# Analyzing Large Spreadsheet Programs

Markus Clermont
Institut für Informatik-Systeme
Universität Klagenfurt
Universitätsstrasse 65–67
A-9020 Klagenfurt
Austria
mark@isys.uni-klu.ac.at

## Abstract

*Although the results of spreadsheet programs are the base for very important decisions and are subject to many changes, they are only poorly documented. In this paper we introduce an approach that extents previous work in the area of spreadsheet visualization. It enables the maintainer to decompose large spreadsheets into self contained parts, that are similar to modules in conventional software. Of course there are important differences, as these modules are only recognized during analysis, and users are not forced to build modular spreadsheets. This is very important, as we aim not to change the spreadsheet users. It has turned out, that attempts to change the users are usually doomed to fail.*

*This approach can be considered orthogonal to semantic classes, that have been introduced in prior work. The generated abstraction is not build upon the formula-contents of the spreadsheet, but on properties of the data flow graph. Therefore, spreadsheets with irregular formulas can be solely analyzed with the new approach. On the other hand, large spreadsheets can be decomposed into data modules at first, that are smaller and easier to understand. Each of the data modules can than be analyszed with a formula based approach.*

## 1. Introduction

The widespread use of the personal computer is inseparably intertwined with the success of spreadsheets as end-user programming language. Up-to-date spreadsheets are of highest importance in most business organizations and fullfil various tasks- from simple data collection up to the simulation of very complex models. Studies [25] have shown, that spreadsheet systems are more often used by the middle and higher management of organizations than word processors.

As various studies (see e.g. [18]) have shown, software quality of the spreadsheets does not correspond to their importance. This problem might arise because spreadsheet programmers are not IT-experts, but domain experts. [16] states that training of spreadsheet programmers is only a partial solution. Well trained spreadsheet programmers tend to make the same number of errors in their spreadsheets because they start to use more complicated and thus more error-prone formulas.

Another reason for the high number of errors is the high complexity of spreadsheets. For the IT-professional it is obvious that spreadsheets are data-flow programs. However, the spreadsheet writers see them as computer support for tools that are on every desktop: *pencil, paper and a pocket-calculator*. For convenience they can redirect the results of the different pocket-calculators to serve as the input for another calculator.

As spreadsheets are a very powerful end-user programming tool, their field of applications is also wide spread (see [3, 11, 14]). Of course, a specific approach for the analysis of spreadsheets cannot cover all possible applications.

The complexity of the comprehension of a large spreadsheet can be reduced, as each of the blocks can be analyzed and understood by any common spreadsheet visualization technique (see e.g. [15, 7, 1, 2, 21]), and the cooperation of the blocks becomes subject to a second phase in the comprehension process.

The spreadsheet visualization approaches known so far deal with analysis of spreadsheets on the formula-level[1]. Therefore, a spreadsheet is considered a set of cells, with each cell possibly containing a formula expression. In our previous work [15], we consider spreadsheets the result of a copy, paste and modify process, and grouped cells with *sim-*

1 Indeed, only one visualization approach that was discovered in literature deals with dataflow based visualization [6]. This approach is discussed in Section 5.

*ilar* contents into so called logical areas. A further abstraction step introduced semantic classes, that contain blocks of similar cells that are repeatedly used throughout the spreadsheet. As it can be seen by the short summary of the approach, it focuses on formulas. In field experiments [8] it turned out that this approach is efficient for spreadsheets with regularly occuring formulas.

However, there is also a different kind of spreadsheet programs, that is usually neglected by conventional spreadsheet visualization approaches. These spreadsheets do not have a regular formula structure and hence, it is difficult to find a helpful criterion for grouping cells into units. The here introduced approach considers a spreadsheet program as a special kind of dataflow program. Cells are grouped into abstract units, so called data modules, by examining the data dependency graph ($DDG$) of the spreadsheet program. As this technique does not rely on any properties of formulas, it is also efficient for spreadsheet programs with irregular formulas.

Usually, these spreadsheet programs are smaller than spreadsheets with regular formulas, because they cannot be created by copying and pasting. Nevertheless, data modules also play a role in the analysis of large spreadsheets, because they can partition a large spreadsheet program into different parts that are only loosely interconnected. Consecutively, each of these parts can be comprehended on its own. As a data module has also a well defined interface to the rest of the spreadsheet program, the knowledge of the functionality of individual data modules can than be reassembled to comprehend how the whole spreadsheet works.

In this paper we will introduce and define data modules. For the definition of the spreadsheet related vocabulary used in this paper we refer to [15, 7]. Only definitions that are crucial for the further understanding of this paper are given in the next section. A toolkit for the decomposition of spreadsheets and the visualization of data modules is introduced in Section 3 by means of an example. Finally, the possible applications of data modules are outlined in a broader context. In the concluding section we will discuss the position of our work among other spreadsheet visualization approaches.

## 2. Data Modules

Subsequently, an abstraction technique that operates on the data flow graph of a spreadsheet program, i.e. the $DDG$, is introduced. The $DDG$ is defined as follows:

**Definition 1: Data Definition Graph ($DDG$)**
The *data dependency graph* ($DDG$) of a spreadsheet is a directed acyclic graph $DDG = (V, E)$, where each cell $c$ in the spreadsheet is represented by a vertex $v \in V$, if the cell is not computationally dead, i.e. not referenced by any other cell and not referencing any other cell, and empty. There is an edge $(v_1, v_2) \in E$, if the formula in the cell corresponding to $v_2$ references the cell corresponding to $v_1$.

Spreadsheet programs have some basic characteristics of data flow programs and of graph-reduction programs, too [7]. Thus, the $DDG$ of a spreadsheet program has an important role for its execution. As stated by Definition 1, in the $DDG$ each cell of the spreadsheet program is represented by a node, and there is an edge from node $n_1$ to node $n_2$, if the cell represented by $n_2$ references the cell represented by $n_1$. As the $DDG$ is a directed, acyclic graph, there are some nodes, that are not sources of further edges, i.e. sink nodes.

To grasp the idea, one can assume that a data module is a set of cells that has a distinguished result cell, that is transitively dependent on all cells in the data module. Cells that are outside the data module may only reference its result cell. Broadly speaking, a data module is a subgraph of the $DDG$, that has only a single sink node (see Definition 2)- namely its result cell. The result cell of such a data module is either a sink node of the $DDG$, i.e. a result cell of the spreadsheet program, or a node that is connected to more than one data module. Obviously, this definition is recursive, but because of the hierarchical organisation of a $DDG$ and its finiteness, this is not a problem.

The construction of data modules will start assuming the $DDG$ sink nodes to be data modules and adds all cells that only contribute to the specific sink node. A cell that transitively contributes to more than one sink node is assumed to be the starting point of a new data module and will be treated in the same way.

However, before the $DDG$ can be partitioned into such data modules, the result cells have to be identified. Obviously, not all sink nodes of the $DDG$ have the semantics of a result of the spreadsheet program, e.g. check-sums.

In contrast to conventional programming where intermediate results are not displayed and each subroutine has a well defined result, in a spreadsheet each intermediate result is visible to the user and all the other formulas. Sometimes, calculations are deliberately formulated in a more complicated way in order to obtain some desired intermediate results. Obviously, most of the cells of a spreadsheet program can be either auxiliary, intermediate or result cells. For sure, cells that are not further referenced by other cells can be considered result cells, because we know that users place them on the spreadsheet, because they want to **see** their contents. If they would not like to see the displayed value, they had not introduced this cell.

Therefore, it is legitimate to start with results, i.e. cells, that are not referenced by other cells, and search those cells, that influence a specific result. As a matter of fact, it is often the case, that the sink nodes in the $DDG$ are not the real results, but check-sums. In this case, the check-sums have to be removed manually, and the remaining $DDG$ is then analyzed.

Subsequently, data modules and their required properties will be formally defined. The identification of result cells and an algorithm for partitioning the $DDG$ into data modules are discussed.

## 2.1. Formal Definition

As stated above, a data module is a set of cells in the spreadsheet program that contribute to a specific result or intermediate result. In the following definition, a data module is defined as a triple of its member nodes ($V_d$), the edges ($E_d$) and a result node ($n$). As a data module is defined as a part of the $DDG$, the nodes represent cells of a spreadsheet program.

**Definition 2: Data Module**
Let $(V, E)$ denote the sets of vertices and edges of a $DDG$ of a given spreadsheet program. A data module $d$ is a triple $(V_d, E_d, n)$, with $V_d \subseteq V$, $E_d \subseteq E$ and $n \in V_d$ that fulfills the following requirements:

1. $n \notin domain(E_d)$
2. $V_d \cap \{n\} = domain(E_d)$
3. $domain(E \setminus E_d) \cap (V_d \setminus \{n\}) = \emptyset$
4. $n \in range(E_d) \vee \{n\} = V_d$
5. $range(E_d) \subset V_d$

The specified properties of a data module guarantee that

1. the result cell of the data module is not referenced by any other cell in the data module,
2. the data module is connected,
3. all edges with a source inside a data module are also part of the data module, only edges having their source in the result node are excluded,
4. the result of the data module is target of an edge in the data module, or the data module consists of a single cell, and
5. a cell in a given data module is only referenced by cells in the same data module.

Obviously, there are arbitrary subgraphs of the $DDG$ that fulfill these requirements, e.g. each single cell can be considered a data module. However, in order to introduce an abstraction step, maximal data modules have to be identified.

**Definition 3: Maximal Data Module**
A data module $d = (V_s, E_s, n)$ in a $DDG = (V, E)$ is called a maximal data module, if $\nexists v \in V \cap V_s$ that can be added to $d$ without violating any of the conditions that are required for a data module.

Hence, a data module is considered maximal if no other cell of the spreadsheet program (or no other node of the corresponding $DDG$) can be added.

Nevertheless, so far only the properties of individual data modules have been described. Additionally, there are some requirements that a valid partitioning of a given $DDG$ into data modules has to fulfill. On the one hand, it has to be guaranteed that the union of all data modules is the original $DDG$, i.e. all nodes and edges of the $DDG$ have to take part in one data module. On the other hand, no node of the $DDG$ is allowed to be in more than one data module. As the $DDG$ on its own, with cells as singleton data modules, already fulfills these properties, the data modules have to be maximal.

For each $DDG$ there is one partitioning into data modules that fulfills the above stated requirements. This kind of partitioning is called a *valid partitioning*.

**Definition 4: Valid partitioning**
The set $D_{Mod}$ of data modules over a $DDG = (V, E)$ of a given spreadsheet program is a valid partitioning if it fulfills the following requirements:

1. $\bigcup\{V_s | \exists n, E_s \bullet (V_s, E_s, n) \in D_{Mod}\} = V$
2. $\bigcup\{E_s | \exists n, V_s \bullet (V_s, E_s, n) \in D_{Mod}\} \cup \cup\{(n, v_1) | (n, v_1) \in \vee \exists (V_s, E_s, n) \in D_{Mod}\} = E$
3. $\forall (V_1, E_1, n_1) \in D_{Mod}, (V_2, E_2, n_2) \in D_{Mod} | V_1 \neq V_2 \bullet V_1 \cap V_2 = \emptyset$
4. $\forall d \in D_{Mod} | d \text{ is maximal}$

The first and the second property ensure that all nodes and edges of the $DDG$ are assigned to at least one data module in $D_{Mod}$. Only edges leaving the result cell of a data module are not considered. The third requirement prevents a node from being assigned to two data modules. The last requirement ensures that all the data modules in a valid partitioning have to be maximal data modules.

The spatial position of member cells in a data module is not considered at all. Therefore, the members of a data module might be located in different parts of the spreadsheet UI. Thus, the concept of spatial nearness is not part of these definitions. However, as it follows from the definitions above, the concept of computational nearness is the key-concept of data modules. Matching spatial and computational nearness of cells turned out to be helpful for auditing spreadsheet programs (see [7]).

## 2.2. Identifying Data Modules

The identification of the data modules in a given $DDG$ starts with the

removal of unnecessary sink nodes, to eliminate check-sums and other calculations that do not have the semantics of a result of the spreadsheet program. This step is necessary, because very often all sections of a spreadsheet are connected by some calculations that either yield a final sum or a check-sum. However, in this case each cell of the spreadsheet program is transitively referenced by the final sum and there will be only one data module.

Thus, the sink nodes of the $DDG$ have to be checked by the users who decide if a sink node should be removed or not. The users have to prune the $DDG$ until all the sink nodes are results of the spreadsheet program. All further consideration in this section concern the pruned $DDG$.

Of course it can be argued that each cell in the spreadsheet program is visible and therefore a result of the spreadsheet program. However, it is important to distinguish between intermediate results that might be introduced only in order to get insight into the calculation or to rewrite a formula in a simpler way, and the final result, that is placed on the spreadsheet because the spreadsheet users really want to see it. Obviously, all sink nodes of the $DDG$, i.e. cells that are not referenced by any other cell any more, are placed on the spreadsheet because the spreadsheet users want to have them there.

The algorithm for the identification of data modules is very straightforward. It operates on the pruned $DDG$ and starts at the sink nodes. Each sink node is considered a data module. Nodes that are the source of edges that target only to one data module are merged with the specific data module. A node that is a source of edges into more than one data module is considered a data module on its own. These steps are repeated until all nodes in the $DDG$ are assigned to a data module (see Figure 1).

The algorithm terminates, as soon as all nodes are assigned to a data module. The first *for* checks all of the nodes that are not yet assigned to a data module, whether they are not referenced by any other unassigned node. For the selected nodes the algorithm checks, whether one data module depends on the cell. If the cell has exactly one depending data module, the cell is added to it. In any other case, i.e. there are either more than one or none dependents, the cell will become a data module on its own. Finally, the cell is removed from the set of unassigned nodes.

The partitioning algorithm does not add edges to data modules, however they can be added in a further step. All edges between two nodes in the same data module are added to this data module. The result cell of a data module is its only sink node.

```
**Algorithm**PartitionDDG: (pruned $DDG$ d)
 **return** *Set of Data Modules*
 **declare**
   $nodeset\ V,\ DM,\ JOIN$
   $node\ v,\ v',\ current_node$
   $edgeset\ E$
   $edge\ e$
   $\mathbb{P}\ node\ identified\_dm\_set$
   *Integer* $amount\_of\_dependents$
   *Boolean* $found$ = true
 **begin**
   $identified\_dm\_set = \emptyset$
   $(V, E) = d$
   **while** ($found$)
     $found$ = false
     **for** $v \in V$
       **if** $\nexists v' \in V \bullet (v, v') \in E$
         $current\_node = v$
         $found$=true
         **break**
       **end if**
     **end for**
     $amount\_of\_dependents = 0$
     $\forall DM \in identified\_dm\_set$
       **if** $\exists (current\_node, v) \in E \bullet v \in DM$
          $\vee JOIN \neq DM$
         $JOIN = DM$
         $amount\_of\_dependents =$
          $amount\_of\_dependents + 1$
       **end if**
     **end for**
     **if** $amount\_of\_dependents = 1$
       $identified\_dm\_set =$
        $identified\_dm\_set \setminus JOIN$
       $JOIN = JOIN \cup \{current\_node\}$
       $identified\_dm\_set =$
        $identified\_dm\_set \cup JOIN$
     **else**
       $identified\_dm\_set =$
        $identified\_dm\_set \cup \{current\_node\}$
     **end if**
     $V = V \setminus current\_node$
   **end while**

   **return** $identifed\_dm\_set$
 **end**
```

**Figure 1. Partitioning of a pruned $DDG$ into node sets of data modules**

## 3. Example

This section aims to demonstrate the analysis of a small example spreadsheet by means of data modules. Therefore, a prototype is introduced that implements various spreadsheet visualization techniques, i.e. logical areas and semantic classes [15] and data modules.

The prototype is freely available as a plug-in for the *gnumeric* spreadsheet system. *Gnumeric* [12] is an open source spreadsheet system that is part of the gnome project for Linux. The toolkit can be started from the spreadsheet system's menu bar and is tightly integrated. All of the tasks that are subsequently described in course of the example are supported by the prototype.

As soon as the spreadsheet is analyzed, the users can choose two different views: a hierarchical view, containing abstract units at top level, e.g. logical areas, semantic classes or data modules, and their contents at a subordinate level. The contents of a data module are cells.

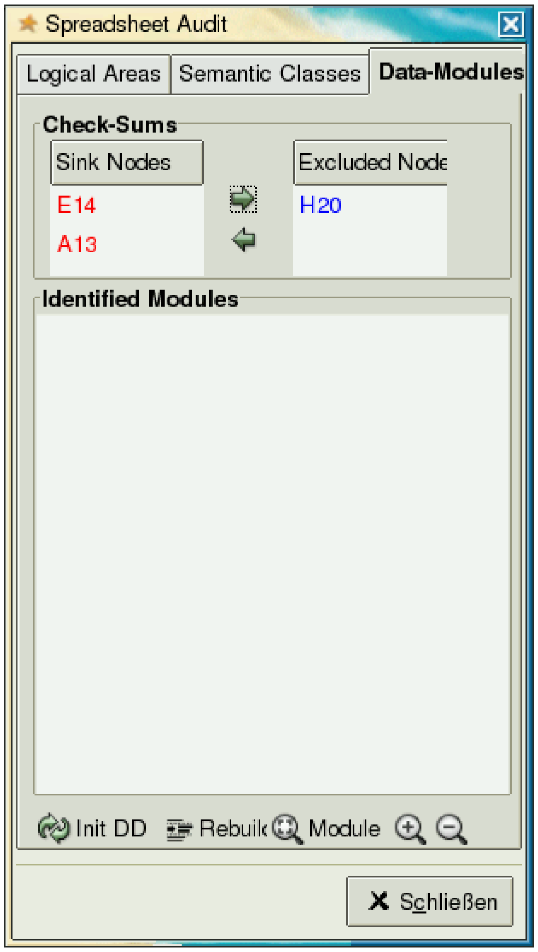
In order to analyze a given spreadsheet program, users will have to remove the $DDG$ sink nodes at first (see the previous section). Therefore, the user interface contains two lists (see Figure 2), one with the identified sink nodes, and a second one with the already removed sinks. Of course, the removal of a sink node from the $DDG$ will turn its predecessors in the $DDG$ into sink nodes. Users can iterate through sink node removal several times, until all nodes they consider check sums, are removed from the $DDG$.

The example spreadsheet (see Figure 3) summarizes the book-keeping of a salesman over three years. There are some constant values, e.g. the fixed cost, the inflation and the earning per unit sold. For each quarter of a year the quantity of sold units is multiplied with the earnings per unit. The difference between fixed costs and earning is the result of the quarter. The result of a year is calculated as the sum of the results of the four quarters. The cash available at the end of the year is calculated by adding the amount of cash available at the end of the last year and the year's result. For the years past 1999, the fixed costs are adjusted with the inflation rate, that is specified in the cell `D1`.

Finally, the three annual results are summed up in order to get an overall result. For the sake of simplicity, our example does not have any check-sums.

Usually, the $DDG$ (see Figure 4) serves as means for comprehending how cells depend on each other. Although the example spreadsheet's $DDG$ is simple, it has only 68 nodes and 86 edges, it is not easy to comprehend at first sight, because the reader is overwhelmed by the complexity of all the nodes and links.

In contrast, the complexity of the $DDG$ is effectively reduced by an $SRG_{DM}$, by collapsing the number of nodes and edges, without loosing any important information. The $SRG_{DM}$ is a directed, acyclic graph. It is an abstraction of



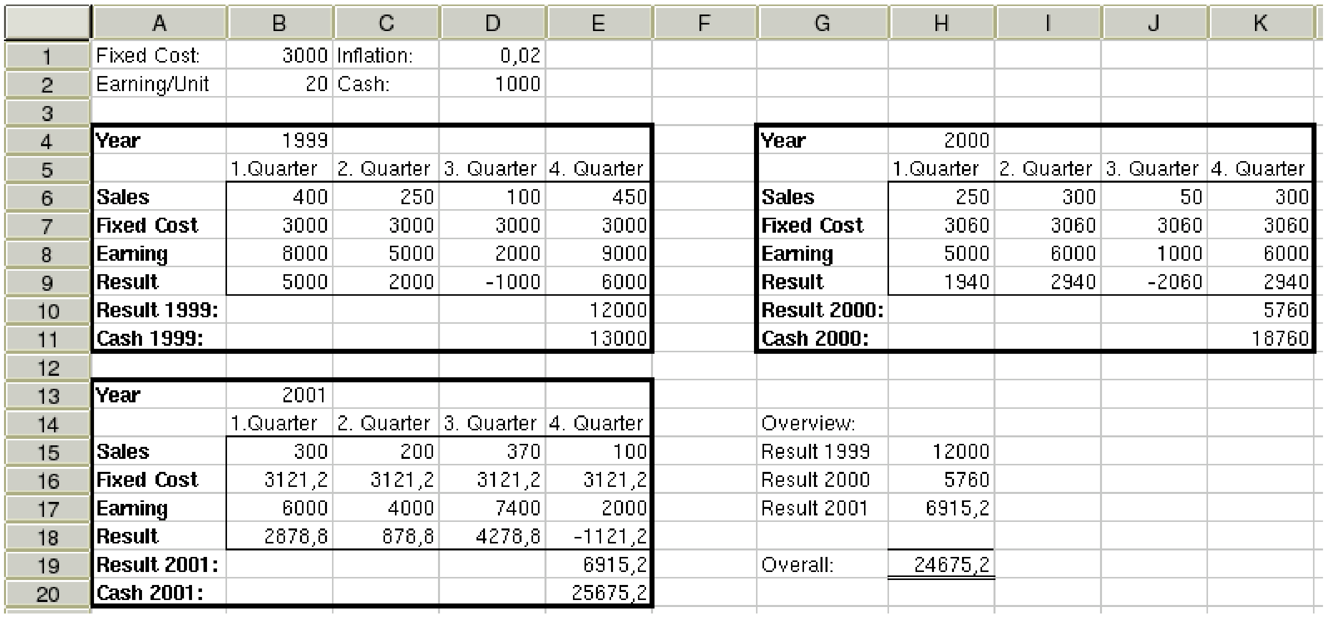
| Sink Nodes | Excluded Node |
|---|---|
| E14 | H20 |
| A13 | |

**Figure 2. Screen-shot of the prototype. The two lists in the check sum frame allow the user to exclude certain sink nodes from the $DDG$ for the further analysis.**

the $DDG$, with each node representing all the cells in one data module. There is an edge between two nodes, if any of the cells in the data module represented by the first node references the result cell of the data module represented by the second node.

In order to construct an $SRG_{DM}$, the spreadsheet has to partitioned into data modules at first. Subsequently, data modules are labelled by the cell adress of their result cells. If we decide, not to remove any sink node from the $DDG$, the following data modules are identified by the partitioning algorithm. Consecutively, each data module is labeled with the cell address of its result cell.

- Singletons: B2, D1, H7
- B7 (B1, B7)



| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Fixed Cost: | 3000 | Inflation: | 0,02 | | | | | | | |
| 2 | Earning/Unit | 20 | Cash: | 1000 | | | | | | | |
| 3 | | | | | | | | | | | |
| 4 | **Year** | 1999 | | | | | **Year** | 2000 | | | |
| 5 | | 1.Quarter | 2. Quarter | 3. Quarter | 4. Quarter | | | 1.Quarter | 2. Quarter | 3. Quarter | 4. Quarter |
| 6 | **Sales** | 400 | 250 | 100 | 450 | | **Sales** | 250 | 300 | 50 | 300 |
| 7 | **Fixed Cost** | 3000 | 3000 | 3000 | 3000 | | **Fixed Cost** | 3060 | 3060 | 3060 | 3060 |
| 8 | **Earning** | 8000 | 5000 | 2000 | 9000 | | **Earning** | 5000 | 6000 | 1000 | 6000 |
| 9 | **Result** | 5000 | 2000 | -1000 | 6000 | | **Result** | 1940 | 2940 | -2060 | 2940 |
| 10 | **Result 1999:** | | | | 12000 | | **Result 2000:** | | | | 5760 |
| 11 | **Cash 1999:** | | | | 13000 | | **Cash 2000:** | | | | 18760 |
| 12 | | | | | | | | | | | |
| 13 | **Year** | 2001 | | | | | | | | | |
| 14 | | 1.Quarter | 2. Quarter | 3. Quarter | 4. Quarter | | Overview: | | | | |
| 15 | **Sales** | 300 | 200 | 370 | 100 | | Result 1999 | 12000 | | | |
| 16 | **Fixed Cost** | 3121,2 | 3121,2 | 3121,2 | 3121,2 | | Result 2000 | 5760 | | | |
| 17 | **Earning** | 6000 | 4000 | 7400 | 2000 | | Result 2001 | 6915,2 | | | |
| 18 | **Result** | 2878,8 | 878,8 | 4278,8 | -1121,2 | | | | | | |
| 19 | **Result 2001:** | | | | 6915,2 | | Overall: | 24675,2 | | | |
| 20 | **Cash 2001:** | | | | 25675,2 | | | | | | |

**Figure 3. Example spreadsheet, value view.**

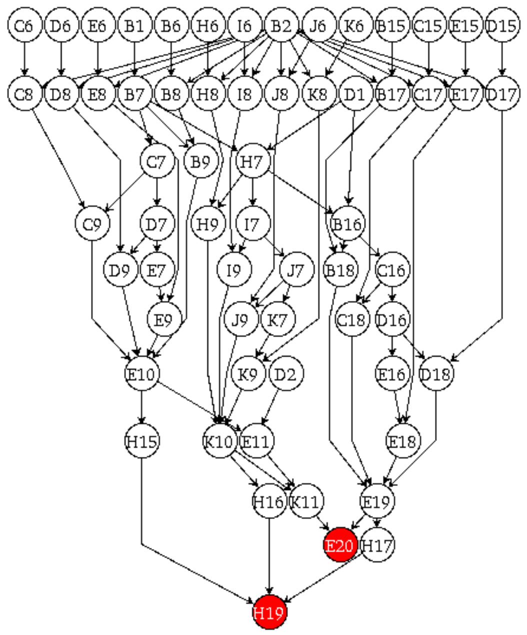


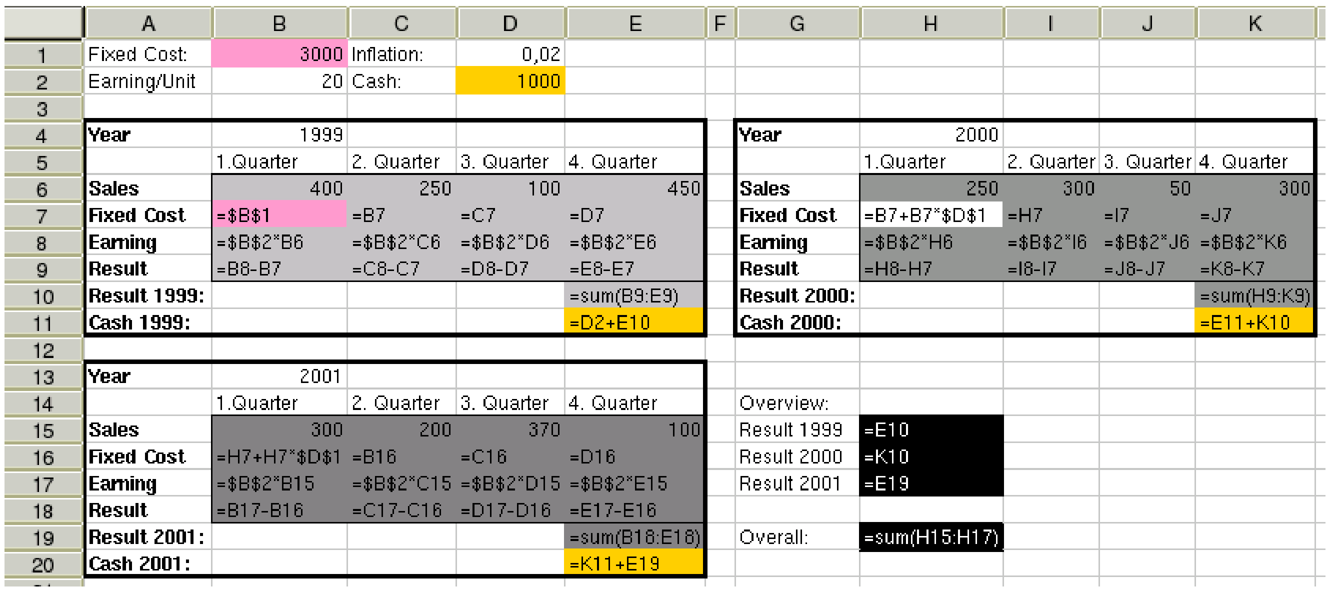
**Figure 4. Example spreadsheet's $DDG$. The sink nodes are shaded in dark gray.**

- E10 (B6, C6, D6, E6, C7, D7, E7, B8, C8, D8, E8, B9, C9, D9, E9, E10)
- K10 (H6, I6, J6, K6, I7, J7, K7, H8, I8, J8, K8, H9, I9, J9, K9, K10)
- E19 (B15, C15, D15, E15, B16, C16, D16, E16, B17, C17, D17, E17, B18, C18, D18, E18, E19)
- E20 (D2, E11, K11, E20)
- H19 (H15, H16, H17, H19)

In Figure 5, cells in the same data module are shaded equally. At first sight, it is interesting, that the *Fixed Cost* of the first quarter of the years 1999 (`B7`) and 2000 (`H7`) are not part of the data modules containing the other cells in the spatial blocks. However, this is clear the formulas when are considered. The corresponding cells are referenced by some cells that are not part of the data module, e.g. `B7` is referenced to calculate the fixed cost for the second quarter of 1999 (`C7`) and to calculate the fixed cost for next year (`H7`). The cash-cells are not part of the years' data modules either, because the remaining cash is always transfered into the next year, e.g. `K11` references `K10` and `E11`.

The $SRG_{DM}$ so generated (see Figure 6) is less complex than the original $DDG$, as it consists of 9 nodes and 15 edges, compared to 68 nodes and 86 edges, but it still contains important information about the spreadsheet programs structure. For instance, it becomes obvious at first sight, that cells in the data module `K10` depend only on cells either in `K10`, in `H7` or in `B2`. This information is important for making local changes or for error tracing (see next section).

Furthermore, the detailed analysis of a specific data module, for instance `K10`, in the broader context of the $SRG_{DM}$ is supported. Therefore, we can zoom into `K10`, replacing the node `K10` in the $SRG_{DM}$ with the subgraph of the $DDG$ that contains only cells, that are members of `K10`



| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Fixed Cost: | 3000 | Inflation: | 0,02 | | | | | | | |
| 2 | Earning/Unit | 20 | Cash: | 1000 | | | | | | | |
| 3 | | | | | | | | | | | |
| 4 | Year | 1999 | | | | | Year | 2000 | | | |
| 5 | | 1.Quarter | 2. Quarter | 3. Quarter | 4. Quarter | | | 1.Quarter | 2. Quarter | 3. Quarter | 4. Quarter |
| 6 | Sales | 400 | 250 | 100 | 450 | | Sales | 250 | 300 | 50 | 300 |
| 7 | Fixed Cost | =$B$1 | =B7 | =C7 | =D7 | | Fixed Cost | =B7+B7*$D$1 | =H7 | =I7 | =J7 |
| 8 | Earning | =$B$2*B6 | =$B$2*C6 | =$B$2*D6 | =$B$2*E6 | | Earning | =$B$2*H6 | =$B$2*I6 | =$B$2*J6 | =$B$2*K6 |
| 9 | Result | =B8-B7 | =C8-C7 | =D8-D7 | =E8-E7 | | Result | =H8-H7 | =I8-I7 | =J8-J7 | =K8-K7 |
| 10 | Result 1999: | | | | =sum(B9:E9) | | Result 2000: | | | | =sum(H9:K9) |
| 11 | Cash 1999: | | | | =D2+E10 | | Cash 2000: | | | | =E11+K10 |
| 12 | | | | | | | | | | | |
| 13 | Year | 2001 | | | | | | | | | |
| 14 | | 1.Quarter | 2. Quarter | 3. Quarter | 4. Quarter | | Overview: | | | | |
| 15 | Sales | 300 | 200 | 370 | 100 | | Result 1999 | =E10 | | | |
| 16 | Fixed Cost | =H7+H7*$D$1 | =B16 | =C16 | =D16 | | Result 2000 | =K10 | | | |
| 17 | Earning | =$B$2*B15 | =$B$2*C15 | =$B$2*D15 | =$B$2*E15 | | Result 2001 | =E19 | | | |
| 18 | Result | =B17-B16 | =C17-C16 | =D17-D16 | =E17-E16 | | | | | | |
| 19 | Result 2001: | | | | =sum(B18:E18) | | Overall: | =sum(H15:H17) | | | |
| 20 | Cash 2001: | | | | =K11+E19 | | | | | | |

**Figure 5. Example spreadsheet, formula view. Cells in the same data module are shaded equally. The result cells of each data module are typeset in an italic font.**

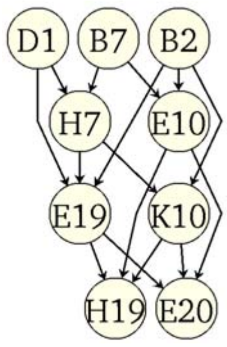


**Figure 6. Example spreadsheet's** $SRG_{DM}$

(see Figure 7). This analysis technique resembles to the application of shrimp views that have turned out to be a successful approach for the comprehension of conventional software [24].

## 4. Applications

As mentioned above, there are several possible applications for data modules in the context of large spreadsheet programs. Consecutively, we will briefly discuss visualization, decomposition, maintenance, and fault tracing of spreadsheet programs by means of data modules.

### 4.1. Visualization

As stated in the previous section, there are at least two ways to exploit the partitioning of a spreadsheet program for the visualization. The $SRG_{DM}$ of a spreadsheet program is a powerful abstraction of the $DDG$ and is, together with shrink-views, an important aid for the comprehension of a spreadsheet program. Additionally, it can also be used for finding structural errors in a spreadsheet program by looking for broken links, i.e. areas of the spreadsheet program that should be in the same data module, but are not, maybe due to a misreference in a formula.

The second possible visualization approach is coloring the cells on the spreadsheet user interface in a way that cells in the same data module will have the same color. This visualization technique supports the user in finding *holes* in data modules, i.e. cells that are referenced from another data module, but should not be. However, it does not give any information about links between different data modules.

A combination of both visualization approaches is most effective, as the user can easily spot the extent of a data module on the colored user interface, and can trace the links in the $SRG_{DM}$ view.

### 4.2. Decomposition

Many problems in the field of spreadsheet analysis is due to the overwhelming complexity that is often due to the size of the spreadsheet programs. In field audits, spreadsheets

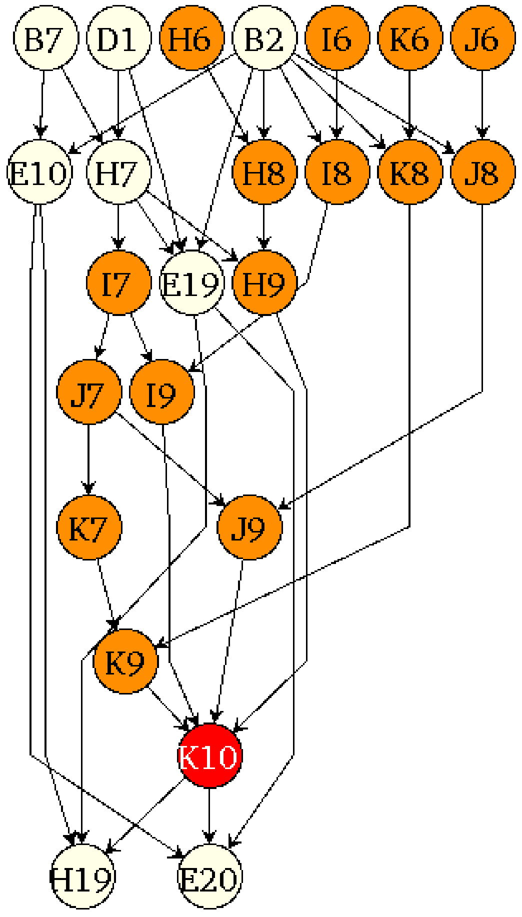


**Figure 7. Example spreadsheet's $SRG_{DM}$, zoomed into `K10`. The members of `K10` are shaded dark.**

with 9000 cells on the average [8] have been reported. Obviously, the comprehension of such large sheets calls for effective abstraction techniques. Data modules can be used to decompose a large spreadsheet into smaller parts and condition it for further analysis of the individual data modules by means of other visualization techniques.

### 4.3. Maintenance

Spreadsheet programs are often subject to short maintenance cycles. Together with the fact, that there is little documentation and only poor testing done, maintenance of spreadsheet programs becomes a critical issue. Data modules can increase the understanding the maintainer has about the internal logic of a spreadsheet. Obviously, there is less risk, if only new data modules are added or a given data module is extended, than if an existing data module is destroyed in course of a maintenance operation. thus, there is also a kind of guidance for the maintainer.

### 4.4. Fault Tracing

Fault tracing is a very common problem in spreadsheet programs, as the symptoms of errors often do not occur at the same place as the faults that cause the wrong results. Hence, most testing techniques also involve techniques for fault tracing that are usually based on the calculation of error probabilities for the predecessors of the faulty cell in the spreadsheet programs $DDG$ (see e.g. Ayalew [1] or Reichwein et al. [20]).

The generation of data modules and the usage of the $SRG_{DM}$ are a powerful support for fault tracing. If an error is detected in the result cell of a data module, it is not necessary to check all the predecessors in the $DDG$ until the error is found. If the spreadsheet auditor is aware of the data module where the symptom of the error occurred, there are only two possibilities:

1. The error occurred inside the data module where it is detected, or
2. the error occurred in a predecessor module in the $SRG_{DM}$.

It is not difficult to decide on which case applies: the spreadsheet auditor has to check only the result cells of the predecessor data modules in the $SRG_{DM}$. If they are correct, the error is buried in the module where the failure occurred. Else it is assumed that the error is propagated from the erroneous module.

For the first case, the $DDG$ of the data module where the failure occurred has to be checked by one of the techniques that are suggested in [1, 20]. Nevertheless, a piece of extra information the auditors are aware is that the error must be in the currently examined subgraph of the $DDG$, and the bug tracing can stop at the module boundaries.

In the second case, the same process is repeated: it has to be checked, whether the fault occurred inside the data module, or in one of its predecessor modules. Depending on the error source, either the module is checked, or the search continues upward in the $SRG_{DM}$.

Obviously, also a combination of error sources is possible, as errors can be hidden inside the module as well as in several predecessor modules. Nevertheless, an iteration of several testing and correction phases will finally find all the errors.

## 5. Related Work

In order to overcome the problems of testing spreadsheets, i.e. the lack of sufficient test data and the lack of testing skills, visual auditing has become a popular review method for spreadsheet programs. The rationale of spreadsheet visualization is to reduce the inherent complexity of spreadsheet programs to a magnitude that is easier to understand for the human auditor (see e.g. Nixon et al. [17]). Usually visual auditing tools color cells that share certain characteristics, i.e. similar formulas, a certain data type or spatial neighborhood, with the same color on the spreadsheet UI. Thus, the auditor does not have to check the spreadsheet on a cell-by-cell basis any more but can check larger units. Examples for these techniques are the S2 and S3 Visualization [22], SpACE [2], the spreadsheet detective [23] and logical areas and semantic classes [15, 7, 9].

However, there might be spreadsheets were formulas are only occasionally copied, and thus, these techniques will not be effective any more. In this case, data flow based techniques have to be applied. The simplest approach, i.e. showing the immediate $DDG$ successors and predecessors, is implemented as a standard feature in most spreadsheet systems [10]. Although this approach is helpful for debugging a specific formula, it is not helpful for the comprehension of the spreadsheet program.

Chan et al. [6, 5] introduce another interesting visualization and debugging approach. It is mainly data flow based and offers support for *local* and *global* debugging. However, both debugging strategies are again tied to the spreadsheet as visualization tool. Therefore, the user can only audit a section of the spreadsheet that corresponds to the size of their screen at a time. In brief, it is assumed that the data flow in a spreadsheet program should correspond to a text: the data should flow from cells that are situated on the upper left corner of the spreadsheet UI to cells on the bottom-right corner of the spreadsheet UI. Data flow that does not correspond to this rule is considered dangerous and will thus be reported to the auditor by colorizing the concerned cells.

However, as the spreadsheet UI is the user interface of the auditing toolkit, the linkage between spatially widespread parts of the spreadsheet is still very hard to understand, and zooming can only be done by adjusting the display size.

In contrast, data modules do not make an assumption like this. Although the debugging effectivity of data modules might suffer from the lack of such an assumption they will reduce the amount of information users have to process, if they are doing maintenance or error tracing. Nevertheless, the approach suggested by [6] is designed for debugging a spreadsheet program, but not for the support of spreadsheet comprehension.

The term of a modulare spreadsheet often appears in the relevant literature [13, 19, 4, 26]. However, these approaches aim to force the user into a design phase in spreadsheet development. It has turned out, that they will efficiently decrease errors and lead to more comprehensive spreadsheet programs. Nevertheless, they are not widely used, as they aim to change the spreadsheet users and require a certain degree of IT-training. Spreadsheets that have been carefully designed with a modular approach in mind, can be very efficiently analyzed with the data modules approach.

Nevertheless, our approach does not force spreadsheet users to change the way they are creating spreadsheets. Of course, a inherently modular spreadsheet will yield a more helpful abstraction than an entangled one. However, it is still the decision of the users, how they build their spreadsheets.

## 6. Conclusion

This paper presents a new technique for spreadsheet decomposition and spreadsheet visualization. Data modules are either an extension to arbitrary formula based spreadsheet comprehension techniques as they can decompose large spreadsheets into smaller units. On the other hand, they are also a visualization technique on their own that can be efficiently applied to large spreadsheets, or regions of spreadsheets, with no regular formula usage. The rationale behind data modules is to find subgraphs of the $DDG$ that are capsuled from the rest of the spreadsheet program.

It has turned out that the approach is suitable for large spreadsheets that are often composed from more than one only loosely related areas. Data modules can find these areas and allow independent analysis, testing and maintenance for each of these areas. Additionally, fault tracing is supported.

In contrast to other techniques that are based on similarities between the formulas, data modules will analyze only the links between cells, neglecting any operations. Thus, even spreadsheets with irregular formula usage can be successfully analyzed.

## References

[1] Y. Ayalew. *Spreadsheet Testing Using Interval Analysis*. PhD thesis, Universität Klagenfurt, Universitätsstrasse 65–67, A-9020 Klagenfurt, Austria, November 2001.

[2] R. Butler. Is This Spreadsheet a Tax Evader ? How H. M. Customs & Excise Test Spreadsheet Applications. In *Proceedings of the 33rd Hawaii International Conference on System Sciences - 2000*, volume 33, 2000.

[3] R. Casimir. Real programmers don't use spreadsheets. *ACM SIGPLAN Notices*, 27(6):10–16, June 1992.

[4] D. Chadwick, K. Rajalingham, B. Knight, and D. Edwards. An Approach to the Teaching of Spreadsheets Using Software Engineering Concepts. In *Proceedings of the 4th International Conference on Software Process Improvement, Research, Education and Training INSPIRE'99*, pages 261–273, 1999.

[5] H. C. Chan, editor. *Easy Steps to Design & Check Your Excel spreadsheets*. Federal Publications, 2001.

[6] H. C. Chan and Y. Chen. Visual checking of spreadsheets. In *Spreadsheet Risks, Audit and Development Methods*, volume 1, pages 75–85. EuSpRIG, University of Greenwich, 7 2000.

[7] M. Clermont. *A Scalable Approach to Spreadsheet Visualization*. PhD thesis, Universität Klagenfurt, Universitätsstrasse 65–67, A-9020 Klagenfurt, Austria, March 2003.

[8] M. Clermont, C. Hanin, and R. Mittermeir. A Spreadsheet Auditing Tool Evaluated in an Industrial Context . In *Spreadsheet Risks, Audit and Development Methods*, volume 3, pages 35–46. EUSPRIG, 7 2002.

[9] M. Clermont and R. Mittermeir. A Pattern Based Approach to Spreadsheet Auditing. In *Proceedings of the 6th International Conference on Information Systems Implementation and Modelling ISIM'03*, April 2003.

[10] J. S. Davis. Tools for spreadsheet auditing. *International Journal of Human-Computer Studies*, 45(4):429–442, 1996.

[11] G. Filby, editor. *Spreadsheets in Science and Engineering*. Springer, Berlin, Heidelberg, 1998.

[12] J. Goldberg. The gnumeric project. http://www.gnumeric.org, January 2003. visited on 10th January 2003.

[13] B. Knight, D. Chadwick, and K. Rajalingham. A structured methodology for spreadsheet modelling. In *Spreadsheet Risks, Audit and Development Methods*, volume 1, pages 43–50. EuSpRIG, University of Greenwich, 7 2000.

[14] P. Kokol. Some Applications of Spreadsheet Programs in Software Engineering. *Software Engineering Notes*, 12(3):45–50, July 1987.

[15] R. Mittermeir and M. Clermont. Finding High-Level Structures in Spreadsheets. In *Proceedings of the 9th Working Conference on Reverse Engineering*, 2002.

[16] B. Nardi and J. Miller. An Ethnographic Study of Distributed Problem Solving in Spreadsheet Development . In *Proceedings of the conference on Computer-supported cooperative work* , pages 197–208. ACM, October 1990.

[17] D. Nixon and M. O'Hara. Spreadsheet auditing software. In *Spreadsheet Risks, Audit and Development Methods*, volume 2. EuSpRIG, University of Greenwich, 7 2001.

[18] R. R. Panko. What we know about spreadsheet errors. *Journal of End User Computing: Special issue on Scaling Up End User Development*, 10(2):15–21, Spring 1998.

[19] K. Rajalingham, D. Chadwick, B. Knight, and D. Edwards. Quality Control in Spreadsheets: A Software Engineering-Based Approach to Spreadsheet Development. In *Proceedings of the 33rd Hawaii International Conference on System Sciences 2000*, volume 33. IEEE, 2000.

[20] J. Reichwein, G. Rothermel, and M. Burnett. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. In *Proceedings of the 2nd Conference on domain-specific languages*, volume 2, pages 25–38. ACM, 2000.

[21] K. Rothermel, C. Cook, M. Burnett, J. Schonfeld, T. Green, and G. Rothermel. Wysiwyt testing in the spreadsheet paradigm: An empirical evaluation. In *ICSE 2000 Proceedings*, pages 230–239. ACM, 2000.

[22] J. Sajaniemi. Modeling spreadsheet audit: A rigorous approach to automatic visualization. *Journal of Visual Languages and Computing*, 11(1):49–82, 2000.

[23] S. C. Software. Operis group plc. http://www.operis.com/oak.htm, 2002. visited on 11th September 2002.

[24] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95)* (Opio (Nice), France, October 16-20, 1995), 1995.

[25] G. E. Vlahos and T. W. Ferratt. The use of information technology by managers of corporations in greece to support decision making. In *Proceedings of the conference on Computer Personal Research*, pages 136–151. ACM, 1992.

[26] N. P. Wilde. A WYSIWYC (What You See Is What You Compute) Spreadsheet. In *Proceedings of the 1993 Symposium on Visual Languages*, pages 72–76. IEEE, IEEE Computer Society Press, 1993.