gets the same value, including permission denied errors.

[0069] In accordance with the present embodiments, every sheet can have zero or more users as its readers. This list can be computed, and can be dependent on substitutable cell values so that, for example, a sheet showing personal information

has a reader list that depends on whose information is being shown. This is useful for letting people read their own information, but not other people's information. A sheet can be read by a user if (s)he can act-as the owner, or at least one of the readers.

[0070] In accordance with the present embodiments, a user can edit the sheet, including the list of readers, if (s)he can act-as the owner. To change the owner of a sheet, the logged in user must be able to act-as both the old owner and the new owner. Since one can only edit a sheet if (s)he can act-as the owner and the sheet is evaluated with the permission of the owner, one cannot gain privilege escalation by writing an expression in a cell.

[0071] If fine-grained permission is not required, every sheet's owner is set to anyone. This is also the default.

[0072] Multiple owners could be allowed for each sheet in accordance with the present embodiments. The contents in the sheet are then executed with the common denominator of all the owners. That is, in order to be able to read a cell from another sheet, all owners must be able to. Subsequently, a user can edit the sheet, including adding new owners, if they can act-as any of the owners. Essentially there is an unnamed single owner which is the common denominator of all the owners.

[0073] Thus, it can be seen that the present embodiments provide a robust user-facing spreadsheet programming language which enhances utilization and operation of spreadsheets. In accordance with present embodiments, spills, sheet substitution as function calls, sheet substitution (thunks) as cell values and callback parameters, barriers, programmatic style definitions, TWITs, web pages using sheet substitution, classes, sheet substitution (thunks) as remote callbacks, and fine-grained permissions in collaborative documents has been defined to provide an easy, flexible and versatile backward-compatible spreadsheet-friendly solution.

[0074] While exemplary embodiments have been presented in the foregoing detailed description of the present embodiments, it should be appreciated that a vast number of variations exist. It should further be appreciated that the exemplary embodiments are only examples, and are not intended to limit the scope, applicability, operation, or configuration of the invention in any way. Rather, the foregoing detailed description will provide those skilled in the art with a convenient road map for implementing exemplary embodiments of the invention, it being understood that various changes may be made in the function and arrangement of steps and method of operation described in the exemplary embodiments without departing from the scope of the invention as set forth in the appended claims.

CLAIMS

What is claimed is:

1. A method for spreadsheet programming comprising:
assigning labels to cells of a spreadsheet, each of the labels assigned to one of a plurality of adjoining variable value cells for data corresponding to a plurality of values of a variable, wherein the plurality of adjoining variable value cells is user defined by a pair of labels indicating a start and an end of a cell range of the plurality of adjoining variable value cells; and
defining a spill area into which data can flow from a cell having an expression defined therein in response to the expression including a variable range defined by the plurality of values of the variable, the spill area automatically sized to a plurality of expression solutions corresponding to the plurality of values of the variable.
2. The method in accordance with Claim 1 wherein defining a spill area comprises defining a non-shrinking spill area into which data can flow from a cell having an expression defined therein before subsequent user input of data into the plurality of adjoining variable value cells.
3. The method in accordance with Claim 1 further comprising defining a barrier across one or more cells to establish an elastic limit to the spill area, and wherein defining the spill area comprises defining cells in an elastic spill area without disturbing a rest of the spreadsheet adjoining the barrier when data entered into the plurality of adjoining variable value cells corresponds to a plurality of expression solutions greater than a number of the cells within the spill area.

4. The method in accordance with Claim 3 wherein defining a barrier across one more cells comprises defining a barrier across less than all the cells of a row or a column of the spreadsheet.

5. The method in accordance with any of the preceding claims further comprising defining a table-within-table (TWIT) circumscribing mini-tables inside a larger table of the spreadsheet, wherein a number of rows, a number of columns and a total area occupied by the mini-table are defined in response to a user TWIT input.

6. The method in accordance with any of the preceding claims comprising displaying cells of the spreadsheet in accordance with user programmable two-dimensional style definitions.

7. The method in accordance with Claim 6 wherein the user programmable two-dimensional style definitions comprise one or more of data font style, data font color, and cell background color.

8. A method for spreadsheet programming comprising substituting another item into a spreadsheet in response to cell values of a cell in the spreadsheet comprising a function call and/or a remote callback.

9. The method of Claim 8 wherein substituting another item into the spreadsheet comprises substituting a dynamically specified spreadsheet reference into the spreadsheet in response to cell values of a cell in the spreadsheet, wherein the

dynamically specified spreadsheet reference comprises a dynamically specified directory reference and a spreadsheet name of another spreadsheet.

10. The method in accordance with any of the preceding claims wherein access to read and/or write code in cells of the spreadsheet is controlled in accordance with fine-grained access control permissions.

11. A method for spreadsheet programming comprising:
substituting dynamic webpage request query parameters into a spreadsheet in response to a dynamic webpage request, and
rendering the spreadsheet as a webpage in response thereto.

12. A spreadsheet comprising:
a plurality of cells arranged in a two-dimensional array, one or more of the plurality of cells comprising cell values for substituting another item into the spreadsheet, wherein the cell values comprise a function call and/or a remote callback.

13. The spreadsheet in accordance with Claim 12 wherein a dynamically specified spreadsheet reference is substituted into the spreadsheet in response to the cell values, and wherein the dynamically specified spreadsheet reference comprises a dynamically specified directory reference and a spreadsheet name.

14. The spreadsheet in accordance with Claim 13 wherein access to read and/or write code in cells of the spreadsheet is controlled in accordance with fine-grained access control permissions.

15. A spreadsheet comprising:

a plurality of cells arranged in a two-dimensional array, one or more of the plurality of cells comprising cell values for substituting dynamic webpage request parameters in response to a dynamic webpage request, and wherein the spreadsheet is rendered into a webpage in response to the dynamic webpage request parameters.

16. A spreadsheet comprising:

a plurality of cells arranged in a two-dimensional array, wherein the plurality of cells comprise a plurality of adjoining variable value cells for data corresponding to a plurality of values of a variable, and wherein the plurality of adjoining variable value cells are user defined by a pair of cell values indicating a start and an end of a cell range of the plurality of adjoining variable value cells, and wherein the plurality of cells further comprises a spill area into which data can flow from a cell having an expression defined therein in response to the expression including a variable range defined by the plurality of values of the variable, the spill area automatically sized to a plurality of expression solutions corresponding to the plurality of values of the variable.

17. The spreadsheet in accordance with Claim 16 wherein the spill area comprises a non-shrinking spill area into which data can flow from a cell having an expression defined therein before subsequent user input of data into the plurality of adjoining variable value cells.

18. The spreadsheet in accordance with Claim 16 wherein a barrier is defined across one or more cells to establish an elastic limit to the spill area, and wherein the spill area comprises cells defined to be in an elastic spill area without disturbing a rest of the spreadsheet adjoining the barrier when data entered into the plurality of adjoining variable value cells corresponds to a plurality of expression solutions greater than a number of the cells within the spill area.

19. The spreadsheet in accordance with Claim 18 wherein the barrier is defined across less than all the cells of a row or all the cells of a column of the spreadsheet

20. The spreadsheet in accordance with any of Claims 12 to 19 wherein the plurality of cells comprises a table-within-table (TWIT) circumscribing mini-tables inside a larger table of the spreadsheet, and wherein a number of rows, a number of columns and a total area occupied by the mini-table are defined in response to a user TWIT input.

21. The spreadsheet in accordance with any of Claims 12 to 20 wherein one or more of the plurality of cells include one or more user programmable two-dimensional style definitions for displaying the cell.

22. The spreadsheet in accordance with Claim 21, wherein the user programmable two-dimensional style definitions comprise one or more of data font style, data font color, and cell background color.