Scientific Software Testing: A Practical Example

BOJANA KOTESKA and ANASTAS MISHEV, University SS. Cyril and Methodius, Faculty of Computer Science and Engineering, Skopje
LJUPCO PEJOV, University SS. Cyril and Methodius, Faculty of Natural Science and Mathematics, Skopje

When developing scientific applications, scientists are mostly interested in getting scientific research achievements. Usually, the first phase in scientific applications development process is coding. Testing is rarely performed or it is not performed if the final result is correct. Many bugs are found later and problems arise when no software documentation can be found. The goal of this paper is to improve the testing of scientific applications by describing the full testing process which follows the software engineering testing principles. In order to confirm the usefulness of the process, we develop and test an application for solving 1D and 2D Schrödinger equation by using the Discrete Variable Representation method (DVR). Using this testing process in practice resulted in requirements specification, test specification and design, finding connection between specified requirements and tests, automated testing and executable tests.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification —*Validation*; G.4 [**Mathematics of Computing**]: Mathematical Software—*Certification and testing, Documentation, Verification*

General Terms: Verification
Additional Key Words and Phrases: Scientific application, Testing process, Software Engineering, Schrödinger equation, Software quality

1. INTRODUCTION

Scientific applications perform numerical simulations of different natural phenomena in the filed of computational physics and chemistry, mathematics, informatics, mechanics, bioinformatics, etc. They are primarily developed for the scientific research community and usually they are used for simulating some I/O and data extensive experiments [Segal 2005]. The performance of such simulation experiments requires powerful supercomputers, high performance or Grid computing [Vecchiola et al. 2009]. Formally, a scientific application is a software applications which turns the object into mathematical models by simulating activities from the real world [Ziff Davis Publishing Holdings 1995].

Scientific applications are different from commercial application, especially in the process of testing and evaluation. They are developed for the specific research community and not evaluated by customers. Testing is usually performed by comparing the obtained results with theory or performed physical experiments. Also, scientists can decide for the correctness of the results visually, by comparing images.

The absence of software engineering testing practices and documentation is confirmed by the results obtained in the survey we conducted among participants in the High Performance - South East Europe

Author's address: Bojana Koteska, FCSE, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje; email: bojana.koteska@finki.ukim.mk; Ljupco Pejov, FNSM, P.O. Box 1001, 1000 Skopje; email: ljupcop@iunona.pmf.ukim.edu.mk; Anastas Mishev, FCSE, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje;email: anastas.mishev@finki.ukim.mk.

In: Z. Budimac, M. Heričko (eds.): Proceedings of the 4th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications (SQAMIA 2015), Maribor, Slovenia, 8.-10.6.2015. Also published online by CEUR Workshop Proceedings (CEUR-WS.org, ISSN 1613-0073)

(HP-SEE) project [Koteska and Mishev 2013]. What we are trying to achieve is to change these practices and to adapt some testing practices of the software engineering which according to the IEEEStd 610.12-1990, is an application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software [The Institute of Electrical and Electronics Engineers 2015a].

The goal of this paper is to describe a practical example of testing a scientific application. It will help scientists to learn to include software engineering practices in order to change the current testing practices in different stages of scientific applications development process. In our previous paper [Koteska et al. 2015], we proposed a framework for a complete scientific software development process which provides a set of rules, recommendations and software engineering practices for all development stages. The framework follows the incremental software development with changes in the process of evaluation, in test-driven design and short reports after each iteration. Each increment of the development process contains the following 8 phases: planning, requirements definition, system design, test cases design, coding, testing, evaluation, writing short report.

In this paper, we give special emphasis on the testing process in different development stages. In addition, we adapt and modify the existing software engineering practices for developing commercial applications to successfully include them in the requirements specifications, test cases design, testing and evaluation phase. The proposed testing process is validated with the testing of an application for solving 1D and 2D Schrödinger equations by using the DVR numerical method. The application is tested by following the suggested recommendations. As a result, we have specified test cases, testing functions and automated testing.

The structure of the paper is organized as follows. Section 2 discusses the related work in the area of scientific application testing. The testing process as a part of the development framework for scientific applications and the testing of the application for solving 1D and 2D Schrödinger equation by using the DVR method is presented in Section 3. Section 4 is devoted to the conclusion and future work.

2. RELATED WORK

There are several papers in which authors find some inconsistencies and give recommendations for improving the testing of the scientific applications, but no paper describes the full testing process. In [Sanders and Kelly 2008], the authors found out that scientists use testing only to show that their theory is correct, not that the software does not work. They claimed that testing the computational engine is important, but they are aware of their disorganized and wrong testing practices.

Cox and Harris [Cox and Harris 1999] elaborated a general methodology for evaluating the accuracy of the results produced by scientific software which has been developed as a part of the National Physical Laboratory research. Their idea is to generate and use reference data sets and corresponding reference results to undertake black-box testing. According to this method, the data sets and results can be generated in a consistency with functional specification of the software. The obtained results for the reference data sets are compared with the reference results.

The efficiency of inclusion of software engineers in the process of scientific applications testing is confirmed in [Kelly et al. 2011]. The authors said that software engineer brings a toolkit of ideas, and the scientist chooses and fashions the tools into some thing that works for a specific situation. The results are: an approach to software assessment that combines inspection with code execution and the suppression of process-driven testing in favor of goal centric approaches.

In [Hook and Kelly 2009], the authors highlight the two main reasons for poor testing of scientific software: the lack of testing oracles and the large number of tests required when following any standard testing technique described in the software engineering literature. They also suggest a small number of well chosen tests that can help in finding a high percentage of code faults.

Baxter et al. [Baxter et al. 2006] proposed testing according to the test plans which should be developed in the design phase. They emphasize the need for unit testing which should ensure that a particular module is working properly. Also, bugs can be identified early if testing is performed iteratively throughout the development process. The authors suggest applying of data standards and quality addressing through performance optimization.

In [Hatton 1997], the authors described two large large experiments: measuring the consistency of several million lines of scientific software written in C and Fortran 77 by static deep-flow analysis across many different industries and application areas and measuring the level of dynamic disagreement between independent implementations of the same algorithms acting on the same input data with the same parameters in just one of these industrial application areas. Their results showed that commercially released C and Fortran software are full of statically detectable fault and irrespective of the existence of any quality system, level of criticality, or application area.

3. TESTING PROCESS OF SCIENTIFIC APPLICATIONS

3.1 Testing Methodology

Scientific software testing is complex mainly because the results can not be evaluated by users, but the they are often compared with results from real experiments or they are based on specific scientific theory. In our previous paper [Koteska et al. 2015], we have explained the steps we used for developing an application for solving 1D and 2D Schrödinger equation and in this paper we are focused on functional testing of the application.

Our testing paradigm is: test small software pieces and test often. In order to achieve this, we perform unit testing of each module, in each increment of the application development process. Integration testing is also performed at the end of an increment. Functional testing is closely related to functional requirement specification in which all application functionalities (also inputs and outputs of a functions) are described.

Well written requirements lead to well specified test cases. Since our application is split in independent modules, there is a functional requirement for every module. Each functional requirement is identified by a specific id, name, version, history of changes and description.

Test cases are defined by using standardized forms such as [The Institute of Electrical and Electronics Engineers 2015b] and also some templates [Assassa 2015]. They are described also by unique identifier, name, goal, preconditions, execution environment, expected and actual results, status (passed/-failed) and history of changes. It is useful to specify the boundary values or range of values when numerical calculations are performed. Test cases should be written before the coding phase.

We recommend white box testing and automated testing. Also, it is very important to eliminate the redundant tests. Testing process is completed when all tests pass successfully and source coverage is achieved. Test automation can be achieved by various testing frameworks. Since our application is written in C language, we recommend the following frameworks (Check, CUnit, AceUnit, CuTest, etc.). These frameworks are well documented and very easy to use. They provide methods and structures, the user should insert code assertions. The connection between modules and functionality of the application is checked by using the integration testing at the end of each iteration.

The errors found in each iteration must be corrected before the next iteration starts. Errors with higher priority, such as critical errors, should be corrected first. Evaluation of results is usually performed by scientists in the research group.

3.2 Scientific Problem - Solving 1D and 2D Schrödinger Equation by using Discrete Variable Representation Method

Schrödinger equation is a partial differential equation which is used to describe the system dynamics at atomic and molecular level.

A time independent Schrödinger equation can be defined as:

$$H * \Psi = E * \Psi \tag{1}$$

where Hamiltionian operator is denoted by H, Ψ is the wave function of the quantum system and E is the energy of the state Ψ.

A time dependent Schrödinger equation can be represented as:

$$i * \hbar \frac{\partial \Psi}{\partial t} = H * \Psi \tag{2}$$

where $\hbar$ is the Planck Constant divided by 2Π and $\frac{\partial}{\partial t}$ is a partial derivative with respect to time t.

One of the ways for solving Schrödinger equation by using matrix representation is the discrete variable representation which is based on truncated standard orthonormal polynomial bases and the corresponding Gaussian quadratures. The matrix elements of differential operators are calculated exactly and those of the operators which are local in the coordinate representation, such as the potential energy operator, are calculated approximately by using Gaussian quadrature accuracy. The Gaussian quadrature gives the most accurate approximation to an integral for a given number of points and weights and it contributes to the DVR method accuracy and efficiency [Szalay et al. 2003].

Discrete variable representation (DVR) methods are widely used because they can solve efficiently numerical quantum dynamical problems. DVR's provide simplification of the kinetic energy Hamiltonian matrix elements calculation and potential matrix elements which represent the potential of the DVR. In a case of multi-dimensional systems, the Hamiltonian matrix is sparse matrix and the operation of the Hamiltonian on a vector is fast when DVR's product is calculated [Light and Carrington Jr 2000].

The goal of this paper is to test a scientific application for solving 1D and 2D Schrödinger equation which is developed in increments. A description of the testing process in each increment is given in the following subsection.

3.3 Testing Process

At the beginning of each increment we have specified the functional requirements. We have identified 13 functional requirements in the first increment and 4 in the second increment. Our application is decomposed in independent modules and functional requirements are tightly connected to modules.

The following modules were developed in the first increment: addition of two matrices, multiplication of two matrices, multiplication of scalar and matrix, printing of matrix elements, matrix transposition, making a diagonal matrix, division of vector by scalar, multiplication of matrix by scalar, making an identity matrix, reading of a file content, comparing two rows of an input file, sorting content in a file, calculation of eigenvalues and module for calculating: arrays of x and y (only in 2D) points in finite basis representation (FBR), transforming matrices for transforming x and y points from FBR to discrete value representation (DVR).

In the second increment, we have developed 4 modules: interpolation function (Hooke method), module for finding the local minimum near given point (x,y), module for transforming the input file of points and function values in those points and module for solving the 1D and 2D Schrödinger equation which outputs the frequencies of various vibrational transitions and vibrational wavefunctions on 1D or 2D grid of points.

Next stage connected to testing is the designing of test cases for each functional requirement. We have used the following template for defining test cases as shown in Tables I and II. Both test cases are written in the first increment before testing.

Table I. An Example of Test Case Definition - Module for Making Diagonal Matrix

Test Case Id	06			
Test Case Name	Diagonal Matrix			
Short Description	A module for making a diagonal square matrix from an array of numbers. If array is denoted by A, and matrix by M, then M[i][i]=A[i], where n is the size of A and M. The elements which not lie on the main diagonal are 0's. The return element is the matrix M.			
Pre - conditions	An array of n double elements must be provided as an input and size n must be a integer number.			
Step	Action	Expected Result	Status (Pass/Fail)	Comments
1	Initialization of a 2D array M of size n x n	Allocated memory for M with size n x n. Elements are of double type	Pass	A dynamic allocation is used
2	Assigning elements to the matrix M	Elements of the array A are assigned to matrix main diagonal, the other elements are 0's. M[i][i]=A[i], M[i][j]=0, i!=j	Pass	/

The next phase is the unit and integration testing of the application. The well specified test cases are very useful for the testing process. In order to perform automatic testing, we use a system for writing, editing and running unit tests in C, called CuTest [CuT 2015]. The testing of a system functionality is finished when the statuses for all steps of the test case which describes that functionality are "pass".

An example of testing function is shown in Figure 1. This test function is connected to test case 13 which is described above. The values of elements of arrays ptsx, fbrtx and matrix Tx are tested. We are comparing the actual and expected results of variables.

One way to evaluate the final results is to test the convergence of the results with increasing the density of grid points which is equivalent to increasing the number of basis functions or to compare them to results obtained for existing application written in another programming language.

4. CONCLUSION AND FUTURE WORK

In this paper, a testing process of scientific applications is described. It is shown that the requirement specification and test cases design phases are tightly connected to the testing phase because we specified test case for each functional requirement and there is a test designed for each test case. We provide an example and identify challenges for scientific application development. We have tested the application for solving 1D and 2D Schrödinger equation by using the DVR method. We specified test cases for each application module and write tests by using the CuTest framework. The testing process is automated since all tests are executed continuously. It is a good practice to repeat the testing of a given function if some code changes are performed inside the function. The most important lessons learned by this research are that current scientific software testing practices must be changed and the software engineering practices can be successfully included in the scientific application testing.

Table II. An Example of Test Case Definition - Module for Generating Arrays of x Values and y Values, Arrays of x and y Values in FBR and Transformation Matrices for Transforming x and y values from FBR to DVR (thcheby).

Test Case Id	13			
Test Case Name	Generating Arrays of x Values and y Values, Arrays of x and y Values in FBR and Transformation Matrices for Transforming x and y values from FBR to DVR.			
Short Description	This module has five input arguments: nx (number of x values - integer) , xmin (minimum value of x - double), xmax (maximum value of x - double), ny (number of y values - integer), ymin (minimum value of y - double), ymax (maximum value of y - double). First, the differences between the maximum and minimum x values (deltax) and the maximum and minimum y (deltay) values are calculated. Next, the array of x values (ptsx) and array of y values (ptsy) are generated by calling the function make_array_ptsxy(nx,nx+1,xmin,deltax) for x values and make_array_ptsxy(ny,ny+1,ymin,deltay) for y values. Then, the elements of arrays of x values(fbrtx) and y values (fbrty) in FBR are calculated by calling the function make_array_fbrtxy(deltax,nx) for x values and make_array_fbrtxy(deltay,ny) for y values. At the end the transformation matrices Tx and Ty are created by calling the function make_matrix_Txy(nx,nx+1) for x values and make_matrix_Txy(ny,ny+1) for y values;			
Pre - conditions	The number of x and y values must be known (nx and ny). Also the minimum and maximum values of x and y must be initialized (xmin, xmax, ymin, ymax).			
Step	Action	Expected Result	Status (Pass/Fail)	Comments
1	Calculation of difference between the maximum and minimum value of x.	deltax=xmax-xmin Elements are of double type	Pass	/
2	Calculation of difference between the maximum and minimum value of y.	deltay=ymax-ymin Elements are of double type	Pass	/
3	Generating the elements of the array ptsx.	ptsx[i]=((i+1)*deltax*1.0)/(nx+1) +xmin Elements are of double type	Pass	The results are tested by testing the function make_array_ptsxy (nx,nx+1,xmin,deltax)
4	Generating the elements of the array ptsy.	ptsy[i]=((i+1)*deltay*1.0)/(ny+1) +ymin Elements are of double type	Pass	The results are tested by testing the function make_array_ptsxy (ny,ny+1,ymin,deltay)
5	Generating the elements of the array fbrtx.	fbrtx[i]= (((i+1)*Π)/deltax)^2 Elements are of double type	Pass	The results are tested by testing the function make_array_fbrtxy (deltax,nx)
6	Generating the elements of the array fbrty.	fbrty[i]= (((i+1)*Π)/deltay)^2 Elements are of double type	Pass	The results are tested by testing the function make_array_fbrtxy (deltay,ny)
7	Generating the elements of the matrix Tx.	Tx[i][j]= $\sqrt{2.0/(nx+1)}$ $*\sin((i+1)*(j+1)*\Pi/(nx+1))$ Elements are of double type	Pass	The results are tested by testing the function make_matrix_Txy(nx,nx+1)
8	Generating the elements of the matrix Ty.	Ty[i][j]= $\sqrt{2.0/(ny+1)}$ $*\sin((i+1)*(j+1)*\Pi/(ny+1))$ Elements are of double type	Pass	The results are tested by testing the function make_matrix_Txy(ny,ny+1)

Our further aim is to propose a software development process for scientific applications. Our future work will be directed in empirical research and number of case studies that would be defined with goal to develop software development process and all supporting guidelines, methods and techniques. Also, we will focus on development of more complex and critical scientific applications where the inclusion of software formal engineering practices is indispensable.

```
void Test_thcheby(CuTest *tc)
{
    struct return_objects result=thcheby(10, 1, 5, 10, 2, 6);
    double *actualptsx=result.ptsx;
    double *actualfbrtx=result.fbrtx;
    double **actualTx=result.Tx;
    int i,j;
    double *expectedptsx=malloc(10*sizeof(double));
    double *expectedfbrtx=malloc(10*sizeof(double));
    double **expectedTx=malloc(10*sizeof(double));
    for(i=0; i<10; i++)
    {
        expectedptsx[i]=((i+1)*(5-1)*1.0)/11+1;
        expectedfbrtx[i]=square(((i+1)*M_PI)/(5-1));
        CuAssertTrue( tc, abs(expectedptsx[i]- actualptsx[i])<0.0000001);
        CuAssertTrue( tc, abs(expectedfbrtx[i]- actualfbrtx[i])<0.0000001);
        expectedTx[i] = malloc(10*sizeof(double));

        for(j=0; j<10; j++)
        {
            expectedTx[i][j]=sqrt(2.0/11)*sin((i+1)*(j+1)*M_PI/11);
            CuAssertTrue( tc, abs(expectedTx[i][j]- actualTx[i][j])<0.0000001);
        }
    }
}
```

Fig. 1. Function for Testing the method ”thcheby”

REFERENCES

2015. CuTest: C Unit Testing Framework. (March 2015). http://cutest.sourceforge.net/

Dr. Ghazy Assassa. 2015. Software Engineering Test Case Template and Examples. (May 2015). [Online]. Available: http://faculty.ksu.edu.sa/ghazy/CSC342_Tools/Test%20Case%20Template.pdf.

Susan M Baxter, Steven W Day, Jacquelyn S Fetrow, and Stephanie J Reisinger. 2006. Scientific software development is not an oxymoron. *PLoS Computational Biology* 2, 9 (2006), e87.

MG Cox and PM Harris. 1999. Design and use of reference data sets for testing scientific software. *Analytica Chimica Acta* 380, 2 (1999), 339–351.

Les Hatton. 1997. The T experiments: errors in scientific software. *Computing in Science and Engineering* 4, 2 (1997), 27–38.

Daniel Hook and Diane Kelly. 2009. Testing for trustworthiness in scientific software. In *Software Engineering for Computational Science and Engineering, 2009. SECSE'09. ICSE Workshop on*. IEEE, 59–64.

Diane Kelly, Stefan Thorsteinson, and Daniel Hook. 2011. Scientific software testing: analysis with four dimensions. *Software, IEEE* 28, 3 (2011), 84–90.

Bojana Koteska and Anastas Mishev. 2013. Software Engineering Practices and Principles to Increase Quality of Scientific Applications. In *ICT Innovations 2012*, Smile Markovski and Marjan Gusev (Eds.). Advances in Intelligent Systems and Computing, Vol. 207. Springer Berlin Heidelberg, 245–254. DOI: http://dx.doi.org/10.1007/978-3-642-37169-1_24

Bojana Koteska, Ljupco Pejov, and Anastas Mishev. 2015. Framework for Developing Scientific Applicatons- Solving 1D and 2D Schrodinger Equation by using Discrete Variable Representation Method. In *The First International Conference on Advances and Trends in Software Engineering SOFTENG 2015, in print.*

John C Light and Tucker Carrington Jr. 2000. Discrete-variable representations and their utilization. *Advances in Chemical Physics* 114 (2000), 263–310. http://dx.doi.org/10.1002/9780470141731.ch4

Rebecca Sanders and Diane Kelly. 2008. Dealing with risk in scientific software development. *IEEE software* 25, 4 (2008), 21–28.

Judith Segal. 2005. When Software Engineers Met Research Scientists: A Case Study. *Empirical Software Engineering* 10, 4 (2005), 517–536. DOI: http://dx.doi.org/10.1007/s10664-005-3865-y

Viktor Szalay, Gábor Czakó, Adám Nagy, Tibor Furtenbacher, and Attila G Császár. 2003. On one-dimensional discrete variable representations with general basis functions. *The Journal of chemical physics* 119, 20 (2003), 10512–10518.

The Institute of Electrical and Electronics Engineers. 2015a. IEEE Standard Glossary of Software Engineering Terminology. The Institute of Electrical and Electronics Engineers, New York, NY, USA. (March 2015). [Online]. Available: http://www.idi.ntnu.no/grupper/su/publ/ese/ieee-se-glossary-610.12-1990.pdf.

The Institute of Electrical and Electronics Engineers. 2015b. Test Case Specification Template (IEEE 829-1998). The Institute of Electrical and Electronics Engineers, New York, NY, USA. (May 2015). [Online]. Available: http://www.ufjf.br/eduardo_barrere/files/2011/06/SQETestCaseSpecificationTemplate.pdf.

C. Vecchiola, S. Pandey, and R. Buyya. 2009. High-Performance Cloud Computing: A View of Scientific Applications. In *Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms and Networks (I-SPAN 2009)*. IEEE Computer Society.

Inc. Ziff Davis Publishing Holdings. 1995. PC Magazine. (1995). http://www.pcmag.com/encyclopedia/term/50872/scientific-application