Design and Rationale of a Quality Assurance Process for a Scientific Framework

Hanna Remmel and Barbara Paech

Institute for Computer Science
University of Heidelberg
Heidelberg, Germany
{remmel, paech}@informatik.uni-heidelberg.de

Christian Engwer

Institute for Computational and Applied Mathematics
University of Münster
Münster, Germany
christian.engwer@uni-muenster.de

Peter Bastian

Interdisciplinary Centre for Scientific Computing (IWR)
University of Heidelberg
Heidelberg, Germany
peter.bastian@iwr.uni-heidelberg.de

Abstract—**The testing of scientific frameworks is a challenging task. The special characteristics of scientific software e.g. missing test oracle, the need for high performance parallel computing, and high priority of non-functional requirements, need to be accounted for as well as the large variability in a framework. In our previous research, we have shown how software product line engineering can be applied to support the testing of scientific frameworks. We developed a process for handling the variability of a framework using software product line (SPL) variability modeling. From the variability models, we derive test applications and use them for system tests for the framework. In this paper we examine the overall quality assurance for a scientific framework. First, we propose a SPL test strategy for scientific frameworks called Variable test Application strategy for Frameworks (VAF). This test strategy tests both, commonality and variability, of the framework and supports the framework's users in testing their applications by creating reusable test artifacts. We operationalize VAF with test activities that are combined with other quality assurance activities to form the design of a quality assurance process for scientific frameworks. We introduce a list of special characteristics for scientific software that we use as rationale for the design of this process.**

Index Terms—**scientific software development, software product line engineering, quality assurance process, test strategy.**

I. INTRODUCTION

In our research, we concentrate on the testing of scientific frameworks. A *framework* consists of common code that provides solutions for several similar applications for specific types of problems [1]. DUNE[1], the software we deal with, is a complex scientific framework for solving partial differential equations supporting a large variety of applications (e.g. fluid mechanics or transport in porous media), mathematical models

and numerical algorithms. DUNE is introduced in more detail in [2] and [3].

The testing of scientific software is a challenging task, since it has to deal with special challenges like missing test oracle, the need for high performance parallel computing, and high priority of non-functional requirements over functional requirements [4]. When testing a scientific framework, we additionally need to find a way to deal with the large variability in a framework. A special challenge of a scientific framework like DUNE is that the variability is hidden in the mathematics the framework implements. The variability is expressed precisely, but not in a form that could be understood as a model by a human or a computer.

Our approach to meet this challenge is to apply *software product line engineering* (SPLE) to handle the framework's variability. In SPLE, the idea is to develop a software platform during *domain engineering* and then, in *application engineering*, use mass customization for the creation of a group of similar applications that differ from each other in specific predetermined characteristics [5].

Similar to the division of the SPLE process into two development processes, SPL testing is broken down into *domain testing* and *application testing*. In domain testing, the goal is to ensure the quality of the reusable platform, including the *commonality* (common characteristics for every application in the product line) and the *variability*, defined for the product line in the domain engineering process. This includes the testing of those artifacts that can be tested as early and often as possible and the creation of reusable test artifacts that can then be reused in application testing. Variable artifacts that are only used in one or few applications are tested in application testing. In application testing, the applications derived from the SPL platform are tested. The test activities in application testing concern parts of the application that are developed during

---

[1] http://www.dune-project.org/

application engineering but also reuse test artifacts from domain testing [6].

For developers of a scientific framework the importance of domain testing is high, since they need to test the functionality of the scientific framework without knowing exactly what kind of applications the users are going to develop. They have a lot less impact on application testing, which is performed by the users of the scientific framework. In other words, the definitions of the roles developer and user in our context are different from the traditional SPLE, where the developers carry out both, domain and application engineering processes. In the context of scientific frameworks, the developers only deal with domain engineering. The application engineering, meaning the development of specific applications, is done by the users of the framework. In scientific frameworks, the users are at the same time the developers in the application engineering process [7].

In scientific software development, the source code is often tested with unit testing. Also in SPLE, unit and integration testing are typically performed in domain testing. During unit and integration testing, most functional failures should already have been found, but in scientific software system testing this is still the only testing level where the interaction between the mathematical model, the numerical model, and its implementation can thoroughly be tested [8]. At system testing level also non-functional requirements like correctness and performance can be tested [9]. This is why we have developed a regression test environment executing system tests for DUNE (introduced in detail in [8]). For the systematical creation of system tests, we use SPL variability modeling.

Those software characteristics, that can vary, are called *variation points* and the possible values for a variation point are called *variants*. A *variability model* is described by variation points and their variants. It also includes the constraints between the variation points and the variants. A special challenge for domain testing is *absent variants*. These are variants that are only created in application engineering and not available in domain engineering [5].

In [7], we discuss a process for creating variability models for DUNE based on the mathematical requirements (the mathematical problems that the framework should solve). Each variability model is associated with a *system test application*, which implements the corresponding mathematical problem. We use each variability model for a system test application as a basis for systematically selecting the set of *test cases* for this test application.

In this paper, we examine the overall quality assurance for a scientific framework. First, we propose a SPL test strategy for scientific frameworks. In a *SPL test strategy*, the partition of responsibilities between domain testing and application testing is defined. The used test strategy strongly influences the activities performed in domain testing and application testing [10]. Second, we introduce the design for a quality assurance process for a scientific framework using SPLE. We explain the quality assurance activities harmonized with the proposed SPL test strategy in detail. As a basis, we choose the RiSE Product Line Engineering TEsting project (RiPLE-TE), a quality assurance process introduced for software product lines by Machado et al. in [11]. We introduce a list of special characteristics for scientific software that we use as rationale for the design of a quality assurance process to suit the needs of the scientific framework.

In Section II, we describe the proposed SPL test strategy. This is followed by the description of the designed quality assurance process for a scientific framework in Section III. After the discussion of related work in Section IV, we summarize our findings and present our future work in Section V.

II. SPL TEST STRATEGY FOR SCIENTIFIC FRAMEWORKS

As described in Section I, a SPL test strategy describes, how the testing responsibility is partitioned between domain testing and application testing. In this section, we first shortly introduce SPL test strategies found in the literature and discuss why they are not suitable for scientific frameworks. Then, we propose a SPL test strategy for scientific frameworks and assess it using the criteria for SPL test strategies introduced by Pohl et al. in [10].

A. Software Product Line Test Strategies

The following SPL test strategies can be found in the literature:
A. Brute Force Strategy (test only in domain engineering) [10]
B. Pure Application Strategy or Testing Product by Product (test only in application engineering) [10], [12]
C. Incremental Testing of Product Lines (test the first application individually and the following applications using regression testing techniques; a special case for B) [12]
D. Sample Application Strategy (SAS) (one or few sample applications are created and used to test domain artifacts; testing for the specific applications in application engineering is still needed) [6]
E. Commonality and Reuse Strategy (CRS) or Design Test Assets for Reuse (test common parts in domain engineering and for the variable parts create reusable test artifacts for the testing in application engineering) [10],[12]
F. Combination of strategies SAS and CRS (use fragments of a sample application in domain testing and create reusable test artifacts for application testing) [10]
G. Opportunistic Reuse of Test Assets (create test artifacts for one application in application testing and use these artifacts for further product line applications) [13]
H. Division of Responsibilities(select testing levels to be applied in both, domain and application engineering, for example, test common parts with unit testing in domain engineering and execute integration, system, and acceptance testing in application engineering) [12]

In a previous paper [7], we mentioned that SAS would be the SPL test strategy we want to use for scientific frameworks. At that time, we were not aware of the exact criteria for a SPL

test strategy a scientific framework needs to fulfill. That is why we need to revise this decision.

Considering the special case of a framework where the framework developers only deal with domain testing and the application testing is accomplished by the framework's users, a SPL test strategy for a framework needs to fulfill the following criteria:

1. Both, commonality and the variability of the framework are tested in domain testing.
2. Application testing is supported with reusable test artifacts created in domain testing.
3. Product line applications still need to be tested, especially parts developed in application testing.

In most SPL strategies, only the common parts of the platform are tested in domain testing. The variability of the platform, meaning possible applications that can be derived from the platform, is tested only in application testing, where concrete applications exist. Since we need to test the whole platform, i.e. the framework's functionality, we also need to test the framework's variability using reference applications in domain engineering already (Criterion 1). Still, the framework's users need to test their applications, because these always include some own functionality and may use the framework in a way the framework's developers could not expect. The users should not assume that since the framework is tested, they do not need to test their own applications (Criterion 3).

In domain engineering, a SPL test strategy for a framework should include the creation of reusable test artifacts, like test applications including variability. These test artifacts can be reused by the framework's users when they are testing their own applications (Criterion 2).

Table I demonstrates which SPL test strategy criteria for a framework are fulfilled by the existing test strategies. Criterion 1 is only fulfilled by the Brute Force Strategy (A), but this test strategy firstly does not fulfill the other criteria and secondly is not recommended by the authors in [10]. Test strategies Incremental Testing of Product Lines (C) and Opportunistic Reuse of Test Assets (G) create reusable test artifacts merely in application testing and therefore only partly fulfill Criterion 2. The most suitable test strategies are CRS (E) and the Combination of strategies SAS und CRS (F). However, these strategies do not test the whole variability in domain testing and therefore do not fulfill Criterion 1.

Since none of the existing SPL test strategies fulfills the criteria for a framework, in the next subsection we propose a new SPL test strategy for frameworks fulfilling all criteria.

TABLE I. FULFILLMENT OF CRITERIA FOR A SPL TEST STRATEGY FOR A FRAMEWORK

Criteria	SPL Test Strategy							
	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*
1.	X							
2.			(X)		X	X	(X)	
3		X	X	X	X	X	X	X

Legend: 'X' = fulfills criterion, '(X)' = partly fulfills criterion

B. Variable Test Application Strategy for Frameworks

What we need is a SPL test strategy similar to CRS, but the strategy also needs to test the variability of the product line in domain engineering. For this purpose, we first take a short look at the variability modeling for a framework. If we had a variability model for the whole framework, we could derive the test applications and test cases thereof as illustrated in Fig. 1. Since it is not feasible to create such a variability model for the whole framework covering a wide range of functionality (as discussed in Section I), we start with the mathematical requirements for the framework (the mathematical problems, which the framework should solve) and create several variability models based on those mathematical requirements (detailed description of the variability model creation for a scientific framework can be found in [7]). Each variability model is associated with a test application, as shown in Fig. 2. For example, one mathematical problem the DUNE framework should support is solving the Poisson equation, an elliptic partial differential equation. A variability model for this problem covers among others the different possible grid configurations and used discretization methods (for details, see [8]). The fulfillment of this mathematical requirement can be tested with the associated test application.

In our SPL test strategy called Variable test Application strategy for Frameworks (VAF), we not only create a few sample test applications in domain testing. Moreover, we attempt to create a set of test applications that cover the range of mathematical requirements of the framework. The commonality of the framework is tested with unit and integration testing, which fulfills Criterion 1. The domain test applications include variability themselves and can be reused by the framework's users to test in application engineering their specific applications created on the basis of framework (Criterion 2). Testing their own applications in
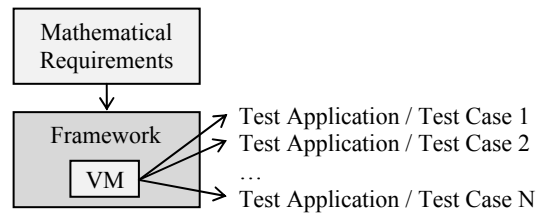

Fig. 1. Derivation of Test Applications and Test Cases for a Framework with Variability Model (VM)
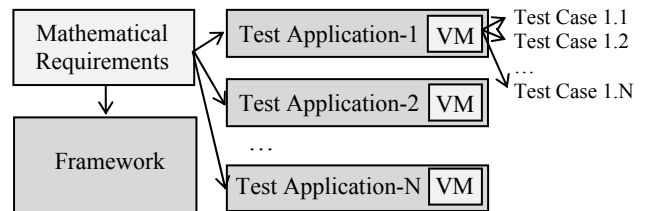

Fig. 2. Derivation of Variability Models (VM) and the Associated Test Applications from the Frameworks Requirements

application testing is still the users' responsibility (Criterion 3). The framework's developers support this with the reusable test applications.

In the next subsection, we will assess VAF and compare it with CRS, which is one of the test strategies most often applied in practice.

C. Assessment

Pohl et al. [10] introduce five essential criteria for a SPL test strategy:

1. Time to Create Test Artifacts (overall time needed for creating test artifacts in domain and application testing; influenced by the amount of test artifacts and by the difficulty to create them)
2. Absent Variants (how well does a test strategy deal with absent variants; for definition see Section I)
3. Early Validation (the time between the finalization of an artifact and its validation should be low to help keeping the costs for repairing defects low)
4. Learning Effort (time it takes for a tester to be able to perform the test activities associated with the test strategy)
5. Overhead (amount of unnecessarily performed activities and/or produced artifacts when the same result could be achieved with lower effort)

In Table II we compare VAF with CRS using these criteria. In our assessment of VAF, we take into account our context with a scientific software development.

In VAF, the required test artifacts are the variability models and test applications. Their creation is time-consuming and mathematical expert knowledge is required. Additionally, the framework's users need to create new test artifacts or extend the reusable artifacts to suit their specific scientific environment. Therefore, we rate the time to create criterion with "-". As for CRS, the handling of absent variants is excellent, since the test applications created in domain testing cover the variability of the framework. If the users of the framework introduce new variants in application engineering, they can extend the reusable test applications to test them.

The assessment of the early validation criterion is better than for CRS, since the test applications created in domain testing can also already be executed in domain testing. The learning effort is high in both cases, since scientists developing the framework are not familiar with SPLE methods, like creating a variability model. On the other hand, this part of the test strategy needs to be created only once at the beginning and does not need to be changed very often afterwards, since the mathematical requirements do not change a lot. As for CRS, the overhead for VAF is low, since the created test artifacts can be reused in application testing.

TABLE II. COMPARING VAF AND CRS TEST STRATEGIES

	Time to create	Absent variants	Early validation	Learning effort	Overhead
VAF	-	+	+	-	+
CRS [10]	+	+	0	-	+

Legend: '+' = positive, '-' = negative, '0' = neutral

III. QUALITY ASSURANCE PROCESS FOR A SCIENTIFIC FRAMEWORK

In this section, we operationalize the domain testing part of VAF (Criteria 1 and 2) with test activities (unit, integration, and system testing). Together with other quality assurance activities (e.g. review, scientific validation) they form a quality assurance process. The use of SPLE is not the only aspect we need to consider for an overall quality assurance process. The fact that we are dealing with scientific software has a major influence on it.

In the following subsections, we first discuss characteristics in scientific software development that need to be considered, when we are designing a quality assurance process for a scientific framework. This includes the most important quality goals for scientific software in general and for the DUNE framework in particular: correctness, performance, portability, and maintainability. After that, we present RiPLE-TE, a quality assurance process for SPL introduced by Machado et al. in [11]. We will show how this quality assurance process needs to be adapted to take into account VAF and the characteristics for scientific software development.

A. Characteristics of Scientific Software Development Relevant for the Design of a Quality Assurance Process

In the literature on scientific software development, several special characteristics compared to traditional software development are mentioned. In a manual literature review, we collected such special characteristics that need to be taken into account when designing a quality assurance process for scientific software. The papers we reviewed were collected between April 2010 and October 2012 using the IEEE and ACM digital libraries. Search strings with most hits were "scientific software engineering", "scientific software development" and "scientific computing software". Furthermore, we collected papers from the previous Workshops on Software Engineering for Computational Science and Engineering and Software Engineering for High Performance Systems Applications. Altogether, we found 201 papers. We looked through the papers to find out if they mentioned any special characteristics for scientific software development relevant for the design of a quality assurance process. We found eight papers describing such characteristics.

The characteristics are presented in Table III. These are used as rationale in the following description of the quality assurance process.

C1: There are different possible sources of a problem in scientific software: the underlying science, the translation of the mathematical model of the field of application to an algorithm and the translation of this algorithm into program code. Each of these should be handled separately: first check the source code for bugs with *code verification* methods and then verify the mathematical algorithm with numerical *algorithm verification* methods. Only after these two steps, knowing that errors in code and mathematical algorithm have already been excluded, the scientists are able to perform the *scientific validation* (evaluate whether the output of the software is a reasonable proximity to the real world).

TABLE III. Characteristics of Scientific Software

		Characteristic	Reference	Considered in
C1		Different possible sources for a software problem. Need support for Code Verification, Algorithm Verification and Scientific Validation.	[14],[15], [16]	Rev, U&I, Sys, SV
C2		Lack of test oracles.	[17]	U&I, Sys, SV
C3		Most software requirements, except for high-level ones, are not known at the beginning of a software project. Requirements stem from science.	[18]	Pla
C4		The cognitive complexity, the difficulty in understanding a concept, thought, or system, is high.	[19]	TR, Pla, Rev, SV
C5		Need for shared, centralized computing resources; high performance computing, parallelism.	[20]	U&I, Sys
C6		Calculations include rounding errors and machine accuracy.	[16]	Sys, SV
C7		Most developers are domain scientists or engineers, not software engineers.	[15],[21], [20],[18]	TR, Rev
C8		There is a high turnover in the development team.	[15]	Pla, Rev, Rep
C9		The most highly ranked project goals: 1. Correctness	[18]	Sys, Val
C10		The most highly ranked project goals: 2. Performance	[18]	Sys
C11		The most highly ranked project goals: 3. Portability	[18]	Sys
C12		The most highly ranked project goals: 4. Maintainability	[18]	Rev, Rep

a. Legend: TR = Test Roles, Pla = Planning, Rev = Review, U&I = Unit and Integration Testing, Sya = System Testing, SV = Scientific Validation, Rep = Reporting

C2: Scientific software is used for gaining research results or solving problems that cannot be solved by other means. The outcome is therefore often not known in advance. This is a problem for testing, since most testing techniques in software engineering assume accurate test oracles.

C3: At the beginning of a scientific software project, the known requirements are often the laws of nature, or, like in our case, stem from mathematics. In most cases, further requirements for the software have not been defined in advance but emerge during software development.

C4: The context of scientific software is usually very complex. Only scientists familiar with the scientific domain in question have the ability to entirely understand the software. This is a problem for testing, since the tester should understand what the software is supposed to do.

C5: Solving complex scientific problems with scientific software often requires special resources like high performance computing. At the same time, a special programming paradigm like the use of parallel computing is applied. This must be taken into account when testing the software.

C6: Scientific calculations use floating-point values, which cannot be represented exactly by a computer. This leads to rounding errors and machine accuracy in the calculations. Testing scientific software must support floating-point arithmetic.

C7: The fact that the developers of scientific software mostly are domain scientists and not software engineers has to be considered in the design of a quality assurance process. An important goal is to keep the process as straightforward and understandable as possible. The process should not include too many technical software engineering terms or structures. There is also a difference in the objective: a software engineer's goal is to produce high quality software, whereas the goal of a scientist is to produce high quality science. The scientists developing the software must be convinced that each step is important and has a real value for the scientific results.

C8: Many developers of scientific software are doctorate students or postdocs who only stay in the team for a few years. Because of this, the overhead of the process should be as low as possible and the method should be easy to learn and quick to adopt.

C9 - C12. In scientific software development, the priority of non-functional requirements is high compared to functional requirements. In a series of case studies, Carver et al. [18] found out, that the most highly ranked scientific software project goals are correctness, performance, portability and maintainability.

We use the characteristics described in this subsection as rationale for adjustments in the quality assurance process RiPLE-TE introduced in the next subsection.

B. RiPLE-TE Quality Assurance Process for SPL

Based on a systematic mapping study on SPL testing and evaluated with an experimental study, Machado et al. [11] designed a quality assurance process for product lines. The process comprises both, domain testing and application testing.

The first activity in both, domain testing and application testing, is the development of a *master test plan* defining what and how will be tested, who will do it, the coverage criteria and a time schedule. The planning should be continuously performed during the other quality assurance activities and the plan should be updated whenever necessary.

Since it is desirable that failures in the source code can be detected as early as possible, the second step is a *technical review*, where the main artifacts in SPL, such as variability model, product map etc., are reviewed before the dynamic tests are being started.

In domain testing, where the product line platform with reusable artifacts is developed, the focus is on *unit* and *integration testing*. The unit testing should ensure that the components may be reused further. After that, integration testing seeks to guarantee that tightly coupled components work together.

In application testing, *integration, system,* and *acceptance testing* is performed. In this stage, integration testing affects the components that will comprise the application. System testing evaluates the application as a whole against system requirements. In acceptance testing, customer feedback on the application is gathered. If any new artifacts have been created in application engineering, these should first be tested with unit testing.

Each of the testing activities above includes four tasks: planning, design, execution, and reporting. After these tasks have been performed, coverage criteria are used to decide whether the testing activity is accomplished or not. If not, the cycle returns to planning.

In the following subsections, we discuss how we need to adjust this quality assurance process in order to make it suitable for a framework. Furthermore, we need to take into account the characteristics of scientific software development collected in Table III. The rationale based on these characteristics is marked in the text in brackets.

This is a plan for a quality assurance process for the DUNE framework. Most of the steps (unit, integration and system testing, scientific validation, regression testing and reporting) have, with some limitations, been implemented already. Other steps still need to be established.

C. Test Roles

In the RiPLE-TE quality assurance process, the activities and tasks are assigned to many different test roles: test manager, test architect, test designer, and tester. In scientific software development, in the most cases, every team member fulfills the role developer and the different test roles all in one person.

The most important reason for this is that the scientist developing a piece of code often is the only person who has the expertise to entirely understand the code (C4). At least in academic projects, even the colleagues in the same group typically are working on different topics. The scientist developing the code must be responsible for creating suitable tests for his or her own code. Another reason not to use many different test roles is that the developers of scientific software normally are not software engineering professionals (C7). The quality assurance process should be as simple as possible.

In our quality assurance process, we are using the roles *developer* and *user,* and as the only test specific role, *test administrator*. The test administrator is responsible for keeping the (nightly executed) test environment running. This role should ideally be fulfilled by technical staff. If this is not possible, the role can also be carried out by a scientist. For some activities in the quality assurance process, it is more suitable to have a team of developers do this. This will be mentioned in the detailed description of the process.

D. Quality Assurance Process Steps

This subsection discusses the steps in the quality assurance process for scientific frameworks. The process is illustrated in Fig. 3. One main difference to RiPLE-TE is that our process only covers domain testing and not application testing, since in the development of a framework we only deal with domain engineering.

Shifting a major part of the quality assurance responsibility from application testing to domain testing, as described in the SPL test strategy for scientific frameworks in Section II, results in introducing system tests already in domain testing. Since we are assuring the quality of scientific software, we need to add a new step scientific validation (C1) at the end of the quality process.
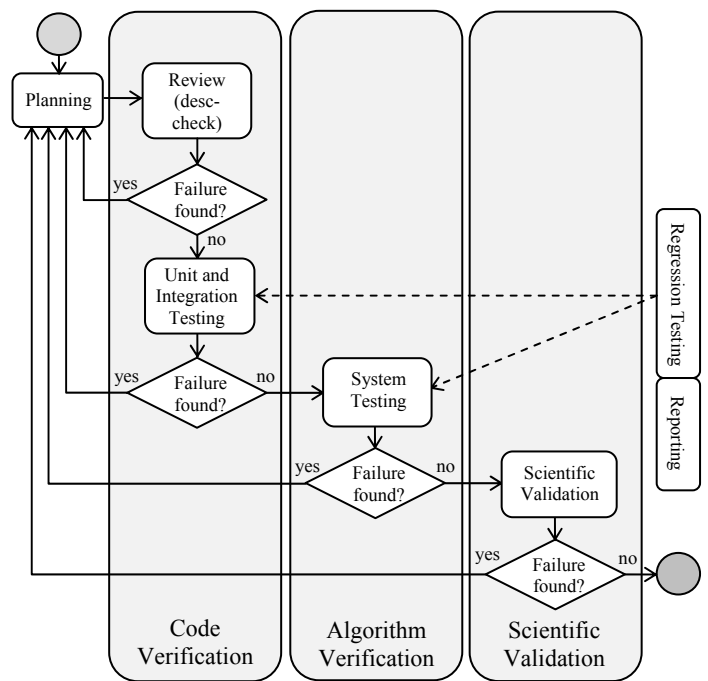

Fig. 3. Quality Assurance Process for scientific frameworks

In the following, the implementation of VAF is also documented.

1) Planning: The activities in this step are critical for the success of the whole process. When a developer makes changes in the source code or develops a new piece of code, she or he has to pay attention to the following quality assurance issues.

The developer is responsible for creating new unit test cases and/or adjusting/removing existing unit tests whenever appropriate. It is very important that the developers take time to create suitable unit tests for their own source code at the very time when they are developing the source code. It is advisable to perform test driven development (TDD) meaning that the unit test cases are created first as a kind of light weight specification for the planned changes, since specifications mostly do not exist in advance (C3). The developer might be the only one to thoroughly understand the source code she or he is developing (C4) and it is very difficult to ensure the quality of the code later, when the developer may have already left the team (C8).

If the mathematical requirements for the framework change, e.g. when a new functionality is included into the framework, it can be necessary to formulate one or more new variability models based on the requirements together with the associated system test applications. In other cases only existing variability models and system test applications must be adjusted, e.g. by including new variation points or variants.

There is no formal planning for the testing activities. As described above, each developer is responsible for preparing tests for his or her own source code. When major changes are planned for the framework and the developers arrange the

responsibility for the development changes, they also should decide who is responsible for preparing the tests.

2) Review: The earlier a failure is found, the lower is the cost of removing it. The earliest possible point for finding failures is right after developing the code. Taking the time for consciously reading one's own code before checking it in (also called desk-checking), can reveal failures before the code is even tested. At the same time, the developer can review the code's readability and structure. Since the software context is complex (C4), the developers should seek to write source code in an understandable way with a sufficient amount of comments. This is also a benefit for new colleagues working with the same software (C8) and it improves the maintainability of the code (C12).

In contrast to the technical review in the RiPLE-TE quality assurance process, our process involves reviewing the source code, not just SPL artifacts like the variability models. Certainly, the developer should review all artifacts she or he created or changed: source code, unit tests, variability models, and system test applications. Review is one part of the code verification needed for the verification and validation (V&V) of scientific software (C1).

If appropriate, the developer can ask a colleague to review her or his changes as well. The development team could also name responsible developers for different software modules who review the changed source code on a regular basis [22]. We do not pursue any structured inspection or review process, since the goal is to keep the quality assurance process practical and simple (C7).

3) Unit and Integration Testing: Together with the review step, unit and integration testing build the code verification part of V&V for scientific software (C1). The goal in unit testing is to verify the functionality of single software units. Since unit and integration testing covers the commonality of the framework, the first part of VAF is implemented. In this step, as in the RiPLE-TE quality assurance process, the communication between software units working closely together is tested with integration testing.

In some contexts of scientific software, where system tests can only run on a high performance computer (C5), the importance of unit testing gets even higher, since the unit tests do not need a long time to run and still have high test coverage [23]. In a similar way, the problem with a missing test oracle for system tests (C2) can be alleviated by comprehensive unit testing. A developer can execute the unit tests manually, but they also run automatically every night in a regression test environment.

When the test environment reports a failure, the scientists first have to find out where the problem is: in the implementation of the unit test or in the source code of the framework. Depending on the situation, the developer can fix the defect right away, if she or he is testing current development or the developer or test administrator creates a ticket in the ticket system, if there isn't one already for the specific failure.

4) System Testing: Our system test environment executes the system test cases derived from the system test applications as described in Section I. The system test applications implement the second part of VAF, because they test the variability of the framework.

If all of the test cases can run at once over night on the available computer resources, they all can build up the *test suite* for the system testing. Mostly, this is not possible and therefore we need a way to reduce the amount of test cases without losing the defect detection power. In the literature, different test suite selection methods can be found to address this problem, for example sampling (test configurations are selected based on domain knowledge), feature interaction (based on statistical analysis the most relevant variant combinations are selected) or the use of regression test techniques [24].

Combinatorial Interaction Testing is especially suitable for us, because it is based directly on the variability model and can be automated. This method selects a subset of all possible variant combinations, where possibly many potential variant interaction failures may happen. For example, in 2-wise testing (also called pairwise testing), those test cases will be chosen, where for every pair of two variants the combinations "both available", "one available" and "none available" are tested [25]. Kuhn et al. have used the technique and found out, that most bugs were found with 6-wise testing. 1-wise found 50%, 2-wise 70% and 3-wise 95% of the bugs. For non-critical product lines, the authors recommend 3-wise test coverage. In one example for 1024 possible variant combinations, 2-wise leads to 41 and 3-wise to 119 test cases [26].

For algorithm verification (C1), the system test applications output includes some significant mathematical quantities like the grid convergence rate or the count of iterations, depending on the used mathematical and numerical model. The expected output values for the mathematical quantities are, if possible, determined analytically. Typically, this is often not possible (C2) and therefore the scientists set up the expected values from a scientifically validated run of the system test application [8]. All expected output values include a manually adjustable tolerance range for taking rounding errors into account (C6). Supporting algorithm verification and testing on different platforms and with different configurations (e.g. count of processors, compiler options) is significant for assuring important quality goal correctness (C9) and portability (C11). System testing is also the suitable step for executing performance testing (C10).

The difference between our system test applications and those used in RiPLE-TE is that our system test applications include variability and can be reused by the framework's users. This implements the reuse part of VAF.

Similar to unit testing, a developer can execute the system tests manually or rely on the nightly running system tests. The automated nightly execution is especially beneficial for the system test environment, because the complex mathematical problems solved mostly take some time to run (C5). Similar to unit tests, a failure means that there is either a problem in the

source code, or the system test application or the expected output must be adjusted to suit the development changes.

5) Scientific Validation: Scientific validation is the last of three steps in V&V for scientific software (C1). The goal is to determine how accurate the computational model simulates the real situation (C9). In an ideal case we can compare the simulation with an analytical solution. Since this is mostly not possible for the kind of simulations that are created with DUNE (C2), our goal in scientific validation is to support the developers in deciding, based on their domain knowledge (C4), whether the simulation result is what they expected or not. The DUNE system test environment supports the scientific validation by comparison of the graphical simulation output files. The values in these output files are compared with the corresponding expected scientific validation output values taking rounding errors and machine accuracy into account (C6) [8].

E. Regression Testing

In contrast to RiPLE-TE, we integrate regression testing in our quality assurance process. The main idea of a regression test environment (automated test running on a regular basis, illustrated in Fig. 3 with a dashed arrow) 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 developers that their changes did not break anyone else's code and that software, which previously passed the tests, still does.

If every developer creates suitable unit and system tests for their own source code in the planning step, the regression test environment proves that the code still works in an evolving framework. Without such tests, the source code could get broken without anyone noticing it. The unit, integration and system tests in the DUNE run every night using the current development version.

F. Reporting

Reporting the results of the quality assurance process is important for the developers so that they can reconstruct which changes caused which effects, in the framework (C12). The log files of unit, integration, and system testing include, beside unexpected or incorrect results also, among others, the information, which source code version and which configuration was used for the test.

A clearly reported instruction for the use of the quality assurance process and the automated regression test environment is crucial so that the knowledge will not get lost, when the developers leave the team (C8).

G. Summary

The quality assurance process for scientific frameworks we introduced in this section implements the SPL test strategy VAF proposed in the section before. In unit testing and system testing, the commonality and variability of the framework is tested, which fulfills the first part of VAF. Since the system test applications include variability and can be reused by the framework's users, the second part is also fulfilled. The third

part is fulfilled, when the framework's users test their own applications.

The special characteristics of scientific software are also taken into account in the quality assurance process. The process is straightforward and the only software engineering method not known by most of the scientists in the DUNE team is the creation of variability models. We want to add this activity to the scientists' work together with software engineers in our future work.

The accomplishment of the important quality goals correctness, portability, and maintainability are already tested by the process. In future work we need to find out, which kind of performance testing is most suitable for DUNE and then adapt it to the system testing step of the quality assurance process.

As in RiPLE-Te, there is no formalized acceptance testing for DUNE. The developers of DUNE stay in close contact with the framework's users and get frequently feedback from the users.

The introduced quality assurance process is suitable for scientific frameworks. If adopted for a framework in another domain, the process should be adjusted to suit to the characteristics of that domain.

IV. RELATED WORK

In this subsection, we consider other quality assurance processes proposed in the literature for scientific software or SPLE. We are not aware of any other cases where both aspects were regarded together.

For scientific software development, some models are introduced in the literature, like an iterative and incremental model by Segal in [27] and a staged delivery model, similar to a waterfall model, used by software projects at a research center by Baxter in [28]. For a development process in general and for quality assurance in particular, we could find in [14], [29], [30], and [31] several lists of recommended software engineering practices, e.g. source control, configuration management, issue tracking, unit testing, verification, and regression testing, but they are not defined as a development and quality assurance process.

In SPLE, besides RiPLE-TE, we found two testing process descriptions. Heider et al. [32] outline existing verification and testing approaches supporting product line evolution: model verification techniques for verifying the variability model and application configurations, unit testing for core assets and application generators and integration and system testing methods, e.g. the use of sample applications in domain testing. They illustrate the interplay of these quality assurance methods, but do not discuss how these steps could form a quality assurance process. Neto et al. [33] propose a very formal regression testing approach for the reference architecture of a SPL, which uses extensive documentation, many detailed process steps and plenty of test roles. Their approach concentrates on the commonality of the SPL and does not apply system testing.

Many key success factors for a test process in agile testing are similar to ours: *a high grade of automation* that we

implement with the regression test environment, which also ensures the *rapid feedback* to the developer about software failures, *a low management overhead,* and *dissolving test roles* [34]. Nevertheless, an agile testing process can only be fully adopted in a scientific software project, if, at the same time, an agile software development process model like Scrum is used.

V. CONCLUSIONS AND FUTURE WORK

In this paper we propose a SPL test strategy for scientific frameworks called VAF. This test strategy tests both commonality and variability of the framework and supports the framework's users in testing their applications by creating reusable test artifacts.

We operationalize VAF with test activities. The commonality is tested by unit testing and the variability by creating a set of system test applications that cover the range of mathematical requirements of the framework. These test applications can be reused by frameworks users. Together with other quality assurance activities (e.g. review, scientific validation), these test activities form a quality assurance process for scientific frameworks.

As a basis for the quality assurance process we use RiPLE-TE, a quality assurance process for SPL introduced by Machado et al. in [11]. We adjust RiPLE-TE so that it implements the SPL test strategy VAF. We introduce a list of special characteristics of scientific software that we use as rationale for the design of the quality assurance process.

In our future work, we plan to implement those parts of the quality assurance process for DUNE that have not been completed yet. Then, we want to evaluate the process. After that, we want to make the reusable test applications available for DUNE users and evaluate the acceptance and benefit of this solution.

REFERENCES

[1] A. Pasetti, "Software frameworks and embedded control systems," Springer-Verlag, 2002.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, "A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework," Computing 82, pp. 103-119, 2008.

[3] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, "A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE," Computing 82, pp. 121-138, 2008.

[4] J. C. Carver, "Report: The Second International Workshop on Software Engineering for CSE," Computing in Science & Engineering, vol. 11, pp. 14-19, 2009.

[5] K. Pohl, G. Böckle, and F. Linden, "Software Product Line Engineering - Foundations, Principles, and Techniques," Springer Berlin Heidelberg, 2005.

[6] K. Pohl, and A. Reuys, "Application Testing," In Software Product Line Engineering, Springer Berlin Heidelberg, pp. 355-370, 2005.

[7] H. Remmel, B. Paech, C. Engwer, and P. Bastian, "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, pp. 10-18, 2011.

[8] H. Remmel, B. Paech, C. Engwer, and P. Bastian, "System Testing a Scientific Framework using a Regression-Test Environment," Computing in Science and Engineering, vol. 14, no. 2, pp. 38-45, 2012.

[9] http://www.swebok.org/

[10] K. Pohl, and A. Reuys, "Domain Testing,"" In Software Product Line Engineering, Springer Berlin Heidelberg, pp. 257-284, 2005.

[11] I. C. Machado, P. A. da M. S. Neto, E. S. Almeida, and S. R. de Lemos Meira "RiPLE-TE: A Process for Testing Software Product Lines," SEKE, Knowledge Systems Institute Graduate School, pp. 711-716, 2011.

[12] A. Tevanlinna, J. Taina, and R. Kauppinen, "Product family testing: a survay", ACM SIGSOFT Software Engineering Notes, vol. 29, no. 2, pp. 1-6, 2004.

[13] A. Reuys, S. Reis, E. Kamsties, and K. Pohl, „The ScenTED Method for Testing Software Product Lines", in Software Product Lines, Springer Berlin Heidelberg, pp. 479-520, 2006.

[14] P. F. Dubois, "Maintaining Correctness in Scientific Programs," Computing in Science & Engineering, vol. 7, no. 3, pp. 80-85, 2005.

[15] J. C. Carver, L. Hochstein, R. P. Kendall, T. Nakamura, M. V. Zelkowitz, V. R. Basili, and D. E. Post, "Observations about Software Development for High End Computing," In CTWatch, vol. 2, no. 4A, pp. 33-38, 2006.

[16] D. Hook, and D. Kelly, "Testing for trustworthiness in scientific software," In Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering, IEEE Computer Society, pp. 59-64, 2009.

[17] D. Kelly, S. Smith, and N. Meng, "Software Engineering for Scientists," Computing in Science and Engineering, vol. 13, no. 5, pp. 7-11, 2011.

[18] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software Development Environments for Scientific and Engineering Software: A Series of Case Studies," ICSE 2007, 29th International Conference on Software Engineering, pp. 550-559, 2007.

[19] D. Kelly, and R. Sanders, "The Challenge of Testing Scientific Software," CAST 2008, Proc Conference of the Association of Software Testing, Toronto, Canada, 2008.

[20] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz, "Understanding the High-Performance-Computing Community: A Software Engineer's Perspective," Software, IEEE, vol. 25, no. 4, pp. 29-36, 2008.

[21] D. Kelly, S. Smith, "2nd CASCON Workshop on Software Engineering for Science," CASCON 2009, pp. 345-347, 2009.

[22] D. Kelly, and R. Sanders, "Assessing the Quality of Scientific Software," in Proceedings of the International Conference on Software Engineering, First International Workshop on Software Engineering for Computational Science and Engineering, Leipzig, Germany, 2008.

[23] C. Morris, "Some Lessons learned reviewing scientific code," in Proceedings of the International Conference on Software Engineering, First International Workshop on Software Engineering for Computational Science and Engineering, Leipzig, Germany, 2008.

[24] C. H. P. Kim, D. Batory, S. Khurshid, "Elimination products to test in a software product line," ASE '10, Proceeding of the IEEE/ACM international conference on Automated software engineering, ACM New York, pp. 139-142, 2010.

[25] M. F. Johansen, Ø. Haugen, F. Fleurey, "Bow tie testing: a testing pattern for product lines," EuroPLoP '11, Proceedings of the 16th European Conference on Pattern Languages of Programs, ACM New York, no. 9, 2012.

[26] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," IEEE Transactions on Software Engineering, vol. 30, no. 6, pp. 418-421, 2004.

[27] J. Segal, "Models of scientific software development," Proceeding of the 2008 Workshop Software Engineering In Computational Science and Engineering, 2008.

[28] R. Baxter, "Software engineering is software engineering," 26th International Conference on Software Engineering, W36 Workshop Software Engineering for High Performance System (HPCS) Applications, pp. 4-18, 2004.

[29] M. A. Heroux, and J. M. Willenbring, "Barely sufficient software engineering: 10 practices to improve your CSE software," SECSE '09 ICSE Workshop on Software Engineering for Computational Science and Engineering, pp. 15-21, 2009.

[30] M. A. Heroux, "Improving the Development Process for CSE Software," PDP '07, 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing, pp. 11-17, 2007.

[31] R. Neely, "Practical software quality engineering on a large multi-disciplinary HPC development team," 26th International Conference on Software Engineering, W3S Workshop Software Engineering for High Performance Computing System (HPCS) Applications, pp. 19-23, 2004.

[32] W. Heider, R. Rabiser, P. Grünbacher, D. Lettner, „Using regression testing to analyze the impact of changes to variability models on products," SPLC '12 Proceedings of the 16th International Software Product Line Conference – Volume 1, pp. 196-205, 2012.

[33] P. A. da M. S. Neto, I. C. Machado, Y. C. Cavalcanti, E. S. Almeida, V. C. Garcia, and S. R. de Lemos Meira, "A Regression Testing Approach for Software Product Lines Architectures," Fourth Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), pp. 41-50, 2010.

[34] L. Crispin, and J. Gregory, "Agile Testing: A practical Guide for Testers and Agile Teams," Addison-Wesley Professional, 2008.