METRIX : a new tool to evaluate the quality of software source codes

Antoine Varet, Nicolas Larrieu, Léo Sartre

METRIX: a new tool to evaluate the quality of software source codes

Antoine Varet[1], Nicolas Larrieu[2] and Léo Sartre[3]
Ecole Nationale de l'Aviation Civile, laboratoire TELECOM
7 avenue Edouard Belin
31055 Toulouse, France

In this paper we will try to answer the question: how to evaluate the quality of software automatically produced by code generation process in the domain of aeronautical embedded systems? To do so, we will introduce our new Open Source tool « METRIX » capable of computing different software quality metrics. Each of them gives information on quality of both the source code and the binary software for the embedded system we want to assess. METRIX is able to evaluate software written in C and ADA languages. There is a specific module which is able to compute the most efficient line of compilation in order to minimize size of code or time of execution for the final binary. Lots of metrics can be considered for software evaluation but in this paper, we will discuss only the three most popular: Lines of Code (LoC), metrics of Halstead (volume of source code, complexity of the algorithm) and cyclomatic complexity of McCabe.

This research may help software engineers to improve their aeronautical system verification and validation process and this paper will give them a complete overview of how METRIX software works, how it produces a quality software comparison and how it proposes enhanced visualization features (Kiviat and City map diagrams for instance). A specific aeronautical case study (secure embedded aeronautical router) will be discussed and will demonstrate how this new software can improve the verification and validation steps of a complete industrial project.

I. Introduction

A. Software development context for aeronautical embedded systems

Software development costs for aeronautical embedded systems are increasing together with their complexity. Thus, to decrease development costs engineers use more and more model converters to generate automatically embedded source codes. The time of code development is decreased but not the amount of line of code generated. Consequently, aeronautical engineers face a difficult question: how to ensure quality and performance of their system taking into account this increase in software complexity? One of the answers is to investigate software quality by assessing **the quality of software automatically produced by code generation process**.

To do so, we will introduce our new Open Source tool « METRIX » which can compute different software quality metrics. Each of them gives information on quality of both the source code and the binary software produced. METRIX is able to evaluate software written in C and ADA languages and many metrics can be considered for software evaluation (the different metrics will be described in detail in the next section).

This research may help software engineers to improve their aeronautical system verification and validation process and this paper will give them a complete overview on how METRIX software works, how it produces a quality software comparison and how it proposes enhanced visualization features (Kiviat and City map diagrams for instance). A specific aeronautical case study (secure embedded aeronautical router) will be discussed and will

[1] PhD Student, SINA Department, avaret@recherche.enac.fr
[2] Assistant Professor, SINA Department, Nicolas.larrieu@enac.fr
[3] Assistant, SINA Department, Sartre@recherche.enac.fr

demonstrate how this new software can improve the verification and validation steps of a complete industrial project.

B. Design methodology for software fast prototyping

In our previous research, we introduced a new methodology to accelerate the embedded software development with two sets of optimization goals. The first set is the minimization of costs and delays for design, source code writing, certification and evaluation. The second set concerns maximization of the safety assurance level in the confidence of the final software. This methodology summarized in figure 1 has been extensively explained in Ref. 1.

The use of model transformers and qualified tools is an improvement for the development and the certification of the final product. Their use produces a formal demonstration of the benefits; and the global development process thus becomes more efficient.

The methodology can be summarized as followed:

1) During the partitioning process, the different requirements are split into logical groups. Each group is referred to as a partition.
2) Then each group of requirements is modeled.
3) The third step is to convert automatically the models into source codes with a qualified tool named transformer; this code is independent of both operating system and hardware constraints.
4) During the fourth step, a piece of code is added in order to make a link between the inputs and outputs of the model-based generated code and the I/O provided by the operating system. This code may be manually written or automatically generated.
5) Following this, all source code files are compiled into a binary form.
6) Binary object files are then aggregated together during the integration step into one binary image per partition defined in the first step.
7) This binary image may be loaded into the hardware embedded target or tested with an emulator.

This methodology has already been successfully applied to elaborate and develop an embedded next generation router. We are now focusing research on new improvements in methodology.

First of all, usage of different and new tools but also new languages may be investigated. Additional sub-steps to evaluate qualitatively and quantitatively the quality of source codes may also be proposed. This quality assessment may help, for instance, to improve the models and the methods applied to write the different models and to generate source files. This paper will be specifically concerned with this last issue; we will try to answer the question: **how to assess the quality of software automatically produced by code generation process?**

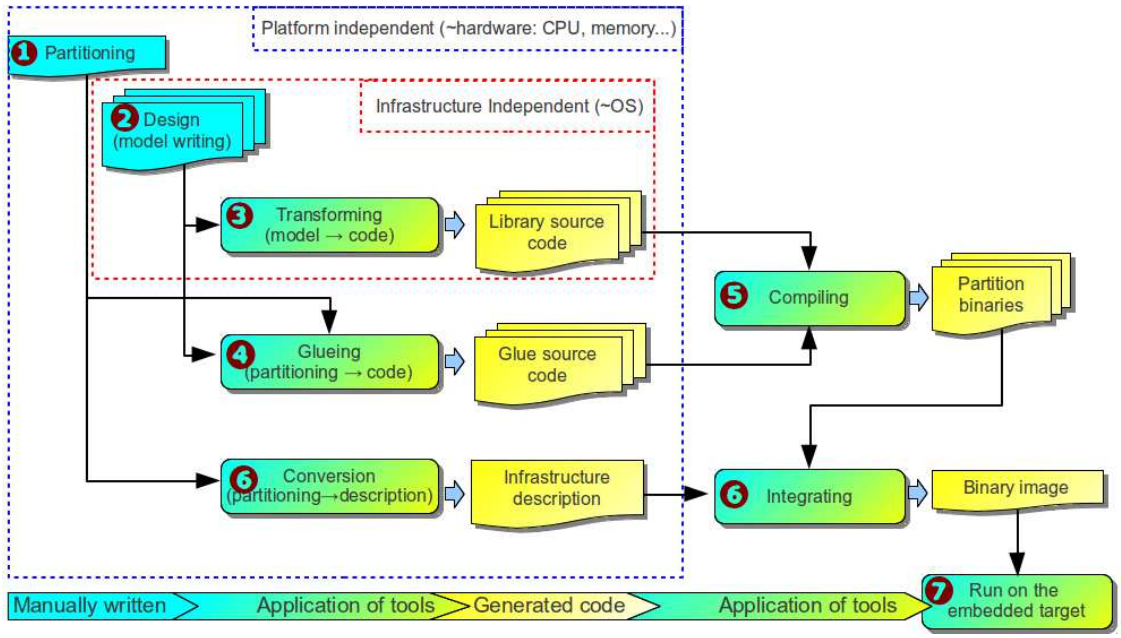

Figure 1. Development methodology for software fast prototyping

C. How to assess software quality?

When considering a model driven approach for software development, there are two levels of software assessment quality. The first one is model-based assessment and the second is code-based assessment. According to the DO-178C standard, model driven development seems to be an interesting track of investigation for future aeronautical system developments (see Ref. 2 for details). Nevertheless, this is still a work-in-progress area for the aeronautical domain. Let us cite Ref. 3 which represents a good example of model based assessment for automotive domain. Their approach could be considered as an interesting starting point for aeronautical embedded systems model assessment.

D. State of the art of code-based quality assessment tools for software development

Most aeronautical industrial projects are working on software code assessment. Different software is available to produce assessment results. Let us cite SourceMonitor[4], Sonar[5], McCabeIQ[6], Polyspace[7] or RSM[8] which represent the most familiar set of tools for the aeronautical industry. SourceMonitor[4] is an open source software with interesting representation choices for quality metrics, but it is a non multiplatform software and which lacks a lot of documentation. Sonar[5] is an open and generic platform to manage code quality (used most of the time for project management). Consequently, Sonar[5] is not really suited to our specific aeronautical embedded system development. References 6, 7 and 8 are industrial software with an expensive license cost. Moreover, we judged them insufficiently flexible for our research needs. They do not seem suitable to be integrated with the development methodology we introduced previously.

Based on this analysis of software quality assessment tools we decided to develop our own code-based quality assessment tool. This software called "METRIX" is able to assess the quality of C and ADA languages and can compute different software quality metrics (each of them is defined in detail in the next section): Lines of Code (LoC), metrics of Halstead (volume of source code, difficulty of the algorithm...) and cyclomatic complexity of McCabe. Moreover, different visualization features are proposed (Kiviat or City map diagrams for instance) that can help improve software quality. We will explain this in the final part of this paper in which we will discuss a specific study case with secure embedded aeronautical router design and development.

In the following sections of this paper, we will first introduce the different quality metrics METRIX is able to assess for a set of codes. Then we will present the different visualization features that can be used for such an assessment. And finally, we will demonstrate through a case study (the software development of an embedded secure router); how METRIX features may help system designers to improve the quality of the final software they want to produce for their real embedded target.

II. Software quality metrics implemented in METRIX

Numerous metrics enable developers to evaluate source code. Reference 9 enumerated more than 200 different metrics. Our main research objective is to assess the quality of software source codes. This is why the first class of metrics we selected is language dependent. This is the quickest way to assess the performance of codes. In a second stage, we considered Halstead and McCabe complexity metrics, which evaluate the complexity of algorithms. Thus the next three subsections describe sequentially the different parameters we took into account for such an analysis.

A. Language-dependant metrics

Our first class of metrics regroups traditional metrics related mainly to the language the developer used to write his software. They allow fast assessment on the performance of a source code.

The first one is the **Lines Of Code (LoC)**. It counts the number of lines for the different files of the entire development project. This metric is probably the most commonly used, mainly because it is easy to understand and to compute. It provides an estimation of development time and costs for the user of this metric. But this metric has a major drawback: its computation may differ according to the way the developer implements this metric. For instance, how to perform an "#include" in language C? Does the counting program count recursively the number of lines in included files or count only one line for this preprocessor instruction? A similar problem exists with the Ada language: how to compute Ada standard package importation?

This is why different variations have been created to deal with this issue:

- **Effective Lines Of Code** (eLoC)[10];
- **Logical Lines Of Code** (lLoC)[10];
- **Code readiness ratio**[10];
- **Function length** (LoC per function)[11];
- **Number of function call**[11].

Effective Lines Of Code (eLoC) is a refinement of the first metric (LoC). This metric excludes from the counting all the lines that are not associated with binary processor instruction. For instance, comment lines and empty lines are excluded from the counting, and also lines with just a brace ("{" and "}").

The Logical **Lines Of Code (lLoC)** is a more restrictive metric than the Effective Lines Of Code. This metric counts only the number of instruction lines, excluding all declarations and all data structure lines, contrary to the Effective Lines Of Code metric.

The **code readiness ratio** is more complex to perform. Here we count the number of lines with comments, the number of spaces and tabulations and we report these counts to the LoC and to the total number of characters of the source code. Most developers considers a lighten source code is easier to read, debug and maintain than a condensed and poorly commented source code.

Function length (aka LoC per function) is a metric useful to improve the readiness of source code. The Linux Kernel Coding Size[11] suggests making a maximum of function in order to reduce each function to one task/objective. However, this metric should be used in conjunction with the **cyclomatic complexity**, another metric defined later in the next section. Consequently, coding style rules imply that low complexity function may be longer and highly complex functions should be short, in order to maintain the product of complexity and length under a user-defined ceiling. The general rule for the function length is to maintain each function under 24 lines of at most 80 characters each. Thus, the function can appear entirely on a terminal or a screen.

The last metric we will introduce in this section is the **number of function call**. This metric can provide profiling information to guide the development team for optimizing their code. A naïve approach considers that a function that is called often requires optimization effort. Nevertheless, in practice, only the interpretation of the code and the execution of the software enable developers to identify which functions spend most of the CPU time and how much CPU time they spend. Some tools are dedicated to this kind of profiling, for instance Valgrind[12].

These metrics are widely used but have one main drawback: with these metrics, you can compare two development projects **if and only if** both have been written with the same language. For example, two projects based on the C language can be compared in terms of line of code. If one is written in Ada and the other is written in C, then the comparison does not provide any useful information.

This is why in the two next sub section we introduce performance metrics related to algorithmic complexity (Halstead and McCabe metrics) and not only the development language used by software engineers.

B. Halstead metrics

Professor Maurice Halstead presented in 1977 different metrics such as the code volume or its algorithmic difficulty, detailed in Ref. 13. These metrics are based on items found in all textual languages of imperative programming. Thus, they depend little on the language. These items are the operators and the operands. The operators are the reserved key-words (for example «+» «*» « >> » «&» in the language C). The operands are the parameters of the operator; it may be numerical values, names of constants or variables, etc. The first four parameters defined by Halstead are:

- **n1** (the **number of different operators**);
- **N1** (the **total number of operators** in the source code);
- **n2** (the **number of different operands**);
- **N2** (the **total number of operands** in the source code).

These four «primary metrics» are measured on the source code, per function, file or set of files. From these metrics, we can derive other metrics:

- **N** (= N1 + N2) is the **size of the program**;
- **n** (= n1 + n2) is the **size of the dictionary**;
- **V** = N * log_2 (n) is **Halstead's Volume** and reports the size of the implementation of the algorithm. Conforming to Halstead's recommendations, V should be between 20 and 1000 for each function. V > 1000 is an indication that the function is too much complex and should be divided into sub-functions.
- **D** = (n1 / 2) * (N2 / n2). D is **Halstead's Difficulty** of the algorithm, also called error-inclination;
- **E** = D * V is called **Halstead's implementation Effort**;
- **T** = E / 18 is **Halstead's Time estimation required for implementation** (in seconds). Maurice Halstead defined the constant 18 by measuring the number of elementary operation per second a human brain can perform.

All these metrics may be used to compare two source codes, and contrary to the metrics of the previous section, Halstead's metrics may be used for any imperative programming language.

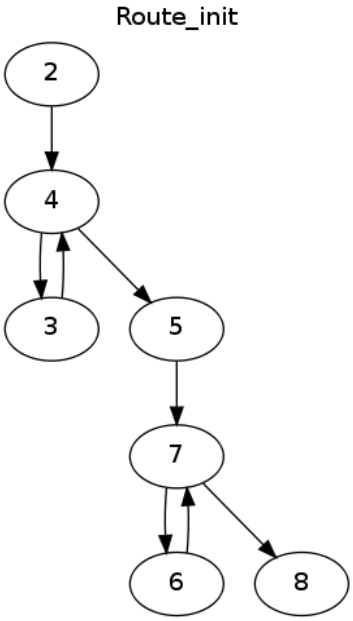

Figure 2. Complexity computation, through the rebuild algorithm

C. Cyclomatic Complexity

The last class of metrics we will present was developed by Thomas J. McCabe. He introduced in Ref. 14 the **cyclomatic complexity of algorithms**, sometimes called **McCabe's metric**.

This complexity is measured by building the flowchart of the source code, then by counting the number of different paths of the graph. For example, figure 2 shows the flowchart of a function called "route_init" in one of our source code. There are three different paths to go from the initial state 2 to the final state 8. Thus, the McCabe cyclomatic complexity of this function is 3.

The minimum complexity of any algorithm is 1 (there is at least one path from the beginning to the end of the algorithm). Each condition, each loop, each branch increases the cyclomatic complexity.

The **cyclomatic complexity iv(G)** can be measured on each module of a program, enabling the developer to perform different code simplifications on the most complex modules. A variant of the cyclomatic complexity evaluates the complexity of variables, which can be used in order to estimate the effort to test the software.

The **essential cyclomatic complexity (eiv(G))** is another variant and measures only the costs of jumping, considering the conditional and unconditional loops have a cost of 1. Nonetheless, in language C, quality standards[15] commonly recommend that the jumping instructions like « **goto** » are kept to a strict minimum, so the essential cyclomatic complexities we measured on our source codes were all **eiv=1**.

We decided to avoid the implementation of other metric measurements in our tool METRIX, because we have evaluated the ratio benefits/cost too low to justify implementing them. Thus, we delayed the implementation of the metrics based on the names of the identifiers and the metrics requiring an abstract interpretation of the code.

We used Perl scripts to perform the computations of metrics, helped by some Open source tools (cfg2dot[16] and graphviz[17]). Another issue we worked on deals with the representation of METRIX measurements. The next section presents two classes of diagram we used in order to highlight the special features of source code.

III. METRIX visualization features for source code performances

Different diagrams enable the user to visualize numeric data. Graphics like line charts, scatter plots and histograms are common, thus we will not expand on them. We will present two less-common classes of diagrams: the radar plots, also called Kiviat diagrams, and the city map diagram, mostly used to represent the cartography of cities. One specific feature of METRIX is to use those two types of visualization for constructing signature and cartography of source codes.

A. Kiviat diagrams

The Kiviat diagram visualizes information through polar coordinates. The distance between the point and the origin is associated to the value we want to represent, while the angle between two points is constant, this constant is uninformative and calculated to uniformly distribute the different points. Moreover, all metric points are linked together, making a plain polygon, which shapes the specificity of the data we want to represent. Of course, this diagram is not adequate to represent only one or two metrics, it requires at least three values to be pertinent. Values may be of the same metrics, representing the metric measured on different parts of the source code. Values may be of very different metrics, computed with heterogeneous units. That enables the user to merge multivariate data on the same diagram. On Kiviat diagrams, values must be strictly positive.

1. Functions signature usage

In figure 3, we represent two Kiviat diagrams coding different metric values for two functions issued from one of our source codes. Metrics are the same on both diagrams, only the numeric values differ between them. By normalizing all values into the range [0, 1], metric per metric, we can obtain a polygon for each evaluated function. Thus, the polygon shape is specific to each function, so we named them the "signature" of the different functions (ComputeChecksum and GetChecksum).

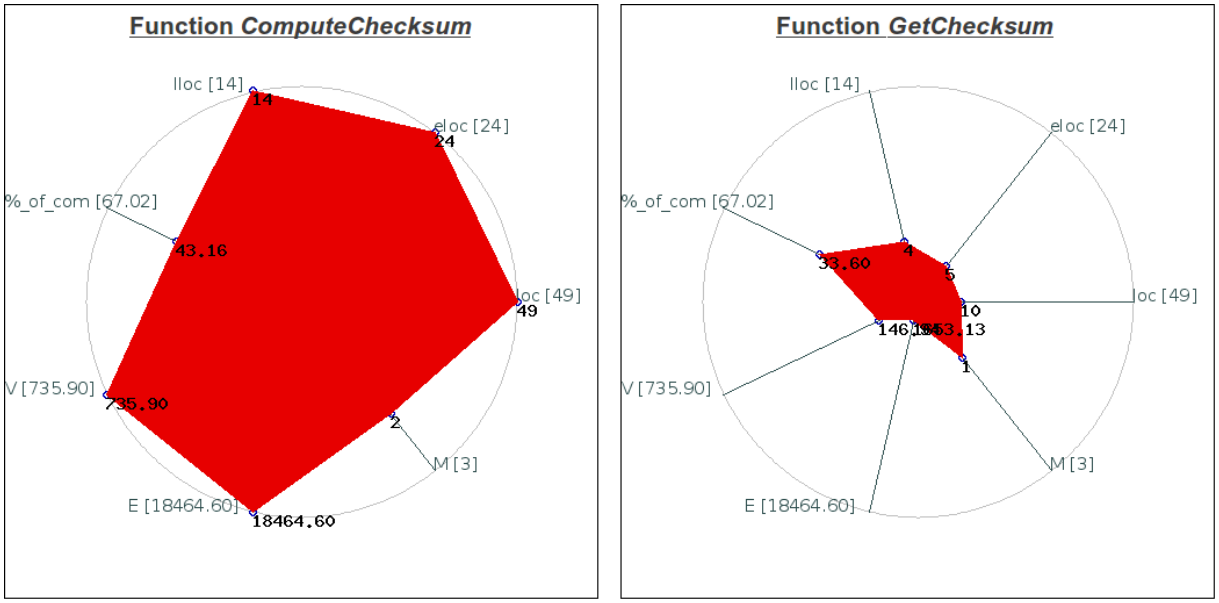

Figure 3. Signatures of the functions ComputeChecksum and GetChecksum

Each signature has its specific shape and differs visually from the signature of the other functions. For instance, figure 3 shows that the function ComputeChecksum contains more lines of code and is more complex than the function GetChecksum. The polygons have graphically different shapes.

To compare different signatures of different functions we need to normalize their signature values with the same maximal values. The algorithm we have used to perform the signature diagrams is the following one:

1) Extract the matrix of data: each function of the source code is associated with a row, each metric is associated with a column. For instance:

	LoC	eLoc	(…other metrics…)	M
Function ComputeChecksum	49	24	…	2
Function GetChecksum	10	6	…	1
(other functions…)	…	…	…	3

2) Compute the diagonal matrix of inverses of maximal values

	LoC	eLoc	…	M
LoC	1/49	0	…	0
eLoc	0	1/24	…	0
…	…	…	…	0
M	0	0	0	1/3

3) Compute the product of the matrices

	LoC	eLoc	…	M
Function ComputeChecksum	1	1	…	2/3
Function GetChecksum	10/49	6/24	…	1/3
…	…	…	…	…

In this new matrix, all values are between 0 and 1, thus they can be represented on a polar diagram of radius 1. That enables us to merge heterogeneous variables on the same diagram.

4) Represent each line of this matrix on a Kiviat diagram (see figure 4 for details).

The different signatures enable the user to rapidly identify the “trouble spots” of the source code, for instance due to human error, just by seeing the area of the signatures of the different functions. Indeed, a complex function should have an area bigger than a simple one. In our example in figure 4, we can see that the function ComputeChecksum is more complex than the function GetChecksum. This does not mean that this function has more bugs, but this is an indication that more effort should be put into to verifying and validating this function.

2. Metrics on the whole project

We used Kiviat diagrams for another usage. We can represent one metric computed on all functions (or a subset) of the development project, as illustrated by figure 4. In this case, the resulting diagram provides a synthetic diagram, easing the extraction of "trouble spots" of the project. This issue has to be analyzed by software engineers and for instance can lead to redesign steps.

In our example in figure 4, we represented the cyclomatic complexity of all the functions of a file "currentpacket.c" in our project. This diagram shows that most functions are simple: the McCabe complexity is 1, which is the lowest value an algorithm can have. Two functions have a complexity of 2 and the function SetCurrentPacket has a complexity of 3, which is the highest complexity measured in our source code.

McCabe recommends in Ref. 14 to redesign functions with a complexity greater or equal to 20. Thus, in our example, we do not have to simplify any function of currentpacket.c.

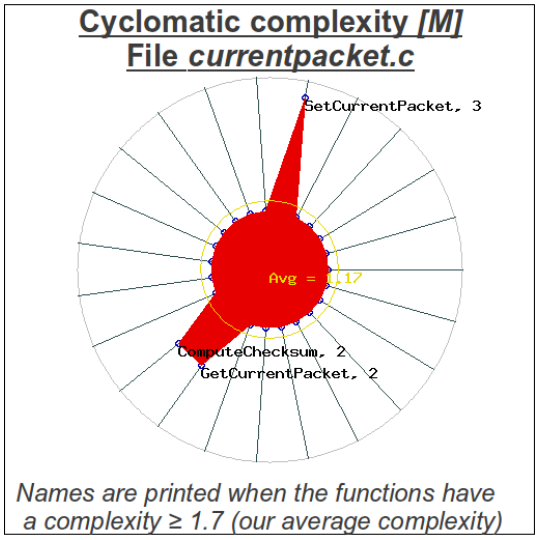

Figure 4. Kiviat to sum up the cyclomatic complexity for the file currentpacket.c

In this diagram, we can insert circles to represent particular values. In our case, we represent a circle with a radius equal to the mean value of all complexity measures. Thus, we will consider for improvement only the functions with a complexity greater than this mean. For instance, as an improvement for software development, we could consider that these functions may be simplified and/or split into simpler sub-functions.

We may have another circle with a radius of 20 (or any other arbitrary value), so we can see immediately all functions with unacceptable complexity.

We searched for variants of Kiviat diagrams: T. Kerren, I. Jusufi and G. Yuhua worked on 3D-Kiviat diagrams in Ref. 18 and 19 and M. Lanza, M. Pinzger and H. Gall worked on Kiviat graphs in Ref. 20, but we found these diagrams harder to use than the one introduced in this section. Thus, we did not select these variants for integration in our software tool METRIX and we will not investigate further this type of diagram in this publication.

B. City map diagrams

In the previous section, we introduced a diagram to deal with software signature. In the rest of this section, we are going to introduce a completely different visualization we worked on: the city map diagram. This diagram is mainly used in urbanism, to represent cities and their buildings in three dimensions. As a software project can contain a very large number of source files, code functions and code variables, we would suggest that this visualization diagram adequately suits our complexity visualization issue.

Historically, a more basic diagram is the treemap diagram. The treemap diagram is used in computing to represent in two dimensions source code metrics with rectangles. Some authors used variants of this class of visualization diagrams to represent values with esthetic considerations (see Ref. 21 for more explanations and examples).

In Ref. 22, Richard Wettel and Michele Lanza proposed using a three-dimensional representation of treemap as an improvement, building the concept of data visualization through city map diagrams. We implemented this representation in our tool METRIX and parameterized it to evaluate our source-code.

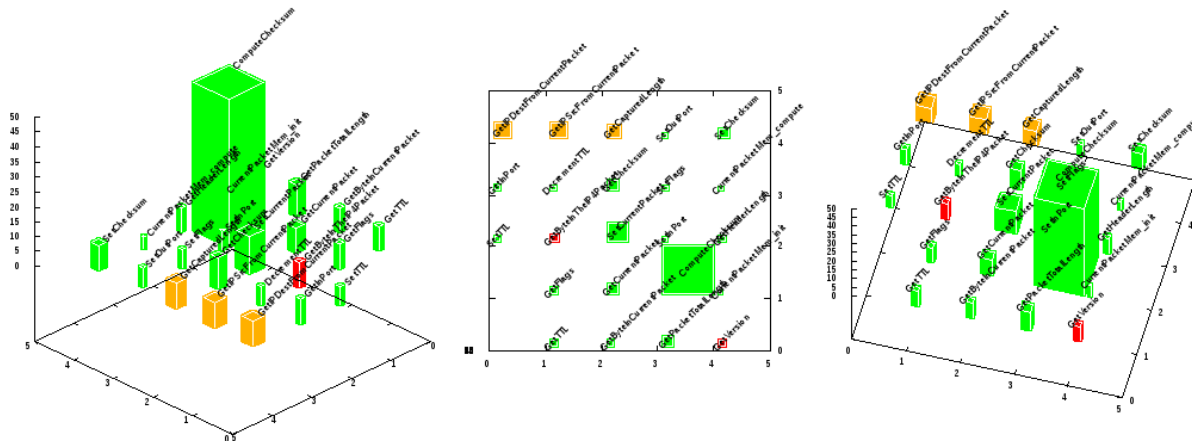

Figure 5. City map for the file currentpacket.c

City map diagrams are indeed multi-parameters diagrams: each "building" is associated with a n–tuple of numeric values. Table 1 presents an example of instantiation of some city map 4-tuples with the metrics extracted from our source code.

We experimented using other parameters for the n–tuple, but our experiments showed that we should not overloading the diagram. For example, all buildings are square because rectangular buildings are difficult to interpret. More than five colors reduce the diagram readiness, except if we use color shading. To position the building, we use a snail-algorithm[4]. We tried forcing a (x;y) placement with two metrics, but the results were not encouraging.

Figure 5 shows three views of the same city map diagram relating to a file "currentpacket.c" of our source code. We used the associations from table 1 to create this diagram. The tool METRIX enables the user to move and turn the diagram, as well as to modify dynamically the angle and the zoom focus. This facilitates the diagram interpretation.

In figure 5, we showed only three views; this is a drawback of this visualization: the city map diagrams are not well adapted for printing on paper; they are more adapted for interactive usage. Our graphical user interface presented in the next section enables the user to explore interactively the diagrams.

Figure 5 shows that function ComputeChecksum is the most complex function of our file currentpacket.c.

- The building associated with this function is the tallest (it means this is the longest function of the file).
- This building is the largest (it means this function has the most important Halstead's volume).
- This building is not at the middle of the map (thus this function has not the maximum cyclomatic complexity).
- The building is green (thus, the ratio of comments is acceptable).

All these diagrams enable the user to interpret graphically the numeric values extracted from his/her source code. The diagrams "sort" data and foreground the values requiring attention, so they provide a way to represent quickly and easily the quality of source code. In any case METRIX remains a tool; the interpretation of its results requires a subjective human analysis, to define what is acceptable and what requires redesign and/or re-implementation. This limit of acceptability is specific for each company and each development project. However, some limits are commonly set. For instance, McCabe complexity should not be more than 20; or maximum volume should not exceed 1000. In the next section, we will give a detailed example of how engineers can use METRIX software on a specific software development project they may have to conduct.

Table 1. Instantiation of city map axes of liberty with metrics issued from source code

4-Tuple parameter city map axe	<==>	Metric measured on our source code
Height of each building		LoC (number of Lines of Code)
Width of each building		Halstead's Volume of the Code
Position of the building on the map		McCabe Cyclomatic Complexity (most complex functions are on the centers, simplest ones are on the bounds)
Color of the building		Comment ratio (red≤20%, orange=between 20% and 50%, green≥50%)

[4] This algorithm places the first item at the center of the map, then places the following items nearby, turning around the center like on a snail shell according to cyclomatic complexity.

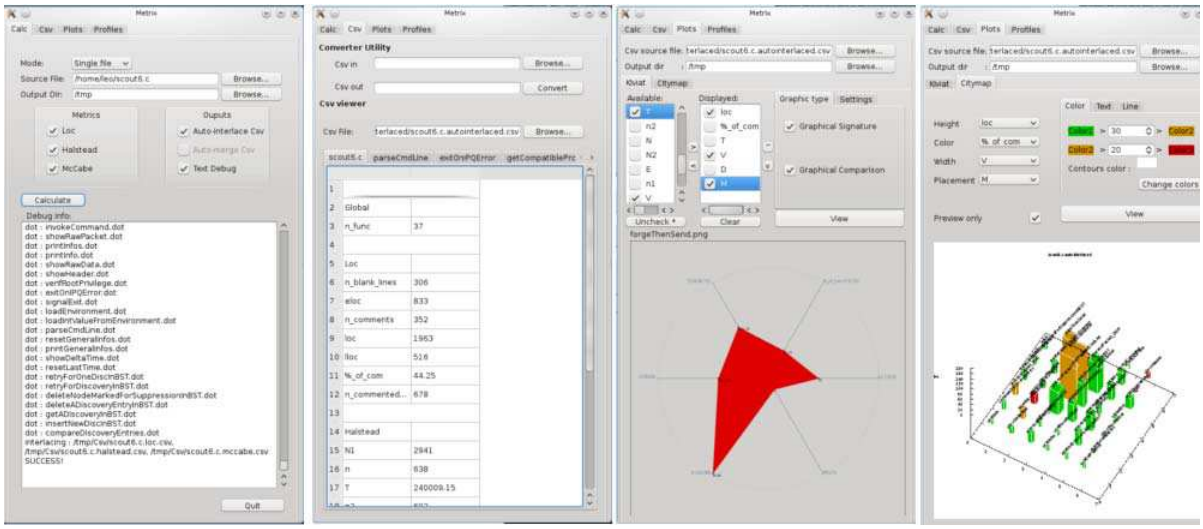

Figure 6. Calc and Csv tabs of the Metrix GUI at left, "Plots" tab of METRIX GUI to represent Kiviat and city maps at right

C. Our Graphical User Interface METRIX

Our first idea was to develop scripts to extract numerical data coming from the metrics computed on the source code and to import this data into a spreadsheet application. Nevertheless, the very large quantity of data generated by our tool lead us to revise our position. In order to ease the data processing and visualization, we developed a graphical user interface (GUI) with a single Window containing three tabs.

The first tab called "Calc" and presented on figure 6 enables the user to indicate the source file(s) to evaluate and to parameterize the metrics to compute. It avoids using hand-start scripts with a prompt, but the open source aspect of our tool allows advanced users to (re)use our measuring scripts manually and/or to integrate them into other software.

The second tab called "Csv" in figure 6 presents the numeric values as a list, with a tab per function and per file. The user may export these values to a spreadsheet application, in order to represent the values with common visualization graphs, like scatter plots, line plots, etc.

The third tab called "Plots" and illustrated by figure 6 provides a way to visualize the city map diagrams for the source code, the signatures of functions and the comparisons between functions through radar/Kiviat plots. This tab enables the user to change the default behavior of the tool, for example to modify the ceiling and floor to insert color into the data, to adapt the placement of the buildings in the city map, etc.

Our tool may generate a report in the form of a LaTeX file, so the user can use it to produce a PDF file. This report summarizes all the values measured on the project. Each function and each file make a section of this report. In each section, numerical values are coupled with the signature of the function (as a Kiviat diagram). Each file produces three views of the associated city map diagram, as well as different Kiviat diagrams for the different metrics measured on the functions of the file.

We have published all METRIX source code on the web site http://www.recherche.enac.fr/~avaret/metrix. The source code is published under the General Public License v3 (GNU GPL v3[23]). We added a Debian package to ease the installation and the deployment of our tool for the final users.

IV. Case study: model improvements of an embedded secure router

A. Aeronautical embedded secure router design context

New aeronautical traffic profiles are growing in usage and complexity. Higher throughputs and new opportunities could be served by multiplexing some different data, but the heterogeneity of their safety and security constraints remains the main problem for promoting multiplexing solutions through a unique network link. For this purpose, we are producing an IP-based Secure Next Generation Router (SNG Router). This SNG Router provides regulation, routing and secure merging of different data sources, as well as preserving their segregation. We have published more details on the design and the implementation of our SNG Router in Ref. 1 and 24.

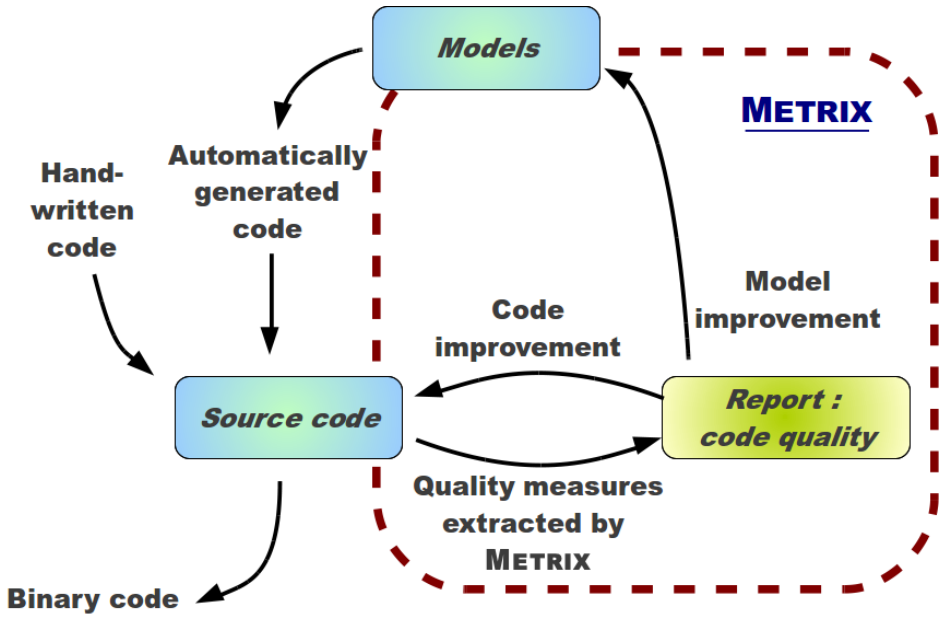

Figure 7. Integration of Metrix in the development process of our router SNG

During the development of SNG Router source code, we designed models for fast prototyping. However, the SNG source code reviews led us to conduct research on how to assess the quality of our source code and to provide a way to improve the automation of this step. The foundations of our tool METRIX were created.

B. METRIX application results for an aeronautical embedded router design improvements

We used METRIX to improve the source code of our embedded router as illustrated by figure 7. We modeled and designed our router through a set of Simulink and Stateflow models, next we automatically generated the source code with a tool called "transformer". We were then able to apply METRIX to evaluate the quality of the source code. Finally we are able to adapt the original models, the transformer or directly the source code.

We used METRIX on source codes generated with GeneAuto[25]. The tool GeneAuto is a transformer of high-level Simulink and Stateflow models into C-language source code. We have applied the tool on the generated source code and have generated a report in pdf format.

Our first utilization of the report was to make improvements of the modeling on trouble spots. GeneAuto translates all the functions of the models into functions in C language. Then, we apply METRIX to extract the

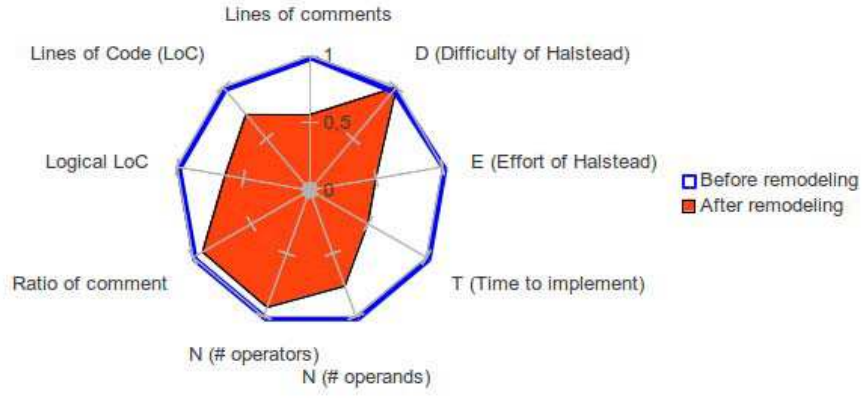

Figure 8. Example of remodeling of the function loadEnvironment

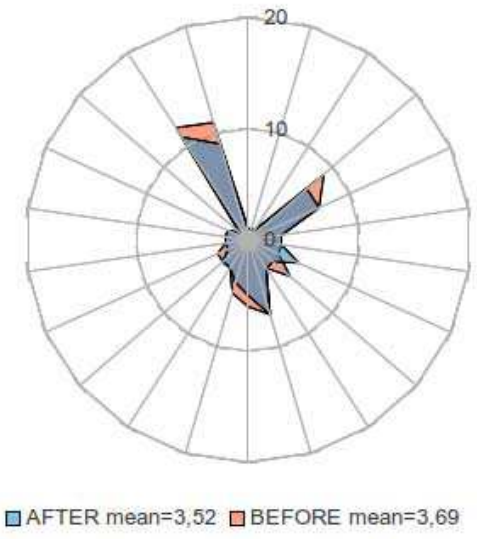

Figure 9. Example of gains for the McCabe cyclomatic complexity metric

signatures of all the functions. Signatures whose polygon has a large area are an indication of complex and long functions (as discussed previously in section 3), thus we redesigned some of them to simplify and shorten these functions. Figure 8 illustrates an example of improvement where we redesigned several functions: we simplified the source code and consequently reduced the number of lines of code in addition to Halstead's metrics "Effort to implement" and "estimation of Time to implement".

The report generated by METRIX contains Kiviat diagrams to synthesize McCabe cyclomatic complexities of all the functions. These diagrams show immediately intolerably complex functions. We remodeled all the functions with complexities higher than 20, modifying the algorithms to avoid some branches and splitting some of them into smaller and easier functions.

Figure 9 shows the McCabe cyclomatic complexities of the functions generated from a model, before and after improvements. The global shape does not change but after improvement, the average complexity decreased from 3.69 to 3.52.

We decided to perform an additional review on the tests to debug the functions on the centre of the city map and on the tallest buildings in this diagram: these functions are the most voluminous and the most complex, thus they are potentially the functions most subject to defects and bugs. This effort leads us to identify and correct some race conditions, correctly specified but incorrectly implemented in the model.

C. METRIX application results to compare C and Ada auto-generated codes

As a further step, we tried an experimental version of GeneAuto to produce Ada source code instead of C source code. We used METRIX to compare the generated code for both languages. Halstead's metrics and McCabe complexity measures are language independent, so for the most part we compare the efficiency of the transformers rather than the efficiency of the languages.

Table 2 summarizes an example of the metrics measured on the generated code from the same Stateflow model called "route". In this example, the GeneAuto generated more functions for the Ada language than for the C language. This resulted in a lower average complexity for the transformer into Ada language.

We performed another comparison between both codes in Ada and C generated by GeneAuto. Table 3 summarizes the data. We can see that Ada code has a tendency to be longer than C code (the average Halstead's volume is greater in Ada than in C), but the Halstead's difficulty is lower in Ada than in C. Halstead's estimation of time shows a slight advantage for the Ada language than for the C language. All these measures are relative to codes generated by two versions of Gene-Auto. Note that improvements of these transformers may inverse these tendencies but these results demonstrate that METRIX can help software designers to compare different projects built in different languages in order to select, for instance, the most efficient one.

Table 2. METRIX measurements of cyclomatic complexity for Ada and C

	C language (route.c)	Ada language (route.adb)
Sum of McCabe complexity	37	38
Number of functions generated	6	10
Average complexity per function	6.2	3.8

Table 3. METRIX measurements of Halstead's metrics on C and Ada source code

	Volume Ada	Volume C	Difficulty Ada	Difficulty C	Estimation of time Ada	Estimation of time C
Commonfunctions	682	337	8.6	9	327	170
Currentpktmem	11109	4968	57.65	51	35582	14085
Filters	6231	3198	42.5	40	14718	7168
Localprocessing	581	472	6.3	14	202	373
Main	5612	9234	13	56	3989	28894
Route	11907	9587	67	108	44416	57572
Verify IP header	3465	3022	27	42	5243	7130
Average	**5655**	**4403**	**32**	**46**	**14925**	**16484**

V. Conclusion and future work

In this paper we have introduced our new Open Source software tool METRIX which enables users to assess quality of source codes. We have enumerated the different source code metrics our tool computes, sorting them into a language-dependant set and a language-independent set. From these metrics, our tool generates various visualizations. Kiviat diagrams enable the user to assign a graphical signature to each function and to compare in a single diagram a metric computed on many functions. City map diagrams provide a way to represent multivariate data, i.e. multiple metrics synthesized on a single diagram. We provide with our tool a Graphical User Interface to facilitate the extraction of metrics and the representation of results. We have concluded the article with the example case study of an embedded router we have been developing and improving through METRIX.

We are currently working on some potential METRIX improvements. Firstly, we are studying new metrics to complete the current set of source code metrics with an additional set of metrics computed directly from the models we use to generate the source code. The models we design are graphical representations and not textual source code. Indeed, some model metrics are very different from the classic metrics we have implemented in our tool. Secondly, we are working on METRIX visualization features: we have already implemented Kiviat and City map diagrams, and we are now looking at other types of diagram. In particular, we are testing different configurations to identify the optimum set of metrics for each diagram in order to improve METRIX ergonomics.

Acknowledgements

We would like to thank Rupert Salmon and John Kennedy for their help in editing this paper.

References

[1]A. Varet, N. Larrieu; "New Methodology To Develop Certified Safe And Secure Aeronautical Software", 14 pages, *Digital Avionics Systems Conference (DASC-2011)*, Seattle, USA, October 2011.

[2]M. Conrad, T. Erkkinen; T. Maier-Komor, G. Sandmann, M. Pomeroy; "Code Generation Verification – Assessing Numerical Equivalence between Simulink Models and Generated Code", http://www.mathworks.com/tagteam/63743_SimTest10_CGV.pdf.

[3]I.Stürmer, H. Pohlheim; "Model Quality Assessment in Practice: How to Measure and Assess the Quality of Software Models During the Embedded Software Development Process", *Proceedings of IEEE ERTS 2012*, http://www.erts2012.org/Site/0P2RUC89/6D-2.pdf.

[4]SourceMonitor software, http://www.campwoodsw.com/sourcemonitor.html.

[5]Sonar, open platform to manage code quality, http://www.sonarsource.org/.

[6]McCabeIQ software official website, http://www.mccabe.com/iq.htm.

[7]Mathworks Polyspace: static analyser for software, http://www.mathworks.fr/products/polyspace/.

[8]IBM Rational Software Modeler (RSM), http://pic.dhe.ibm.com/infocenter/rsmhelp/v7r5m0/index.jsp.

[9]Horst Zuse, "A framework of software measurement", Walter De Gruyter Inc, 1997.

[10]M. Squared Technologies LLC, "Metrics Definitions", http://msquaredtechnologies.com/m2rsm/docs/rsm_metrics.htm, 2012.

[11]Linux community, "Linux Kernel Coding Style", https://www.kernel.org/doc/Documentation/CodingStyle, 2000.

[12]J. Seward, "Valgrind, a GPL licensed programming tool for memory debugging, memory leak detection, and profiling", www.valgrind.org, 2002.

[13]M. Halstead, "Elements of software science", Elsevier, New York, 1977.

[14]T. J. McCabe, A. H. Watson, “Structured testing: a testing methodology using the cyclomatic complexity metric”, http://www.mccabe.com/iq.htm, 1996.
[15]“Introduction to Motor Industry Software Reliability Association for C language (MISRA C)”, http://www.embedded.com/columns/beginerscorner/9900659?_requestid=427335, 1998.
[16]R. Fechete, G. Kienesberger, “Flow World – Controlling the flow since 2005 – cfg2dot”, http://cfg.w3x.org/cfg2dot/cfg2dot.html, 2012.
[17]J. Ellson and al. “GraphViz, a Graph Visualization Software”, AT&T, 2011.
[18]T. Kerren, I. Jusufi, “Novel Visual Representations for Software Metrics using 3D and animation”, *Software Engineering Workshop band*, 2009.
[19]G. Yuhua, “Implementation of 3D Kiviat diagrams”, 2008.
[20]M. Lanza, M. Pinzger, H. Gall, “Visualizing multiple evolution metrics”, 2005.
[21]R. Vliegen, K. van Wijk, E-J. van der Linden, “Visualizing Business Data with Generalized Treemaps”, http://www.magnaview.nl/documents/Visualizing_Business_Data_with_Generalized_Treemaps.pdf, *IEEE Transactions on visualization and computer graphics*, Vol. 12, No.5, 8 pages, 2006.
[22]R. Wettel, M. Lanza, “Visualizing software systems as cities”, in *4th IEEE International Workshop on Visualizing Software for understanding and analysis, VISSOFT*, pages 92-99, 2007.
[23]Free Software Foundation, “Welcome to GNU GPLv3 General Public License version 3”, http://gplv3.fsf.org/, 2007.
[24]A. Varet, N. Larrieu, C. Macabiau; ” Design and Development of an Embedded Aeronautical Router With Security Capabilities”, 14 pages, *Integrated Communication, Navigation and Surveillance Conference (ICNS-2012)*, Washington DC, USA, May 2012.
[25]A. Toom, T. Naks, M. Pantel and al., “GeneAuto: an Automatic Code Generator for a safe subset of Simulink/Stateflow and Scicos”, in *Embedded Real Time Software and Systems (ERTS²)*, 2008.
[26]C.-B. Chirila, D. Juratoni, D. Tudor, V Cretu; “Towards a software quality assessment model based on open-source statical code analyzers”, 2011 *6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI).*