How to Review, Verify, and Validate Software

M. Colom

miguel.colom@cmla.ens-cachan.fr
http://mcolom.perso.math.cnrs.fr/

July 21, 2013

1 How to Review, Verify, and Validate Software

The objective of reviewing an implementation of a published algorithm is simple: verifying that the program behaves exactly as described in the corresponding article. However, the level of complexity of software and interactions of the running program is really high, making the review a difficult process. The number of variables and different cases that must be taken into account is much larger in software engineering than in the rest of classic engineering processes.

Even worse, even if the authors of an algorithm are able to give a precise and rigorous mathematic description of an algorithm, they don't necessarily have a background in software engineering. This means that usually they don't follow well-known design and programming techniques, as for example design patterns [2]. From the point of view of the reviewer, it makes the source code difficult to revise, since it's difficult to recognize the structure of the program. The problem of bad designed programs can only be solved by instructing the authors on some basic software engineering and refactoring techniques [1] to improve it. Not only the authors but the reviewers too, of course.

However, even if the design of the program is correct from the point of view of the software engineering, it may present functional flaws. Clearly, a program with a modular and recognizable structure allows for the identification and isolation of the faulty parts with less difficulty. An example of a functional flaw is a program that runs correctly when running sequentially but fails or gives different results when parallelized. A race condition in a parallel loop with openMP is a classic example.

Therefore, to review and verify correctly a program, the following steps should be performed:

1. Analyze the structure of the program and suggest a refactoring if needed, even if the program runs correctly. If the program doesn't

run correctly, this step should be mandatory.

2. Verify that the description of the algorithm coincides exactly with the operations that the program actually executes.

3. Run a battery of tests of the program, to detect functional flaws.

At least, the second and third steps should always be performed.

In order to verify the behavior of the program automatically, the author should provide an script that determines if the execution has been successful or not. For example, if the program runs the FFT of an image, it could run it on a known image and compare it with the expected theoretical result.

The set of tests might include these:

- Compare the execution with and without openMP support. The Makefile of the program should allow to enable and disable openMP easily (with flag, for example).

- Compare the execution of the same executable generated with different compilers.

- Compare the execution of the program in different machines with different architectures.

- Use tools as Valgrind[1] and others to detect buffer overruns and memory leaks.

- Use tools as kcachegrind[2] and others to analyze the performance of the code.

It's also important that the server constantly monitors the execution of the programs and detects any failure. For example, in IPOL right now the logcheck[3] sends an email to the Tech team each time an executable fails during its execution.

For the final validation, a program should only considered validated if a prudential time (say, three months) has elapsed without any notice of failure. Of course, this doesn't ensure that a program considered valid doesn't contain any design or functional problem. Therefore, if a problem is detected after the validation of a program, it should be put in a special state ("in revision", for example) to inform the users that a problem has been detected on the program. The authors must provide a correction and upload a new version of their algorithm.

[1]http://valgrind.org/
[2]http://kcachegrind.sourceforge.net/
[3]http://logcheck.org/

References

[1] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[2] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49:120, 1995.