Testing Computer Vision Applications An Experience Report on Introducing Code Coverage Analysis in the Field

Iulia Nica and **Franz Wotawa**[1] and **Gerhard Jakob** and **Kathrin Juhart**[2]

Abstract. In this paper we present our work in progress in defining a suitable testing and validation methodology to be used within computer vision (CV) projects. Typical quality assurance (QA) measures, targeting the applicability in real-world scenarios, are meant here to complement the research on specific computer vision methods. While inspecting the existing literature in the domain of CV performance evaluation, we first identified the main challenges the CV researchers have to deal with. Second, as every vision algorithm eventually takes the form of a software program, we followed the classic software development process and performed an in depth code coverage analysis in order to assure the quality of our test suites and pinpoint code areas that need to be reviewed. This further leaves us with the questions of which test coverage tool to prefer in our situation and whether we can introduce some specific evaluation criteria for identifying the right tool to be used within a CV project. In this article we also contribute to answering these questions.

1 Motivation

Computer vision (CV) is used today in a wide range of real-world applications, from industrial inspection and safety relevant vehicle functions to 3D model generation by photogrammetric methods, medical imaging and fingerprint recognition. Although a vast variety of literature covering evaluation techniques in subfields of the whole topic is available, still no study reports on testing a complete vision system, i.e., comprising hardware, software, data communication and control. Obviously the high quality of CV applications has a great impact on their usability in real world scenarios. Hence beside traditional CV evaluation techniques such as using test data sets as input and comparing the algorithms output against a manually established ground truth - we have to control the quality of the involved applications by means of applying a more generic evaluation strategy. In this context, quality assurance (QA) activities like peer reviews, coding guidelines, or the usage of software quality tools (static and dynamic analyzers) offer many benefits, from being able to track the CV projects progress and estimate its relative complexity to helping us realize when we have achieved the desired state of quality [4].

Still, what is different about testing CV applications and why is it so difficult to test whether computer vision algorithms can live up to their claims?

Regarding algorithmic correctness on one side, it is often very hard to get a consistent and exact definition of the desired output for a specific input. Especially in classification tasks, it is tough to decide, when the obtained results are still correct and when we are dealing with an abnormal behavior.

Regarding the evaluation of the complete, often very complex vision system on the other side, the QA team has to manage and run a high amount of tests on all levels - from unit tests, to integration, function and system tests. Therefore one needs to understand the system as a whole, as well as all of its components and their interdependencies. Furthermore we have to cover also possible hardware faults when identifying use cases, based on the defined system requirements and specifications. Fortunately, today there are well-established QA practices and many quality management tools available on the market, meant to ease the generic evaluation of products and processes, that the only challenge is to find a proper manner to integrate them in the vision project.

The remainder of this paper is organized as follows. In Section 2 we review the existing literature in the domain of CV performance evaluation and introduce some basic quality assurance terms. Afterwards, in Section 3, we identify and discuss the requirements a code coverage tool has to fulfill in order to be used in the CV domain. Further on, we give a short overview of our four best ranked tools. In Section 4 we first introduce the case study and compare the tools based on their integration with the example application. Additionally, we present the first success story in improving our code coverage. With Section 5 we conclude this paper.

2 Related Research

In our work, we have first inspected the existing literature in the domain of performance evaluation in computer vision. General overviews of empirical evaluations were found in [5], [2], [3], and [11] and will be further presented here in a chronological order. They all review the commonly used techniques for performance characterization of algorithms in different subfields of CV.

In the early 90s, [5] was discussing the evident lack of performance evaluation in the literature on vision algorithms. In the author's opinion, this situation has been tolerated because the ability to perform a CV task was interesting enough, so that the performance of the new algorithm became a secondary issue. In order to quickly design a machine vision system, which works efficiently and meets requirements, [5] suggests an analogy with a system's engineering methodology. Thus, a well-defined protocol containing a *modeling component*, an *experimental component* and a *data analysis component* was envisioned. The modeling component would describe the ideal input image population (real or synthetic images), the random perturbation model (by which non-ideal images arise), the random

[1] Technische Universität Graz, Austria, email: {inica,wotawa}@ist.tugraz.at
[2] JOANNEUM RESEARCH, email:{gerhard.jakob,Kathrin.Juhart}@joanneum.at

perturbation process (that characterizes the output random perturbation as function of input random perturbation) and the criterion function (by which one can quantify the difference between the ideal output and the computed output). The experimental component describes the performed experiments, whilst the data analysis determines the performance characterization based on the experimentally observed data.

In the absence of acknowledged methods for the evaluation of algorithmic performance, [2] proposed the definition of performance as function of mathematical sophistication. However, as the number and specificity of assumptions made in the mathematics underlying a vision algorithm increase (i.e., the sophistication of an algorithm increases), the performance of the CV application not necessarily does. This is the case when the assumptions made do not match the application characteristics. Furthermore, the need of standard databases, evaluation protocols and scoring methods/performance metrics available to researchers was identified by the authors.

Regarding the typology of test data, [3] differentiate first between data without noise and data with noise. Moreover, they mention three types of empirical testing: testing using real data with full control, empirical testing with partially controlled test data and testing in an uncontrolled environment. Depending on the distribution of the available data into training and testing sets, test protocols have been proposed. Another discussed issue in [3] is again the necessity to define a metric, which can be used to quantify performance. The authors associate such performance metrics with the failure modes of an algorithm. For each type of vision algorithm, specific evaluation metrics were defined according to the function performed by the given algorithm. Some examples are the ROC (Receiver Operating Characteristic) curve in case of a feature detector, the confusion matrix in case of object recognition, or the true and false matches when dealing with matching algorithms, such as those used in stereo or motion estimation.

Similarly to [3], the authors of [11] outline two different levels of analysis for vision systems:

- *technology evaluation*, which concerns the characteristics of the algorithms using generic metrics, such as ROC curves. Standardized data sets are used and the results are therefore repeatable and depend on the size and scope of the test data sets. Generally, this evaluation stage requires simple metrics related to the fulfilled function- detection, estimation, classification.
- *scenario evaluation*, which concerns the system's behavior in particular situations - for a specific functionality with its sets of variables (e.g, number of users, type of lighting). The test data is based on a controlled real world and is therefore only partly reproducible. More complex metrics are to be used here, e.g., system reliability expressed as mean time between failures.

[11] takes the topic of technology evaluation a step further by defining a set of eight key questions, thought to highlight the best practices and the state of evaluation methodology in several representative areas of computer vision discipline: sensor characterization, feature detection, shape- and grey-level-based object localization, shape-based object indexing: recognition, lossy image and video compression, differential optical flow, stereo vision, face recognition, measuring structural differences in medical images. From the guiding questions formulated in [11], we selected those, which are in our opinion first to be answered in algorithmic testing:

1. *Is there a data set for which the correct answers are known?*
2. *Are there data sets in common use?*
3. *Are there any known algorithms that can be used as benchmarks for comparison?*
4. *What should we be measuring to quantify performance? What metrics are used?*

Though the analysis in [11] touches also other aspects of building a complete vision system, it excludes testing the hardware. Furthermore, the mentioned software validation is limited to ensuring that the software implementation of an algorithm correctly instantiates its mathematical foundation [11]. Hence, the collected answers for each of the considered visual tasks indicate the fact that performance characterization techniques are mostly application/algorithm specific and that currently they do not refer to the integrated system as a whole, i.e., comprising hardware, software, data communication and control.

More currently published research like [13], [14] emphasizes the role of test data generation and test data validation in vision testing. For the purpose of evaluating CV algorithms, there are today some publicly available data sets, such as the FERET database [9] for face recognition algorithms, Middlebury [10] and KITTI [7] test data sets for stereo vision, or VOT datasets [6] for visual tracking. The usage of this large amount of test images brings yet some problems. One of them is that the test data sets are not specially designed for a particular vision application, but for a class of algorithms. Hence, a 100% coverage of the possible scenarios can not be guaranteed. As introduced in [13] and further elaborated in [14], a solution to this problem would be the automatic generation of datasets, so that they contain all the typical scenes and hazards, without including too much redundancy, so that the testing effort could be manageable.

For a change, as the vision algorithm will take eventually the form of a software program, we see no reason why we should not take advantage of the great progress in the domain of quality assurance and software testing in particular. The usage of standardized QA methods, metrics and tools can ease the work of any CV developer and quickly improve the overall process, especially in terms of system's resilience and end user's satisfaction.

"Quality control activities determine whether a product conforms to its requirements, specifications, or pertinent standards"[12]. In addition to the traditional testing practices, QA activities encompass peer reviews, coding guidelines, and also the usage of software quality tools, like static analyzers that examine source code for possible errors or code coverage analysis tools, that can measure the actual coverage of the software with the available test data sets. For more information on software testing and other QA techniques we refer the interested reader to [8], [1], [12].

3 Code Coverage Analysis

Among the first quality assurance metrics invented for systematic software testing, code coverage is used to describe the degree to which the source code of a program is tested by a particular test suite. Test coverage can be used in unit testing, regression testing, for test case order optimization, test suite augmentation or test suite minimization.

The code coverage analysis process is generally divided into *code instrumentation*, *data gathering*, and *coverage analysis*. Code instrumentation consists in inserting some additional statements, that monitor the execution of the source code. The instrumentation can be done basically at code level in a separate pre-processing phase or at runtime.

In order to be self-contained, we briefly introduce here the most commonly used code coverage metrics, as they might be new to the

computer vision community. We further refer to the following small code snippet to quickly highlight their major advantages/ disadvantages in practice:

```
if (x>1 && y==0) {
    z=z+1;
}
if (x==2 || z>1) {
    z=z+2;
}
```

As already mentioned, several kinds of instrumentation are possible. The most common are for:

- *line or statement coverage*: where the tool instruments the execution of every executable source code line; this coverage criterion is a rather poor one, as it is completely insensitive to some control structures and logical operators.
  For instance, one could execute every statement (reaching a 100% line coverage) from our example by writing a single test case: `T1(x=2, y=0, z=4)`. Now, let us assume that the second decision should have stated `z>0`. If so, this error would not be detected. Or perhaps in the first decision should be an *or* rather than an *and*. This error would also go undetected.
- *decision or branch coverage*: it reports whether each decision has a `true` and a `false` outcome at least once; this criterion is stronger than line coverage, but it is still rather weak.
  For instance, with our previous test-case inputs `T1(x=2, y=0, z=4)` and a new one `T2(x=3, y=1, z=1)`, we can reach full decision coverage. However, if in the second decision we should have had `z<1` instead of `z>1`, the mistake would not be detected by the two test cases.
- *condition coverage*: in this case, one has to write enough test cases to ensure that each condition in a decision takes on both `true` and `false` outcomes at least once; this metric is similar to decision coverage, but has better sensitivity to the control flow. However, full condition coverage does not guarantee full decision coverage. For instance, the following test cases: `T3(x=1, y=0, z=4)` and `T4(x=2, y=1, z=1)` cover all conditions' outcomes, but they cover only two of four decisions' outcomes.
- *function coverage*: reports whether each function is called (and how many times); it is useful during preliminary testing to quickly find coarse deficiencies in a test suite.

3.1 CV tailored Evaluation Criteria

Following the classic software development process depicted in Figure 1, we first learned that code coverage analysis does not exist in most of the CV projects. As a result, we tried to identify the must-have and nice-to-have features of a code coverage tool to be used in the CV application domain. Like in any tool selection process, one has to clarify first the user's requirements. We will further present only those particular requirements related to computer vision software, and neglect general questions such as: what platforms can the tool run on, what is the target application's language or which are the supported compilers. We will not mention here requirements coming from the quality assurance team, which are to be discussed in the next section.

The following list ranks the priorities of these specific features, as discussed with CV software developers:

1. *working with templates*: due to the great variety of data types (pixel and parameter types), there is a tremendous number of templates defined in CV applications and which have to be taken into consideration when analyzing the code coverage. Tools that cannot handle templates appropriately are dismissed.
2. *unit testing support*: in our case, CPPUnit unit testing framework support is needed, as this is the most frequently used framework in C/C++ CV applications.
3. *excluding 3rd party libraries from the coverage analysis*: as most of the CV applications make use of third party libraries, whose analysis is obviously not desired, the tool has to provide a simple way to hook/instrument only certain files.
4. *automated testing/non-interactive testing*: taken into consideration the high complexity of the currently developed CV software, an easy automation of the test coverage analysis is essential.
5. *performance under big test data amounts*: There is no doubt that the insertion of instrumentation will increase the code size and affect the instrumented applications performance, i.e., it will use more memory and run slower. A low performance overhead is of course desired, however, considering the complexity of the target programs, our requirement is that the analysis tool does not crash.

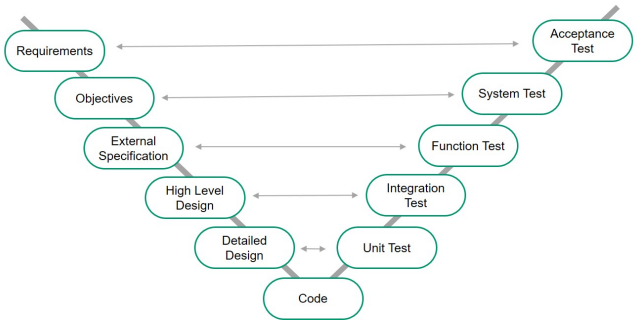

Figure 1. One-to-one correspondence between development and testing processes.

3.2 Four state-of-the-art Code Coverage Tools

Identifying the right tool for code coverage analysis in vision applications can lead to major productivity improvements and implicitly to increases in the release quality of the overall computer vision system. Hence, various free and commercial coverage analyzers have been inspected and compared. As a large variety of coverage metrics exist (see the preceding summary), the QA team imposed as requirement that the code coverage tool should be able to measure at least condition coverage. This requirement together with the previously presented CV tailored evaluation criteria have led to limiting our comparative evaluation to the following four state-of-the-art commercial coverage tools: C++ coverage validator[1] , Squish Coco code coverage tool[2] , BullseyeCoverage tool[3] , Testwell CTC++ analyser[4] .

4 Case Study: `Dibgen` and `Dibgiom` Libraries

`Dibgen` is a collection of basic C++ libraries used particularly, but not exclusively, in computer vision applications implemented

[1] http://www.softwareverify.com/cpp-coverage.php
[2] http://www.froglogic.com/squish/coco/index.php
[3] http://www.bullseye.com/measurementTechnique.html
[4] http://www.verifysoft.com/de_cmtx.html

by JOANNEUM RESEARCH (JR). Included libraries cover basic, mostly matrix based mathematical operations, color handling and evaluation, as well as generic parameter storage, progress information handling, different types of basic file IO methods often used in computer vision, and value-to-string conversion (and back-conversion). All the libraries are implemented using template-heavy C++ code allowing the usage of different data types (pixel types, parameter types) for most of the operations. In terms of volume, `Dibgen` consists of approximately 100000 LOC.

The other partially analyzed collection was `Dibgiom`. Seen as OpenCV counterpart and based on `Dibgen`, it contains 15 libraries, which are all used for image processing tasks. The library consists of approximately 9 MB of source code and approximately 255000 LOC. We further provide a brief description of those `Dibgiom` libraries, which were yet analyzed:

- `Band`: Various representations of image data in the memory (tiled with FileIO for huge satellite data, pure memory-based for rapid CPU access, specially aligned memory layout for acceleration using Intel Performance Primitives, special layout for CUDA acceleration), transparently accessible via the same interface to both user and algorithms.
- `BandIterator`: Generic access iterators for bands regardless of memory layout (see above)
- `Calibration`: Simple radiometric calibration methods
- `Convolve`: Image filter based on convolution (Gauss, Laplace, etc.)
- `Detect`: Various detectors (Extrema, Bright Spot, Corner, etc.)
- `Filter`: General image-filter (arithmetic, logic, etc.,), that convert, in principle, a pixel in the source image(s) to a pixel in the target image
- `KernelFilter`: Core-based image filters (mean, median, etc.), which do not calculate any convolution
- `KeyPoint`: Description of key points for various detectors
- `Operation`: Operations on images whose result, or whose source is not an image (source no image: filling images, etc., or target no image: the sum of all the pixels in the image)
- `Pyramid`: Generation of pyramid representations (Gauss, etc.)
- `Segmentation`: Image-based operations that compute segmentations from arbitrary source images (Watershed, RegionGrowing, etc.)
- `Sift`: Special version of a Sift detector.

For the `Dibgen` experiments we used the same unit test suites and the same configuration for all the four coverage tools. Although each tool features more than just decision and function coverage, we will merely present the comparison of these two types of coverage measurements, as only they are computed by all the four tools.

The tests carried out for the `Dibgiom` experiments are also unit tests, in which the source data is generated either directly by means of using unit-test programs (usually only for very simple algorithms), or by reading the image data from files. In the latter case, the expected outcome is generated with other reference implementations chosen from the literature (like MATLAB, OpenCV, etc.) and it is further compared with the outcome produced by `Dibgiom`.

In Table 1 we list the global results for the whole `Dibgen` test application, while in Table 2 and Table 3 we present the coverage results per directory. It is worth noting that with Testwell CTC++, the coverage results are extremely low, while the other three tools compute comparable coverage results. Table 4 depicts the running times for the normal, uninstrumented program and for the instrumented programs. Note that the tests were run on a notebook with Intel(R) Core(TM) i7-4500U CPU 1.80 GHz and 8 GB of RAM running under Windows 10 Pro. Although the running time for the program invoked by Coverage Validator is approximately six times higher, we have no source code instrumentation involved, i.e., there is no need to recompile or relink the target program. The only requirement is the existence of PDB files with debug information and/or MAP files with line number information. Therefore we chose to further use the Coverage Validator tool for the first `Dibgiom` experiments. The results can be seen in Table 5.

Table 4. Running Times for the unit tests defined for the `Dibgen` Solution

For non-instrumented programs	*68,44 sec*
For programs invoked by Coverage Validator	475,68 sec
For CTC++ instrumented programs	74,16 sec
For Bullseye instrumented programs	68,97 sec
For Squish Coco instrumented programs	70,81 sec

Table 5. `Dibgiom` Coverage Results computed with **C++ Coverage Validator**

Library	Decision Coverage	Function Coverage
`Band`	36,84%	53,55%
`BandIterator`	13,94%	67,12%
`Calibration`	73,58%	13,33%
`Convolve`	14,52%	34,77%
`Detect`	56,25%	41,72%
`Filter`	30,02%	73,66%
`KernelFilter`	40,08%	48,86%
`KeyPoint`	86,99%	70,64%
`Operation`	54,53%	68,21%
`Pyramid`	2,01%	12,71%
`Segmentation`	87,72%	87,27%
`Sift`	78,10%	66,34%

4.1 First Success Stories

One of the most complex and frequently used basic libraries in the JR's CV applications is the library `ParameterPool` from the `Dibgen` collection. With about 17.000 LOC, the library is used to store any kind of parameters of arbitrary types in one container. Each parameter can be combined with validity information, access level permission for user interface based parameter modifications, as well as several kinds of descriptive text (unit, help text). Additionally, parameters can be grouped together and it is possible to define several types of parameter dependencies. Since this library is used heavily in nearly every JR CV application, the JR developers particularly paid attention to test it thoroughly from the very start of development.

However, first code coverage analysis showed dissenting results, especially in branch, function and line coverage, while at least file coverage could reach nearly acceptable results (see Figure 2). More detailed analysis showed that only 12 out of 36 source code files had a line coverage better than 90%, while 9 files were not tested at all (see Figure 4). Although the remaining 15 files were tested at least partially from the line coverage point of view, especially their branch coverage showed very poor results. After particular review of the tested source code, the used test code as well as the used test data, the test code has been adapted in some places and some test data sets have been slightly modified.

Additionally, some new test functions were developed, especially for previously untested files or functions. One meanwhile unused

Table 1. Overall `Dibgen` Coverage Results

	C++ Coverage Validator	Testwell CTC++	BullseyeCoverage	Squish Coco
Decision Coverage	31,27%	9%	40%	45,30%
Function Coverage	39,86%	8%	52%	51,95%

Table 2. `Dibgen` Coverage Results per Library (Decision Coverage)

	C++ Coverage Validator	Testwell CTC++	BullseyeCoverage	Squish Coco
`Color`	52,91%	9%	75%	78,65%
`Exception`	N.A.	30%	12%	40%
`Fileio`	15,15%	15%	44%	39,65%
`Internationalisation`	55,81%	77%	61%	70,89%
`Math`	62,36%	3%	30%	39,54%
`ModuleInterface`	14,05%	5%	23%	28,54%
`ParameterPool`	12,39%	6%	46%	58,95%
`ParameterPoolDocumentation`	N.A.	0%	0%	N.A.
`ProgramOptions`	0%	0%	59%	0%
`Progress`	55,07%	12%	47%	54,51%
`ResultDataPool`	5,31%	1%	20%	29,51%
`Serialization`	10,14%	11%	86%	76,31%
`Strings`	45,50%	36%	44%	50,66%
`Types`	70%	1%	8%	20,28%
`UserDataBase`	10,94%	77%	77%	81,51%
`Utilities`	0%	3%	0%	23,18%

Table 3. `Dibgen` Coverage Results per Library (Function Coverage)

	C++ Coverage Validator	Testwell CTC++	BullseyeCoverage	Squish Coco
`Color`	59,41%	6%	81%	81,17%
`Exception`	32,50	36%	45%	45,45%
`Fileio`	16,81%	26%	60%	56,52%
`Internationalisation`	77,67%	95%	94%	94,44%
`Math`	74,22%	3%	47%	47,08%
`ModuleInterface`	23,49%	4%	37%	35,68%
`ParameterPool`	36,13%	6%	72%	69,17%
`ParameterPoolDocumentation`	N.A.	0%	0%	N.A.
`ProgramOptions`	0%	0%	50%	0%
`Progress`	14,30%	9%	63%	68,46%
`ResultDataPool`	8,60%	2%	37%	34,69%
`Serialization`	11,22%	6%	86%	86,20%
`Strings`	49,46%	37%	65%	64,66%
`Types`	51,87%	1%	22%	22,88%
`UserDataBase`	10,45%	80%	86%	86,20%
`Utilities`	0%	7%	26%	30,76%

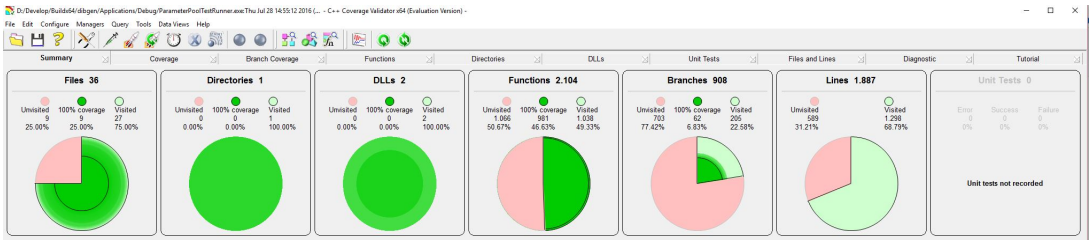

Figure 2. Coverage Validator's summary tab before improvements.

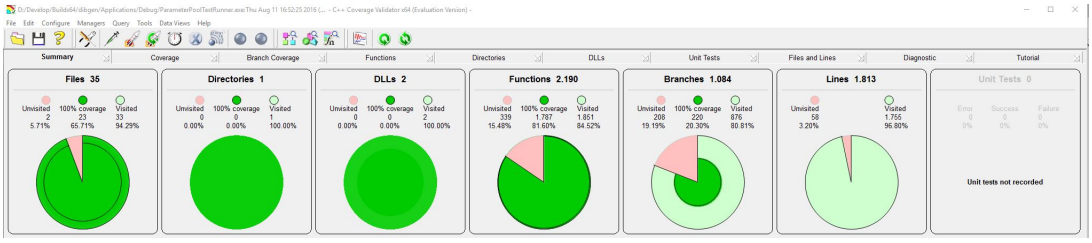

Figure 3. Coverage Validator's summary tab after improvements.

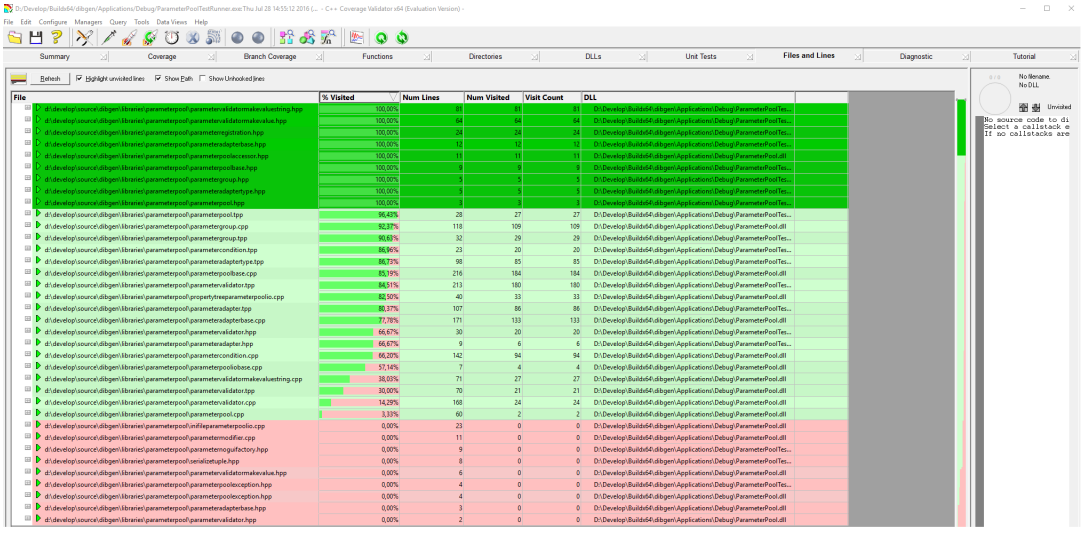

Figure 4. Coverage Validator's Files and Lines tab before improvements.

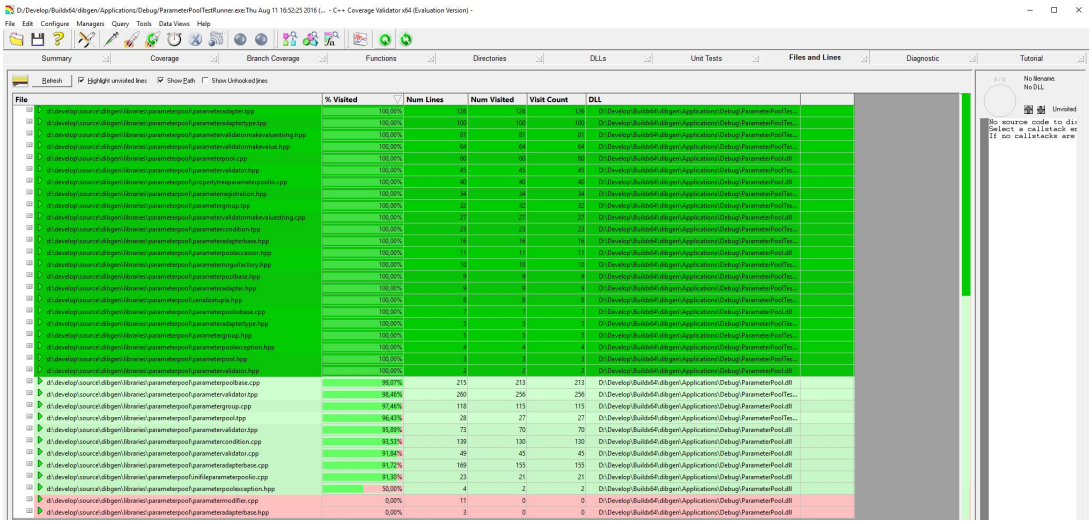

Figure 5. Coverage Validator's Files and Lines tab after improvements.

source code file could be entirely removed. Two of the untested source code files contained only source code that is used to disable default class behavior (make default constructor, copy constructor and/or assignment operator private), which makes this code untestable by design. Altogether, all of these mentioned modifications did not touch more than 10% of the test code, but resulted in a huge improvement in all code coverage measures (see Figure 3). As one can see in Figure 5, now from the remaining 35 source code files, 32 reach a line coverage above 90% (23 of which even reach 100% - compared to only 9 before the modifications were made). The 2 still remaining untested files contain the above mentioned disabling source code.

By improving the test code and the test data for the exemplarily chosen library, 3 implementation errors were found and corrected, 2 of which can be considered to potentially cause major problems in applications. Spending some effort in QA and improving the coverage of the tested source code will already pay off in the near future in several stages of the testing process; especially in regression and integration tests.

5 Conclusions

In this paper we presented our first steps in the direction of constructing a generic testing and evaluation protocol for CV applications. In our view, the performance characterization methodology in the domain can successfully be complemented with well known techniques borrowed from a typical quality assurance process.

The conducted experiments on JR's source code demonstrated that with little effort, by means of using a code coverage analysis tool for the available unit tests, the CV developers can considerably improve their code, and implicitly the release quality of the overall CV system.

After finishing unit/module-testing the program, we have to perform higher-order testing, as for instance integration and system tests (see Figure 1), in order to complete the testing process. Therefore, together with JR, we analyzed the requirements and possible use cases/hazards of one CV application, which was chosen as representative candidate in the Vision+ project[3]. We paid particular attention to the process of test case definition, with focus on: requirement(s)(from the requirements specification) related to a particular test case, its prerequisites (any conditions that must be fulfilled prior to executing the test), its detailed setup and preferred execution procedure (automated/manual). However, as usually a test management tool is used to accomplish the task, we further encourage CV developers to consider the integration of such a tool in their projects. Our colleagues from JR have already started out on analyzing the test management tools available for managing functional software and hardware testing in agile development projects. Some of the benefits one gains are the assurance of the complete test cycle, the repeatability of tests as well as the automatic generation of statistics and reports.

Finally, we would like to summarize the main ideas, which will further lead our work presented in this paper. On one hand, as resources are always limited, we have to find the right mixture of QA techniques and to focus towards specific CV pain points. In order to do this, it is important to determine the desired quality attributes for CV applications. On the other hand, we have to find a way to derive applicability rules for certain sets of CV algorithmic classes. Due to the vast diversity of CV algorithms, these tasks are rather difficult, however, the classical hierarchy of vision systems, which groups the them into low-, mid- and high-level processing levels, could serve as a starting point. At low-level vision, code structure and data representation are still in close correlations (in other words, every pixel has to be treated by some kind of operation/code), thus code improvement by QA directly affects the data quality. For example, filtering operations by convolutions (as those contained in our `Dibgiom` library) are many simple code snippets executed many times sequentially or in parallel, thus even small code discrepancies produce a large effect, which easily propagate further to higher processing levels. Mid- and high-level vision algorithms on the other hand, are more difficult to tackle, because the representations fall into one of the exponentially many branches of different meta-data types, where often the same meta-data can be produced by fundamentally different code pieces.

[3] http://comet-visionplus.at/

ACKNOWLEDGEMENTS

This work was partly funded by BMVIT/BMWFW under COMET programme, project no. 836630, by "Land Steiermark" trough SFG under project no. 1000033937, and by the Vienna Business Agency.

REFERENCES

[1] B. Beizer, *Black-box testing: techniques for functional testing of software and systems*, NY, USA, 1995.

[2] Kevin Bowyer and P. Jonathon Phillips, *Empirical Evaluation Techniques in Computer Vision*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edn., 1998.

[3] Patrick Courtney and Neil A. Thacker, 'Imaging and vision systems', chapter Performance Characterisation in Computer Vision: Statistics in Testing and Design, 109–128, Nova Science Publishers, Inc., Commack, NY, USA, (2001).

[4] Robert B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[5] Robert M. Haralick, 'Performance characterization in computer vision', *CVGIP: Image Underst.*, **60**(2), 245–249, (September 1994).

[6] Matej Kristan, Jiri Matas, Ales Leonardis, Tomas Vojir, Roman P. Pflugfelder, Gustavo Fernández, Georg Nebehay, Fatih Porikli, and Luka Cehovin, 'A novel performance evaluation methodology for single-target trackers', *CoRR*, **abs/1503.01313**, (2015).

[7] Moritz Menze and Andreas Geiger, 'Object scene flow for autonomous vehicles', in *Conference on Computer Vision and Pattern Recognition (CVPR)*, (2015).

[8] G. J. Myers, *The Art of Software Testing*, New Jersey, Second Edition edn., 2004.

[9] P. Jonathon Phillips, Hyeonjoon Moon, Syed A. Rizvi, and Patrick J. Rauss, 'The FERET Evaluation Methodology for Face-Recognition Algorithms', *IEEE Trans. Pattern Anal. Mach. Intell.*, **22**(10), 1090–1104, (October 2000).

[10] Daniel Scharstein, Heiko Hirschmller, York Kitajima, Greg Krathwohl, Nera Nesic, Xi Wang, and Porter Westling, 'High-resolution stereo datasets with subpixel-accurate ground truth.', in *GCPR*, eds., Xiaoyi Jiang, Joachim Hornegger, and Reinhard Koch, volume 8753 of *Lecture Notes in Computer Science*, pp. 31–42. Springer, (2014).

[11] N. A. Thacker, A. F. Clark, J. Barron, R. Beveridge, C. Clark, P. Courtney, W. R. Crum, and V. Ramesh. Performance characterisation in computer vision: A guide to best practices, 2005.

[12] K. Wiegers, *Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2002.

[13] Oliver Zendel, Wolfgang Herzner, and Markus Murschitz, 'VITRO - vision-testing for robustness', *ERCIM News*, (97), (2014).

[14] Oliver Zendel, Markus Murschitz, Martin Humenberger, and Wolfgang Herzner, 'CV-HAZOP: introducing test data validation for computer vision', in *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pp. 2066–2074, (2015).