A Disclosure Rules Language for  
Deciding Publishability of Frequency Tables

**Keywords:** disclosure control, software, language, rules, census.

# **Introduction**

When publishing population characteristic frequency tables for small geographic areas, such as in a census, it is usual that some cross-tabulations would be disclosive in some areas but not in others. Areas with higher diversity in person characteristics lead to more tables being non-disclosive. Some countries are taking an approach of using flexible dissemination and automating the process of evaluating output tables on an area by area basis to allow more data to be published [1].

Automating the process of output table evaluation requires specifying formal rules that can be encoded in computer software. This paper describes a new language which facilitates specification of rules by statistical disclosure control experts. An implementation of the language has been constructed. The resulting software is fast enough such that the rules written in the new language can be evaluated in real time by a flexible table builder system when an output frequency table is requested.

# Rationale

Whilst it is possible to use existing computer languages to specify the disclosure control rules, using such an approach would have the following disadvantages:

– The core software code and data would need protecting from user authored rules code in order to maintain resilience in the face of bugs. This incurs significant communication and performance penalties.

– Using close coupling for performance reasons creates difficulties with version management between user code and core software code (e.g. changes in data representation).

– A general purpose language has more scope for unintended behaviour and side effects, e.g. by unintentionally adding global state.

– Programs in general purpose languages are typically not amenable to parallel execution unless great care is taken and authors are knowledgeable in appropriate techniques.

– General purpose languages are typically large in scope and thus present more of a learning challenge for a program author.

– User authored code in a general purpose language would typically have more diverse ways of specifying a given rule and thus present more of a challenge for a reader.

For this reason we have attempted the design of a domain specific language for specifying which queries are allowed and formal rules which output tables must pass.

Our approach allows rules to be evaluated by the flexible table builder software instead of an external process. The rules can also be kept secret from the software programmers.

The rules themselves are most easily understood as algorithms. It might be possible to contrive some declarative form to represent them, however history suggests that most people find imperative computer code easier to comprehend and construct [2]. Hence our language has an imperative form. The aim was to create a language as small and simple as possible whilst still being sufficiently expressive to be able to specify rules in a concise manner.

# Implementation

Our language has been implemented using a yacc parser-generator. An abstract syntax tree is constructed from the parser, is further analysed and optimized for eventual interpretive execution in memory. Some common computations on the output table are implemented in native code to provide good runtime performance.

Alternative implementations of the language are possible, for example generating source code for later compilation. Our language definition is open, allowing anyone to make such an implementation.

# Program Structure

Rule program text in our language consists of a set of clauses. We split the program in to two primary types of clause:

– querytest : these are run before a query is executed by the flexible table builder to determine whether a query is allowed.

– tabletest : these are run after a query is executed but before the result is returned to the user. If the table involves a geographic dimension then the tests are run independently for each geographic area to determine whether the table can be published for that area.

The entry point for each set of tests is named “main”. The “main” test can call other sub-tests with different names. This improves readability and also enables diagnostic reporting in the software as to which test caused a table to be barred from publication, or an audit of all the pass/fail status of all tests independently.

Data is made available to rules program code via a set of predefined named objects with properties accessed using a familiar “objectName.propertyName” syntax. Such data includes variables, categories, cell counts, marginal totals, margin tables and their cells and whether those margin cells are disclosive or not. A margin cell is deemed disclosive if only one cell in the main table contributes to it. The corresponding index of the cell in the main table is provided as a property of the margin cell.

## querytest rules

querytest rules define the criteria for a query to be accepted and a candidate output table to be generated. We use rules before we generate an output table to save computation time and to alert the user early that their proposed table is not allowed.

An example of why a table might not be allowed is that the combination of variables would lead to a table with too many cells which is thus inherently sparse.

A simple example of some querytest rules in our language is shown below:

querytest main()  
 fail if not checkMaxVars(4,6)  
 fail if not checkMaxCells(10\*1000\*1000)  
end

querytest checkMaxVars(maxSmall, max)  
 if (query.vars[0] sourceof "SMALL\_GEO") &&  
 ((len query.vars) > (maxSmall + 1))  
 fail "Maximum vars at SMALL\_GEO is " & maxSmall  
 end

if "SMALL\_GEO" sourceof query.vars[0]  
 if (len query.vars) > (max + 1) // any geographic query  
 fail "Maximum vars for geographic query is " & max  
 end  
 else // not a geographic query  
 if (len query.vars) > max  
 fail "Maximum variables in query is " & max  
 end  
 end  
end

querytest checkMaxCells(max)  
 var numCells = 1  
 for v in query.vars  
 numCells \*= (len v.cats)  
 end  
 fail "maximum table size exceeded" if numCells > max  
end

## tabletest rules

tabletest rules are applied to a candidate output table or a sub-table (i.e. for a specific geographic area). They are always evaluated before any filtering of variable categories requested by the user so that the output is consistent across any filtered views.

These rules can measure various aspects of disclosivity, for example sparsity and dominance. They automate decisions which might have previously been made by statisticians using manual analysis techniques.

Additional program clauses are provided to define filtered views of the output tables (“tabledef”) for tabletests to inspect and computed properties (“tableprops”) of the tables to test against.

A typical use case for a filtered view is to exclude “not applicable” categories. An example use case for a table property might be to count the number of zeros in a table.

An simple example of some tabletest rules in our language is shown below:

tabletest main()  
 fail if not MaxPermilleZeros(718)  
 fail if not MaxMarginalTotalPermille(828)  
 fail if not MaxPermilleDisclosiveCells(45, 2, 5)  
end

// define a filtered table view without "Not Applicable" categories  
tabledef tableWithoutNA  
 for c in table.cats  
 fail if c.code eq "N/A"  
 end  
end  
  
tableprops total, nZeros  
for cell in tableWithoutNA.cells  
 total += cell.count  
 if cell.count == 0  
 nZeros += 1  
 end  
end  
  
tabletest MaxPermilleZeros(perMilleCells)  
 fail if (tableWithoutNA.nZeros \* 1000) >  
 (perMilleCells \* (len tableWithoutNA.cells))  
end

tabletest MaxMarginalTotalPermille(maxPerMilleTotal)  
 var maxTotal = (maxPerMilleTotal \* tableWithoutNA.total) / 1000  
 for c in tableWithoutNA.cats  
 fail if c.total > maxTotal // c.total is total count for category  
 end  
end  
  
// check maximum fraction "per-thousand" of disclosive cells  
// (if it is the sole contributor to at least one margin cell)  
  
tabletest MaxPermilleDisclosiveCells(perMilleCells, minCats, maxCount)  
 var threshold = (perMilleCells \* (len tableWithoutNA.cells)) / 1000  
 var count = 0  
 for margin in tableWithoutNA.margins  
 continue if (len tableWithoutNA.vars[margin.index].cats) < minCats  
 for cell in margin.cells  
 continue if cell.disclosiveCellIndex < 0  
 continue if cell.count > maxCount  
 continue if marked tableWithoutNA.cells[cell.disclosiveCellIndex]  
 mark tableWithoutNA.cells[cell.disclosiveCellIndex]  
 count += 1  
 fail if count > threshold  
 end  
 end  
end

# Conclusion

We have successfully created working software which implements a new language that allows disclosure control experts to independently specify table publication rules in a safe manner within a flexible dissemination system. The language has been successfully tested with proposed rules for a census publication system.

# **References**

1. Blanchard, S.: The methodological challenges of protecting outputs from a Flexible Dissemination System. Survey Methodology Bulletin 79; 1-15. (2019)
2. Martin Fowler and Rebecca Parsons: Domain-Specific Languages. Addison-Wesley Signature Series (2011)