RAPID: RAPid Interpretation of Data

A.D. Pimentel L.O. Hertzberger

Dept. of Computer Science
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam
The Netherlands
{andy,bob}@fwi.uva.nl

Abstract

The RAPID tool facilitates the analysis and the visualization of data residing in arbitrary text files. It differs from other analysis applications in the way it selects and operates on data. In RAPID, data elements are directly retrieved from the unstructured text files using pattern-matching. For this purpose, we use the Perl language. This language was especially designed for efficient text scanning and thus provides powerful support for pattern-matching. Moreover, the fact that Perl is an interpreted script language enhances our tool's extensibility and portability significantly. In this paper, we describe the RAPID tool and discuss some of its design and implementation issues.

1 Introduction

Computer experiments, such as simulations, typically generate a vast amount of output. Since manual interpretation of the data is generally not feasible, it is important to have the disposal of good analysis tools. Using all kinds of statistical methods and visualization techniques, these tools are essential for the successful interpretation of the experimental data. During the past few decades, many analysis applications, such as S [1], SPSS [5] and SAS [2], have evolved. They all allow for a thorough, interactive, statistical analysis of data. Once these programs have read in the data, the user can "play" interactively with it. For many purposes, this is a natural and flexible approach.

Despite the existence and the power of these tools, we felt the need for a less sophisticated tool that facilitates quick, batch-oriented, analysis of arbitrary text-based data. This need originates from the computer architecture simulation work that is being performed within the Mermaid project [7, 6]. The output of the Mermaid simulators is unstructured text and is completely defined by the user. This implies that every time a new simulation is constructed, a *filter* program has to be written in order to extract the required numbers from the simulation output for further analysis. Typically, these filters are shell scripts utilizing the basic Unix tools such as `grep`, `awk` and `sed`. Furthermore, the interpretation of the data can often be performed using simple statistical methods only. As a consequence, the powerful, but sophisticated, analysis applications are often seen as overkill and therefore not frequently used or not used at all. Instead, the analysis is directly programmed into the filters. This is undesirable because the writing of such filter programs can be quite tedious, especially when this has to be done every time the format of the simulation output changes.

In this paper, we present the RAPID (RAPid Interpretation of Data) tool which allows for the automatic generation of filter programs. A simple language is provided in which the user specifies what statistical methods should be used on which parts of the simulation data. According to this specification, RAPID generates the analysis filter. Since the output format of the Mermaid simulators is not formally defined and can thus assume any shape, pattern-matching is used as a data selection mechanism. As we will show, the use of pattern-matching provides for a great amount of flexibility and expressiveness.

RAPID is not intended to be a competitor of the previously listed analysis applications. It merely is an alternative tool with which unstructured text-based data can be analyzed swiftly in the cases where using a more sophisticated application would require too much effort. Here, the term effort relates both to the learning cycle of a large software package and to

the work that should be done in order to supply the application with data in the correct input format. Of course, RAPID can always be used to just extract the required data from the simulation output after which an application like SPSS is used for a more extensive analysis.

The next section gives an overview of the basic framework of RAPID, emphasizing the portability and the extensibility of the tool. Section 3 describes how filters are specified and demonstrates the power of pattern-matching within the specifications. In section 4, the available (statistical) operations and visualization techniques are discussed. Section 5 describes the optimization techniques applied to obtain better analysis performance. Finally, Section 6 concludes the paper.

2 The RAPID tool

The RAPID framework consists of a specification language, a compiler and a function library. This is illustrated in Figure 1. The specification language is a simple rule-based language. It allows the user to select areas of interest within the simulation output and to specify (statistical) operations that should be performed on the selected areas of the data. A more detailed description of how this works is given in the next section.

The compiler translates the specification into an executable shell script. Both the compiler and the generated script are written in Perl [8], which is an interpreted language optimized for scanning arbitrary text files. Using this language has several consequences. Perl programs do not need to be compiled; they can be readily executed (i.e. interpreted) whenever Perl is available on the host system. So, since Perl is available on many platforms, this makes RAPID a highly portable tool. Moreover, Perl provides for a powerful support of pattern-matching. This feature is transparently embedded into RAPID as well. On the other hand, a disadvantage of using Perl might be its inefficiency with respect to certain computational functions. For instance, Perl scripts are not really suited for matrix computations necessary for some analysis techniques. We tolerate this inefficiency however, since the occasions in which we need this kind of computations are reasonably scarce. Besides, if efficiency is the primary objective, then it is possible to implement the number-crunching functions in a C library which is then linked to the Perl analysis filter.

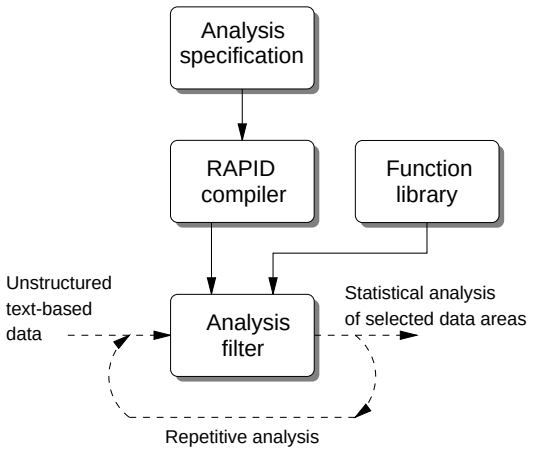

Figure 1: The RAPID framework.

Since we want RAPID to be easily extensible, a separate Perl library contains the functions that can be applied to the data. New, user-defined, functions can simply be added to the library. When this has been done, the user only needs to add a prototype of the new function specifying the number and the types of its arguments after which the compiler is ready to use it. Besides some basic functionality, the current library mainly contains functions that are commonly used in the performance analysis of computer systems [3].

The function library together with the compiler-generated script form the analysis filter. Because the quick, possibly repetitive, analysis of (multiple) data files is a requirement, the filter operates in a batch-oriented manner. This implies that the specified types of analysis are performed one after each other. So, user interaction was not considered to be a necessity.

3 Specifying the analysis

The language that specifies the required analysis is a relatively straightforward rule-based language which does not contain assignments, explicit[1] loop constructs or recursion. There is a slight resemblance with the expressions of formulas that can be constructed within spreadsheet applications. The RAPID language is, however, directly referring to data items within the text files and not to items in a worksheet.

A specification consists of one or more rules where the syntax of a single rule can be described as

[1] The language has a notion of implicit looping, as will be explained further on.

```
<desc> <output-format> [id] --> <expr> ;
```

Here, <desc> is a string describing the purpose of the rule and <output-format> is a format specifier like it is used in the C function `printf`. Thus, "%d" means that the output should be an integer. The field [id] is an optional string and is used as a tag when the output contains more than one result. Each result is tagged with [id] and a sequence number. Finally, the <expr> field specifies what operations should be performed on which data. To illustrate the above, consider the following output of a simple network simulation.

```
Elapsed time: 1667288 time units
Node 0: bytes sent: 99000922
Node 1: bytes sent: 90665112
Node 2: bytes sent: 97999003
Node 3: bytes sent: 87231230
```

The line "Elapsed time" indicates the simulated time, whereas the remaining four lines refer to the number of bytes each node has sent within the time interval. This sample simulation output will be used throughout this paper for illustrative purposes.

To calculate the total network throughput, the following rule could be used:

```
Network throughput %.2f -->
   sum("bytes sent:#") / "Elapsed time:#";
```

In this example, the string "Network throughput" is the description. It appears in front of the result in the output. The result is specified to be a floating point number having two digits after the comma. At the right-hand side of the arrow the actual operation is given. The data file is first searched for numbers preceded by the string "bytes sent:". Here, the # stands for the required number, which may be in any format (integer, floating point, etc.). Of the resulting list of numbers the sum is taken. Subsequently, the numbers with the preceding string "Elapsed time:" are selected. In this case, we assume that only a single number is matched. Dividing the total number of bytes sent by the (simulated) time gives us the throughput, which is the final result. After compilation of this specification and using the above data file as input for the generated analysis filter, the following output is shown:

```
Network throughput = 224.85
```

From this example can be seen that pattern-matching plays an essential role within RAPID specifications. It provides a natural and powerful interface for selecting data items.

Because of the simplicity of the specification language, it only contains two types of data to which functions can be applied: numbers and lists of numbers. Single numbers can be immediately specified within expressions or can be the result of a pattern-match (like the `"Elapsed time:#"` in the previous example). A list of numbers can only be obtained by a pattern-match. As a consequence of this simple typing-scheme, functions can easily be type checked, which improves the user-friendliness of the tool considerably.

3.1 Data selection

Data can be selected on the basis of either their appearance or their contents. Selection on the basis of appearance is done by pattern-matching, as was already illustrated in the previous section. Patterns must be specified between quotes and should contain the "#" character indicating the location of the required number. RAPID allows for a transparent use of the complete pattern-matching functionality of Perl. So, to get the number of bytes sent by the nodes 0 and 1 from the network simulation output, for instance, a pattern like

```
"Node [0-1].*sent:#"
```

could be used. In this pattern, the `.*` skips all characters between the node number and the word "sent".

Selecting data on the basis of their contents is performed by special selection operations that are available within the language. For example, to select transmission quantities of more than 90 Mbytes only, the expression

```
gt("bytes sent:#", 90*1024*1024)
```

can be applied. As one would expect, the function `gt` stands for "greater than". In a complete rule, this may look like

```
Nodes > 90Mb %d -->
   count(gt("bytes sent:#", 90*1024*1024));
```

Clearly, this particular rule counts the number of nodes having sent more than 90 Mbytes. A more detailed overview of the available functions, including the selection operations, is presented further on in this paper.

3.2 Analyzing multiple data files

To analyze a data file, its filename is passed as an argument to the filter program. When examining multiple data files and thus passing more than one argument to the filter, operations are by default performed globally. In other words, the multiple data files are essentially seen as one large data file. It may occur, however, that "local" operations must be performed on a subset of the data files. For this purpose, a tagging mechanism is used. A tag is either an integer or a range of integers. It is glued together with a (search-)pattern using a colon, thereby specifying the data file(s) in which the search must be performed. This means that the tag equals to the filter's argument number(s) from which should be read. To illustrate this, consider the following example in which a speedup is calculated:

```
Speedup in percents %.2f -->
      100 * (1:"Elapsed time:#" /
             2:"Elapsed time:#");
```

In this example, a tag numbered 1 is glued to the first pattern and a tag numbered 2 to the second pattern. Consequently, the rule computes the speedup using the simulated times found in the data files referred to by the filter's first and second argument.

As mentioned above, a tag can also be a range of integers. Such a range tag is given by the tuple $a.b$ where a and b denote the lower and upper bounds respectively. Here, a and b are either an integer or the special $ character which stands for "last file". Furthermore, both a and b must be in the domain of [1,$] with $a \leq b$. When using the $a.b$ tag in a rule, the rule is performed $b - a + 1$ times. In iteration i, with $0 \leq i \leq b - a$, the range-tagged patterns are applied to the data file referred to by the filter's $a + i$th argument. So, basically, these tags allow for implicit looping of rules. For example, the rule

Selection functions	Basic computational functions
lt(*pattern*, *number*)	sum(*pattern*)
gt(*pattern*, *number*)	prod(*pattern*)
eq(*pattern*, *number*)	max(*pattern*)
leq(*pattern*, *number*)	min(*pattern*)
geq(*pattern*, *number*)	count(*pattern*)
	abs(*pattern*)
	floor(*pattern*)
	ceil(*pattern*)
	log(*pattern*, *base*)
	diff(*pattern*, *pattern*)
Statistical functions	
{a,g,h}mean(*pattern*)	
{a,g,h}standev(*pattern*)	
median(*pattern*)	
variance(*pattern*)	
correl(*pattern*, *pattern*)	
quantile(*pattern*, *fraction*)	
fit(*pattern*, *pattern*, *function*)	
meanconf(*pattern*, *confidence*)	
cmp_[un]paired(*pattern*, *pattern*, *confidence*)	
ml_regres(*pattern*,...,*pattern*)	

Table 1: An overview of the available analysis functions.

```
Maximum sent %d -->
      max(1.$:"bytes sent:#");
```

determines the maximum number of bytes sent by a node for all data files mentioned as a filter argument. So, if three filenames were passed as an argument, then the result would consist of three maxima.

4 Function types

To give an impression of the functionality of the RAPID specification language, this section presents an overview of the available analysis and visualization functions. The function library may seem relatively limited, but it is sufficiently equipped for the types of analysis required in the Mermaid project. Besides, as was explained in Section 2, the library can easily be extended with new functions when this is needed.

Table 1 lists the analysis functions available in the language. The parameters called *pattern* refer to pattern-matching strings resulting in a list of one or more numbers. The other parameters indicate single

numbers, unless specified otherwise. With a *selection function*, data can be selected on their contents. In general, these functions compare the data to a specific number and return a list of numbers that satisfy the given selector function.

As the name already suggests, a *basic computational function* performs a straightforward operation on a list of numbers. The upper five functions in the list of this category return a single number, whereas the remaining five return a list of numbers. We will not describe these functions in detail as they all are more or less self-explanatory.

The category of *statistical functions* requires a little bit more explanation. Calculating a mean can either be done arithmetically (*amean*), geometrically (*gmean*) or harmonically (*hmean*). The same holds for computing of standard deviations. A regression model for linear or curvilinear functions can be calculated using the *fit* function. The first two parameters of this function contain the observed x and y values while the third parameter specifies the function to which the data should be fitted. The latter is a string that equals to one of the strings in the left column of Table 2. For example,

```
Cache-model %.2f -->
   fit("cache size:#",
       "hitrate:#", "y = a + bx");
```

computes a fit to the function $y = a + bx$ for data from a cache simulation. Here, the matched cache sizes (x values) have a one-to-one correspondence with the matched hit rates (y values). Besides calculating the regression parameters a and b, the goodness of the fit and some confidence intervals of the computed parameters are calculated as well.

Using the function *meanconf*, confidence intervals of averages are calculated. The associated *confidence* parameter specifies the required level of confidence in percents. Comparison of two alternative observations can be performed by either the *cmp_paired* or the *cmp_unpaired* function. The first one deals with paired observations, in which there is a one-to-one correspondence between each pair of observations selected by the two pattern-matching strings. For instance, when comparing two computer systems by measuring the run-time of different workloads, the nth workload executed on both systems results in two paired observations. Evidently, when there are no such correspondences, the *cmp_unpaired* function is used instead. This operation is formally called the t-test. Finally, *ml_regres* is similar to the *fit* function as it computes a multiple linear regression model. This means that multiple predictor variables (x values) are allowed. The *ml_regres* operation fits data to the linear equation

$$y = b_0 + b_1 x_1 + b_2 x_2 + \ldots + b_k x_k + e$$

Here, $b_0, b_1, ..., b_k$ are $k + 1$ regression parameters to be computed and e is the error term. Like the *fit* function, *ml_regres* also returns several statistics on the goodness of the fit and on the confidence intervals of the regression parameters. Moreover, a so-called *Analysis Of Variance* (ANOVA) is performed as well. This analysis, which applies the well-known F-test, essentially determines whether or not the model is good enough to explain a significant fraction of the response variation (the y variable).

Function string	Meaning
y = a + bx	$y = a + bx$
y = a + b/x	$y = a + \frac{b}{x}$
y = 1 / (a+bx)	$y = \frac{1}{a+bx}$
y = x / (a+bx)	$y = \frac{x}{a+bx}$
y = abˆx	$y = ab^x$
y = a + bxˆn	$y = a + bx^n$
y = bxˆa	$y = bx^a$

Table 2: Functions that may be used in the *fit* function. In this list, any integer may be substituted for the character n.

Visualization functions
create_graph(*pattern*, *pattern*, *graph_type*)
add2graph(*pattern*, *pattern*, *graph_type*)
hist(*pattern*, *cell_size*)
add2hist(*pattern*, *cell_size*)
quantnorm_plot(*pattern*, *average*, *stan_dev*)
quantexp_plot(*pattern*, *average*)
drawgraph()

Table 3: An overview of the available visualization functions.

4.1 Visualization functions

The functions which allow for drawing of graphs are listed in Table 3. These functions generate a separate data file (containing the data to be visualized) and a plotfile. The latter is used as input for the widely-used (interactive) plotting program Gnuplot [4]. After generation of a graph, the user is free to adjust

the Gnuplot file to regenerate the graph or to start an interactive visualization session with Gnuplot. So, providing an interface towards Gnuplot enhances the flexibility considerably.

Currently, RAPID only supports the construction of two-dimensional graphs, which suffices for the Mermaid analysis work. The *create_graph* and *add2graph* functions allow for the generation of a whole range of different graph-types. The first two pattern parameters contain the observed x and y values respectively, whereas the third parameter specifies the required graph-type. This type is a string which equals to one of the plot-styles of Gnuplot (lines, points, linespoints, etc.). The *create_graph* operation initializes a new graph while *add2graph* adds new curves to an existing graph, thereby allowing multiple curves per graph. After applying one of these functions, the ranges of the x and y axes are automatically (re-)computed. Furthermore, a graph is said to be *active* in between calling the *create_graph* and the *drawgraph* functions. The latter operation actually generates the Gnuplot file. So, all intermediate uses of *add2graph* are applied to the active graph.

To give an example, consider the following three rules. They generate a graph consisting of multiple curves, where each curve refers to a different data file.

```
Graph %d -->
    create_graph(1:"cache size:#",
                 1:"hit rate:#", points);
Graph cont'd %d -->
    add2graph(2.$:"cache size:#",
              2.$:"hit rate:#", points);
Draw graph %d --> drawgraph();
```

With the *create_graph* function, the graph is initialized and the curve based on the first data file is drawn. Subsequently, *add2graph* adds all the curves from the remaining data files by using the "2.$" tag. Finally, the graph is generated (i.e. the Gnuplot file is produced) by means of the *drawgraph* operation. The format %d found in all three rules specifies that the data to be visualized should be handled as integers. Clearly, this example illustrates the convenience of range tags: only three rules are needed to draw a graph consisting of two or more curves.

The *hist* and *add2hist* functions are similar to *create_graph* and *add2graph* respectively, but they can only produce one type of graph: a histogram. The parameter called *cell_size* determines the resolution as it specifies the size, and inherently to this, the number, of cells used in the histogram.

With *quantnorm_plot* and *quantexp_plot* quantile-quantile scatter plots for the normal and exponential distributions respectively are generated. In these plots, the quantiles of the observed data are shown against the theoretical quantiles of the selected distribution. By doing this, the distribution of the observed data can be determined. The *average* and *stan_dev* parameters specify the average and, if appropriate, the standard deviation of the theoretical distribution. A more comprehensive explanation of the statistical methods mentioned in this section can be found in [3].

5 Function and list caching

Although RAPID analyzes data files in a batch-oriented manner, which may suggest that analysis efficiency is not of great importance, we applied several optimizations to deliver good analysis performance. To minimize file I/O, for instance, data files are by default analyzed "from memory". This means that the generated filters read all the data at the start of the analysis after which all operations are applied to the data stored in memory. Clearly, this approach trades efficiency for memory usage. However, analysis "from disk" is also provided in the case the data files are too large to fit in memory.

A more sophisticated performance optimization involves the removal of common subexpressions within analysis specifications. It regularly happens that certain data selections are performed multiple times or, even worse, that identical functions are applied to the same data more than once. For instance, the rules

```
Maximum sent %d --> max("bytes sent:#");
Average sent %d --> amean("bytes sent:#");
Standard deviation %.2f -->
              astandev("bytes sent:#");
```

would perform overlapping work if no precautions are taken. First, the number of transferred bytes is extracted from the data file three times in a row. Second, the function *astandev* needs to calculate the mean before computing the standard deviation. However, this has already been done in the previous rule (containing the *amean*). So, to prevent the same work being performed multiple times, the RAPID filters cache the results of functions and pattern matches whenever this is required. For the above example this would mean that the pattern match is only performed once, after which every succeeding

matching-request is serviced by the cache containing the resulting list of numbers. Furthermore, the computed mean of the second rule is also cached and directly used in the calculation of the standard deviation in the third rule. The difference in precision of the second and third rule, i.e. %d versus %.2f, is not a problem here. The analysis filter always performs calculations based on Perl's *number* type, which is internally represented as a double precision floating point value.

The RAPID compiler has to find out which of the (sub)expressions are used multiple times. Only these expressions will be cached. Thus, there will never be an entry within the cache that is not reused. Moreover, the entities allowed to be cached essentially include all the (sub)expressions appearing at the right-hand side of the arrow in a specification rule. So, if one, for instance, would specify two identical rules, then only one rule is really computed while the other is serviced entirely from the cache.

However, the use of regular expressions within patterns may complicate the search for cacheable pattern-matches considerably. For instance, the patterns

`"Node.*sent:#"` and `"bytes sent:#"`

will extract the same data for the sample simulation output presented in Section 3. But when both patterns are used within a specification, the compiler cannot see that they are identical. It is therefore the user's responsibility that such aliasing problems do not occur.

Like the minimization of file I/O, caching also involves a tradeoff between efficiency and memory usage. The caching may require large quantities of buffer space, which can pose a problem when the analysis filter runs out of memory. Therefore, cached results can be removed from the cache whenever they are not needed anymore. If this is not enough to fit objects in the cache, caching may be turned off entirely.

6 Conclusions

In this paper, we presented the RAPID tool which allows for swift analysis of data residing in arbitrarily structured text files. The analysis approach of RAPID is rather different from other analysis applications as it directly operates on the unstructured text files instead of reformatting the data first to a particular standard format, e.g. a spreadsheet-like format. Moreover, the tool was kept as simple as possible to allow easy utilization. For this reason, RAPID analyzes in a batch-oriented manner rather than it provides interactivity with the user. As a consequence, we can quickly perform many, often straightforward, types of analysis without requiring much effort.

Since RAPID operates on unstructured text, the language Perl is used for efficient scanning of the text data. Consequently, data items are selected by means of pattern-matching. As we have shown, this selection mechanism is natural and provides great flexibility. Using the script language Perl also implies that our tool is portable and easily extensible. With respect to the latter, RAPID features a separate function library which can simply be extended by the user.

For the work being done within our department involving the simulation of computer architectures, RAPID proves to be a convenient tool. Its, seemingly limited, analysis functionality has sufficed our analysis requirements so far. Nevertheless, the tool will undoubtedly be enhanced with new features in the future.

References

[1] R. A. Becker and J. M. Chambers. Design of the S system for data analysis. *Communications of the ACM*, 27(4):486–495, May 1984.

[2] J. T. Helwig. *SAS : Introductory Guide*. Raleigh, NC: SAS Institute Inc., 1978.

[3] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[4] A. Liaw and D. Crawford. *Gnuplot 3.5 User's Guide*, Nov. 1994.

[5] P. Lovie. Statistical software for microcomputers - SPSS/PC. *Journal of Mathematical and Statistical Psychology*, 41(4):151–154, 1988.

[6] A. D. Pimentel and L. O. Hertzberger. An architecture workbench for multicomputers. In *Proc. of the International Parallel Processing Symposium*. IEEE Computer Society Press, April 1997.

[7] A. D. Pimentel, J. van Brummen, Th. Papathanassiadis, P. M. A. Sloot, and L. O. Hertzberger. Mermaid: Modelling and Evaluation Research in MIMD ArchItecture Design. In *Proc. of the High Performance Computing and Networking Conference, LNCS*, pages 335–340, May 1995.

[8] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.