Towards the Design and Implementation of Aspect-Oriented Programming for Spreadsheets

Pedro Maia*†, Jorge Mendes*†, Jácome Cunha*‡, Henrique Rebêlo§, João Saraiva*†

*HASLab / INESC TEC, Portugal
†Universidade do Minho, Portugal
{pedromaia,jorgemendes,jas}@di.uminho.pt
‡Universidade Nova de Lisboa, Portugal
jacome@fct.unl.pt
§Universidade Federal de Pernambuco, Brazil
hemr@cin.ufpe.br

Abstract—A spreadsheet usually starts as a simple and single-user software artifact, but, as frequent as in other software systems, quickly evolves into a complex system developed by many actors. Often, different users work on different aspects of the same spreadsheet: while a secretary may be only involved in adding plain data to the spreadsheet, an accountant may define new business rules, while an engineer may need to adapt the spreadsheet content so it can be used by other software systems. Unfortunately, spreadsheet systems do not offer modular mechanisms, and as a consequence, some of the previous tasks may be defined by adding intrusive "code" to the spreadsheet.

In this paper we go through the design and implementation of an aspect-oriented language for spreadsheets so that users can work on different aspects of a spreadsheet in a modular way. For example, aspects can be defined in order to introduce new business rules to an existing spreadsheet, or to manipulate the spreadsheet data to be ported to another system. Aspects are defined as aspect-oriented program specifications that are dynamically woven into the underlying spreadsheet by an aspect weaver. In this aspect-oriented style of spreadsheet development, different users develop, or reuse, aspects without adding intrusive code to the original spreadsheet. Such code is added/executed by the spreadsheet weaving mechanism proposed in this paper.

I. INTRODUCTION

Spreadsheet systems are the software system of choice for many non-professional programmers, often called end-user programmers [1], like for example, teachers, accountants, secretaries, engineers, managers, etc. In the last century such end users would develop their spreadsheets individually in their own desktops, and sharing and reuse was not the usual procedure. The recent advent of powerful mobile devices, and, as a consequence, the availability of powerful cloud-based spreadsheet systems (like, for example, Google Drive), has dramatically changed this situation. Nowadays, spreadsheets are complex software systems, developed and maintained by many (end) users.

Very much like in the development of other software systems, different developers (end users in this case) are concerned with different aspects of the functionality of the system. However, while modern programming languages offer modularity mechanisms providing powerful abstractions to develop software collaboratively, spreadsheet systems offer no such support to their users. As a result, a spreadsheet tends to evolve into a single software artifact where all business logic from all different users is defined! In such a collaborative setting if a new user needs to express a new business rule on the spreadsheet data, he/she has to do it by intrusively adding formulas to the existing spreadsheet.

The programming language community developed advanced modularity mechanisms to avoid this problem in regular programming languages. In that sense, aspect-oriented programming (AOP) is a popular and advanced technique that enables the modular implementation of the so-called cross-cutting concerns. The crosscutting structure tends to appear tangled and scatted across several artifacts of a software system. While implementing such crosscutting concerns, the crosscutting structure can appear in common software development concerns, such as distribution and persistence [2], error handling [3], [4], certain design patterns [5], tracing [6], or design by contract [6]–[8].

In this paper we introduce the concept of AOP to spreadsheets. We start by introducing a running example in Section II. Our first contribution is presented in Section III where we adapt the AOP concept to spreadsheets. For instance, a key feature of a common aspect-oriented language is the possibility to add advice before, after or around a join point. Since spreadsheets are two-dimensional, AOP features like advising need to be adapted in order to be applied to spreadsheets. The second contribution we do in this work is the design of a new language to implement AOP for spreadsheets. This language is presented in Section IV. An overview of the architecture of the system is presented in Section V. Finally, Section VI discusses related work, and Section VII presents our conclusions.

II. MOTIVATION

In this section we present an example of a popular setting to use spreadsheets: manage the students marks in a course. Such a spreadsheet is shown in Fig. 1.

In this simple spreadsheet, the final mark of a student is the average of three evaluation items: two exams and one essay. Thus, this mark is obtained by the formula =AVERAGE(B2:D2) (for the student in line 2).

Fig. 1. Student grading spreadsheet.

Let us consider now a real scenario where three different users share, access, and update this spreadsheet: a *teaching assistant*, that mainly structures the spreadsheet and compiles the different marks, the *teaching coordinator* who has to validate the spreadsheet, decide on borderline cases, and send the final marks to the academic services, and finally, the *Erasmus coordinator* who has to adapt the marks of Erasmus students to the ECTS system[1].

Because spreadsheets offer no modularity mechanisms, a typical spreadsheet development environment allows a spreadsheet to quickly evolve into a more complex one. That is, each of the users adds/updates data and formulas in order to express the (crosscutting) logic they need. In the following, we show the spreadsheet after the teaching and Erasmus coordinator update it.

Fig. 2. Example of a student grading spreadsheet with mark adjustments (cell E3) and ECTS marks (column F).

The data edited by one user can be seen as intrusive code for the others. For example, the teaching coordinator and its assistant are not concerned with the ECTS system, although this concern is now part of their spreadsheet.

Let us analyze the teaching coordinator role in more detail. Because he decides on borderline cases, he decided to approve only one of such cases (student in line 3). Instead of changing that particular final mark formula, and as often occurs in reality, he just replaces this formula by the constant 5.0. Although the result of such an operation is a correct spreadsheet, this action has several problems. First, there is no documentation on how borderline cases have been decided (the threshold is not known). Second, the original spreadsheet data is lost, and it may be difficult to recover it (for example, if other borderline students ask for explanations).

This is the natural setting to apply AOP. In this style of programming users do not intrusively modify the original program (a spreadsheet in our case), but instead, a new software fragment is defined in order to specify the program transformations needed to express such new concern. These software fragments are called *aspects*. Then, a specific AOP

[1]For more details on the ECTS system please refer to http://ec.europa.eu/ education/tools/docs/ects-guide_en.pdf.

mechanism, called *weaver*, given the original program and the aspect(s), weaves them into a coherent executable software program.

We present in Listing 1 our first spreadsheet aspect, the one that specifies the coordinator concerns on borderline cases.

Listing 1. An aspect to handle borderline marks.
```
aspect BorderlineCase
finalmark : select sheet{*}.column{*}.cell{*}
around finalmark {
    #{cell.result >= 4.8 && cell.result < 5 ? 5 :
    cell.value}
  } when {
    cell.column[0].value = "Final Mark"
  }
end
```

An aspect-oriented language has three main parts. First it is necessary to select the *join points* of interest by means of pointcut declarations. In the example, the pointcut `finalmark` selects any cell within any worksheet. This is done by the *select* command in our language. Then, the *around* advice declaration defines the actions (transformations in our case) to be applied. Hence, it specifies when the *result* of evaluating a formula is greater than of equal to 4.8, and if it is, the cell is replaced by the constant 5.0. To access the result of the computation of a cell, that is, the result of the formula in the cell, we use `cell.result`. This accesses a cell, and after that, its computed *result*. The *when* statement specifies when the action is applied (in this case to all columns labeled `Final Mark`).

This aspect clearly and non-intrusively defines the crosscutting rule used to decide on borderline cases: all students with marks greater than or equal to 4.8 are approved in the course.

Similarly, the Erasmus coordinator can define an aspect where the rules to define the ECTS marks are expressed. This aspect is presented in Listing 2.

Listing 2. Aspect to add ECTS marks.
```
aspect AddECTSMark
finalmarks : select sheet{*}.column{*}.cell{*}

right finalmarks {
    =IF(#{cell.name}<=10 && #{cell.name}>=9.5
     , "A"
     , IF(#{cell.name}<9.5 && #{cell.name}>=8.5
       , "B"
       , IF(#{cell.name}<8.5 && #{cell.name}>=6.5
         , "C"
         , IF(#{cell.name}<6.5 && #{cell.name}>=5.5
           , "D"
           , IF(#{cell.name}<5.5 && #{cell.name}>=5
             , "E"
             , "F")))))
  } when {
    column[0].value == "Final Mark"
      && cell.row <> 0
  }

right finalmarks {
    ECTS Mark
  } when {
    column[0].value = "Final Mark"
      && cell.row = 0
  }
end
```

In this aspect, we use `cell.name` to access the cell address (for instance "E4"). Thus, when the `IF` formula is placed in the corresponding cells, the correct references will be calculated and inserted in the formula replacing the use of `cell.name`.

An aspect weaver can then weave this aspects into the original spreadsheet individually, or by composing them. Actually, Fig. 2 shows the result of weaving the aspect *AddECTS-Mark* after the aspect *BorderlineCase*. Thus, aspects can be combined. It should also be noticed that such a spreadsheet based weaver has to dynamically weave aspects since for some aspects (*BorderlineCase*) only after computing formulas, aspects can be weaved. Indeed, it is necessary to execute the spreadsheet to get the results of formulas, for instance, when the `cell.result` operator is used.

We have just presented two aspects of our AOP spreadsheet language and briefly explained how aspects are weaved into a spreadsheet system. Next sections present in detail the design and implementation of this language and weaver.

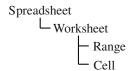## III. APPLYING ASPECT-ORIENTED PROGRAMMING TO SPREADSHEETS

In this section, we discuss how AOP concepts can be applied to spreadsheets. Specific examples are described in Section IV.

A. Spreadsheet Join Point Model

In order to apply AOP to a language, it is necessary to identify the *join point* model that the new aspect language supports. A join point is a well-defined point in the program that is specified by a *pointcut*, that is, an expression to match specific elements within the language. In our case, these elements are set to be the main elements of the spreadsheet, where users will want to perform operations on, for instance, use an alternate worksheet for testing or add more cells for debugging. For spreadsheets, we define the following join points of interest:

- worksheets;
- ranges; and,
- cells.

With these join points the spreadsheet is accessible from aspects, hence users can separate concerns at different levels:

```
Spreadsheet
    └─Worksheet
            ├─ Range
            └─ Cell
```

When a join point is instantiated, it is possible to perform an action using advice. Advice are additional behavior that one wants for the underlying program, that can be new worksheets, ranges of cells, or single cells. This allows to separate the business logic of the spreadsheet into several concerns when developing the spreadsheet and then join everything in order to achieve the wanted application.

B. Worksheet

A spreadsheet file is composed of a set of worksheets, each of which containing the cells with the data and formulas. Worksheets are the top-level artifacts in a spreadsheet.

As a join point, a worksheet (see Fig. 3) can be modified by the standard AOP advice in the following manner:

- **before** – insert a worksheet before the current join point;
- **after** – insert a worksheet after the current join point;
- **around** – insert a worksheet before and/or after the current join point and/or define an alternative worksheet for the current join point.
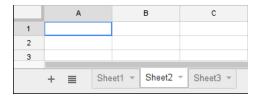

Fig. 3. List of three worksheets within a spreadsheet file.

As an example, consider Fig. 3 with `Sheet2` as the join point. A *before* advice for this join point results in a new worksheet between the worksheets `Sheet1` and `Sheet2`. On the other hand, if we consider an *after* advice, the resulting worksheet is between worksheets `Sheet2` and `Sheet3`.

To specify a worksheet, two options are available: either defining the cells for the new worksheet, or referencing a worksheet to be duplicated.

C. Cell

Cells are the finest grained join points possible in the spreadsheets world. Unlike common programming languages, they are inserted in a two-dimensional plan. Thus, the usual *before* and *after* advice declarations are not completely adequate since ambiguities may arise about which of the two dimensions (vertical or horizontal) should be used. To overcome this issue, each kind of advice declaration is separated in two for each dimension in the plane, resulting in the following:

direction \ advice	*before*	*after*
vertical	above	below
horizontal	left	right

The advice that can be defined for cell join points is:

- **left** – add a cell, or range of cells, to the left of the current join point;
- **above** – add a cell, or range of cells, above the current join point;
- **right** – add a cell, or range of cells, to the right of the current join point;
- **below** – add a cell, or range of cells, below the current join point;
- **around** – define an alternative cell for the current join point.

A cell is specified just by stating its contents. A range of cells is specified by defining the cells that it contains.

Fig. 4. Two-dimensional advice.

D. Range

A range is a set of cells contained in a worksheet. As it regularly happens in spreadsheets, our setting only handles rectangular ranges, that is, the range must be a rectangle and all the cells in that rectangle must part of the range.

Advice declaration for ranges is similar to the ones for cells, but special care must be taken to match range sizes when adding new cells, or ranges before or after it. Thus, we have the following possible advice declarations:

- **left** – add a range of cells, to the left of the current join point;
- **above** – add a range of cells, above the current join point;
- **right** – add a range of cells, to the right of the current join point;
- **below** – add a range of cells, below the current join point;
- **around** – define an alternative range for the current join point.

IV. LANGUAGE TO SPECIFY ASPECTS FOR SPREADSHEETS

In this section we present our language for aspects for spreadsheets. This language is based on existing ones for common purposes programming languages, and implements the vision we described in Section III. The proposed language allows to specify aspects defining its pointcuts and advice. For each component of the aspects to be written by the user we present next the corresponding grammar. This grammar is the artifact used to validate, though a compiler, the correctness of the aspects written by the users. We start by introducing the grammar for pointcuts.

A. Pointcuts

Pointcuts are defined by an expression pattern, which is specified by the following grammar:

⟨*join_point*⟩ ::= ⟨*jp_name*⟩ ':' `select` ⟨*pExpr*⟩

where ⟨*pExpr*⟩ is a pattern expression. The expression is used to define the kind of join point to be selected: *sheet*, *range*, or *cell*.

The allowed expressions in join points are instances of the following productions:

⟨*pExpr*⟩ ::= ⟨*pSheet*⟩
  | ⟨*pSheet*⟩ '.' ⟨*pRange*⟩
  | ⟨*pSheet*⟩ ('.' ⟨*pRange*⟩)? '.' ⟨*pCell*⟩

⟨*pSheet*⟩ ::= `sheet` '{' ⟨*pSheetExpr*⟩ '}'

⟨*pSheetExpr*⟩ ::= 'name' ⟨*bComp*⟩ ⟨*string*⟩
  | 'number' ⟨*bComp*⟩ ⟨*integer*⟩
  | '*'

⟨*pRange*⟩ ::= 'range' '{' ⟨*pRangeExpr*⟩ '}'
  | 'column' '{' ⟨*pRangeExpr*⟩ '}'
  | 'row' '{' ⟨*pRangeExpr*⟩ '}'

⟨*pRangeExpr*⟩ ::= 'name' ⟨*bComp*⟩ ⟨*string*⟩
  | '*'

⟨*pCell*⟩ ::= 'cell' '{' ⟨*pCellExpr*⟩ '}'

⟨*pCellExpr*⟩ ::= 'name' ⟨*bComp*⟩ ⟨*string*⟩
  | 'match' ⟨*bComp*⟩ ⟨*string*⟩
  | '*'

⟨*bComp*⟩ ::= '==' | '<' | '<=' | '>' | '>=' | '<>'

The pattern expression ⟨*pExpr*⟩ allows to select worksheets ⟨*sheet*⟩, ranges ⟨*range*⟩, or cells ⟨*cell*⟩.

When specifying a worksheet as a join point, it is possible to select a worksheet relative to a given name, to a worksheet index, or to select all worksheets.

For ranges, there are three kinds that can be selected: a column range (`column`) with a width of one cell, a row range (`row`) with a height of one cell, or a range with any rectangular shape (`range`).

For cells, it is possible to specify its address (`name`), or a pattern-match expression (`match`). When no specific worksheet, range or cell is necessary, the wildcard symbol '*' can be used.

With this, we can select, for instance, the second worksheet

```
worksheet_jp: select worksheet{number=2}
```

or a rectangular range of 3 columns by 2 rows starting at cell A2 in any worksheet:

```
range_jp: select worksheet{*}.range{name="A2:C3"}
```

or the cells in the first row of any worksheet:

```
cell_jp: select worksheet{*}.range{row=1}.cell{*}
```

Depending on the kind of join point selected, different artifacts are made available to work with within the advice. For worksheets, we can use the variable `worksheet`, and then one of the attributes: `name` or `number`. For ranges, depending on its kind, the available variable can be: `range`, `column`, or `row`; they have the attribute `name`. Moreover, since ranges are sets of cells, indexation can be used, for instance, to select the first row of a column, one can write `column[0]`. For cells, we have the variable `cell` which has the attribute `name` (its cell reference).

B. Advice

Advice are the actions to apply to the join points. They are defined by the following grammar:

⟨*advice*⟩ ::= (⟨*advice_name*⟩':')?   ⟨*advice_position*⟩
  ⟨*jp_name*⟩ '{' ⟨*code*⟩ '}' ⟨*advice_condition*⟩

⟨*advice_position*⟩ ::= 'left' | 'above' | 'right' | 'below' | 'around'

⟨*advice_condition*⟩ ::= ('when {' ⟨*bExpr*⟩ '}')?

where ⟨*code*⟩ is a cell or a list of cells and respective contents:

⟨*code*⟩ ::= ⟨*string*⟩ | ⟨*cellList*⟩

⟨*cellList*⟩ ::= ⟨*cellRef*⟩ '=' ⟨*string*⟩ (';'⟨*cellList*⟩)?

The contents of the cell ⟨*string*⟩ can be defined using interpolation of values made available in the context of the advice (for instance, join point contents) using interpolation. For example, to add a row which evaluates the total of a column (for instance, the join point is a column range) where the join point is named *myColumn*:

```
below myColumn { =SUM(#{range.name}) }
```

In the above example, interpolation is used to introduce a value that is available in the join point, but is not accessible from common spreadsheet formulas. If the column range is `C1:C20`, then the formula would be `=SUM(C1:C20)`.

Note that the order by which the advice are applied is important. They are applied according to their precedence which is defined by the order they are defined. In the case of `before` the ones defined first are the ones with more precedence. Thus, the ones defined earlier are applied first. For the `after` it is the other way around, that is, although the ones declared first are the ones with more priority, they are executed last. This is the common behavior of aspects for other programming languages.

Advice can be applied conditionally, that is, when a specified criterion is met. This is specified with a boolean expression as defined by the following grammar:

⟨*bExpr*⟩ ::= '!' ⟨*bExpr*⟩
  | ⟨*bExpr*⟩ ('&&' | '||') ⟨*bExpr*⟩
  | ⟨*expr*⟩ ⟨*bComp*⟩ ⟨*expr*⟩

⟨*expression*⟩ ::= ⟨*bExpr*⟩ | ⟨*var*⟩ | ⟨*string*⟩ | ⟨*number*⟩
  | ⟨*cellRef*⟩ | ⟨*rangeRef*⟩

C. Aspect

An aspect is composed of the definition of the pointcuts and the advice to apply to them, as defined by the following grammar:

⟨*aspect*⟩ ::= 'aspect' ⟨*string*⟩
  ⟨*join_point*⟩+ ⟨*advice*⟩+
  'end'

V. A WEAVER FOR SPREADSHEETS

In this section we present our architectural model for aspect spreadsheets. The aspects are defined using the language presented in Section IV, and together with the spreadsheet, they are interpreted by the *weaver*. In this context, the spreadsheet is only complete when considered together with the aspects.

Some aspects can be handled statically, that is, without executing the spreadsheet. For instance, the example shown is Listing 2 does not need to execute the spreadsheet to know what to do.

However, some aspects require the execution of the program, in our case, of the spreadsheet (formulas). This is the case illustrate in Listing 1 where it is necessary to *compute* the student grade to decide if the final grade is changed or not. In that case we used the operator `cell.result`, which requires to evaluate the formula of the underlying cell to obtain its result. Thus, the weaver must be integrated with a spreadsheet execution engine. Indeed we intend to build the weaver inside Excel itself, so it can reuse its recalculation mechanism.

In these cases, the spreadsheet is kept untouched, but since now the spreadsheet is only complete when considered with the aspects defined, the values it shows may change.

If the user wants to see the original spreadsheet then, it is only necessary to deactivate the aspects.

Fig. 5 illustrates the integration of our weaver with a spreadsheet system to create an aspect-oriented spreadsheet system.
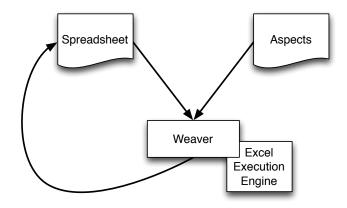

Fig. 5. A weaver for aspect-oriented spreadsheets.

VI. RELATED WORK

Aspect-oriented programming has been applied to several programming languages, for instance, Java [9], C++ [10], or MATLAB [11], [12], targeting specific characteristics of each language in order to improve them. Some of these applications contributed to shape the use of aspects, and introduced new concepts to this paradigm.

Java was one of the first languages to be exposed to aspects through a language dubbed AspectJ [9]. This language is similar to Java so it feels familiar to Java users. It provides a dynamic join point model, where some advice are applied at compile time, but others are applied only during run time when the complete information about the execution is available. The nature of spreadsheets, where both data and computation are at the same level, imposes a dynamic join point model in order to have access to run-time values.

AOP was also used to support development of embedded systems. The LARA language [13] was purposely designed with this goal, but has a wider range of applications. Since it can target several languages, we inspired ourselves on it as the basis for the specification of aspects for spreadsheets.

In the context of spreasheets, there are several works presenting techniques to transform spreadsheets by using spreadsheet specific transformation languages [14], [15], and querying languages [16], [17]. Moreover, such transformations can be refactorings to remove spreadsheet smells [18]–[21] and thus improve their usage and reduce possible error entry points.

BumbleBee [14] is a Microsoft Excel add-in mainly for performing refactorings to remove smells, but can also perform other kinds of transformations. It finds cells where it can apply a previously defined set of transformations, lets the user select the transformation to apply, and then applies the transformation to a selected range, to the entire worksheet, or to the entire file. A limitation is that BumbleBee only supports intra-formula transformations. Our language uses the BumbleBee's transformation language to support cell value transformations.

RefBook [15] is another tool to perform refactorings to remove spreadsheet smells. It implements a set of seven refactorings in a Microsoft Excel add-in allowing users to perform refactorings when working with their spreadsheets. Our approach can also be used to perform these refactorings.

More generic transformations for spreadsheets, introduced by end-users' needs, can be performed using program synthesis. This technique added the ability to transform strings of text [22], or tables of data [23] from user-supplied examples, providing a familiar way to solve common tasks when dealing with spreadsheets.

Another kind of transformation, targeting spreadsheet testing, is mutation [24]. The goal is to perform mutations in the spreadsheet, that is, small modifications, in order to analyze a test set for the spreadsheet being tested. Using our aspect system, mutation is also possible, by using the BumbleBee transformation language to specify cell mutations.

In [25], [26] we presented techniques to infer the business logic of spreadsheet data. Such techniques restructure the spreadsheet data into different (relational) tables. Such tables can be viewed as aspects of the business logic/spreadsheet data. Thus, this approach may be used to infer aspects in spreadsheets and to evolve a legacy spreadsheet into an aspect-oriented one.

VII. CONCLUSION

This paper proposes the use of aspect-oriented programming for spreadsheets. We have designed an aspect language that considers spreadsheet peculiarities, and a dynamic weaver that is embedded in the evaluation mechanism of a spreadsheet system.

Although AOP provides a powerful modular approach that is particularly suitable to be used in software that is shared and being collaboratively developed, our work opens some important questions that we intend to answer in future work by conducting empirical studies, namely:

- Are end users able to understand the abstractions provided by AOP and to use it in practice?
  This is not only related to our proposed language as it is a more general question. Nevertheless, we need to answer it so we can understand how to better make our AOP language available for user. This is specially important when dealing with end users.
- Does AOP improves end-users' productivity?
  We have shown that some of the model-driven approaches we introduced in the past can do that. We will conduct similar studies to evaluate this new proposal.
- Is the textual definition of the AOP language adequate or should we use a more spreadsheet-like one?
  When dealing with more advanced users, it is not always the case they prefer visual languages [27]. However, for end users this is probably the case. We will extend our work with a visual language to allow end users to define aspects in a more friendly way.

REFERENCES

[1] B. A. Nardi, *A Small Matter of Programming: Perspectives on End User Computing*, 1st ed. Cambridge, MA, USA: MIT Press, 1993.

[2] S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with AspectJ," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '02. New York, NY, USA: ACM, 2002, pp. 174–190. [Online]. Available: http://doi.acm.org/10.1145/582419.582437

[3] M. Lippert and C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming," in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 418–427. [Online]. Available: http://doi.acm.org/10.1145/337180.337229

[4] F. Castor, N. Cacho, E. Figueiredo, A. Garcia, C. M. F. Rubira, J. S. de Amorim, and H. O. da Silva, "On the modularization and reuse of exception handling with aspects," *Softw. Pract. Exper.*, vol. 39, no. 17, pp. 1377–1417, Dec. 2009. [Online]. Available: http://dx.doi.org/10.1002/spe.v39:17

[5] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '02. New York, NY, USA: ACM, 2002, pp. 161–173. [Online]. Available: http://doi.acm.org/10.1145/582419.582436

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with AspectJ," *Commun. ACM*, vol. 44, no. 10, pp. 59–65, Oct. 2001. [Online]. Available: http://doi.acm.org/10.1145/383845.383858

[7] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm, "AspectJML: Modular specification and runtime checking for crosscutting contracts," in *Proceedings of the 13th International Conference on Modularity*, ser. MODULARITY '14. New York, NY, USA: ACM, 2014, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/2577080.2577084

[8] H. Rebêlo, R. Lima, and G. T. Leavens, "Modular contracts with procedures, annotations, pointcuts and advice," in *SBLP '11: Proceedings of the 2011 Brazilian Symposium on Programming Languages*, 2011.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP 2001—Object-Oriented Programming*. Springer, 2001, pp. 327–354.

[10] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An aspect-oriented extension to the C++ programming language," in *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, ser. CRPIT '02. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2002, pp. 53–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=564092.564100

[11] J. M. Cardoso, P. Diniz, M. P. Monteiro, J. M. Fernandes, and J. Saraiva, "A domain-specific aspect language for transforming MATLAB programs," in *Domain-Specific Aspect Language Workshop (DSAL'2010), part of AOSD*, 2010, pp. 15–19.

[12] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren, "AspectMatlab: An aspect-oriented scientific programming language," *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, ser. AOSD '10. New York, NY, USA: ACM, 2010, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/1739230.1739252

[13] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "Lara: an aspect-oriented programming language for embedded systems," in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM, 2012, pp. 179 –190.

[14] F. Hermans and D. Dig, "BumbleBee: A refactoring environment for spreadsheet formulas," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 747–750. [Online]. Available: http://doi.acm.org/10.1145/2635868.2661673

[15] S. Badame and D. Dig, "Refactoring meets spreadsheet formulas," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, Sept 2012, pp. 399–409.

[16] O. Belo, J. Cunha, J. P. Fernandes, J. Mendes, R. Pereira, and J. Saraiva, "QuerySheet: A bidirectional query environment for model-driven spreadsheets," in *Proceedings of the 2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2013, pp. 199–200.

[17] J. Cunha, J. P. Fernandes, J. Mendes, R. Pereira, and J. Saraiva, "Querying model-driven spreadsheets," in *Proceedings of the 2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2013, pp. 83–86.

[18] F. Hermans, M. Pinzger, and A. van Deursen, "Detecting code smells in spreadsheet formulas," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, Sept 2012, pp. 409–418.

[19] J. Cunha, J. P. Fernandes, H. Ribeiro, and J. Saraiva, "Towards a catalog of spreadsheet smells," in *Proceedings of the 12th international conference on Computational Science and Its Applications - Volume Part IV*, ser. ICCSA'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 202–216. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31128-4_15

[20] J. Cunha, J. P. Fernandes, P. Martins, J. Mendes, and J. Saraiva, "SmellSheet detective: A tool for detecting bad smells in spreadsheets." in *VL/HCC*, M. Erwig, G. Stapleton, and G. Costagliola, Eds. IEEE, 2012, pp. 243–244. [Online]. Available: http://dblp.uni-trier.de/db/conf/vl/vlhcc2012.html#CunhaFMMS12

[21] J. Cunha, J. Fernandes, P. Martins, R. Pereira, and J. Saraiva, "Refactoring meets model-driven spreadsheet evolution," in *9th International Conference on the Quality of Information and Communications Technology (QUATIC)*, Sept 2014, pp. 196–201.

[22] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 317–330. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926423

[23] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 317–328. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993536

[24] R. Abraham and M. Erwig, "Mutation operators for spreadsheets," *Software Engineering, IEEE Transactions on*, vol. 35, no. 1, pp. 94–108, Jan 2009.

[25] J. Cunha, M. Erwig, and J. Saraiva, "Automatically inferring ClassSheet models from spreadsheets," in *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE CS, 2010, pp. 93–100.

[26] J. Cunha, M. Erwig, J. Mendes, and J. Saraiva, "Model inference for spreadsheets," *Automated Software Engineering*, pp. 1–32, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10515-014-0167-x

[27] J. Cunha, J. P. Fernandes, R. Pereira, and J. Saraiva, "Querying Spreadsheets: An Empirical Study," *ArXiv e-prints*, Feb. 2015.