Supporting the Testing of Scientific Frameworks with Software Product Line Engineering – A Proposed Approach

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 and Peter Bastian
Interdisciplinary Centre for Scientific Computing (IWR)
University of Heidelberg
Im Neuenheimer Feld 368, 69120 Heidelberg, Germany
+49 6221 54 – {8881,8261}
{christian.engwer, peter.bastian}@iwr.uni-heidelberg.de

ABSTRACT
Testing scientific software involves dealing with special challenges like missing test oracle and different possible sources of a problem. When testing scientific frameworks, additionally a large variety of mathematical algorithms and possible applications for the framework has to be handled. We propose to use concepts of software product line engineering to handle this variability.

The contribution of this paper is a two-step process for reengineering a variability model out of a framework for scientific software. This process is explained with a real case study. Furthermore, we sketch how the variability model can be used to systematically derive system test applications for the framework.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.13 [**Software Engineering**]: Reusable Software – *Domain engineering;* G.1.8 [**Numerical Analysis**]: Partial Differential Equations

General Terms
Verification, Documentation.

Keywords
Scientific software, testing, framework, software product line engineering, variability modeling.

1. INTRODUCTION
Testing scientific software is different from software testing in general. The special challenges include missing test oracle, the high priority of non-functional requirements over functional requirements and the need for high performance parallel computing [6]. Missing test oracle means that the expected output of the software is not known. This is due to the fact that scientists use software as a tool for their research. Hook and Kelly [10] state that since the test oracle is missing, the scientists do not expect to be able to prove the correctness of the scientific software. Rather, they test its trustworthiness. Hook and Kelly propose using a well chosen set of tests that may reveal a high percentage of code faults and thus allow scientists to increase their trust. One question that is not answered by Hook and Kelly is how to choose the set of tests. This is something we are investigating.

When testing scientific software it is important to distinguish between different possible sources of a problem: the underlying science, the translation of the mathematical model of the field of application to an algorithm and the translation of that algorithm into program code [7]. Each possible source of problem should be handled separately. Hook and Kelly point out that ideally 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 mathematical 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 proximity to the real world) knowing that errors in code and mathematical algorithm are already excluded.

Several methods have been introduced for each of these steps of testing scientific software. Oberkampf et al. [13] 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.

Scientific software engineering can help scientists in code verification and algorithm verification. Code verification is often done by unit testing in scientific software. On the other hand, system tests are seldom systematically adopted for complex scientific software. Case studies like [1] confirm the fact that there is a lack in system testing. Ackroyd et al. found testing actions in the analyzed scientific software insufficient and pushed the developers to focus on an intensive usage of unit testing. The authors admit that the problem that still remains is how to ensure that code changes through continuous integration do not cause problems, if system testing is insufficient. Even though unit tests can demonstrate that every unit works as expected, they still do not ensure that these units work together.

Our research concentrates on system testing, in particular algorithm verification, of scientific frameworks which provide solutions for several similar mathematical problems like numerical solving of partial differential equations (PDEs). DUNE[1], the software we deal with, is a complex scientific framework with a large variety of fields of application (i.e. fluid mechanics or heat transport), used mathematical algorithms and numerical solutions. Developers of DUNE use unit testing to test new and changed functionality. However there is no systematical system testing. Our common goal with the developers of DUNE is to set up system tests that cover all relevant fields of application for the DUNE framework. Some questions we have to face are:

- How to model all the different possible applications for a framework?
- How to choose a suitable set of applications for testing from the many different possible applications?
- How can we systematically establish algorithm verification in scientific frameworks that deals with several mathematical models and fields of application?
- How to ensure reasonable test coverage for the system tests?

Since it is not feasible to test every possible application of a framework, we want to use the idea of Hook and Kelly to carefully choose a set of tests. But how do we do this? Our idea is to define the framework as a product line and use the variability modeling of software product line engineering (SPLE) [17] to model the necessary parts of the frameworks variability with a variability model. Then, we use the created variability model as a basis for systematically selecting the set of test applications.

In this paper, we discuss the feasibility of SPLE for scientific software and especially for scientific frameworks. We show how to create a product line variability model from an existing scientific framework and sketch how SPLE can help testing a scientific framework.

In the following sections we first present simulation of PDEs and DUNE, the scientific framework in the focus of our research, and then introduce SPLE for scientific software, especially for scientific frameworks. After that we describe the creation of a reengineering variability model for DUNE and the derivation of system test applications from the variability model. After the discussion of related work we summarize our findings and present our future work.

2. SIMULATION OF PDES

Before we introduce the DUNE framework and SPLE, in this section we explain some terminology in the context of numerical simulations. For further reading we refer to [9].

2.1 Numerical Simulation Terminology

Starting from observations the first step is to describe a system of components and their interaction. In natural science and engineering these interactions are usually *natural phenomena* like gravitation, fluid mechanics or heat transport, which are then formulated as *mathematical model*, often in terms of partial differential equations (PDEs) like the Poisson equation, Euler equation or heat transport equation. In general, it is not possible to solve these PDEs, or systems of PDEs analytically, thus *numerical methods* are used to find an approximation for the inaccessible analytical solution. The actual solution is obtained by a computer simulation. This should scale from the scientists laptop to high performance computers with thousands of cores. In the following, we provide a small glossary of the terminology used: A *PDE* is a relation involving an unknown function of several independent variables and their partial derivates with respect to those variables. They can be classified in elliptic (i.e. Laplace equation, stationary heat equation), parabolic (i.e. instationary heat equation) and hyperbolic (i.e. transport equation) PDEs. The function is usually spatial varying and can be scalar, like a temperature distribution, or vector valued, like a velocity field. The analyzed problem can be *stationary,* meaning that it does not depend on time or it can be *instationary,* meaning that some characteristics like position or temperature change with time.

We consider a bounded domain for which the mathematical model is assumed to be valid. This domain of dimension d can be embedded into a higher dimensional space of dimension w (e.g. a surface in a three-dimensional world). *Boundary conditions* complement the PDEs and describe the behavior of the solution on the boundaries of the region. Appropriate boundary conditions are necessary to guarantee the uniqueness of the solution. Additionally the solution of the PDE can depend on spacially varying parameters or functions, like source terms, material parameters or external forces.

To solve the PDEs numerically the exact solution is approximated by a discrete solution. Creating a numerical problem out of a mathematical problem is called *discretization*. Different discretization methods are possible and lead to different approximations with different properties. The most well-known classes of discretization methods are finite element methods (FEM), finite volume methods (FVM) and finite difference methods (FDM). All mentioned discretization methods are *grid based*. A *grid* is a partition of the computational domain into non-overlapping sub-regions called *grid elements*.

Instationary problems also need to be discretized in time. This usually means different solutions are computed for different discrete time steps. How the evolution from one time step to the next can be computed depends on the chosen *time stepping scheme*. For well-posedness of the problem, initial values are needed in addition.

This discretized problem yields a large system of linear or non-linear equations. *Solvers* are root-finding algorithms, which are used to numerically solve the equation system. For non-linear systems a *non-linear solver* (i.e. Newton's method, fixed point method) is used. Iterative non-linear solvers create a sequence of linearized systems. For solving linear systems two types of *linear solvers* are applicable: *direct solvers* (only for small problems) and *iterative solvers* (i.e. Richardson, Krylov subspace methods). The performance of an iterative linear solver can be improved by applying a *preconditioner* (i.e. Jacobi, Gauss-Seidel, SOR, ILU, multigrid) to the linear equation system.

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

2.2 Grid Terminology Example

As an example of a field of application for DUNE we take a closer look at the grid terminology. This example will be further used in the following sections. A detailed definition of a *grid* in DUNE can be found in [3].

A grid is a partition of a bounded domain into a set of grid elements, which can be described by a *reference element*, (e.g. cube or simplex) and a transformation into global coordinates that are transformations of specific *reference element types*. For simplicity, all figures in this section use 2D grids.

A grid element consists of different subentities, like *faces*, *edges* or *vertices*. A face is an entity of dimension *d*-1, in 2D it is the line. Edges are entities of dimension 1 and vertices of dimension 0. An *intersection* describes the contact area between two neighboring elements or an element and the domain boundary, like faces they are of dimension *d*-1. As we will describe later, intersections do not necessarily correspond to the faces.

A grid is *single-element-type* when all elements correspond to the same reference element. In a *multi-element-type* grid different reference elements are allowed. Figure 1 and 2 show examples of single-element-type and multi-element-type grids.

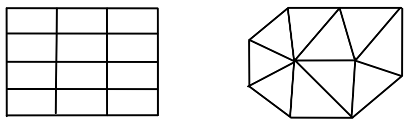

Figure 1. Single-element-type grids.

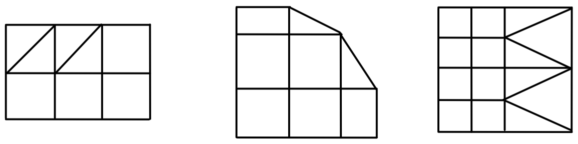

Figure 2. Multi-element-type grids.

A *structured* grid is a grid with congruent grid elements. An *unstructured* grid is more flexible, since the grid elements may be used in an irregular pattern. Figures 3 and 4 show examples of structured and unstructured grids. Note that a structured grid is always a single-element-type grid.

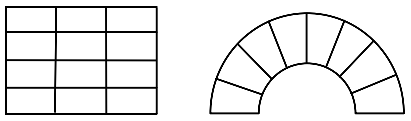

Figure 3. Structured grids

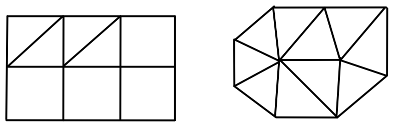

Figure 4. Unstructured grids.

A *conforming* grid is one where the intersection of two elements is either empty or a face of each of the two elements. Otherwise the grid is called *nonconforming*. Figure 5 shows examples of conforming and non-conforming grids.

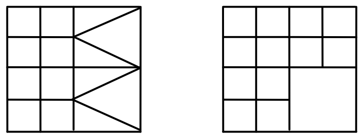

Figure 5. A conforming and a non-conforming grid.

To obtain a better numerical solution, it is possible to refine the grid. The refined grid is obtained by sub-dividing elements into smaller elements. Successive refinement leads to a hierarchy of grids.

A grid can be *globally* or *locally* refined. Global refinement means that all elements are refined, whereas local refinement means that only a subset of the elements is refined. Figures 6 and 7 illustrate the difference between global and local refinement.

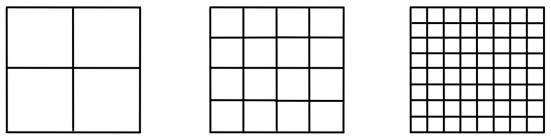

Figure 6. Hierarchy of globally refined grids.

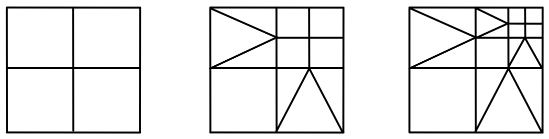

Figure 7. Hierarchy of locally refined grids.

Note that locally refined conforming grids are either multi-element-type or simplicial grids.

3. DUNE – A SCIENTIFIC FRAMEWORK

DUNE, the distributed and unified numerics environment, is a free software licensed framework for solving PDEs with grid-based methods [3], [4]. It supports the easy implementation of discretization methods like finite element, finite volume and finite difference methods. DUNE makes several grids and powerful mathematical implementations available. Its main principles are the separation of data structures and algorithms by abstract interfaces, efficient implementation of these interfaces using generic programming techniques and the reuse of existing finite element packages (i.e. UG[2], ALBERTA[3] and ALUGrid[4]) with a large body of functionality.

[2] http://atlas.gcsc.uni-frankfurt.de/~ug/

[3] http://www.alberta-fem.de/

[4] http://aam.mathematik.uni-freiburg.de/IAM/Research/alugrid/

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), an iterative solver template library (dune-istl [5]), an interface for finite element shape functions (dune-localfunctions) and tutorials for using and implementing 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.

Development of DUNE started about eight years ago. The distributed development team for the core modules consists of 4-8 scientists from mathematics, computer science and physics. Additionally there are up to 20 developers working on external modules. DUNE core modules consist of about 200.000 LOC in C++. It supports parallelism based on 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. In the following, we focus only on DUNE users who implement their own DUNE applications. Users of DUNE are mostly mathematicians, computer scientists and physicists at universities in Germany and abroad. Recently it was adopted for industrial applications for flow and transport processes in porous media[5]. 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. Big code changes are planned as milestones with some kind of a prioritization, however, without defined scheduling. The development is done when resources are available. The documentation consists of detailed code documentation, a user documentation and tutorials on mathematical concepts and their implementation. The documentation is available online.

The most important software quality goals for DUNE are 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. Still unsolved challenges for DUNE are the systematic adaption of algorithm verification and system tests.

4. SPLE AND SCIENTIFIC SOFTWARE

In this section we describe how we adopt SPLE to support testing scientific frameworks. We only explain some basics of SPLE. A more detailed description of SPLE can be found in the book "Software Product Line Engineering - Foundations, Principles, and Techniques" by Pohl et al. [17].

4.1 SPLE and Variability Modeling

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 [16]. The characteristics that can vary are called *variation points* and the possible values for a variation point are called *variants*. An example for a variation point for DUNE is a "reference element type" and its variants, restricted in 2D, are "cube" and "simplex". There are always a finite number of possible variants for a variation point.

Variation points are divided into *external* and *internal* variability. External variation points are visible to the users and stakeholders, like the variation point "reference element type". Internal variation points are hidden from the users and are only of interest to the developers of the product line. Typical causes for internal variation points are technical issues. Hiding such technical details leads to reduced complexity for the users. It may be *mandatory* to select a variant for a variation point or it may be *optional*. The variation point "reference element type" is mandatory. However, a variation point "grid refinement type" is optional, since it is only needed, if the grid is locally refined. It can also be defined how many variants (min, max) may be selected for a variation point. Usually only one variant can be selected for a variation point. In our example, the variation point "reference element type" constitutes an exception, as for a multi-element-type gird more than one variant can be selected.

Typically there are constraints between the different variation points and variants. A variation point (or variant) may *require* or *exclude* another variation point (or variant). For example, if we have a variation point "grid structure type" with variants "structured" and "non-structured" and a variation point "grid element type quantity" with variants "single-element-type" and "multi-element-type", then the variant "structured" requires the variant "single-element-type", since a structured grid always consists of only one element type.

A *variability model* includes all variations points and their variants of a product line. It also includes constraints between the variation points and variants. A variability model should answer at least the following questions: what varies (variation points), why does it vary (stakeholder needs, management decisions, technical variability etc.), how does it vary (available variants) and for whom is it documented (internal and external variability). A variability model should also include traceability information consisting of links to other development artifacts like use cases, design models, test cases or source code.

Our goal is not to create a detailed variability model over the whole range of applications that can be implemented with a scientific framework. This would be oversized for our goal to support the testing of a framework. With its huge range of functionality (based on mathematics) a scientific framework often includes almost unlimited variability. For our needs, the creation of a high level variability model of the framework is sufficient and more appropriate.

4.2 SPLE Development Processes

The SPLE process is divided into two development processes: domain engineering and application engineering. In the *domain engineering* process a reusable platform, including the *commonality* (common characteristics for every application in the product line) and variability, is defined for the product line [15]. The *application engineering* process is responsible for deriving

[5] http://www.sintef.no/Projectweb/GeoScale/Simulators/

applications from the product line platform that was established during domain engineering.

There are five key sub-processes in domain engineering: product management, domain requirements engineering, domain design, domain realization and domain testing. In the first sub-process, product management, a *roadmap* describing the scope and the goals of the product line is created [17]. The roadmap is used as an input when the first version of a variability model is created in domain requirements engineering. In traditional SPLE, the commonality analysis in domain requirements engineering is used to define the commonality for all product line applications. A high amount of commonality reduces the effort in designing the variability. Since we consider a scientific framework where the commonality and variability are already implemented, we can neglect the commonality analysis and concentrate on the roadmap and variability modeling.

Traditionally, the variability model is described in more detail in every further sub-process. The portion of internal variability grows, when design and realization variability are included. Since we consider an existing application, we are not particularly interested in domain design or domain realization. As described in Section 5, we first completed the sub-processes product management and domain requirements engineering. Then we used the variability model as an input for the domain testing sub-process.

In the sub-process of domain testing the challenge is to test the commonality and variability without access to the separate applications that are only created in application engineering. Pohl et al. [19] introduce several domain test strategies. One of the two recommended test strategies is Sample Application Strategy (SAS), where a few sample applications are used to test the domain artifacts. As not all possible applications are tested, application testing is still needed for the separate applications created in application engineering.

SAS is the test strategy we want to use for testing the scientific framework. Pohl et al. do not show how to choose the sample applications for SAS. In Section 5.3 we demonstrate how we use the high level variability model to do this.

The reusable domain artifacts (requirements, architecture, components and tests) that result from the domain engineering sub-processes include the product lines variability. In the application engineering sub-processes application requirements engineering, application design, application realization and application testing this variability is *bound* meaning that a specific variant is chosen for each variation point. Binding each variation point in the variability model, results in a separate application. The benefits of SPLE include a reduction of development effort, since new applications can be implemented by reusing the platform, an enhancement of quality, since the artifacts in the platform are thoroughly tested in many applications and a reduction of maintenance effort, since new variants can be inserted for a variation point with a reasonable effort.

4.3 SPLE and Scientific Frameworks

We apply SPLE to a scientific framework since we want to systematically describe parts of the variability of the framework. A framework consists of common code providing generic functionality for specific fields of application. 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 [14]. In our approach, we consider the framework as the product line platform. The applications developed by the users of the framework are then regarded as the product line applications.

This leads to a specific definition for the terms *developers* and *users* in the case of scientific frameworks. Developers in the sense of traditional SPLE carry out the domain engineering and application engineering processes. The users of traditional product lines at best can take a look at the set of external variation points, choose the desired variants and at the end get to use the separate application. When we apply SPLE to a scientific framework, the developers only deal with domain engineering. The developers set up a scientific framework including a huge amount of variability. The application engineering, that is the binding of the variability and the development of the application, is done by the users of the scientific framework. When we are talking about developers and users in this paper, we mean their specific roles in the context of scientific frameworks. In these terms the users of a scientific framework are at the same time the developers in the application engineering process.

This also means that the borderline between internal and external variability as a separation between the variability visible to developers only and the variability also visible to users is shifted. The users of a scientific framework have a more technical view on the frameworks variability. Yet, there is still variability dealing with implementation details that is only visible to the developers of the framework. At this point of our research this internal variability is not in our focus, since testing such implementation details are covered by unit testing.

SPLE supports the users of a scientific framework as they are developing their applications. Since most of the users of a scientific framework are scientists in a specific field of research who are not professional software developers, they often start a new application as a copy of a similar application and simply adjust it to their own needs. It can easily happen that the users do not understand the source code in full detail. SPLE can help the users to understand the source code better and to be aware of the development decisions they have to make. When the users follow the variability model and carefully bind every variation point they know that every important decision for their separate application has been made.

To developers of a scientific framework the importance of domain engineering rises. They have a lot less impact on application engineering which is performed by the users of the scientific framework. In domain requirements engineering the developers must keep in mind the needs of a wide range of different applications. In fact, the developers can not foresee all applications the users want to develop using the framework. In the case of scientific frameworks, the mathematical models used set some natural boundaries to their variability. Domain testing has a high importance to the developers, since they need to test the functionality of the scientific framework without knowing exactly what kind of applications the users are going to develop.

5. A REENGINEERING VARIABILITY MODEL FOR DUNE

In this section, we describe our process of creating the reengineering variability model for DUNE. Reengineering means the adjustment of a software system to improve the software quality. Thereby the software functionality remains mostly the same [2]. In our research, we created a variability model for an existing software and therefore we call it a reengineering variability model.

5.1 Reengineering Product Management

We decided to follow the instructions for creating a variability model as described by Pohl et al. [15]. Like described in section 4.2, the first sub-process in domain engineering is product management, where scope and goals of the product line are described in a product roadmap for the product line. A product roadmap determines the major common and variable features of the products. In our case, we can ignore the marketing and scheduling aspects of product management, since the framework already exists.

DUNE is a framework that enables the implementation of various applications of the product line, but does not include the implementation of these applications itself. In product management, we consider all possible applications of DUNE as goals. We are not describing the framework itself but its applications. Together with the scientists, we wrote down the procedure a DUNE user follows when creating a DUNE application and recorded the decisions she or he has to make during this procedure. Additionally we analyzed the documentation of DUNE and example DUNE applications, that are part of the user documentation, to find out the alternatives a DUNE user has. At this point, we did not analyze the source code, since we wanted the variability model to be based on DUNE's requirements. We had to be careful not to get lost in details. Since the whole framework already exists, it could easily happen that we start writing down detailed features of the software that do not belong to product management. This sub-process focuses on the stakeholders' view of the application. Thus, we were only interested in the goals of the application, not the implementation details. The following description sketches a rough version of the roadmap we developed for DUNE. The terminology used in it is explained in Section 2, including examples. The roadmap follows the procedure of deriving a solution for a realistic problem. The starting point of the procedure is the natural phenomenon that is in the focus of the users' research. The arrows symbolize the steps of formulating the mathematical model for this natural phenomenon and then choosing a numerical model for the mathematical model (some detail decisions are listed as nested bullet points).

Natural phenomenon

- ➔ Define the characteristics of the mathematical model:
  - o Create systems of PDEs
  - o For each equation: note if it is linear or non-linear
  - o Note if the problem is stationary or instationary
  - o Define boundary conditions, material parameters, etc.
  - o If instationary: define initial values
- ➔ Decisions for the numerical model
  - o Decide whether the systems of PDEs are split or solved monolytically
  - o If instationary: choose appropriate time stepping scheme
  - o Set up a spacial discretization of the PDEs
    - ▪ Define the used grid (dimension, reference elements etc.)
    - ▪ Select a discretization method (FVM, FEM or FDM)
    - ▪ If adaptive: choose adaption strategy and error estimator
  - o For non-linear equations: choose a non-linear solver
  - o Choose direct or iterative linear solver
    - ▪ If iterative: choose preconditioner

The first part of the roadmap for DUNE characterizes the different details of the mathematical model for the problem the user is solving. This is something the user cannot choose, since these characteristics are given by the mathematical problem. On the other hand, the second part, which concerns the numerical model, consists of decisions the user has to make mostly based on the mathematical model. This is where the variable features of DUNE can be seen. For example, the user has to decide whether the PDEs should be split or not and what kind of grid should be used. Many decisions depend on the characteristics of the mathematical problem (i.e. if a time stepping scheme is needed) or previous decisions (i.e. whether a preconditioner is needed or not). The roadmap also shows which characteristics are common for every DUNE application. Every application needs a grid, a discretization method, a linear solver etc.

The main goal for DUNE is to support the implementation of all grid-based methods in numerical solving of PDEs. In the roadmap this goal is described in separate goals on a high level, like supporting stationary and instationary problems or supporting different discretization methods.

5.2 Domain Requirements Reengineering

The next sub-process in domain engineering is domain requirements engineering. The goal is to create a high level variability model based on the roadmap created in product management.

The first step in the creation of the variability model for DUNE is the identification of the variation points. We did this by examining the roadmap and writing down the characteristics where the applications differ from one other. For example, the formulation "choose appropriate time stepping scheme" implies that "time stepping scheme" is a variation point. Other examples for variations points of the roadmap are the characteristics of a grid: "grid dimension", "grid reference element type" etc. Every goal in the roadmap results in a variation point. After the variation points were written down, the set of variants was defined for each variation point. If the variants were not documented in the roadmap, other documentation for DUNE or the mathematical theory was used as a source for the variants.

Variation points from the first part of the roadmap describe the characteristics of the mathematical model. Such variation points are, for example, “equation linearity” with the variants “linear” and “non-linear” or “stationarity” with the variants “stationary” and “instationary”. Some characteristics of the mathematical model (i.e. boundary conditions) cannot be formulated as variations points with a finite number of variants. These characteristics are handled separately when we derive the test applications from the variability model (see Section 5.3).

Next, we defined the dependencies between the variation points and the variants. This was done based on the scientists’ knowledge in the field of research and the mathematical theory underlying the applications. Tables 1, 2 and 3 and Figure 8 demonstrate an example of the variability model. The notation we used in the variability model is based on the notation by Pohl et al. [16]. We extended their graphical notation with a textual notation similar to Yu and Smith [21] who first introduced SPLE for scientific software. Since the notation is already described in detail in these two references, we only show a small example of the variability modeling for DUNE. We consider the variation points “Grid structure type”, “Grid conformity type” and “Grid refinement type”.

Table 1. Textual notation of the variation point “Grid structure type”

Variation Point Number	VP1	
Variation Point Name	vpGridStructureType	
Description	Defines whether the grid is structured or non-structured.	
Variation dependency	Mandatory	
Variant Number	Variant Name	Possible Values
V1_1	vStructured	Structured
V1_2	vNonStructured	Non-structured

Table 2. Textual notation of the variation point “Gridconformity type”

Variation Point Number	VP2	
Variation Point Name	vpGridConformityType	
Description	Grid conformity type defines whether the grid is conform or non-conform.	
Variation dependency	Mandatory	
Variant Number	Variant Name	Possible Values
V2_1	vConform	Conform
V2_2	vNonConform	Non-conform

Table 3. Textual notation of the variation point “Grid refinement type”

Variation Point Number	VP3	
Variation Point Name	vpGridRefinementType	
Description	Grid refinement type defines how the grid is refined.	
Variation dependency	Mandatory	
Variant Number	Variant Name	Possible Values
V3_1	vLocal	Local
V3_2	vGlobal	Global

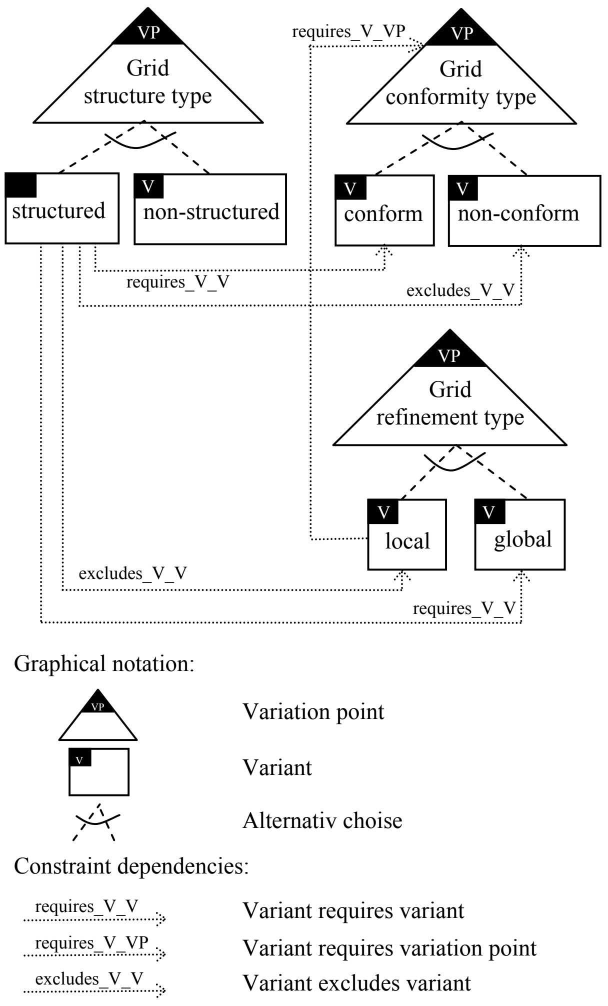

Figure 8. Example of the graphical variability model.

5.3 Deriving System Test Applications from the Variability Model

In this section, we describe the process we want to use to derive system test applications for a scientific framework using the variability model. This process corresponds to domain testing using SAS test strategy (see Section 4.2).

For a specific test application we have to choose a mathematical model with all its characteristics as described in the first part of the roadmap. The variation points describing the characteristics of the mathematical model are bound at this point. The characteristics that could not be described as variation point (like boundary conditions) are carefully documented and used when the test application is implemented.

There are two possibilities to derive test applications. The scientists can carefully choose mathematical models that DUNE’s users typically want to solve and then bind the variability suitable to these mathematical models. It may be possible to derive several test applications based on one mathematical model. The other possibility is to create allowed combinations of the variants in the variability model and then complete the mathematical model including the boundary conditions etc. These two ways can also be combined when the set of test applications is being defined.

For the example of the part of DUNE's variability model in Figure 8, there should be at least one test application that uses a structured grid. Because of the dependencies between the variants, the only possibility for this test application is to use a conforming grid with global grid refinement. To reach broad test coverage, there should also be test applications that use non-structured grids. In this case, there are no dependencies on the other variants or variation points. Therefore, for reaching a 100% test coverage on the high level, there should be test applications with the combinations of conforming grid with local refinement, conforming grid with global refinement, non-conforming grid with local refinement and a non-conforming grid with global refinement.

Ideally, the test applications cover the whole variability of the high level variability model. This would mean a 100% test coverage on the high level. This may be difficult to archive depending on the amount of variation points, variants and dependencies between them in the high level variability model. Since it is not always feasible to construct test applications for all possible combinations, it remains the responsibility of the scientists to reject the combinations that don't need to be included in the test applications since they are unlikely to occur.

Using the variability model as a basis for choosing the test applications 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. They can ensure that the selected test applications cover all typical uses of the framework. There should be no critical gaps between the set of test applications and the variability model that is every variant should be used at least once in a test application and all typical variant combinations should be covered by the test applications.

Next, every test application is implemented. At this point algorithm verification is included in the test applications. Depending on the used mathematical model in the test applications, suitable mathematical tests, like grid convergence testing, symmetry and conservation tests, are implemented for the test application. We still have to take a closer look at establishing algorithm verification in our research and to develop a way to systematically include it in the test applications.

The output of the tests depends on the used mathematical tests. There are self-contained tests which include evaluation logic and deliver "Passed" or "Not passed". Other tests may need a reference output value that was gained in a previous test application run. The test application output will then be compared to the according reference output.

The implemented test applications can be integrated into an automated system test environment for the framework. Developers can use the test environment for system testing when they commit changes in the source code. The test environment can be automated and run on a regular basis.

Our high-level goal is to build a system test environment in which a developer can choose the desired variants for each variation point she or he wants to test. After that, the environment would execute system tests which match to the selected combination of variants.

6. RELATED WORK

Easterbrook and Johns [8] investigated a software development process for a climate change simulation. There was an overnight automated regression test for this software as a part of the verification and validation process. These regression tests consisted of a bit-wise comparison of simulation results with a reference result to ensure the reproducibility of experiments. This kind of regression testing is not practical in a general case for at least two reasons. First, the differences in the outputs may also be caused by an enhancement in the code. Second, it is not possible to trace a problem in the code based on a change in the output. Easterbrook and Johns admit that the overall quality is hard to assess and the scientists tend to treat some errors as modeling approximations, rather than defects. Later on, some of these presumed approximations will be reported as bugs by the users.

As Kelly et al. [11] were testing scientific software for detecting artifacts in astronomical imagery, they found out that the goal of reaching 100% code coverage is not always necessarily a worthwhile pursuit for scientific software. It makes more sense that the scientists carefully consider which scientific goals the software should fulfill and design test scenarios that suit these scientific goals. Test data should be selected in a way that all typical use cases for the software are covered. Kelly et al. do not consider how the scientists can systematically determine the different scenarios that the code must handle.

SPLE was first introduced in scientific software by Yu and Smith [21] who used SPLE to describe the variability of one mathematical model (beam analysis problems) and its solution using PDEs and discretization method finite element analysis. First, differing from their approach, we use SPLE for the reengineering of an existing framework for several mathematical models and the descretization methods for solving PDEs. The second difference is the goal of using SPLE: Yu and Smith wanted to create a variation model that supports scientists in creating applications in the same field of application. Our purpose is to model the variability in the scientific framework to use it for systematically organizing the system testing of this framework.

SPLE is also adopted for the reengineering of legacy software in other fields of research than scientific software. This is typically applied when standalone applications in the same field of application with a similar content are maintained separately. There are several methods, architecture-centric or based on a feature model, for merging such standalone applications into a product line [12]. Yoshimura et al. [20] describe how they created a product line out of standalone applications. The focus is in merging software code of separate variants into one source code. The variability and commonality emerge from the source code or architecture analysis.

In our case we also apply SPLE to an existing application. Still, there are some significant reasons why we did not want to apply any existing product line reengineering methods in our research. First, we are not dealing with standalone applications. We do not have to merge any duplicated source code, since we are dealing with an existing framework. Second, and even more important, we do not want to derive the variability model from the source code or architecture. The reason is that we want to use the variability model for testing the framework. We want to test the software against its requirements and goals. That is why we create the variability model out of the requirement documentation. If the source code does not fit to theis variability model, then the software does not fulfill its requirements. Finding such mismatches is one important goal of testing.

7. SUMMARY

In this paper, we showed the feasibility of applying some SPLE sub-processes to scientific frameworks and introduced a way to create a reengineering variability model from an existing scientific framework. We discussed which parts of the SPLE development process are essential for the creation of a reengineering variability model. We argued for the high importance of the domain engineering process when SPLE is applied to a framework. We also sketched how the variability model can be used to derive a set of system test applications for this framework.

We believe that the combination of already existing unit tests for code verification and the system tests with algorithm verification described above build a stable structure for testing a scientific framework. However, there are still many aspects that we have to consider. Our work in the near future will include deriving and realizing a system test applications for DUNE as described in Section 5.3. In the course of a planned implementation sprint we will apply the system test applications and analyze the outcome.

After this trial phase we are going to refine the variability model of DUNE and insert more internal variability, including parallelism, and traceability links to the source code. We also need to take a closer look at the algorithm verification and its systematical use in the system test applications.

8. REFERENCES

[1] Ackroyd, K.S., Kinder, S.H., Mant, G.R., Miller, M.C., Ramsdale, C.A. and Stephenson, P.C. 2008. Scientific Software Development at a Research Facility. *IEEE Software* 25, 44-51.

[2] Arnord, R.S. 1994. Software Reengineering. IEEE Compuer Society Press.

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

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

[5] Blatt M. and Bastian, P. 2007. The Iterative Solver Template Library. In *Applied Parallel Computing. State of the Art in Scientific Computing* 4699, 666-675. Springer.

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

[7] Carver, J.C., Hochstein, L., Kendall, R.P., Nakamura, T., Zelkowitz, M.V., Basili, V.R. and Post, D.E. 2006. Observations about Software Development for High End Computing. In *CTWatch* 2(4A), 33-38.

[8] Easterbrook, S.M. and Johns, T.C. 2009. Engineering the Software for Understanding Climate Change. *Computing in Science & Engineering* 11, 65-74.

[9] Eriksson, K., Estep, D., Hansbo, P. and Johnson C. 1996. Computatuional Differential Equations, Campridge University Press.

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

[11] Kelly, D., Thorsteinson S. and Hook D. 2010. Scientific Software Testing: Analysis with Four Dimensions, Eds. IEEE Software.

[12] Obbink, H., Pohl, K., Kang, K.C., Kim, M., Lee, J. and Kim, B. 2005. Feature-Oriented Re-engineering of Legacy Systems into Product Line Assets – a Case Study. In *Software Product Lines* Springer Berlin / Heidelberg, 45-56.

[13] Oberkampf, W.L., Trucano, T.G. and Hirsch, C. 2004. Verification, Validation, and Predictive Capability in Computational Engineering and Physics. In *Applied Mechanics Rev*, 345-384.

[14] Pasetti, A. 2002. Software frameworks and embedded control systems, Springer-Verlang.

[15] Pohl, K., Böckle, G. and Linden, F. 2005. A Framework for Software Product Line Engineering. In *Software Product Line Engineering,* Springer Berlin Heidelberg, 19-38.

[16] Pohl, K., Böckle, G., Linden, F. and Lauenroth, K. 2005. Principles of Variability. In *Software Product Line Engineering,* Springer Berlin Heidelberg, 57-88.

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

[18] Pohl, K., Böckle, G., Linden, F. and Niehaus, E. 2005. Product Management. In Software Product Line Engineering Springer Berlin Heidelberg, 163-192.

[19] Pohl, K., Böckle, G., Linden, F. and Reuys, A. 2005. Domain Testing. In Software Product Line Engineering Springer Berlin Heidelberg, 257-284.

[20] Yoshimura, K., Ganesan, D. and Muthig, D. 2006. Defining a strategy to introduce a software product line using existing embedded systems. In *Proceedings of the Proceedings of the 6th ACM & IEEE International conference on Embedded software*, Seoul, Korea2006 ACM.

[21] Yu, W. and Smith, S. 2009. Reusability of FEA software: A program family approach. *In Proceedings of the Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, IEEE Computer Society, 43-50.