

©2012 IEEE. Reprinted, with permission, from **Rommel H., Paech B., Bastian P., Engwer C., System Testing a Scientific Framework using a Regression Test Environment, Computing in Science and Engineering Vol. 14, No. 2, March 2012.**

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Heidelberg's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

System Testing for a Scientific Framework Using a Regression Test Environment

Authors

Hanna Remmel and Barbara Paech

Institute for Computer Science
University of Heidelberg
Im Neuenheimer Feld 326, 69120 Heidelberg, Germany
+49 6221 54 – {5817, 5810}
{remmel, paech}@informatik.uni-heidelberg.de

Christian Engwer

Institute for Computational and Applied Mathematics
University of Münster
Einsteinstrasse 62, 48149 Münster, Germany
+49 251 83-35067
christian.engwer@uni-muenster.de

Peter Bastian

Interdisciplinary Centre for Scientific Computing (IWR)
University of Heidelberg
Im Neuenheimer Feld 368, 69120 Heidelberg, Germany
+49 6221 54 –8261
peter.bastian@iwr.uni-heidelberg.de

Abstract

Testing a scientific framework is a challenging task, since, among other typical challenges in testing scientific software, one has to find a way to deal with the large variability in a framework. Our approach is to apply software product line engineering to handle the frameworks variability. We use variability modeling to support the selection of test applications and test cases.

In this article we introduce the regression test environment we developed for DUNE, a complex scientific framework. The test environment consists of system tests that are constructed to take the huge variability of possible uses for the framework into account. We use system testing, since it is the only testing level where the interaction between the mathematical model, the numerical model and its implementation can be thoroughly tested. The tests also support algorithm verification and scientific validation.

Keywords

D.2.5.o Test execution
D.2.19.c Methods for SQA and V&V

Introduction

Scientific software development teams find the testing of scientific software very challenging. The special difficulties stem from missing test oracle, the high priority of non-functional requirements and the need for high performance parallel computing [1]. Most existing testing methods cannot be applied straight away without adjusting them. Still many teams consider verification and validation as essential elements of the development. As Baxter states in [2]: “if software is meant to do something, then that something can – and should – be tested for.” Although the context of scientific software is special, the benefits of well-planned testing are just as favorable.

When testing scientific software, it is important to distinguish between the different possible sources of a failure: the underlying mathematical model, the translation of the mathematical model to a particular numerical algorithm, and the implementation of that algorithm into program code. Each possible source of failure should be handled separately. Hook and Kelly [3] point out that these steps should be carried out in a strict order: first check the program code for bugs with code verification methods and then verify the numerical algorithm with numerical algorithm verification methods. Only after these two steps, the scientists are able to perform the scientific validation (evaluate, whether the output of the software is a reasonable approximation to the real world) knowing that defects in the code and the numerical algorithms have already been excluded.

For each of these steps of testing scientific software, several methods have been introduced. Oberkampff et al. [4] give a broad overview of existing methods for verification, validation and prediction capability in computational science. Especially the suitability of methods for algorithm verification (i.e. grid convergence testing, symmetry and conservation tests) strongly depends on the mathematical model used in the scientific software. It is a challenge to choose a suitable combination of different verification and validation methods for an application.

Our research concentrates on testing scientific frameworks. A framework consists of common code that provides solutions for several similar applications for specific types of problems. Frameworks differ from software libraries, among other things, in the following two ways. First, the flow of control is not dictated by the caller, but by the framework (inversion of control). Second, a framework can be extended by the user by overriding functionality or by implementing interfaces [5]. DUNE, the software we deal with, is a complex scientific framework for solving partial differential equations supporting a large variety of applications (i.e. fluid mechanics or transport in porous media), mathematical models and numerical algorithms [<http://www.dune-project.org/>].

On the component level, the scientists developing DUNE use unit testing to check new and changed functionality. The code verification for DUNE is done by the unit tests. Recently we have developed a regression *test environment*, an infrastructure for testing, on system testing level for DUNE. The *test applications* we implemented for the test environment each use the DUNE framework to solve a specific problem including the mathematical and numerical models. Each of these test applications has a set of parameters. A *test case* for a test application can be defined by selecting a value for each test application parameter. The test applications support numerical algorithm verification and scientific validation.

Testing a scientific framework is a challenging task, since we need to have a way to deal with the large variability in a framework. Our approach is to apply software product line engineering (SPLE) to handle the frameworks variability [6]. We use variability modeling to support the selection of test

applications and test cases. System test planning and the selection of test cases in SPLE is a complex task. The risk of double testing as well as the risk of overlooking important tests are high. The testing is done across three dimensions: the different testing levels (unit, integration and system), regression testing as the system evolves over time and the testing of the huge variability in the product line. It is a big challenge to balance the testing effort across these three dimensions [8].

Case studies like [7] confirm the fact that there is a lack in system testing in scientific software development. In that paper Ackroyd et al. point out that testing actions in the analyzed scientific software are insufficient and urge the scientists to focus on an intensive usage of unit testing. The authors admit that a remaining problem is how to ensure that code changes do not cause regression defects, if system testing is insufficient. Even though unit tests can demonstrate that every component works as expected, they still do not ensure that the components work together in a correct way when integrated.

In the following, we first introduce DUNE in more detail. Then we discuss the concepts we use in the DUNE test environment and demonstrate our approach with a concrete example. Finally, we discuss our experiences while using the test environment.

DUNE – a Scientific Framework

DUNE, the Distributed and Unified Numerics Environment, is a free software licensed framework for solving partial differential equations with grid-based methods [9] [10]. It supports the easy implementation of discretization methods like finite element, finite volume and finite difference methods. DUNE makes several grids and powerful numerical implementations available. Its main principles are the separation of data structures and algorithms by abstract interfaces, the efficient implementation of these interfaces using generic programming techniques and the reuse of existing finite element packages (i.e. UG [<http://atlas.gcsc.uni-frankfurt.de/~ug/>], ALBERTA [<http://www.alberta-fem.de/>] and ALUGrid [<http://aam.mathematik.uni-freiburg.de/IAM/Research/alugrid/>]) with a large body of functionality.

DUNE consists of several separate modules. Its users can put together a certain set of modules depending on their needs. The core modules deliver the basic classes (dune-common), an abstract grid interface (dune-grid), a linear solver library (dune-istl), an interface for finite element shape functions (dune-localfunctions) and tutorials for the use and implementation of the grid interface.

Additional to the core it is possible to use external modules in DUNE. There are several of them including modules for complete simulations, additional grid managers and discretization. In our research, we concentrate on dune-pdelab, a discretization module for a wide range of methods.

The development of DUNE started about ten years ago. The distributed development team for the core modules consists currently of 10 scientists from mathematics, computer science and physics. Additionally there are up to 20 scientists working on external modules. DUNE core modules consist of over 250.000 LOC in C++. DUNE supports parallelism based on Message Passing Interface (MPI).

Some users use DUNE's interfaces to implement their own external modules. Most of the users use core and external modules to implement their own applications. Still others just use ready implemented DUNE applications. Users of DUNE are mostly mathematicians, computer scientists and physicists working in academia. Recently it was adopted for industrial applications for flow and

transport processes in porous media [<http://www.sintef.no/Projectweb/GeoScale/Simulators/>]. Altogether, there are about 50-100 users.

The development team applies software engineering best practices like version management and configuration management. New requirements are collected using mailing lists and an issue tracking tool. Rapid prototyping is used to some extent. The high use of software engineering practices is untypical for scientific software [1] and may be due to the fact that some members of the development team are computer scientists.

Big code changes are planned as milestones with some kind of a prioritization, however, without defined scheduling. The development is done whenever resources are available. The documentation, available online on the DUNE project's web page, consists of detailed code documentation, user documentation and tutorials on mathematical concepts and their implementation.

The most important software quality goals for DUNE are efficiency, flexibility, numerical correctness and portability, especially on high performance computers. The quality of single modules is tested with unit tests and there are some automated configuration tests which are run on every commit or overnight.

Test Environment for DUNE – used concepts

In this section we introduce the concepts used in the DUNE test environment. Although the different DUNE framework modules have already been tested using unit testing, there is a need for further quality assurance which is not covered by the unit tests. Firstly, as unit tests only test single components, there is a need for system tests that test the collaboration of different modules for concrete uses of the framework. Secondly, we needed a way to test the huge variability of different possible applications in the framework. Thirdly, it should be ensured that changes in the source code due to new development or bug fixing do not have unexpected side effects on other applications of the framework. This is done with regression testing. In addition to this we wanted to apply suitable mathematical testing in the form of algorithm verification and scientific validation in the planned test environment. These elements of the test environment are introduced in more detail in the following subsections.

Please note that the ideas for a test environment introduced in this section are not specific for scientific frameworks but can also be used for other types of scientific software.

System Testing

Software testing can be divided into three levels depending on the target of the testing: unit testing, integration testing and system testing. Each of these testing levels is important and should not be neglected. In unit testing, the goal is to verify the functionality of single software units, small pieces of the software that are separately testable. In integration testing, the interaction between different software components are verified. System testing concentrates on the behavior of the whole software system. At this level the focus can also be set on non-functional system requirements like performance or accuracy [www.swebok.org/].

The DUNE development team uses unit testing to check the functionality of DUNE algorithms. Integration testing in the DUNE development team is partly done by using the unit testing tool, but in future work we plan to formalize this testing level, too. Most functional failures should already be

identified during unit and integration testing, but system testing is still the only level of testing that considers concrete DUNE applications.

For scientific software, especially scientific frameworks, system testing is the only testing level where the interaction between the mathematical model, the numerical model and its implementation can be thoroughly tested.

While unit testing is widely used for scientific software, currently the use of system tests is hardly mentioned in the literature. This may be partly due to the fact that there are several good tools for adopting unit testing in many different programming languages. It is easier to plan and implement tests for smaller pieces of the software. These steps can easily be integrated into the everyday work of scientific software developers when they are implementing or changing pieces of the software.

System tests on the other hand, still mostly have to be planned and implemented from scratch. Their form and goals can be very different from one scientific software product to another. It needs a high level of domain understanding to plan system tests.

In the next subsection we explain how we selected these test applications and the test cases for each test application. One example test application is introduced in the next section.

Dealing with the Huge Variability in a Framework

One interesting research question in testing scientific frameworks is how to choose a suitable set of test applications and test cases for each test application out of the numerous possibilities. Since it is not feasible to test every possibility, we need to find a way to deal with the large variability in a framework. Our idea is to define the framework as a product line and use the variability modeling of software product line engineering (SPLE) to model the necessary parts of the framework's variability with a variability model [11].

At the moment the test environment uses the DUNE core modules and the external module `dune-pdelab` for the solving of partial differential equations. Later on, it can be expanded to test other external DUNE modules, too.

The test applications were chosen so that they cover typical uses of `dune-pdelab`, meaning that typical problems, mathematical models and numerical algorithms are covered. The goal was to reach possible high coverage for the available numerical algorithms in the DUNE framework. Due to the use of C++ templates in the DUNE implementation comprehensive code coverage analysis is not possible with any tool familiar to us.

We also included some special test problems in these test applications, for example complex grid structures, that are known to be tricky for the used numerical methods. After creating a variability model for each test application, we use this model as a basis for systematically selecting the set of test cases for a test application.

In SPLE the idea is to develop a software platform and use mass customization for the creation of a group of similar applications that differ from each other in specific predetermined characteristics [6]. The characteristics that can vary are called *variation points* and the possible values for a variation point are called *variants*. A *variability model* includes all variation points and their variants. It also includes the constraints between the variation points and variants.

We developed a process for creating variability models for DUNE. A detailed description of the process can be found in [11]. Together with the scientists, we first created a roadmap, which determined the procedure a DUNE user follows when creating a DUNE application and recorded the decisions she or he has to make.

After that the goal was to create a variability model based on the roadmap. The first step in the creation of the variability model for DUNE was the identification of the variation points. We did this by examining the roadmap and writing down the characteristics where the applications differ from one another. After the variation points were written down, the set of variants was defined for each variation point. Next, we defined the constraints between the variation points and the variants. This was done based on the scientists' knowledge in their field of research and the mathematical theory underlying the applications.

At last, we derived the test cases for each test application using the variability models. This was done by binding the variability in the variability model which means that we choose a specific variant for each variation point.

Ideally, the test cases cover the whole variability in the variability model. Since it is not always feasible to construct test applications for all possible variant combinations, it remains the responsibility of the scientists to reject the combinations that need not to be included in the set of test cases since they are unlikely to occur. Using the variability model as a basis for choosing the test cases for a test application gives the scientists the confidence that they do not miss anything important. Being able to comprehend the test coverage of the variability model, the scientists can gain trust in their choice of test applications.

In the next section we demonstrate this approach for an example test application in the DUNE test environment. Before that, in the next subsections, we explain how we apply regression testing in our test environment.

Automated Regression Testing

Automating test runs on a certain level of testing and repeating them on a regular basis leads to a regression test environment. The main idea is to show that modifications in the software code do not cause any unwanted side effects. In other words, running regression tests demonstrates to the scientists that their changes did not break anyone else's code and that software, which previously passed the tests, still does. The tests in the DUNE test environment run every night using the current development version of DUNE.

Most regression test environments compare the behavior of two program versions to find out if there are any changes. Deviations in the program behavior can be intended, such as bug fixes, or unintended, such as regression faults. Traditionally regression testing techniques characterize the program behavior due to the program output. Sometimes an old version's output is stored as an expected output. There are also methods, like program spectra, that compare the program versions' internal behavior in a black box testing manner [12].

When we apply such regression testing techniques for a scientific framework, we need to take some specialties in the software's characteristics into account. In our case the output of the software consists of floating point values. When we compare two floating point values, we have to take rounding errors into account. This is why we use stored expected output values with a tolerance

range as a reference. The expected output values are defined very carefully as described in the next subsection.

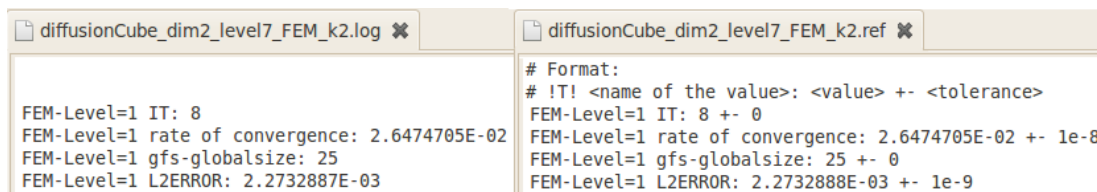
Algorithm Verification

Algorithm verification is a process that focuses on the correct implementation of the numerical algorithms. It addresses the software reliability of the implementation of all the numerical algorithms that affect the numerical accuracy and efficiency of the code. Depending on the used numerical algorithm, there are several algorithm verification techniques, like grid convergence test, convergence of the iterative solvers, consistency and symmetry tests, and benchmark solutions. One specific technique for developing a special type of analytical solution to be used for the testing of numerical algorithms is the Method of Manufactured Solutions (MMS) [4].

For algorithm verification in the DUNE test environment, the scientists extend each test application's output in a way that, depending on the used mathematical and numerical model, it includes some significant mathematical quantities like the grid convergence rate or the count of iterations. If possible, the expected output values for these mathematical quantities are determined analytically. If this is not possible, like it typically isn't for scientific software, the scientists set up these values from a scientifically validated run of the test application.

A change in these expected values always indicates a change in the test applications' behavior. In most cases this means that there is a defect in the DUNE framework. The other possibility is that the framework was changed in a way that intended a change in this specific test application. In this case, the scientists can update the expected output values for the test case. Such changes always have to be scientifically justified and carefully documented.

In our test environment, the output for the algorithm verification consists of a free text description of the mathematical quantity and the according value followed by a colon. An example of a test application output is shown in the left part of Figure 1.



The image shows two side-by-side windows from a text editor. The left window, titled 'diffusionCube_dim2_level7_FEM_k2.log', contains the following text:

```
FEM-Level=1 IT: 8
FEM-Level=1 rate of convergence: 2.6474705E-02
FEM-Level=1 gfs-globalsize: 25
FEM-Level=1 L2ERROR: 2.2732887E-03
```

The right window, titled 'diffusionCube_dim2_level7_FEM_k2.ref', contains the following text:

```
# Format:
# !T! <name of the value>: <value> +- <tolerance>
FEM-Level=1 IT: 8 +- 0
FEM-Level=1 rate of convergence: 2.6474705E-02 +- 1e-8
FEM-Level=1 gfs-globalsize: 25 +- 0
FEM-Level=1 L2ERROR: 2.2732888E-03 +- 1e-9
```

Figure 1. Example of an algorithm verification output and the according expected values for one test case.

For the example above, the expected values can be found in the right part of Figure 1. Since numerical calculations are carried out using floating point arithmetics and thus will include rounding errors, the test environment cannot expect an exact value for each algorithm verification test value. The tolerance range for each expected value is determined due to the scientist's expert knowledge.

Scientific Validation

In scientific validation the goal is to determine how accurate the computational model simulates the real world [4]. In an ideal case we can compare the simulation with experimental data. Since this is mostly not possible for the kind of simulations that are created with DUNE, our goal in scientific

validation is to support the scientists in deciding whether the simulation result is what they expected or not.

The DUNE test environment supports the scientific validation by comparison of the graphical simulation output files. The values in these output files are compared with according expected scientific validation output values. Both, the absolute and the relative difference between the output file values and the expected values are tested. As described in the subsection before, the output values include some rounding errors and machine accuracy, which is taken into account when calculating the absolute and relative differences. In this way a minor change in the values is automatically accepted. If there any bigger changes in the values in these files, the test environment will report it.

Again, it is the responsibility of the scientists to decide whether changes in the output are due to a defect or due to a planned change in the framework.

Example Application Diffusion

In this section we demonstrate the process of creating a test application and the according test cases with a concrete example. First we define the mathematical problem we want to solve and decide which numerical algorithm we want to use to solve this problem. Next we illustrate a part of the according variability model and describe how the set of test cases is derived using the variability model.

The Mathematical Problem

As a test problem we consider the Poisson equation, an elliptic partial differential equation. The mathematical model reads: Given a domain $\Omega \subset \mathbb{R}^d$. Find $u \in H^1$ such that

$$\Delta u = f \text{ in } \Omega \quad (1)$$

$$u = g \text{ on } \partial\Omega \quad (2)$$

where g denotes the Dirichlet boundary conditions. For an arbitrary solution of u we can choose f and g such that Eq. (1) is fulfilled. For our test we have chosen $u = e^{-|x|^2}$ to be a Gauss bell. This yields $g = e^{-|x|^2}$ and $f = (2 * \dim - 4 * |x|) * e^{-|x|^2}$. Given a particular numerical model, the numerical results can be compared to the analytic solution of u .

The numerical model requires the scientist to choose from a range of different numerical methods and parameters. In the next subsection we will elaborate these options and formalize the variability in the numerical model.

Variability Model for the Test Application Diffusion and Derivation of Test Cases

The variability model for the numerical model consists of all possible grid and discretization configurations. Figure 2 shows the part of the variability model for the grid configuration. It contains variation points for the different possible characteristics of a grid, the possible variants for each characteristic and the constraints between them.

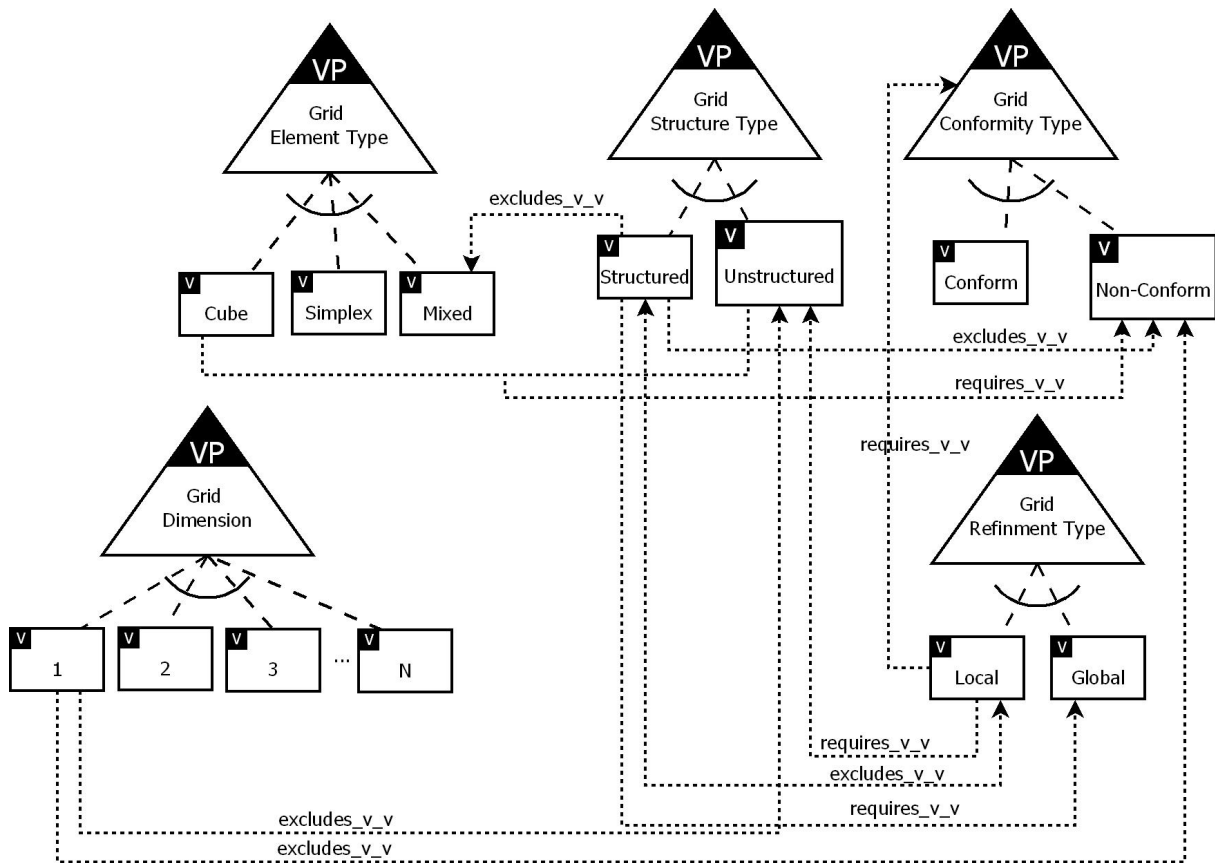


Figure 2. Variability model for the grid configuration

The test application should enable the testing of each allowed combination of these characteristics. Part of the variation points, like Grid Element Type and Grid Dimension, can be found as input parameters for the test application. Some other variation points are included in the test applications implementation. For example, the diffusion test application uses different kind of grid implementations: YaspGrid is a structured grid, AluGrid is an unstructured grid. This enables the use of both variants of the variation point Grid Structure Type.

To set up all possible test cases for a test application we need to figure out the possible combinations of variants from the variability model. First we make some presumptions for the possible variants: we only want to test the grid dimensions 2D and 3D. If at first we ignore the constraints between the different variants, this leads to 48 possible variant combinations. In this example, the constraints between the variants decrease the number of allowed variant combinations to 24.

Implementation of the Test Application

The diffusion test application is a C++-program consisting of about 650 LOC. It uses the DUNE framework modules dune-common, dune-grid, dune-istl and dune-pdelab. The test application has input parameters that enable us to create test cases with different grid and discretization configurations.

The implementation of the diffusion test application is based on an existing test application that was previously used for manual test runs. Most of the variability was already implemented in the test application. It was possible to select the used grid element type, grid dimension and discretization method. As we compared the implementation with the variability model, we noticed that still only

four of the possible 24 combinations of variants were supported by the test application. Thus the variability modeling helped us to detect missing functionality in our test application and to enhance the test environment.

The output for algorithm verification for the test application diffusion includes the count of iterations and the rate of convergence for the linear solver used. In addition the global size for the grid function space and the L_2 -error for the numerical solution are calculated and given out. For scientific validation the simulation output is redirected into a separate output file.

Experiences with the Test Environment

During the last implementation sprint of the dune-pdelab module we applied the test environment to test the development changes and to evaluate how the test environment supports the scientists' work. The sprint took over one month and involved about ten DUNE developers. The changes consisted of over 400 commits with about 32000 new LOC and 19000 deleted LOC.

The implementation sprint included some major changes in the dune-pdelab module. These include the replacement of a grid operator and interface changes for a local operator and an AMG solver. Since these operators are used in most DUNE applications, this also led to some major changes in the test applications in the test environment.

We used the following approach for applying the test environment for the development sprint. First we run the test environment for the previous version of dune-pdelab. Those expected output values for algorithm verification and scientific validation that are not defined analytically were determined according to the test run on the previous version. After that the test applications were adjusted to the new development. As soon as the changed operators were implemented, we adopted them to the test applications and rerun the tests on the changed version of dune-pdelab.

The unit tests were always run before the test application was executed. This way we observed that even though the single units were tested using unit tests, some faults could only be found through the test environment.

Using the test environment for such a major development sprint was challenging. Each test application had to be adjusted to the changed framework functionality which caused some extra work for the scientists. The test applications could only be changed after the development for a specific development change was completed. This sometimes led to a delay in testing these development changes.

When the test environment reported a problem, the scientists first had to find out where the problem is: in the implementation of the test application or in the functionality of the framework. In the second step the scientists had to figure out if this change in the output was intended or unintended.

Altogether, this process helped the scientists to evaluate the development changes made in the framework. After this process they were more confident about the good quality of the new dune-pdelab version. Finding some defects in the DUNE framework (even some that existed for months or years) motivated the scientists to use the test environment.

Future Work

After we have included the concept of modeling the variability in the framework and selecting the characteristics for the test applications accordingly, we want to automate the test case creation for the test applications.

Some minor future extensions for the test environment include the definition of different kind of test runs: one for the nightly runs and one more complex one for release acceptance testing. We also plan to extend the test output to include some information about the hardware and compile options that were used to compile and run the test applications. We also want to extend the DUNE test environment in a way that it additionally tests the performance of the test applications.

Major effort in regression testing research is done in reducing the cost of regression testing without a reduction in the benefit. Several regression test selection or test prioritization techniques have been introduced to select a subset of existing tests to retest a new version of the program [12]. Using such techniques is part of the future work for the DUNE test environment.

Conclusions

With this article we introduced the regression test environment using system testing we developed for DUNE, a scientific framework. With an example we showed how we use our software product line approach to deal with the huge variability in a framework and how we arrange the set of test applications and test cases for the test environment.

Acknowledgements

The foundations for the test environment, including the structure of the configuration files and the scripts running the tests, were implemented by Felix Heimann from the DUNE development team for his own daily work. The original version of the script for the scientific validation was implemented by Jorrit Fahlke from the DUNE development team.

References

1. J.C. Carver, "Report: The Second International Workshop on Software Engineering for CSE," Computing in Science & Engineering, vol. 11, no. 6, 2009, pp.14-19.
2. R. Baxter, "Software Engineering Is Software Engineering," *Proceedings of the First International Workshop on Software Engineering for High Performance Computing System Applications*, IEEE Computer Society, 2004, pp. 14-18.
3. D. Hook and D. Kelly, "Testing for trustworthiness in scientific software," *Software Engineering for Computational Science and Engineering*, IEEE Computer Society, 2009, pp. 59-64.
4. W.L. Oberkampf, et al., "Verification, Validation, and Predictive Capability in Computational Engineering and Physics", Applied Mechanics Rev., 2004, pp.345-384.
5. A. Pasetti, "Software frameworks and embedded control systems," Springer-Verlag, 2002.
6. K. Pohl, et al.. "Software Product Line Engineering - Foundations, Principles, and Techniques," Springer Berlin Heidelberg, 2005.
7. K.S. Ackroyd, et al., "Scientific Software Development at a Research Facility," *IEEE Software*, vol. 25, no. 4, 2008, pp. 44-51.

8. E. Engström and P. Runeson, "Decision Support for Test Management and Scope Selection in a Software Product Line Context," *Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2011, pp.262-265.
9. P. Bastian, et al., "A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework," *Computing*, vol. 82, no. 2, 2008, pp. 103-119.
10. P. Bastian, et al., "A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE," *Computing*, vol. 82, no. 2, 2008, pp. 121-138.
11. H. Rimmel, et al., "Supporting the testing of scientific frameworks with software product line engineering: a proposed approach," *Proceeding of the 4th international workshop on Software engineering for computational science and engineering (SECSE '11)*, ACM, 2011, pp. 10-18.
12. T. Xie and D. Notkin, "Checking inside the black box: regression testing by comparing value spectra," *IEEE Transactions on Software Engineering*, vol.31, no.10, 2005, pp. 869- 883.

Biographies

Hanna Rimmel is a graduate student at the Institute for Computer Science, University of Heidelberg, Germany. Her research interests include software engineering for computational science and engineering, in particularly testing scientific frameworks. She has a master's degree in computer science from the University of Jyväskylä, Finland. Contact her at remmel@informatik.uni-heidelberg.de.

Barbara Paech is the chair of software engineering at the Institute for Computer Science, University of Heidelberg, Germany. She is also the spokesperson of the section Software Engineering of the German computer science society. Her research interests include achieving quality with adequate effort, requirements engineering, and rationale management. She received a PhD in computer science from the Ludwig-Maximilians-Universität München, Germany, and a habilitation in computer science from the Technische Universität München, Germany. Contact her at paech@informatik.uni-heidelberg.de.

Christian Engwer is a junior professor at the University of Münster, Germany. His research interests include numerical methods for the solution of partial differential equations, their application to complex or coupled problems, and the development of efficient and maintainable numerics software. Christian Engwer studied physics at the University of Heidelberg, Germany, and hold a PHD in mathematics. Contact him at christian.engwer@wwu.de.

Peter Bastian holds the chair of scientific computing at the Interdisciplinary Center for Scientific Computing, University of Heidelberg, Germany. His research interests include the numerical solution of partial differential equations, parallel algorithms and simulation of flow and transport in porous media. Peter Bastian graduated in computer science from the University of Erlangen-Nürnberg, Germany, received a PhD in mathematics from the University of Heidelberg, Germany, and did his habilitation in Computer Science at the University of Kiel, Germany. Contact him at peter.bastian@iwr.uni-heidelberg.de.