DISSERTATION

submitted

to the

Combined Faculty for the Natural Sciences and Mathematics

of the

Ruperto-Carola University of Heidelberg, Germany

for the degree of

Doctor of Natural Sciences

Put forward by

M.Sc. Hanna Remmel

Born in Varkaus, Finland

Oral examination: ....

Supporting the Quality Assurance of a Scientific Framework

Advisors: Prof. Dr. Barbara Paech
Prof. Dr. Peter Bastian

To my children.

Abstract

The quality assurance of scientific software has to deal with special challenges of this type of software, including missing test oracles, the need for high performance computing, and the high priority of non-functional requirements. A scientific framework consists of common code, which provides solutions for several similar mathematical problems. The various possible uses of a scientific framework lead to a large variability in the framework. In addition to the challenges of scientific software, the quality assurance of a scientific framework needs to find a way of dealing with the large variability.

In software product line engineering (SPLE), the idea is to develop a software platform and then use mass customization for the creation of a group of similar applications. In this thesis, we show how SPLE, in particular variability modeling, can be applied to support the quality assurance of scientific frameworks.

One of the main contributions of this thesis is a process for the creation of reengineering variability models for a scientific framework based on its mathematical requirements. Reengineering means the adjustment of a software system to improve the software quality, mostly without changing the software's functionality. In our research, the variability models are created for existing software and therefore we call them reengineering variability models. The created variability models are used for a systematic development of system test applications for the framework. Additionally, we developed a model-based method for test case derivation for the system test applications based on the variability models.

Furthermore, we contribute a software product line test strategy for scientific frameworks. A test strategy strongly influences the test activities performed. Another main contribution of this thesis is the design of a quality assurance process for scientific frameworks, which combines the test activities of the test strategy with other quality assurance activities. We introduce a list of special characteristics for scientific software, which we use as rationale for the design of this process.

We report on a case study, analyzing the feasibility and acceptance by developers for two parts of the design of the quality assurance process: variability model creation and desk-checking, a kind of lightweight review. Using FeatureIDE, an environment for feature-oriented software development as well as an automated test environment, we prototypically demonstrate the applicability of our approach.

Zusammenfassung

Die Qualitätssicherung wissenschaftlicher Software muss sich mit den speziellen Herausforderungen dieser Art von Software befassen. Diese Herausforderungen beinhalten fehlende Testorakel, den Bedarf von Hochleistungsrechenmaschinen und die hohe Priorität von nichtfunktionalen Anforderungen. Ein wissenschaftliches Rahmenwerk besteht aus gemeinsamem Quelltext, der Lösungen für mehrere ähnliche mathematische Probleme bereitstellt. Die verschiedenen möglichen Verwendungen eines wissenschaftlichen Rahmenwerks führen zu einer großen Variabilität innerhalb dieses Rahmenwerks. Zusätzlich zu den Herausforderungen der wissenschaftlichen Software muss die Qualitätssicherung wissenschaftlicher Rahmenwerke einen Weg finden, um diese große Variabilität zu bewältigen.

Die Idee bei der Produktlinienentwicklung (SPLE) ist es, zunächst eine Softwareplattform zu entwickeln und dann individualisierte Massenfertigung zu verwenden, um eine Gruppe von ähnlichen Anwendungen zu erstellen. In dieser Dissertation werden wir zeigen, wie SPLE und insbesondere Variabilitätsmodellierung eingesetzt werden kann, um die Qualitätssicherung von wissenschaftlichen Rahmenwerken zu unterstützen.

Ein Hauptbeitrag dieser Dissertation ist ein Prozess für die Erstellung von Reengineering-Variabilitätsmodellen für ein wissenschaftliches Rahmenwerk, basierend auf den mathematischen Anforderungen des Rahmenwerks. Das Reengineering bedeutet die Umstellung eines Software-Systems, um die Softwarequalität zu erhöhen - meistens ohne die Funktionalität der Software selbst zu ändern. In unserer Forschung werden die Variabilitätsmodelle für eine existierende Software erstellt und deswegen nennen wir sie Reengineering-Variabilitätsmodelle. Die erstellten Variabilitätsmodelle werden für eine systematische Entwicklung von Systemtestanwendungen für das Rahmenwerk verwendet. Zusätzlich haben wir eine modellgestützte Methode für die Herleitung von Testfällen entworfen. Diese Methode verwendet die erstellten Variabilitätsmodelle.

Darüber hinaus entwickelten wir eine Produktlinien-Teststrategie für wissenschaftliche Rahmenwerke. Eine Teststrategie hat einen starken Einfluss auf die ausgeführten Testaktivitäten. Ein anderer Hauptbeitrag dieser Dissertation ist der Entwurf eines Qualitätssicherungsprozesses für wissenschaftliche Rahmenwerke. Dieser Entwurf verbindet die Testaktivitäten aus der Teststrategie mit weiteren Maßnahmen der Qualitätssicherung. Wir stellen eine Liste besonderer Eigenschaften von wissenschaftlicher Software vor, die wir als Begründung für den Entwurf des Prozesses verwenden.

Wir berichten über eine Fallstudie, die die Machbarkeit und die Akzeptanz von zwei Bestandteilen des Entwurfs für den Qualitätssicherungsprozess untersucht: die Erstellung eines Variabilitätsmodells und den Schreibtischtest, eine Art von leichtgewichtigem Review. Anhand der Verwendung von FeatureIDE, einer Umgebung für die Feature-orientierte Software-Entwicklung, und mittels einer automatisierten Testumgebung haben wir die Anwendbarkeit unseres Vorgehens nachgewiesen.

Acknowledgements

This work was carried out during 2010-2014 at the Combined Faculties for the Natural Sciences and for Mathematics, in collaboration with the Interdisciplinary Center for Scientific Computing (IWR) at the Ruperto-Carola University of Heidelberg.

I owe my deepest gratitude to my first supervisor Professor Barbara Paech. I am grateful for her friendly and encouraging support over the years of my research. Her feedback on my work always pushed me forward and guided me in the right direction.

I would like to express my very great appreciation to my second advisor Professor Peter Bastian for introducing me to the world of numerics and scientific computing. I am particularly grateful for the possibility to work with DUNE during my research.

I would particularly like to thank Professor Christian Engwer for his essential support in clarifying numerics and DUNE specific issues. I am very thankful for his cooperation in writing scientific publications.

I am very grateful to the Heidelberg Graduate School of Mathematical and Computational Methods for the Sciences (HGS MathComp) and particularly Dr. Michael Winckler, for the scientific and practical support during my research.

I would like to offer my special thanks to my colleagues in the Software Engineering Group at the University of Heidelberg for their continued support and friendship. I would also like to thank my colleagues in the Scientific Computing Group for their support in the DUNE development, the participation in our case study and their feedback on our research. Thank you Professor Sven Apel and Thomas Thüm for your kind support on questions concerning FeatureIDE and FeatureC++.

I am deeply grateful to my husband and family for cheering and supporting me during my work. Very special thanks are reserved for Oma and Opa: without your support with own son, this work would not have been possible.

Contents

Part I

Preliminaries

Chapter 1

Introduction

This chapter provides an introduction to the subject of the thesis. The first section explains the motivation for our research on quality assurance for scientific frameworks. Subsequently, the goals and contributions of our research are introduced in Sections 1.2 and 1.3. Section 1.4 outlines the thesis and Section 1.5 provides an overview of previous publications relating to our research.

1.1 Motivation

Scientific computer simulations are increasingly important for decision making in politics and industry. One example of this is climate modeling used for multi-trillion dollar decisions concerning how to prevent or adapt to climate change. At the same time, the quality of such simulation software is subject to increasingly intense inquiry. Software defects that change the simulation output can negatively impact important decisions, and reduce decision makers' confidence in the science. Unfortunately, simulation software development teams often struggles with the consequences of extremely limited investments in software engineering processes, e.g. verification processes [33].

Scientific software can be anything from small scripts to large computationally intensive software packages. It is used for gathering and analysing data as well as report generation [77]. Shared, centralized computing resources and high performance computing are often needed for such large scientific software [7]. The primary interest of the developers and users of scientific software is gaining scientific results. The developers are predominantly domain scientists (e.g. mathematicians, physicists, or biologist) and not computer scientists. Most of them do not have any formal training in computer science or software engineering. Often, the primary users of scientific software are the developers themselves. They add functionality to the source code for the needs of their own research. There is a high turnover in the development teams of scientific software. Project members are often graduate students or postdoc, who are only involved in the development for a few years [19].

Scientific simulation software is often based upon a *scientific framework*. A scientific framework consists of common code, which provides solutions for several similar mathematical problems, such as the numerical solving of partial differential equations (PDEs). It is not a running application itself but can be used to develop various applications in the framework's domain. A framework can be extended by the user by overriding functionality or implementing interfaces [59].

Scientific software development teams find the *quality assurance (QA) of scientific software* very demanding. Most existing QA methods cannot be applied straight away without adjusting them [11]. Nonetheless, many teams consider verification and validation as among the essential elements of the development. As Baxter states in [11]: "if software is meant to do something, then that something can - and should - be tested for." Although the context of scientific software is special, the benefits of well-planned QA are just as favorable.

The *QA of a scientific framework* is a challenging task. On the one hand, it has to deal with special challenges of scientific software, like missing test oracles, which 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 [47]. Furthermore, the requirements of a scientific software project are often not known at the beginning of the project. The requirements stem from science and the priority of non-functional requirements (e.g. correctness, performance, portability or maintainability) over functional requirements is high [20]. The cognitive complexity, the difficulty in understanding a concept, thought, or system, is high for scientific software [45]. The calculations in the software can often only be performed on high performance computers [21] and include rounding errors and machine accuracy [38].

On the other hand, when ensuring the quality of a scientific framework, one additionally needs to find a way of dealing with the large variability, namely the various possible uses, of a framework. The variability is often hidden in the mathematics that the framework implements. The variability is expressed precisely, albeit not in a form that could be understood as a model by a human or computer. The question is: how can we systematically capture the variability of a scientific framework? How can we generate a set of tests for a scientific framework that cover the whole range of the frameworks functionality? Furthermore, what else do we need to consider when ensuring the quality of a scientific framework?

Our approach to meet this challenge is to apply *software product line engineering* (SPLE). In SPLE, the idea is to develop a software platform and then use mass customization for the creation of a group of similar applications that differ from each other in specific predetermined characteristics [62]. In particular, we apply variability modeling to systematically capture the framework's variability. We call the created variability models *reengineering variability models*. Reengineering is the adjustment of a software system to improve the software quality without changing its functionality. The created variability models are used for a systematic development of system test applications for the framework. In the QA of scientific software, the source code is often tested with unit testing. During unit testing, most functional failures should already have been found, whereas for scientific software system testing is still the only testing level where the interaction between the mathematical model, the numerical model, and its implementation can be thoroughly tested.

There is a need for QA methods for scientific simulation software and scientific frameworks. In particular, practices for verifying complete simulation runs (e.g. with system testing) are missing [1]. The main goal of our research is to develop efficient software engineering methods for the QA of scientific frameworks. We consider the overall QA management but mainly focus on the systematic development of system tests for the framework using the SPLE methods.

1.2 Research Goals

Our research has the following goals:

Goal 1: Develop software engineering methods for the QA of scientific frameworks that support an efficient quality management. The methods should consider special characteristics of scientific frameworks.

Goal 2: Examine the adoption of SPL methods for a systematic capturing of the variability in a scientific framework. The goal is to use the captured variability for a systematic development of tests for the framework.

Goal 3: Evaluate the extent to which the developed software engineering methods are feasible and acceptable for the development of a scientific framework.

Goal 4: Implement the developed software engineering methods prototypical for the scientific framework DUNE.

1.3 Research Contributions

This section summarizes the contributions of this thesis, which are connected to the research goals defined in the last section.

For Goal 1 (methods for the QA of scientific frameworks):

Contribution 1: A list of special characteristics of scientific software that need to be taken into account when designing QA for scientific software.

Contribution 2: Design of an overall QA process for a scientific framework, which takes the special characteristics in Contribution 1 into account.

For Goal 2 (adaption of SPL methods for scientific frameworks):

Contribution 3: A SPL test strategy for scientific frameworks that defines the activities in a SPL test process.

Contribution 4: A process for creating reengineering variability models for a scientific framework based on the framework's requirements.

Contribution 5: A way of systematically developing system test applications for a framework based on the variability models developed in the process in Contribution 4.

Contribution 6: A model-based method for test case derivation for scientific frameworks based on the variability models created with the process in Contribution 4.

For Goal 3 (evaluation of the developed methods):

Contribution 7: The design and execution of a case study evaluating the feasibility and acceptance of parts of the designed QA process defined in Contribution 2. The design of the case study was carefully documented so that it can be conducted for further cases.

For Goal 4 (prototypical implementation of the developed methods):

Contribution 8: A prototypical implementation of a system test application for DUNE using the methods in Contributions 4, 5, and 6.

1.4 Thesis Outline

The thesis is structured as follows:

Chapter 2 introduces the basic definitions and general concepts used in this thesis. It provides background information about QA for scientific software, development and QA of SPLs and introduces DUNE, the scientific framework with which we deal with in our research.

Chapter 3 explaines how methods of SPL development have been used in scientific software development and reengineering in the past, as well as how the software product line engineering (SPLE) development processes need to be adjusted to suit scientific frameworks. Furthermore, it presents a SPL test strategy named Variable test Application strategy for Frameworks (VAF) that we developed for scientific frameworks.

Chapter 4 presents a process for creating reengineering variability models based on the requirements of a scientific framework. It explaines how these variability models can be used for a systematic development of system test applications for a scientific framework.

Chapter 5 introduces VAF-Pro, an overall QA process for scientific frameworks. The detailed steps in the process are reasoned with characteristics of scientific software, the use of SPL test strategy VAF and essential QA demands.

Chapter 6 introduces a case study analyzing the feasibility and acceptance by DUNE developers for two parts of the QA process: variability model creation and desk-checking.

Chapter 7 deals with the practical implementation for the system testing part of the VAF-Pro QA process. It explaines how the tool FeatureIDE can be used for the implementation of the system test applications, as well as how the system test applications can be run in an automated test environment.

Chapter 8 summarizes the thesis and outlines some directions for the future research.

1.5 Previous Publications

Parts of the thesis have been published as scientific publications. The following list provides an overview of the relevant publications in chronological order, including the chapters and sections to which they contribute:

1. Remmel H, Paech B, Engwer C, Bastian P, **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, Waikiki (Honolulu), May 28, 2011, pp. 10-18, USA ACM — Sections 2.3, 2.4, 2.6, 3.1, 3.2, 4.1

2. Remmel H, Paech B, Bastian P, Engwer C, **System Testing for a Scientific Framework using a Regression-Test Environment,** In: Computing in Science and Engineering Vol. 14, No. 2, March 2012, pp. 38-45 — Sections 2.1, 2.3, 4.1.2.2, 4.2, 7.3.2

3. Remmel H, Paech B, Engwer C, Bastian P, **Design and Rationale of a Quality Assurance Process for a Scientific Framework,** Proceeding of the 5th international workshop on Software engineering for computational science and engineering, San Francisco, May 18, 2013, pp. 58-67, USA IEEE — Sections 1.1 2.1.8, 2.3, 2.4.1, 3.3, 5.1, 5.2, 5.3

4. Remmel H, Paech B, Bastian P, Engwer C, **A Case Study on a Quality Assurance Process for a Scientific Framework,** submitted for: Computing in Science and Engineering — Section 2.5

Chapter 2

Background

This chapter provides background information on topics that are relevant for this thesis. Sections 2.1 and 2.2 define basic concepts and acronyms frequently used in this thesis. In Section 2.3 we explain basics on QA for scientific software. Section 2.4 deals with the development of a SPL and Section 2.5 introduces methods for the QA for SPLs. Section 2.6 introduces DUNE, the scientific framework we deal with in our research. At last, Section 2.7 summarizes the chapter.

2.1 Basic Definitions

This section defines basic concepts handled in this thesis.

2.1.1 Software Engineering for Computational Science and Engineering

According to IEEE Computer Society, *software engineering* provides systematic, disciplined, and quantifiable approaches that support the development, operation, and maintenance of software [72].

Society for Industrial and Applied Mathematics defines *Computational Science and Engineering* (CSE) as "a rapidly growing multidisciplinary area with connections to the sciences, engineering, mathematics and computer science. CSE focuses on the development of problem-solving methodologies and robust tools for the solution of scientific and engineering problems [71]."

Software Engineering for CSE identifies and develops software engineering methods, tools and techniques for CSE [41].

2.1.2 Framework

A *framework* consists of common code, which provides solution for several similar applications for specific types of problems. Frameworks differ from software libraries, among other things, in the following two ways: First, the flow of control is not dictated by the caller, but by the framework (inversion of control). Second, a framework can be extended by the user by overriding functionality or by implementing interfaces [59].

2.1.3 Quality Assurance Process

According to IEEE Computer Society, "a *quality assurance process* is a process for providing adequate assurance that the software products and processes in the project life cycle conform to their specified requirements and adhere to their established plans [72]."

2.1.4 Unit Testing, Integration Testing, System Testing

Software testing can be divided into three levels depending on the target of the testing: *unit testing, integration testing* and *system testing.* Each of these testing levels is important and should not be neglected. In unit testing, the goal is to verify the functionality of single software units, small pieces of the software that are separately testable. In integration testing, the interaction between different software components is verified. System testing concentrates on the behavior of the whole software system. At this level the focus can also be set on non-functional system requirements like performance or accuracy [72].

2.1.5 Regression Testing

Automating test runs on a certain level of testing and repeating them on a regular basis leads to a *regression test* environment. The main idea is to show that modifications in the software code do not cause any unwanted side effects. In other words, running regression tests demonstrates to the developers that their changes did not break anyone else's code and that software, which previously passed the tests, still does [78].

2.1.6 Software Product Line

According to the Carnegie Mellon Software Engineering Institute (SEI), a *software product line* (SPL) is "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [17]."

2.1.7 Software Product Line Engineering

According to Pohl et al [62], the idea in *Software Product Line Engineering (SPLE)* is to develop a software platform and use mass customization for the creation of a group of similar applications (a SPL) that differ from each other in specific predetermined characteristics. The characteristics that can vary are called *variation points* and the possible values for a variation point are called *variants*. A *variability model* includes all variation points and their variants. It also includes the constraints between the variation points and variants [62].

2.1.8 Software Product Line Test Strategy

In a SPL test strategy, the partition of responsibilities between domain testing and application testing (for definition, see 2.4.1) is defined. The presence of variability is also considered. The used test strategy strongly influences the activities performed in domain testing and application testing [64].

2.1.9 Partial Differential Equation

In mathematical terms, a *partial differential equation (PDE)* is any equation involving a function of more than one independent variable and at least one partial derivative of that function. Many elementary phenomena, i.e. heat or fluid flow, can be described with PDEs. For computer simulations, numerical methods are used to solve PDEs by means of a computer [2].

2.1.10 Case Study

Yin [79] defines *case study* as "an empirical enquiry that investigates a contemporary phenomenon within its real life context, especially when the boundaries between phenomenon and context are not clearly evident."

2.2 Acronyms

This is a list of acronyms frequently used in this thesis:

CSE	Computational Science and Engineering
DC	Desk-Checking
DUNE	Distributed and Unified Numerics Environment
FDM	Finite Difference Method
FEM	Finite Element Method
FOP	Feature-Oriented Programming
FVM	Finite Volume Method
IDE	Integrated Development Environment
LOC	Lines of Code
PDE	Partial Differential Equation
QA	Quality Assurance
RQ	Research Question
SPL	Software Product Line
SPLE	Software Product Line Engineering
V&V	Verification and Validation
VAF	Variable test Application strategy for Frameworks
VAF-Pro	Quality Assurance Process for the Variable test Application strategy for Frameworks
VM	Variability Model

2.3 QA for Scientific Software

For the QA of 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 [19] (see Figure 2.1).

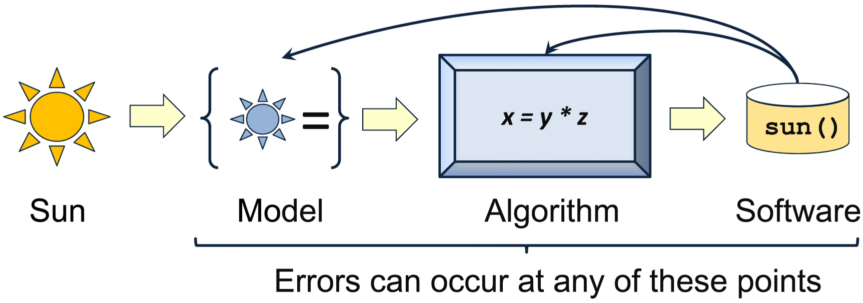


Figure 2.1: Possible Sources of a Software Problem in Scientific Software [28].

Each possible source of problem should be handled separately. Hook and Kelly [38] point out (as illustrated in Figure 2.2) 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.

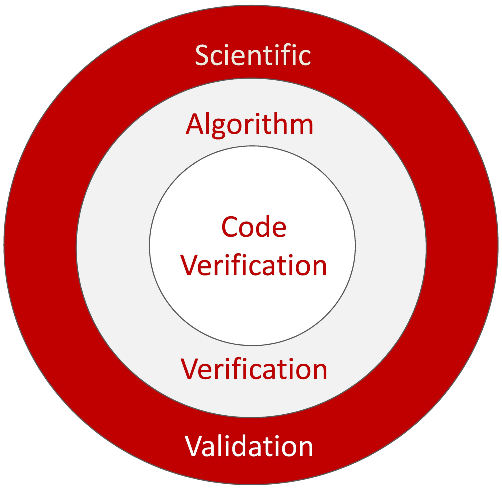


Figure 2.2: Model of Testing for Scientific Software [38].

The goal in *code verification* is to check whether the source code contains any faults or not. Typical software testing methods (e.g. black-box and white-box testing) can be applied. For scientific software, code verification is often performed using unit testing tools [58].

Algorithm verification is a process that focuses on the correct implementation of the numerical algorithms. It addresses the software reliability of the implementation of all the numerical algorithms that affect the numerical accuracy and efficiency of the code. Depending on the used numerical algorithm, there are several algorithm verification techniques that can be applied. The most confident method is comparing the software output with an *exact analytical solution*. This is followed by semianalytical and highly accurate *benchmark solutions*. Often, analytical solutions are only available for very small mathematical problem. Benchmark solutions are very accurate numerical solutions to special cases of a general PDEs (see Section 2.1.9). They commonly result from simplifying assumptions, such as simplified geometries [58].

The most widely used method of calculation verification is *grid convergence testing*. For single-grids (one discretization level) there are alternatives, like *energy methods* or *a posteriori methods*, but grid convergence testing is more general and more reliable [67]. Although the correct output of a simulation cannot be known in advance, it is often possible to identify *symmetries* in the theory being simulated that must be honored by a correct simulation, e.g. translational, rotational symmetry, or symmetry in time. Furthermore, any *conserved property* of the underlying theory (e.g. energy, momentum, or particle number), should also be preserved in the simulation code. The tests should stress parts of the system where errors may not be obvious, like on the boundaries [16].

In *scientific validation* the goal is to determine how accurate the computational model simulates the real world. In an ideal case we can compare the simulation with experimental data. Since this is mostly not possible, the goal in scientific validation is to support the scientists in deciding whether the simulation result is what they expected or not [58].

Several methods have been introduced for each of these steps of testing scientific software. Oberkampf et al. [58] give a broad overview of existing methods for verification, validation and prediction capability in computational science. Especially the suitability of methods for algorithm verification (e.g. 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 V&V methods for an application.

Software engineering for CSE (see Section 2.1.1) can help scientists in code verification and algorithm verification. The following subsections 2.3.1 and 2.3.2 introduce general software engineering methods that are useful for scientific software development. Subsection 2.3.3 discusses related work on QA for scientific software.

2.3.1 Statical Analysis and Dynamic Testing

Some methods of classic software quality assurance (SQA) are also adapted in scientific software development. Methods of *statical analysis* (e.g. reviews, inspections or audits) are not a regular part of development activities for scientific software, although it is widely recognized that statical analysis provides very effective QA methods [44].

According to IEEE Standard [40], a *technical review* is a "systematic evaluation of a software product by a team of qualified personnel that examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards. Technical reviews may also provide recommendations of alternatives and examination of various alternatives."

Methods of *dynamic testing* are widely used for scientific software. The source code is often tested with *unit testing.* During unit and *integration testing*, most functional failures should already have been found, but for scientific software, especially scientific frameworks, *system testing* is still the only testing level where the interaction between the mathematical model, the numerical model, and its implementation can thoroughly be tested. At system testing level also non-functional requirements like correctness and performance can be tested [72].

While unit testing is widely used for scientific software (e.g. [1], [34], [28]), currently system tests are seldom systematically adopted for complex scientific software. This may be partly due to the fact that there are several good tools for adopting unit testing in many different programming languages. It is easier to plan and implement tests for smaller pieces of the software. These steps can easily be integrated into the everyday work of scientific software developers when they are implementing or changing pieces of the software. System tests on the other hand, still mostly have to be planned and implemented from scratch. Their form and goals can be very different from one scientific software product to another. It needs a high level of domain understanding to plan system tests. 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.

2.3.2 Regression Testing

When developing scientific software in a team, regression testing gives the confidence that changes in the source code work together with the rest of the source code and changes made by other developers. Regression testing should be performed frequently and for every software modification. The main use of regression testing is to minimize the effort devoted to fixing incorrect source code changes [58].

Most regression test environments compare the behavior of two program versions to find out whether there are any changes or not. Deviations in the program behavior can be intended, such as bug fixes, or unintended, such as regression faults. Traditionally regression testing techniques characterize the program behavior due to the program output. Sometimes an old version's output is stored as an expected output. There are also methods, like program spectra, that compare the program versions' internal behavior in a black box testing manner [78].

When regression testing techniques are applied for a scientific framework, we need to take some specialties in the software's characteristics into account. For example, the output of scientific software often consists of floating point values. When we compare floating point values with each others, we have to take rounding errors into account. Often, stored expected output values with a tolerance range are used as a reference.

2.3.3 Related Work on QA Processes for Scientific Software

We could not find any other descriptions of QA processes for scientific software in the literature. For scientific software development, some models are introduced in the literature, like an iterative and incremental model by Segal in [70] and a staged delivery model, similar to a waterfall model, used by software projects at a research center by Baxter in [11]. For a development process in general and for QA in particular, we could find in [28], [37], [36], and [55] 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 QA process.

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* [23]. 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.

2.4 Development of a SPL

In this section, we introduce the development processes of SPLE (see Section 2.1.7) that are used in this thesis (Subsection 2.4.1). For variability modeling, we introduce two different approaches in Subsection 2.4.2.

2.4.1 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 [60]. The *application engineering* process is responsible for deriving 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 [62]. The roadmap is used as an input when the first version of a variability model is created in *domain requirements engineering.*

In *domain design*, a reference architecture that provides a common, high-level structure for all product line applications, is defined. *Domain realisation* deals with the detailed design and the implementation of reusable software components [60].

In *domain testing*, the goal is to ensure the quality of the reusable platform, including the commonality 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 [63].

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.

In *application requirements engineering* one major concern is the detection of deltas between application requirements and the available domain artifacts. It results in *requirement documentation* for a particular application. *Application design* uses the reference architecture as input and creates an *application architecture.* At last, *application realization* creates the considered application [60].

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 application engineering but also reuse test artifacts from domain testing. Variable artifacts that are only used in one or few applications are tested in application testing [63].

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 [62].

Traditionally, the *developers* of a SPL carry out both domain engineering and application engineering. They provide completed SPL applications to the *customers* or *users* of the SPL.

2.4.2 Variability Modeling

In SPLE, a *variability model* is created in domain requirements engineering and refined in further sub-processes in domain engineering. A variability model illustrates at least what varies and how does it vary. In the following subsections we introduce two different possibilities to represent a variability model: the orthogonal variability model by Pohl et al. in Subsection 2.4.2.1 and the feature diagram by Thüm et al. in Subsection 2.4.2.2. At first we used the orthogonal variability model in our research, but this model was rejected by the developers in a case study (see Section 6.3.1). As a consequence we changed to the feature diagram by Thüm et al.

2.4.2.1 Orthogonal Variability Model

Figure 2.3 demonstrates an example of the variability model notation by Pohl et al. [61]. It shows a small example of the variability modeling for DUNE. We consider the variation points "Grid structure type", "Grid conformity type" and "Grid refinement type" (for an explanation of the terms, see Section 2.6.3).

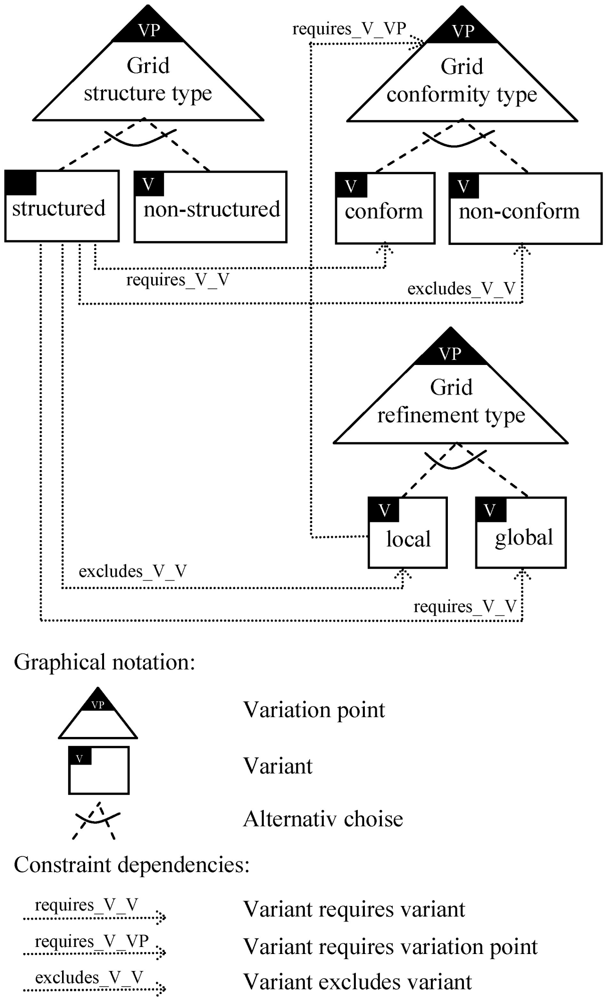


Figure 2.3: Example of the variability model notation by Pohl et al [61].

In an orthogonal variability model, the characteristics in a SPL that can vary are called *variation points* and the possible values for a variation point are called *variants*. An example for a variation point in Figure 2.3 is "Grid structure type" and its variants are "structured" and "non-structured". 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 "Grid structure 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 "Grid structure 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.

A special challenge for domain testing is *absent variants*. These are variants that are only created in application engineering for one or few applications and are not realized in domain engineering.

Typically there are *constraints* between the different variation points and variants. Variation point(s) (or variant(s)) may *require* or *exclude* other variation point(s) (or variant(s)). In our example, the variant "structured" for the variation point "Grid structure type" requires that for the variation point "Grid conformity type" the variant "conform" needs to be selected.

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), how does it vary (available variants) and for whom is it documented (internal and external variability). A variability model could also include traceability information consisting of links to other development artifacts like use cases, design models, test cases or source code.

2.4.2.2 Feature Diagram

This is the description of a variability model by Thüm et al. in [76]. Please note that since this variability model is a hierarchical tree, it does not distinguish between variation points and variants like the orthogonal variability model by Pohl et al. In a feature diagram both variation points and variants are called *features*:

> "A *variability model* defines the valid combinations of features in a domain [43].

> Variability models have a hierarchical structure, whereas each feature can have subfeatures [24]. The graphical representation of a variability model is a *feature diagram* and an example is shown in Figure 2.4. Connections between a feature and its group of subfeatures are distinguished as *and-, or-,* and *alternative-* groups [10]. The children of *and*-groups can be either *mandatory* or *optional.* A feature is *abstract*, if it is not mapped to implementation artifacts and *concrete* otherwise [75]."

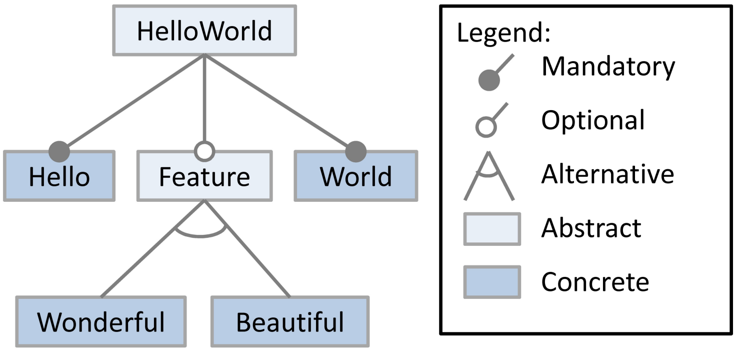


Figure 2.4: Example of the graphical variability model [76].

If a feature has more than one child features, there are three possibilities for the relationship:

- and: none, one or more than one child feature can be used at the same time
- or: one or more than one (but at least one) child feature can be used at the same time
- alternative: only one of the child features can be used at the same time

If a feature in the feature diagram is selected, its parent feature is automatically also selected. Furthermore, all mandatory subfeatures of an *and*-group must be selected, too. In *or*-groups, at least one subfeature must be selected and in *alternative*-groups, exactly one subfeature has to be selected [76].

A variability model may have *cross-tree constraints* to define dependencies which cannot be expressed otherwise. These constraints are expressed as logical formula over the set of features.

More about variability modeling using the feature diagram can be found in Section C.4.

2.5 QA for SPLs

This section introduces QA concepts for SPL (see Section 2.1.6). We could find three QA process descriptions for SPL in the literature. RiPLE-TE is introduced in detail in Subsection 2.5.1. Heider et al. [35] 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 QA methods, but do not discuss how these steps could form a QA process. Neto et al. [56] 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 (see Section 2.4.1) of the SPL and does not apply system testing.

The variability in a SPL leads to two interesting issues for preparation of tests for a SPL: test case derivation and test suite selection. Subsections 2.5.2 and 2.5.3 deal with these issues.

2.5.1 RiPLE-TE QA Process for SPLs

Based on a systematic mapping study on SPL testing and evaluated with an experimental study, Machado et al. [51] designed a QA process for product lines. The process, illustrated in Fig. 2.5, comprises both, domain testing and application testing. In domain testing the assets are designed considering the possible variable parts to be further reused in 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 QA 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

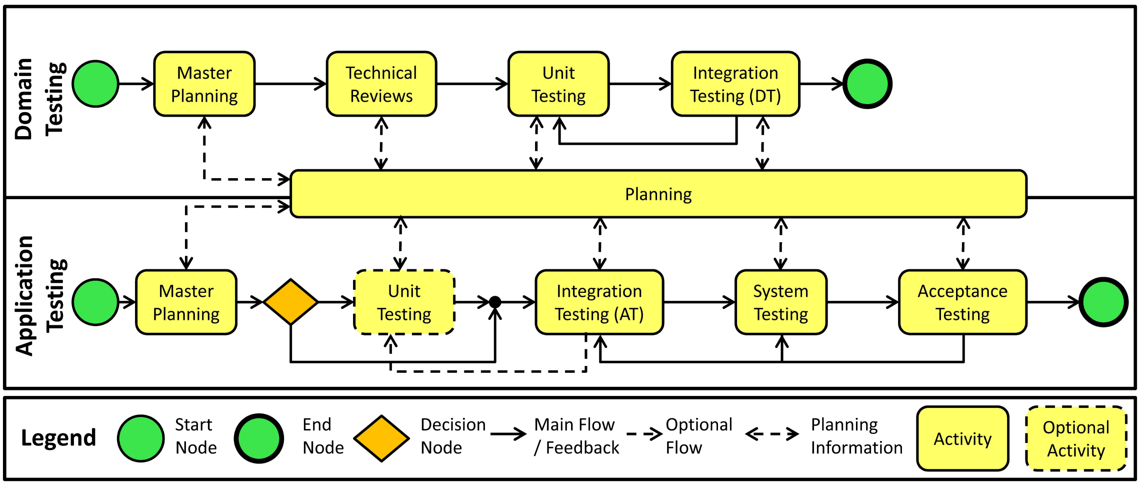


Figure 2.5: RiPLE-TE Domain Engineering and Application Engineering Workflow [51]

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 is based on the assets previously developed and tested and 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.

In special cases, when a new feature that does not yet belong to the product platform is included to a specific application, unit tests may be performed in application testing, too.

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 RiPLE-TE QA process, different test engineering roles are used: Test Manager, Test Architect, Test Designer and Tester. A test engineer can assume more than one role, and one role can be filled in by more than one engineer. Besides, other stakeholders like SPL Manager or Product Manager can be applicable to RiPLE-TE. Divers documents are the output of the RiPLE-TE process: Master Test Plan, Test Plan, Test Case, Test Software, Test Reports and Test Suites [50].

2.5.2 Test Case Derivation

The variability in a SPL leads to diverse requirements for the different SPL applications. It is a challenging task to systematically develop suitable test cases for all these applications. In the SPL literature one can find several methods for test case derivation. Most of them are *model-based*, meaning that they use some kind of a model for the presentation of the SPL requirements and use this model for the derivation of test cases.

Nebut et al. [54] present an automated approach of deriving system test cases for SPL applications from the SPL requirements. The requirements are modeled using enhanced UML use cases. Bertolino et al. [14] have developed a further modification of the use case notation for SPL, called Product Line Use Cases (PLUCs), which also supports the derivation of test cases for a specific SPL application.

ScenTED, a method for test case derivation by Reuys et al. [66] also uses use case scenarios. The method is based on the systematic refinement of generic use case scenarios to generic system and integration test case scenarios. Stricker et al. [73] have extended the method to ScenTED-DF, which detects redundant test cases based on data path analysis of the use cases.

Our model-based test case derivation method is based on variability models, as explained in Section 4.1.3.

2.5.3 Test Suite Selection

Once we have a set of test cases for a SPL (see Subsection 2.5.2), the next challenge is to build a suitable *test suite* out of the complete set of test cases available. Reducing the amount of used test cases is necessary, if the effort for executing all test cases exceeds the available resources for running the tests. However, the use of test selection techniques is only justifiable when the cost to select test cases is less that running the entire test suite [56].

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) and feature interaction (the most relevant feature combinations are selected based on statistical analysis) [48].

The *sampling method* should be chosen in a way that it retains as much of detect detection power of the original test suite as possible. The method should also concentrate on high-priority test cases. Possible criteria for sampling are for example [52]:

- Include all test cases that failed in the last test run.
- Include all test cases that cover the changes made to the source code.

Combinatorial Interaction Testing is especially suitable for us, because it is based directly on the variability model (see Section 2.4.2) and can be automated. This method selects a subset of all possible feature combinations, where possibly many potential feature 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 features the combinations "both available", "one available" and "none available" are tested [42]. 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 feature combinations, 2-wise leads to 41 and 3-wise to 119 test cases [49].

2.6 DUNE - a Scientific Framework

This section introduces DUNE, the scientific framework that is used in the case study and practical application for our research. Subsections 2.6.2 and 2.6.3 explain specific terminology that is used in examples in this thesis.

2.6.1 A Scientific Framework for the Simulation of PDEs

The Distributed and Unified Numerics Environment (DUNE[1]), is a free software licensed framework for solving PDEs (see Section 2.1.9) with grid-based methods [8], [9]. 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 (e.g.

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

UG[2], ALBERTA[3], and ALUGrid[4]) with a large body of functionality.

DUNE consists of several separate modules. Its users can put together a certain set of modules depending on their needs. The core modules deliver the basic classes (dune-common), an abstract grid interface (dune-grid), an iterative solver template library (dune-istl [15]), 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.

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

Some users use DUNE's interfaces to implement their own external modules. Most of the users use core and external modules to implement their own applications. Still others just use ready implemented DUNE applications. 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. Altogether, there are about 50-100 users.

The development team applies software engineering best practices like version management and configuration management. New requirements are collected using mailing lists and an issue tracking tool. Rapid prototyping is used to some extent. The high use of software engineering practices is untypical for scientific software [21] and may be due to the fact that some members of the development team are computer scientists.

Big code changes are planned as milestones with some kind of a prioritization, however, without defined scheduling. The development is done 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

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

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

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

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.

2.6.2 Numerical Simulation Terminology

This subsection explains some terminology in the context of numerical simulations. For further reading we refer to [30].

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 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 (e.g. Laplace equation, stationary heat equation), parabolic (e.g. instationary heat equation) and hyperbolic (e.g. 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* (e.g. 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* (e.g. Richardson, Krylov subspace methods). The performance of an iterative linear solver can be improved by applying a *preconditioner* (e.g. Jacobi, Gauss-Seidel, SOR, ILU, multigrid) to the linear equation system.

2.6.3 Grid Terminology

As an example of a field of application for DUNE we take a closer look at the grid terminology. A detailed definition of a *grid* in DUNE can be found in [8].

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 subsection 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. Figures 2.6 and 2.7 show examples of single-element-type and multi-element-type grids.

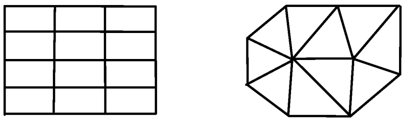


Figure 2.6: Single-element-type grids.

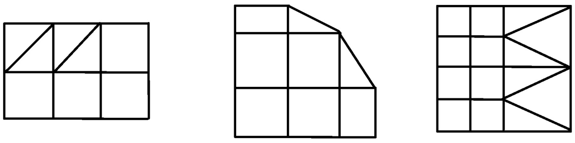


Figure 2.7: Multi-element-type grids.

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 2.8 and 2.9 show examples of structured and unstructured grids. Note that a structured grid is always a single-element-type grid.

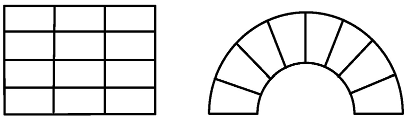


Figure 2.8: Structured grids.

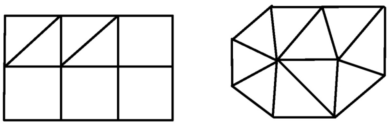


Figure 2.9: Unstructured grids.

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 2.10 shows examples of conforming and non-conforming grids.

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

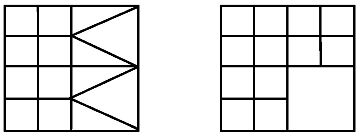


Figure 2.10: A conforming and a non-conforming grid.

refined, whereas local refinement means that only a subset of the elements is refined. Figures 2.11 and 2.12 illustrate the difference between global and local refinement.

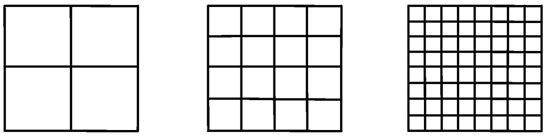


Figure 2.11: Hierarchy of globally refined grids.

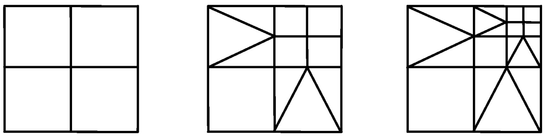


Figure 2.12: Hierarchy of locally refined grids.

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

2.7 Chapter Summary

This chapter provides basic definitions and background information on the topics handled in this thesis.

For the *QA of scientific software* it is important to consider different possible source for a software failure and to handle these separately: first check the source code for bugs with *code verification* methods, then verify the mathematical algorithm with *algorithm verification* methods, and at last evaluate whether the output is a reasonable proximity to the natural phenomenon in question with *scientific validation.* Software engineering for CSE can help scientists in code verification and algorithm verification. Recommendable methods of classic SQA for scientific software include reviews, dynamic testing, and regression testing.

In *SPLE* a software platform is developed in *domain engineering* development process and mass customization is then used in *application engineering* for the creation of a group of similar applications (a SPL) that differ from each other in specific predetermined features. In the first two domain engineering sub-processes, product management and domain requirements engineering, a *roadmap* describing the scope and the goal of the product line is created and used as an input for a *variability model.* A variability model illustrates the variable features of the product line and the constraints between them. In *domain testing*, the quality of the SPL platform, including the commonality and the variability, is ensured. *Application testing* tests the applications derived from the SPL platform reusing test artifacts from domain testing.

RiPLE-TE is a *QA process for SPLs* introduced by Machado et al. [51]. It comprises both domain testing and application testing. In this process, the activities in domain testing include the development of a master test plan, a technical review, and unit and integration testing. In application testing, integration, system, and acceptance testing is performed.

In *test case derivation for a SPL* the challenge is to systematically develop suitable test cases for all of the SPL applications. Most methods for test case derivation are *model-based*, meaning that they use some kind of a model for the presentation of the SPL requirements (e.g. UML use case diagram) and use this model for the derivation of test cases. The next challenge is to build a suitable test suite of the complete set of test cases. Several *test suite selection* methods can be found in the literature, for example sampling and combinatorial interaction testing.

DUNE is a *scientific framework for solving PDEs* with grid-based methods. It supports the use of different discretization methods like finite elements, finite volume, and finite difference methods. DUNE consists of several core modules and additional external modules, e.g. a discretization module dune-pdelab. This chapter explains some numerical simulation and grid terminology used in the examples in this thesis.

Part II

Comprehensive Quality Assurance of Scientific Frameworks

Chapter 3

SPLE in the QA of Scientific Frameworks

In this chapter we will explain how we apply SPLE for a scientific framework to systematically describe the variability of the framework. This is needed for the development of system test applications.

The first subsection provides an overview how SPLE has been used for scientific software before. After that we will explain how SPLE suites for a framework and how it supports framework's developers. Section 3.3 then introduces VAF, the SPL test strategy (see Section 2.1.8) we have designed for scientific frameworks. VAF-Pro, the QA process for a scientific framework introduced in Chapter 5 implements this SPL test strategy. Section 3.4 gives a summary of the chapter.

3.1 SPLE for Scientific Software and Reengineering

SPLE was first introduced in scientific software by Yu and Smith [81] 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 discretization methods for solving PDEs. The second difference is the goal of using SPLE: Yu and Smith wanted to create a variability 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 [57]. Yoshimura et al. [80] 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 (see Section 2.4.1) 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 (see Section 2.4.2) 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 the variability model, then the software does not fulfill its requirements. Finding such mismatches is one important goal of testing.

3.2 SPLE and Scientific Frameworks

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 (see Section 2.4.1). The users of traditional product lines at best can take a look at the set of external features, choose the desired features 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 thesis, 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 framework's variability. Yet, there is still variability dealing with implementation details that is only visible to the developers of the framework.

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 (see Section 2.4.1) 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. They have a lot less impact on application testing, which is performed by the users of the scientific framework.

3.3 SPL Test Strategy for Scientific Frameworks

As described in Section 2.1.8, an SPL test strategy defines, how the testing responsibility is partitioned between domain testing and application testing. Such considerations are necessary in order to achieve the goal of testing and to avoid problems related to the handling of variability [64].

In this section, we first briefly introduce SPL test strategies found in the literature and discuss why they are not suitable for scientific frameworks. Then, we introduce an SPL test strategy for scientific frameworks and assess it using the criteria for SPL test strategies introduced by Pohl et al. in [64].

3.3.1 Criteria for an SPL Test Strategy for Scientific Frameworks

In a manual literature review about SPL literature, we collected papers that mention SPL test strategies. The following SPL test strategies could be found:

A. Brute Force Strategy (test only in domain engineering) [64]

B. Pure Application Strategy or Testing Product by Product (test only in application engineering) [64], [74]

C. Incremental Testing of Product Lines (test the first application individually and the following applications using regression testing techniques; a special case for B) [74]

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) [63]

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) [64],[74]

F. Combination of strategies SAS and CRS (use fragments of a sample application in domain testing and create reusable test artifacts for application testing) [64]

G. Opportunistic Reuse of Test Assets (create test artifacts for one application in application testing and use these artifacts for further product line applications) [66]

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) [74]

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, an 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 in application testing.

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

Table 3.1: Fulfillment of Criteria for an SPL Test Strategy for a Framework

In most SPL strategies (B - H), at the most 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, an 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 3.1 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 [64]. 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. More or less, out strategy is a combination of the Brute Force Strategy (A) and CRS (E).

[1] Legend: 'X' = fulfills criterion, '(X)' = partly fulfills criterion

3.3.2 VAF - Variable Test Application Strategy for Frameworks

CRS test strategy distributes test activities between domain engineering and application engineering. Commonality, meaning the common code for all applications, is tested during domain engineering. Furthermore, test artifacts that contain variability are prepared for testing the variable parts is application engineering.

What we need is an 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 (see Section 2.4.2). If we had a variability model for the whole framework, we could derive the test applications and test cases thereof as illustrated in Fig. 3.1. Since it typically is not feasible to create such a variability model for the whole framework covering a wide range of functionality, 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 as illustrated in Fig. 3.2 (detailed description of the variability model creation for a scientific framework can be found in Chapter 4).

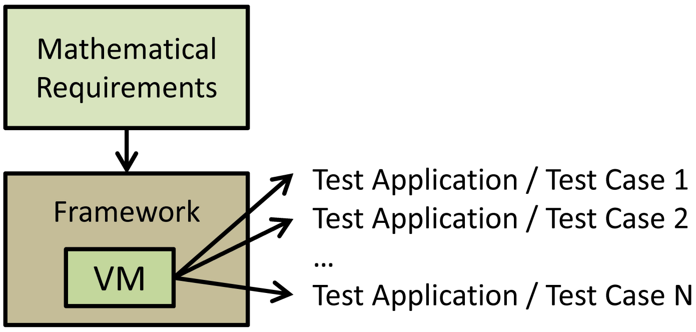

Mathematical Requirements
Framework
VM
Test Application / Test Case 1
Test Application / Test Case 2
...
Test Application / Test Case N

Figure 3.1: Derivation of Test Applications and Test Cases for a Framework with Variability Model (VM)

The mathematical requirements for the variability modeling should be chosen so that they cover typical uses of the framework, meaning that typical problems, mathematical models and numerical algorithms are covered. The goal is to reach possible high coverage for the available numerical algorithms in the framework. It is also possible to include special test problems in these test applications, for example complex grid structures, that are known to be tricky for the used numerical methods.

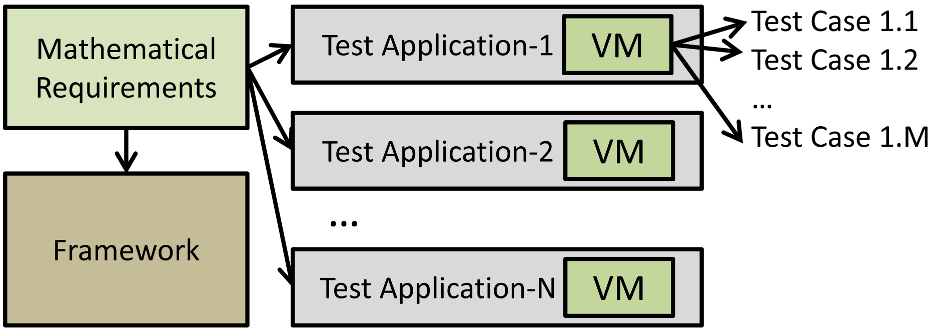


Figure 3.2: Derivation of Variability Models (VM) and the Associated Test Applications from the Framework's Requirements

For each variability model we develop an associated system test application, as shown in Fig. 3.2. Using the framework, the system test application solves the mathematical problem represented with the variability model. For example, one mathematical problem the DUNE framework (see Section 2.6.1) should support is solving the Poisson equation, an elliptic PDE. A variability model for this problem covers among others the different possible grid configurations and the discretization methods used (for details, see Section 2.6.2). The fulfillment of this mathematical requirement can be tested with the associated test application.

The users of the framework can reuse the unit and integration tests, the variability models and the system test applications when they are testing their own applications in application engineering.

Summarized, this is our SPL test strategy called Variable Test Application strategy for Frameworks (VAF). It fulfills the criteria for an SPL test strategy for a framework in Section 3.3.1 :

1. Test the commonality of the framework in domain engineering (e.g. using unit and integration testing).

2. Test the variability of the framework in domain engineering by creating variability models based on the mathematical requirements of the framework and developing system test applications associated to these variability models.

3. Provide reusable test artifacts (unit and integration tests, variability models and system test applications) that framework's users can use to test their specific applications in application engineering.

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.

3.3.3 Assessment

Pohl et al. [64] introduce five essential criteria for an 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 2.4.2.1)

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

Pohl et al. evaluate CRS positive with respect to the time to create test artifacts, since the inclusion of variability in the test artifacts significantly reduces the amount of tests needed to create from scratch in application testing. 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 "-".

For both, CRS and VAF, 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 features in application engineering, they can extend the reusable test applications to test them.

	Time to create	**Absent variants**	**Early validation**	**Learning effort**	**Overhead**
VAF	-[2]	+	+	-	+
CRS [64]	+	+	0	-	+

Table 3.2: Comparing VAF and CRS Test Strategies

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.

3.4 Chapter Summary

In this chapter, we explain how SPLE supports the developers of a framework. We introduce a SPL test strategy for scientific frameworks.

Yu and Smith [81] use *SPLE in scientific software* to describe the variability of one mathematical model and its' solution. In other fields of research (e.g. Yoshimura et al. [80]), SPLE is also adopted for the *reengineering*, i.e. for merging several standalone applications in the same field of application to a SPL by analyzing the source code or architecture. Differing from these approaches, we use SPLE for reengineering an existing framework for several mathematical models and methods. Since we want to test the framework against its requirements, we do not use the source code as a resource for the variability.

We consider the framework as the product line platform. The applications developed by the users of the framework are the product line applications. Therefore, the *developers* of a scientific framework only deal with *domain engineering*. They have only little impact on application engineering which is performed by the users of the framework. Domain

[1] Legend: '+' = positive, '-' = negative, '0' = neutral

testing has a high importance for 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.

A *SPL test strategy* defines how the testing responsibility is partitioned between domain testing and application testing. This chapter introduces SPL test strategies that can be found in the literature, but none of them fulfills the criteria for a SPL test strategy for a scientific framework. The test strategy needs to consider the special case of a framework where the developers only deal with domain testing.

In our *SPL test strategy for frameworks, VAF*, we create several variability models based on the mathematical requirements for the framework, i.e. the mathematical problems the framework should solve. The goal is to reach possible high coverage for the available numerical algorithm in the framework. For each variability model we develop an associated system test application that solves the mathematical problem represented with the variability model. Summarized, this is the VAF test strategy: in domain testing, test the commonality using unit and integration testing and the variability by creating variability models and system test applications as described above. The unit, integration, and system tests including the variability models can be reused by the framework's users in application testing.

Chapter 4

Creating Reengineering Variability Models and System Test Applications for a Scientific Framework

In this chapter, we concretize steps two and three in the VAF test strategy (see Section 3.3.2): creating reengineering variability models for a framework, developing system test applications for these variability models and providing reusable test artifacts for the application engineering. Reengineering means the adjustment of a software system to improve the software quality. Thereby the software functionality remains mostly the same [3]. In our research, we create variability models for existing software and therefore we call them reengineering variability models.

In the first step, reengineering product management, we create roadmaps that considers all possible applications of the framework (Section 4.1.1). In the second step, domain requirements engineering, we specify the mathematical problems and create variability models for them (Section 4.1.2). The system test applications can then be implemented with the help of the results from reengineering product management and domain requirements engineering (Section 4.2). In Section 4.3 we explain how framework's users can reuse the created artifacts in application engineering. Section 4.4 summarizes the introduced process.

4.1 Creating Reengineering Variability Models

Variability models (see Section 2.4.2) are created in the first two sub-processes in domain engineering: product management and domain requirements engineering (see Section 2.4.1). Traditionally, a 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 (for definition, see Section 2.4.2.1). Since we consider an existing framework, we are not particularly interested in domain design or domain realization (see Section 2.4.1). As described in the following sections, we first complete the sub-processes product management and domain requirements engineering. Then we use the results as an input for the domain testing sub-process (see Section 2.4.1).

4.1.1 Reengineering Product Management

In the test strategy VAF (see Section 3.3.2) we create several variability models based on the mathematical requirements of the framework meaning the mathematical problems, which the framework should solve. We follow the instructions for creating variability models as described by Pohl et al. [60]. Like explained in Section 2.4.1, 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 product line applications [60]. In our case, we can ignore the marketing and scheduling aspects of product management, since the framework already exists. Depending on the range of the mathematical requirements for the framework in question, we define one or few separate roadmaps, as described in detail in the following.

A framework 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 for the framework. We are not describing the framework itself but its applications.

Input for the definition of the roadmaps can mostly be found in framework's documentation and in example applications. Further input can be collected through interviewing the developers of the framework. At this point, it is not advisable to analyze the source code, since we want the roadmap to be based on framework's requirements, not the implementation. We have 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 are only interested in the goals of the application, not the implementation details.

We create the roadmap in three steps:

1. Define the goal(s) for the framework.
2. Define the general mathematical model for each goal described in Step 1.
3. Describe the general approach for solving the mathematical problem defined in Step 2.

4.1.1.1 Step 1: Define the Goal(s) for the Framework

In the first step we define one or more high level goals for the framework. If the mathematical requirements are so diverse that they cannot reasonably be formulated using one goal, several goals are defined. Each goal should describe what kinds of mathematical problems are solved with the framework and what kinds of approaches are used to solve these mathematical problems. The following questions can help in formulating goals:

- What is within the *scope* of the framework and what is outside?
  - What kind of *mathematical problems* can be solved (cannot be solved) with the framework?
  - What kind of *mathematical methods* are used (are not used) to solve the problems?
  - Are there limitations with respect to the *domain* (field of application)?

Example: Goal for the DUNE framework The examples in this chapter deal with the scientific framework DUNE introduced in Chapter 2.6. More precisely the example handle the DUNE module dune-pdelab, which is a descretization module.

The goal for the DUNE framework can be formulated as: "DUNE is a framework for solving *PDEs* with *grid-based* methods." This includes the definition of the group of mathematical problems (PDEs) and used mathematical methods (grid-based). There is no limitation in domain, since DUNE can be used in any field of application.

4.1.1.2 Step 2: Define the General Mathematical Model

In the next step we define a general mathematical model for each goal described in Step 1. We focus on the "mathematical problem" included in the goal definition. The following questions can help in defining the mathematical model:

- What is needed to define and restrict the mathematical problem in detail?
- What decisions does a user of the framework need to make when choosing the mathematical model?

Example: General mathematical model for the DUNE framework

The goal in DUNE framework is to solve PDEs. The following list shows what needs to be done to characterize the different details of the mathematical model for solving a PDE. The terminology used in the example is explained in Section 2.6.2.

- Create systems of PDEs
- Determine type of PDEs (elliptic, parabolic, or hyperbolic)
- For each equation: note if it is linear or non-linear
- Note if the problem is stationary or instationary
    - If instationary: define initial values
- Define boundary conditions, material parameters, etc.

4.1.1.3 Step 3: Describe the General Approach

There are probably several ways to solve the mathematical model(s) defined in Step 2 using the framework (with respect to possible limitations defined for the mathematical methods in Step 1). In the last step we consider, what can be said about general decisions that need to me made to solve the mathematical model using the framework without knowing the exact mathematical model.

Example: Decisions for the numerical model for solving a PDE with the DUNE framework

These are general decisions that need to be made when solving a (system of) PDE(s) with the DUNE framework:

- Decide whether the systems of PDEs are split or solved monolytically
- If instationary: choose appropriate time stepping scheme
- 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
- For non-linear equations: choose a non-linear solver
- Choose direct or iterative linear solver
  - If iterative: choose preconditioner

4.1.1.4 Result: the Product Roadmap

The outcomes from these three steps define the product roadmaps for the frameworks. Each roadmap gives guidance when defining specific mathematical problems and developing the corresponding variability models in domain requirements engineering (see Section 4.1.2). If new features are introduced during the life cycle of the framework, it may be necessary to adjust the product roadmaps.

The major common and the variable features of the framework that are by definition described by the product roadmap can be seen in the outcome of Step 3 (see Subsection 4.1.1.3). The outcome consists of decisions the user of the framework needs to meet to solve the mathematical model. Some characteristics always need to be defined (common features) and others are optional or can have different values (variable features).

Example: Common and Variable Features in DUNE Roadmap

Some major variable and common features of the DUNE framework can be seen in the

DUNE example is Step 3.

As an example for common features, every application needs a grid, a discretization method, and a linear solver.

The roadmap also shows which characteristics are variable. 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).

4.1.2 Domain Requirements Engineering

The next sub-process in domain engineering is domain requirements engineering. According to Bühne and Pohl it is "a process of defining the requirements for all foreseeable applications to be developed in the SPL" [18]. Based on the roadmaps provided by product management, detailed common and variable requirements are formulated in domain requirement engineering. The variable requirements are described in form of a variability model. We use the process for defining commonality and variability (see Section 2.4.1) for the system test applications for the framework.

First, in *commonality analysis*, we establish a list of common requirements for all system test applications for one roadmap.

Second, in *variability analysis*, the goal is to identify variabe requirements and to define the set of features. We create a variability model in three steps:

1. Define a concrete mathematical model.
2. Identify features and their dependencies.
3. Identify constraints between the features.

These three steps are repeated for each planned system test application.

4.1.2.1 Commonality Analysis

If possible, it is advisable to begin the commonality analysis with the creation of a list of all mathematical models that should be solved by the system test applications. This makes it easier to consider which requirements these system test application have in common.

A *checklist-based analysis* is one possibility of finding common requirements. Each item on the checklist represents a category of requirements that should be considered as candidates for common requirements [18]. A meaningful checklist for the common requirements depends on the particular framework in question and should be determined by the development team. Possible items for the checklist are:

- Basic requirements that every application must fulfill
- High priority requirements for the framework in question
- Strategic requirements are foreseeable basic needs that will appear in the framework's lifetime
- Requirements that are prescribed by mathematical laws and standards
- Requirements that are necessary for the technical support, like error handling, maintenance, communication, graphical output, etc.

Example: Common requirements for DUNE system test applications

DUNE system test applications that solve the kind of mathematical models described in the example in Subsection 4.1.1.2 have following common requirements:

- Support for parallelism (for parallel runs of the system test application).
- Error handling
- Output for algorithm verification (see Section 5.3.4)
- Graphical output for scientific validation (see Section 5.3.5).

4.1.2.2 Variability Analysis, Step 1: Define a Concrete Mathematical Model

In this step we concretize the general mathematical model defined in a product roadmap. For a specific system test application the developers choose a mathematical problem with all its characteristics as described in 4.1.1.2.

In an ideal case, the mathematical model for a system test application is well defined and not very simple or very complex. This makes it easier to create the variability model in the next step and leads to a reasonable system test application.

Example: Concrete mathematical model for "diffusion" system test application We define the conrete mathematical model using the general mathematical model defined in 4.1.1.2.

As a concrete test problem we consider a stationary test problem named *"diffusion"* and to simulate it, use the Poisson equation, an elliptic PDE. The mathematical model reads: Given a domain $\Omega \subset \mathbb{R}^d$. Find $u \in H^1$ such that

$$\Delta u = f \; in \; \Omega \tag{4.1}$$

$$u = g \; on \; \partial\Omega \tag{4.2}$$

where g denotes the *Dirichlet boundary conditions.* For an arbitrary solution of u we can choose f and g such that Eq. 4.1 is fulfilled. For our test we have chosen $u = e^{-|x|^2}$ to be a Gauss bell. This yields $g = e^{-|x|^2}$ and $f = (2 \cdot dim - 4 \cdot |x|) \cdot e^{-|x|^2}$. Given a particular numerical model, the numerical results can be compared to the analytic solution of u.

The equations in this test problem are all linear.

4.1.2.3 Variability Analysis, Step 2: Identify Features and Their Dependencies

Solving the mathematical model formulated in Step 1 requires the developers to choose from a range of different numerical methods and parameters. For the variability model we elaborate these options and formalize the variability.

The first step in the creation of the variability model is the identification of the variability. This starts with the analysis of the roadmap, which defines the major variable features (see Section 4.1.1.3). This definition is concretized for a concrete mathematical model formulated in Step 1. Since the variability defined in the roadmap is only high-level, solving a concrete mathematical model probably requires further variability, too. Framework's documentation and the mathematical theory are useful sources for variability definition. In general, each difference in structure, functionality or behavior between different numeral methods is a candidate for variability [18].

Drawing the variability model is explained in Section 2.4.2. We use the notation by Thüm et al. [76] as integrated in FeatureIDE, the tool we use for the implementation of the system test applications. More about variability modeling with FeatureIDE can be found in Section C.4.

Example: Variability model for "diffusion" system test application

For the definition of a variability model for the "diffusion" system test application we go through the description of the general approach for a DUNE system test application defined in Section 4.1.1.3. For each item in the description we need to decide how we want to implement the specific item. If we want to enable more than one possibilities for one item this lead to a new feature.

- Decide whether the systems of PDEs are split or solved monolytically

  → the PDE for the "diffusion" system test application is solved monolytically

- If instationary: choose appropriate time stepping scheme

  → does not apply since "diffusion" is a stationary problem (see 4.1.2.2)

- Set up a spatial discretization of the PDEs

  - Define the used grid (dimension, reference elements etc.)

    → "diffusion" system test application enables dimension 2D and 3D (feature *"dim"* with childfeatures *"dim_2"* and *"dim_3"*) and refenrence elements *"cube"* and *"simplex"* (feature *"mesh"*). The maximum grid width (used for global refinement of the grid) can be set to 2-8 (feature *"maxlevel"*).

  - Select a discretization method (FVM, FEM or FDM)

    → we enable two different FEM methods (feature *"method"*) for solving the "diffusion" problem: Lagrangian Finite-Elemente Method (*"FEM"*, [5]) and

Symmetrical Interior Penalty Galerkin method (*"SIPG"*, [27]). As a maximal polynomial order of the trial and test base the values 1-4 can be chosen (feature *"degree"*).

  - If adaptive: choose adaption strategy and error estimator

    → does not apply since used grids are not adaptive

- For non-linear equations: choose a non-linear solver

  → does not apply since we only have linear equations

- Choose direct or iterative linear solver

  → for solving the "diffusion" problem we use an iterative linear solver, more precisely a sequential Conjugate Gradient (CG) solver

  - If iterative: choose preconditioner

    → the "diffusion" system test application uses an ILU-preconditioner mit zero fill-in (feature "ILU0")

Please note: mathematically there are even more possibilities for the different features. The "diffusion" system test application could enable further dimensions (1D - 4D), discretization methods, solvers and preconditioners. Furthermore, the maximal polynomial order of the trial and test base (feature "maxlevel") and grid width (feature "degree") could mathematically enable any integer value. Anyhow, for a concrete implementation we can, e.g. for performance reasons, and also need, e.g. because only a finite number of features is technically possible, to make some restrictions.

The feature definition above leads to a variability model as shown in Figure 4.1. The variability model has a root feature "diffusion". All other features depend on this root feature.

4.1.2.4 Variability Analysis, Step 3: Identify Constraints Between the Features

Constraints between the features determine the permissible combinations of features for a system test application (see Section 2.4.2.2). The developers define the constraints based on their knowledge in the field of research and the mathematical theory underlying the applications.

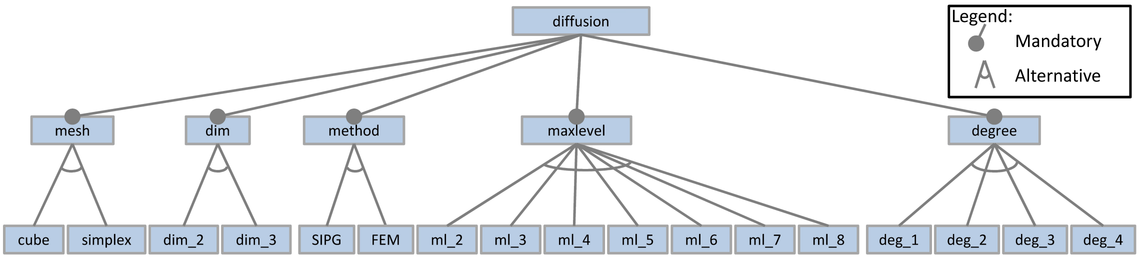


Figure 4.1: Variability model for the "diffusion" system test application.

Example: Constraints for "diffusion" system test application

Mathematically, there are no constraints between the features defined above. Nevertheless, for the system test application "diffusion" some constraints were defined for a better performance. The constraints for variability model of the "diffusion" system test application are illustrated in Figure 4.2.

$$\begin{gathered}
\text{cube} \wedge \text{dim\_2} \wedge \text{SIPG} \Rightarrow \neg \text{deg\_4} \\
\text{cube} \wedge \text{dim\_2} \wedge \text{FEM} \Rightarrow \neg(\text{deg\_3} \vee \text{deg\_4}) \\
\text{cube} \wedge \text{dim\_3} \wedge \text{FEM} \Rightarrow \neg(\text{deg\_3} \vee \text{deg\_4}) \\
\text{simplex} \wedge \text{dim\_2} \wedge \text{SIPG} \Rightarrow \neg \text{deg\_4} \\
\text{simplex} \wedge \text{dim\_2} \wedge \text{FEM} \Rightarrow \neg \text{deg\_4} \\
\text{simplex} \wedge \text{dim\_3} \wedge \text{SIPG} \Rightarrow \neg \text{deg\_4}
\end{gathered}$$

Legend:

$\wedge$	and
$\vee$	or
$\neg$	not
$\Rightarrow$	implies

Figure 4.2: Contraints definition for the "diffusion" system test application.

4.1.3 Deriving Test Cases for a System Test Application from the Variability Model

Our research contributes a new model-based method for test case derivation (see Section 2.5.2). The used model for the presentation of the requirements is the variability model. Different *test cases* for a system test application can be formed by selecting a value for each alternative feature in the variability model. Ideally, the system tests cover the whole variability of the variability model meaning that each possible combination of features is tested. This would mean a 100% test coverage. It may be difficult to achieve depending on the amount of features and dependencies between them in the variability model.

If all of the system 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. This can be accomplished with test suite selection methods introduced in Section 2.5.3.

In FeatureIDE, the test cases are realized using configuration files. Section C.7 explaines the use of configuration files in detail.

4.2 Developing System Test Applications

At last, the system test applications are implemented. Used technology (programming language etc.) depends on the framework in question. While developing a system test application for solving a concrete mathematical model, the developer needs to take into account

- the common requirements defined in commonality analysis (see Section 4.1.2.1),
- the concrete mathematical model (see Section 4.1.2.2),
- the implementation decisions made when creating the variability model in Step 2, that did not lead to variable features (see Section 4.1.2.3),
- and the variability model for the mathematical model and its constraints (see Section 4.1.2.3).

FeatureIDE is a tool that supports SPLE in different programming languages. We recommend using this tool for the development of system test applications. FeatureIDE is introduced in detail in Section 7.1.1.

Example: "Diffusion" system test application

Altogether, the requirements for the "diffusion" system test application are:

- common requirements (see 4.1.2.1): support for parallelism, error handling, and output for algorithm verification and scientific validation
- concrete mathematical model (see 4.1.2.2)
- implementation decisions (see 4.1.2.3): PDEs solved monolytically, use of an

iterative linear solver with "ILU0"-preconditioner

- variable features: mesh, dim, method, maxlevel and degree (see 4.1.2.3)

The "diffusion" system test application is a C++-program consisting of about 650 LOC. The feature-oriented source code (about 100 LOC, included in Appendix B) enables an easy use of the variability model in the program code. Feature-oriented programming is introduced in detail in Section 7.1.2.

Please note that the possibilities of feature oriented programming have not been exhausted in this example system test application. For example the different methods for solving the PDE (methods "runDG" and "runFEM") could also be included in the feature oriented source code.

The structure of the source code for the main program is included below. The complete source code can be found in Appendix B. The structure reveals how the requirements are combined in the source code.

```
// file: diffusion.cc
//
//-------------------------------------------------------
// Includes for the required dune modules
//-------------------------------------------------------
...
#include<dune/common/parallel/mpihelper.hh>
...

//-------------------------------------------------------
// Include for the feature oriented source code
//-------------------------------------------------------
#include"src/Features.h"

//------------------------------------------------------------------------
// Mathematical methods supporting the solving of the PDEs
//   class Parameter: definitions for tensor diffusion coefficient,
//                    velocity field, sink term, source term, and
//                    boundary conditions
//
//   class DifferenceSquaredAdapter: calculation of L2-error
//------------------------------------------------------------------------

template<typename GV, typename RF>
```

```
class Parameter
{
  ...
};
...

//----------------------------------------------------------------
// Methods runDG and runFEM for solving the PDE
//----------------------------------------------------------------

template<class GV, class FEM, class PROBLEM, int degree,
         int blocksize>
void runDG (const GV& gv, const FEM& fem, PROBLEM& problem,
            std::string basename, int level, std::string method,
            std::string weights, double alpha)
{
  ...
      //----------------------------------------------------------------
      // Use linear solver with "ILU0"-preconditioner
      //----------------------------------------------------------------
      typedef Dune::PDELab::ISTLBackend_SEQ_CG_ILU0 LS;
      LS ls(10000,1);
      typedef Dune::PDELab::StationaryLinearProblemSolver<GO,LS,U> SLP;


  ...
      //----------------------------------------------------------------
      // Output for the algorithm verification (common requirement)
      //----------------------------------------------------------------
      TEST_OUTPUT("dG-Level=" << level << " IT", ls_result.iterations)
      TEST_OUTPUT("dG-Level=" << level << " rate of convergence",
                                           ls_result.conv_rate)
  ...
  //----------------------------------------------------------------
  // Output for the scientific validation (common requirement)
  //----------------------------------------------------------------
  if (graphics)
    {
      ...
      vtkwriter.write(fullname.str(),Dune::VTK::ascii);
    }
}
```

```
...

//————————————————————————————————————————————
// Main program
//————————————————————————————————————————————

int main(int argc, char** argv)
{
  //————————————————————————————————————————————
  // Support for parallelism (common requirement)
  //————————————————————————————————————————————
  Dune::MPIHelper& helper = Dune::MPIHelper::instance(argc, argv);
  ...

  try
    {
    //————————————————————————————————————————————
    // Use of the feature oriented source code
    //————————————————————————————————————————————
      Features features;
      if (features.mesh()=="CUBE")
        {
          const int dim = Features::F_DIM;
          ...
        }
    ...
    }
  //————————————————————————————————————————————
  // Error handling (common requirement)
  //————————————————————————————————————————————

  catch (Dune::Exception &e)
    {
      std::cerr << "Dune reported error: " << e << std::endl;
      return 1;
    }
   ...
}
```

The implementation of the "diffusion" test application is based on an existing test application that was previously used for manual test runs. Most of the variability was already implemented in the test application as input parameters. It was possible to select the used grid element type, grid dimension and discretization method. As we compared the implementation with the variability model, we noticed that still only four of the possible 24 combinations of features were supported by the test application. Thus the variability modeling helped us to detect missing functionality in our test application and to enhance the test environment.

The output for algorithm verification for the test application "diffusion" includes the count of iterations and the rate of convergence for the linear solver used. In addition the global size for the grid function space and the L_2-error for the numerical solution are calculated and given out (for further information about algorithm verification see Section 2.3). For scientific validation the simulation output is redirected into a separate output file.

4.3 Supporting Application Engineering with Reusable Test Artifacts

In application engineering (see Section 2.4.1), framework's users develop and test their own applications (see Section 2.4.1). Artifacts created in reengineering product management and domain requirements engineering as explained in this chapter can be reused by the framework's users in application engineering.

For the development of their own applications the users can reuse the general mathematical models, the descriptions of general approach and the common requirements for applications created by the framework's developers in domain engineering. Hereby the users can follow the instruction for defining a concrete mathematical model in Section 4.1.2.2 and concretizing the general approach as explained in Section 4.1.2.3. The only difference is that in concretizing the general approach the users do not define variable features but decide on specific features for each item in the description of the general approach. This way the result is not a variability model but a list of concrete requirements for an application.

Since most of the users 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. Reusing artifacts from domain engineering can help the users to understand the source code better and to be aware of the development decisions they have to make.

The framework's users can naturally also develop their own system test applications for testing their own implementations for mathematical methods. The users can follow the instructions in this chapter for developing system test applications. They can also reuse (adjust or extend) existing system test applications created by framework's developers in domain testing. This is possible, if the application domain and solved mathematical problems are suitable.

4.4 Chapter Summary

This chapter introduces a process for creating reengineering variability models and system test applications for a scientific framework. This concretizes the steps two and three in the VAF test strategy. Each step is demonstrated with a concrete example.

In *reengineering product management* we create *product roadmaps* that describe the scope and the goals of the scientific framework and determine the major common and variable features of all possible applications for the framework. A roadmap is created in three steps. First, we define one or more *high level goals* for the framework, depending on how diverse the mathematical requirements for the framework are. Each goal should describe what kinds of mathematical problems are solved with the framework and what kinds of approaches are used for it. Second, we define a *general mathematical model* for each goal. Hereby we focus on the mathematical problem included in the goal definition and restrict it in detail. Third, we describe a *general approach* for solving the general mathematical problem(s). In this step we consider, what kind of general decisions need to be made by a user of the framework when solving the mathematical problem without knowing the exact mathematical model.

Domain requirements engineering consists of two parts. First, in *commonality analysis*, we establish a list of common requirements for all system test applications for a roadmap. A checklist-based analysis can be helpful in finding common requirements. Second, in *variability analysis* we specify the mathematical problems and create variability models for them. This is done in three steps. In the first step we *choose a concrete mathematical model* for a system test application and formulate it with all its characteristics as described in the definition of the general mathematical model. In the second step we *identify features and their dependencies*. We analyze the roadmap, which defines the major variable features and concretize this definition for the concrete mathematical model. In general, each difference in structure, functionality or behavior between different numerical methods is a candidate for variability. In the third step we *identify constraints between* *the features* that determine the permissible combinations of features for the system test application. Different *test cases* for a system test application can be formed by selecting a value for each alternative feature in the created variability model.

At last, the *system test applications are implemented.* A developer needs to take into account the common requirements, the concrete mathematical model, and the variability model resulting from the previous process steps. The tool FeatureIDE supports the system test application development in different programming languages.

Framework's users can *reuse the created artifacts in application engineering.* For the development of their own applications the users can reuse the general mathematical models, the descriptions of the general approch and common requirements. Furthermore, the users can also develop their own system test applications following the instructions in this chapter or adjust existing system test applications.

Chapter 5

VAF-Pro, a QA Process for a Scientific Framework

This chapter presents the design of VAF-Pro, an overall QA process (see Section 2.1.3) for scientific frameworks. Since we apply SPLE in our research to handle the framework's variability, as explained in Chapter 3, VAF-Pro also reflects SPLE principles. It implements VAF, the SPL test strategy for frameworks introduced in Section 3.3.2. However, the use of SPLE is not the only aspect we need to consider for an overall QA process. The fact that we are dealing with scientific software has a major influence on it.

As discussed in Section 2.3.3 we could not find any other QA process for scientific frameworks in the literature. The design is based on RiPLE-TE, a QA process for SPL presented in Section 2.5.1. VAF-Pro was designed to fulfill the requirements of the scientific framework DUNE (see Section 2.6.1), but is also suitable for other scientific frameworks. If adopted for a framework in another domain, the process should be adjusted to suit the characteristics of that domain.

The QA is done across three angles of view: the different testing levels (unit, integration and system testing), different goals in V&V of scientific software (code verification, algorithm verification and scientific validation), and regression testing as the system evolves over time.

In the following sections, we first in Section 5.1 discuss characteristics in scientific software development that need to be considered, when designing a QA 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 show how the RiPLE-TE QA process needs to be adapted to take into account the characteristics for scientific software development. Section 5.2 introduces the test roles we use in the QA process. The detailed steps in the QA process are explained in Section 5.3. Sections 5.4 and 5.5 introduce the use of automated regression testing and reporting. Section 5.6 includes some additional remarks to the QA process and Section 5.7 summarizes the chapter.

5.1 Characteristics of Scientific Software Development

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. Afterwards, we filtered out those characteristics, that need to be taken into account when designing a QA 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 QA process. We found eight papers describing such characteristics.

The characteristics are presented in Table 5.1. These are used as rationale in the following description of VAF-Pro. The table includes for each characteristic a label (C1-C12), a short description, the references where the characteristic was found and a list of the QA process steps, where the characteristic is considered.

	Characteristic	**Reference**	**Considered in**
C1	Different possible sources for a software problem. Need support for Code Verification, Algorithm Verification and Scientific Validation.	[28],[19],[38]	Rev, U&I, Sys, SV[1]
C2	Lack of test oracles.	[47]	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.	[20]	Pla
C4	The cognitive complexity, the difficulty in understanding a concept, thought, or system, is high.	[45]	TR, Pla, Rev, SV
C5	Need for shared, centralized computing resources; high performance computing, parallelism.	[7]	U&I, Sys
C6	Calculations include rounding errors and machine accuracy.	[38]	Sys, SV
C7	Most developers are domain scientists or engineers, not software engineers.	[19],[46],[7],[20]	TR, Rev
C8	There is a high turnover in the development team.	[19]	Pla, Rev, Rep
C9	The most highly ranked project goals: 1. Correctness	[20]	Sys, Val
C10	The most highly ranked project goals: 2. Performance	[20]	Sys
C11	The most highly ranked project goals: 3. Portability	[20]	Sys
C12	The most highly ranked project goals: 4. Maintainability	[20]	Rev, Rep

Table 5.1: Characteristics of Scientific Software

[1] Legend: TR = Test Roles, Pla = Planning, Rev = Review, U&I = Unit and Integration Testing, Sys = System Testing, SV = Scientific Validation, Rep = Reporting

In the following, the characteristics are discussed in more detail.

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 approximation of the real world).

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 means that rounding errors and machine accuracy (the accuracy to which floating-point numbers can be presented on a particular computer) need to be taking into account 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 QA 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 of the process 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. [20] found out, that the most highly ranked scientific software project goals are correctness, performance, portability and maintainability.

We use the characteristics described in this section as rationale for adjustments of the QA process RiPLE-TE introduced in Section 2.5.1.

In the following sections, we discuss how we need to adjust this QA 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 5.1. The rationale based on these characteristics is marked in the text in parentheses.

5.2 Test Roles

In the RiPLE-TE QA 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 QA process should be as simple as possible.

In VAF-Pro, 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 VAF-Pro, it may be more suitable to perform them in a team of developers than by one developer alone. This will be mentioned in the detailed description of the process.

5.3 QA Process Steps

This section discusses the steps in the VAF-Pro QA process. The process is illustrated in Fig. 5.1. The scientific software angle of view can be seen in the separation of the process in code verification, algorithm verification and scientific validation (blue rectangles in Fig. 5.1). This separation in the QA of scientific software is explained in Section 2.3. The different testing levels unit, integration and system testing (see Section 2.3.1) are presented in the process as well as the regression testing (see Section 2.3.2) angle of view.

Table 5.2 provives an overview about the adjustments made in VAF-Pro QA process compared to RiPLE-TE.

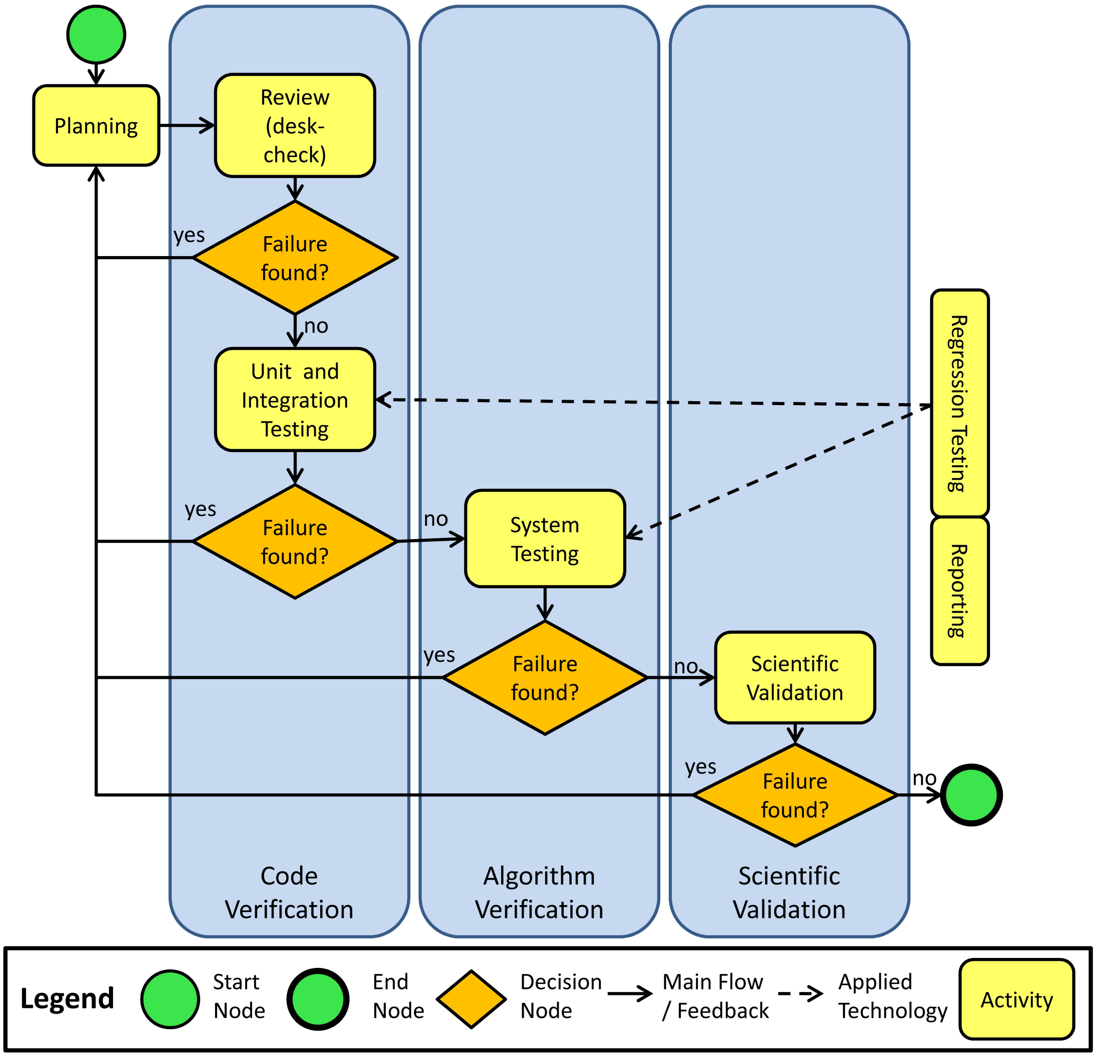


Figure 5.1: VAF-Pro QA Process for Scientific Frameworks

Adjustment	**Description**
Only Domain Testing	The main difference to RiPLE-TE is that VAF-Pro only covers domain testing and not application testing, since in the development of a framework we only deal with domain engineering (see Section 3.2). RiPLE-TE considers both domain testing and application testing.
System Testing already in Domain Testing	In VAF-Pro, system testing is already conducted in domain testing. Shifting a major part of the QA responsibility from application testing to domain testing in the VAF test strategy, as described in Section 3.3.2, results in introducing system tests already in domain testing. When developing a scientific framework, domain testing should ensures the quality of the whole framework and system testing is needed to ensure this (for details see Subsection 5.3.4).
Scientific Validation	Since VAF-Pro deals with scientific software, the QA step scientific validation is included to the process (see Section 2.3).
Planning	RiPLE-TE expects continuous planning with several predefined output documents. Instead of creating documents, since the goal is to keep the process practical and simple (C7), QA planning in VAF-Pro includes the development of unit, integration and system tests (for details see Subsection 5.3.1).
Review	Review in RiPLE-TE is a technical review on SPL main assets like feature model, product map and so on. A timetable for reviews is established in project planning and the reviews are done during group meetings [50]. In VAF-Pro, the review step is adjusted since the scientists are mostly developing the source code alone. The review considers SPL assets (variability model) as well as development outcome and is conducted by the developer simultaneously to the development (for details see Subsection 5.3.2).
System Testing	In RiPLE-TE, system testing tests a set of SPL applications previously defined during application engineering. Each application is tested with traditional testing methods for single applications. In VAF-Pro, we first need to define a suitable set of test applications and then include test methods for scientific software, as described in Subsection 5.3.4.

Table 5.2: Adjustments Made in VAF-Pro QA Process Compared to RiPLE-TE

5.3.1 Planning

QA planning in VAF-Pro is performed during the software development and includes the development of unit, integration and system tests. 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 QA issues:

- Do unit tests exist for the adjusted or new source code?
  - If yes, adjust the unit tests if necessary.
  - If no, create new unit tests.

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) [12] 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). Further discussion about unit and integration testing can be found in Subsection 5.3.3.

If the mathematical requirements for the framework change, e.g. when a new functionality is included into the framework, the developer needs to consider the following:

- Do system test applications exist for the adjusted or new functionality?
  - If yes, adjust the system test applications (e.g. insert new tests for algorithm verification) and the associated variability models if necessary.
  - If no, develop new system test applications and the associated variability models.

The developer may need to formulate one or more new variability models based on the requirements together with the associated system test applications, as described in detail in chapter 4. In other cases only existing variability models and system test applications must be adjusted, e.g. by including new features.

VAF-Pro does not expect any formal plan documentation. 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 distribute the responsibility for the development changes, they also should decide who is responsible for preparing the tests.

5.3.2 Review

The review step is conducted right after developing source code, unit and integration tests, system test applications, or variability models. A suitable moment is right before checking in the changes in a *version control system*. This is the earliest possible point for finding failures.

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 strive to write comprehensible source code with a sufficient amount of comments. This would also be beneficial to 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 QA process, VAF-Pro involves a review of 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 and integration tests, variability models, and system test applications. Review is one part of the code verification needed for the V&V of scientific software (C1).

A *desk-checking checklist* reminds the developers on all important aspects of reviewing. The checklist could include following points:

- Before checking in, please go through the source code and other created artifacts one more time:
    - Is the desired functionality or change implemented correctly?

- Is the source code sufficiently documented?
- Does the source code follow the coding style?
- Were unit and integration tests created for new functionality? Were existing unit and integration tests adjusted for the changed source code?
- If new mathematical requirements were implemented, were variability models and system test applications created/adjusted?

In a DUNE case study, the developers discussed a suitable desk-checking checklist for the DUNE development. The most important point for the developers was to check that the source code is sufficiently documented and that suitable tests have been created for the source code. More details to the case study can be found in Section 6.

If appropriate, the developer can ask a colleague to review her or his changes as well. The development team could also name developers responsible for different software modules who review the changed source code on a regular basis [44]. We do not demand any structured inspection or review process, as the goal is to keep the QA process practical and simple (C7).

5.3.3 Unit and Integration Testing

In this step, the unit and integration tests (see Section 2.1.4) prepared in the planning step are executed. A developer can execute the unit tests manually, but they also run automatically every night in a regression test environment (see Section 5.4).

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. The communication between software units working closely together is tested with integration testing. There are numerous tools available for unit testing, depending on the used development language. For C++, for example, one could use CppUnit[2] or googletest[3].

In some contexts of scientific software, where system tests can only run on a high performance computer (C5), the importance of unit testing gets very high, since the unit tests do not need a long time to run and still have high test coverage [53]. In a similar

[2] http://freedesktop.org/wiki/Software/cppunit/

[3] http://code.google.com/p/googletest/

way, the problem with a missing test oracle for system tests (C2) can be alleviated by comprehensive unit testing.

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.

5.3.4 System Testing

Similar to unit and integration tests, in this step the system test applications (see Section 2.1.4) prepared in the planning step are executed.

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. All expected output values include a manually adjustable tolerance range for taking rounding errors into account (C6) (see also Section 2.3).

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

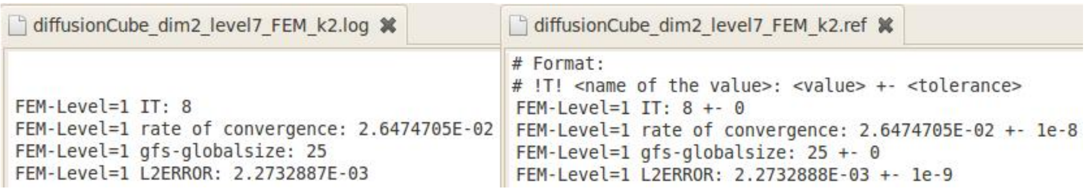
Supporting algorithm verification and testing on different platforms and with different configurations (e.g. count of processors, compiler options) is significant for assuring the important quality goals correctness (C9) and portability (C11). System testing is also the suitable step for executing performance testing (C10).

Similar to unit and integration 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.

For DUNE, the system test applications are executed in an automated system test environment introduced in detail in Chapter 7.

Example: Algorithm Verification in the DUNE Test Environment

In the DUNE test environment, the output for the algorithm verification consists of a free text description of the mathematical quantity and the according value followed by a colon. An example of a test application output is shown in the left part of Figure 5.2.


diffusionCube_dim2_level7_FEM_k2.log
diffusionCube_dim2_level7_FEM_k2.ref

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

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

Figure 5.2: Example of an algorithm verification output and the according expected values for one test case.

For the example above, the expected values can be found in the right part of Figure 5.2. Since numerical calculations are carried out using floating point arithmetics and thus will include rounding errors, the test environment cannot expect an exact value for each algorithm verification test value. The tolerance range for each expected value is determined due to the scientist's expert knowledge.

5.3.5 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).

The outcome of the system test applications includes graphical simulation output files (see Section 4.2). 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).

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 a framework (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 developers can form their opinion about the result through examination of the comparison and the graphical output.

5.4 Automated Regression Testing

In contrast to RiPLE-TE, we integrate regression testing in VAF-Pro (illustrated in Figure 5.1 with a dashed arrow). If every developer creates suitable unit, integration and system tests for their own source code in the planning step, the regression test environment (see Section 2.1.5) 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. The test environment is introduced in detail in Section 7.

5.5 Reporting

Reporting the results of the QA 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 other things, the information, which source code version and which configuration was used for the test.

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

5.6 Additional Remarks

VAF-Pro, the QA process for scientific frameworks we introduced in this section takes the special characteristics of scientific software introduced in Section 5.1 into account. The process is straightforward and the only software engineering methods not known by most of the scientists in the DUNE team in advance were the creation of variability models and desk-checking. These were introduced to the developers during the case study (see Section 6).

The accomplishment of the important quality goals correctness, portability, and maintainability is already tested by VAF-Pro. In 2013 there was a Google Summer School project with the goal of integrating performance testing to the DUNE QA process[4]. The resulting tool Dune-perftest runs performance measurements for DUNE applications (e.g. compile and run time).

In contrast to 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.

5.7 Chapter Summary

In this chapter we present the design of VAF-Pro, an overall QA process for scientific frameworks.

In a manual literature review, we collected *characteristics of scientific software development* that need to be taken into account for the design of a QA process for a scientific framework. First of all, different possible sources for a software failure and the lack of test oracles need to be considered. Most software requirements are not known at the beginning of a software project. The cognitive complexity of scientific software is high. For the calculations special utilities like high performance computing are often needed. Furthermore, the calculations include rounding errors and machine accuracy. Most of the developers are domain scientists and not software engineers and there is a high turnover in the development team. These characteristics are used as rationale for the design of the QA process.

In VAF-Pro, we have three *test roles*: developer, user and test administrator. The scientists developing the source code are responsible for creating suitable tests for his or her own code. Some activities may be better performed in a team of developers. The test administrator is responsible for keeping the regression test environment running.

The QA process has three *angles of view*: the different testing levels (unit, integration, and system testing), different goals in V&V of scientific software (code verification,

[4] http://www.dune-project.org/gsoc/2013/index.html

algorithm verification, and scientific validation), and regression testing. Code verification is carried out in the review and unit and integration testing steps of the QA process. Algorithm verification is integrated in the system testing step.

The first step in the QA process, *planning*, is performed during the software development. Each developer is responsible for creating new unit tests or adjusting existing unit tests whenever appropriate. If the mathematical requirements for the framework change, the developer may also need to formulate one or more new variability models together with the associated system test applications.

The *review* step is conducted right after changing any source code, unit and integration tests, variability models or system test applications. Taking time to go through the changes before checking them in, also called *desk-checking*, can reveal failures even before testing. At the same time, the developer can review the source code's readability and structure. A desk-checking checklist reminds the developers on all important aspects of reviewing, like to check that the source code is sufficiently documented and that suitable tests have been created.

In the *unit and integration testing* step, the tests prepared in the planning step are executed. The tests can be executed manually or run automatically in a regression testing environment.

Similarly, the prepared system tests are executed in the *system testing* step. The system tests also run in the automated regression testing environment. For algorithm verification, the outputs of system test applications include some significant mathematical quantities. The expected values for the mathematical quantities are determined analytically. If this is not possible, the scientists set up expected values from a validated run of the system test application.

The outcomes of the system test applications include graphical simulation output for the *scientific validation* step. The developers can examine a comparison of the graphical output with expected values to build their opinion on whether the simulation result is what the expected or not.

Part III

Evaluation and Practical Application

Chapter 6

Case Study DUNE

In this Chapter, we report on a case study (see Section 2.1.10) based on the method described by Runeson et al. [68] analyzing the feasibility and acceptance by DUNE developers (see Section 2.6.1) for those parts of the design of the VAF-Pro QA process introduced in Chapter 5 that have not yet been familiar to the DUNE developers: variability model creation (using the orthogonal variability model by Pohl et al. presented in Section 2.4.2.1) and desk-checking.

The use of variability modeling for the systematic creation of system tests is a new method in scientific software development, which makes case study results interesting for the computational science and engineering community in general. Desk-checking on the other hand, is a relatively well-known technique and has already been mentioned in software engineering for CSE literature [44], but no experimental results on the feasibility and acceptability of the method in the context of scientific software were available so far.

The main results of the case study are that variability modeling is feasible for the development team, but we needed to find a different variability modeling language to be able to represent all important aspects. The acceptance of variability modeling was positive. Desk-checking was also found feasible and was clearly accepted.

In Section 6.1 we introduce the case study design. After that, Section 6.2 presents the results of the case study. Section 6.3 includes a discussion of these results and the threats to validity for the case study are examined in Section 6.4. This is followed by the discussion of related work in Section 6.5. Section 6.6 provides a summary of the case study.

6.1 Case Study Design

In this section we introduce the case study design in detail. This includes the defined research questions and used research methods.

We designed and conducted the case study according to instructions by Runeson et al. in [68]. The objective of the case study was to analyze the feasibility and the acceptance of variability model creation and desk-checking as part of the VAF-Pro QA process. This objective was chosen as these aspects had not yet been familiar to the DUNE developers. The goal was to find out, if DUNE developers are satisfied with the design of the VAF-Pro QA process and if they think it is useful for them. The advantage of analyzing the process before it was established completely is that we still had the possibility to adjust the design according to the case study results without much overhead.

The conducted case study can be categorized as an *embedded single-case study with two units of analysis*: variability modeling and desk-checking. The design of the case study may be used for possible replications with other teams developing a scientific framework. The design is based on a *case study protocol* as described by Runeson. The contents of the case study protocol are introduced in the following sections.

The case study was executed with a group of six DUNE developers. They all work in the same academic group and developed DUNE for 2.5 - 3.5 years, with one developer having worked in the group even for 10 years. Four of the developers are mathematicians and two of even them are computer scientists.

6.1.1 Research Questions

According to the Goal Question Metric approach (GQM) of Basili et al. [6], in order to measure in a purposeful way, we first need to specify our goals and then define how we intend to collect and interpret data with respect to the stated goals.

These were our goals for the case study:

- Goal 1: Assess the feasibility of variability modeling in the QA from the developer's viewpoint.
- Goal 2: Assess the feasibility of desk-checking in the QA from the developer's viewpoint.

- Goal 3: Assess the acceptance of variability modeling in the QA from the developer's viewpoint.
- Goal 4: Assess the acceptance of desk-checking in the QA from the developer's viewpoint.

The research questions formulated according these goals are presented in the following subsections.

6.1.1.1 Feasibility

For the assessment of the feasibility, we reflected on possible advantages and disadvantages of variability modeling and desk-checking. Then we formulated research questions that would check if these assumptions apply. Tables 6.1 and 6.2 contain our research questions and the associated hypotheses. Each research question is followed by the data sources that are used to collect the data.

First, we want to observe how the developers perform variability modeling (F_RQ_VM1) (respectively desk-checking (F_RQ_DC1)). During this step in the case study, we also observe possible discussions or comments about advantages or disadvantages of the methods (F_RQ_VM2 and F_RQ_VM3 for variability modeling and F_RQ_DC2 and F_RQ_DC3 for desk-checking). For variability modeling, we additionally want to observe, if the method can be used to capture the variability of mathematical problems (F_RQ_VM4). Afterwards, the developers are asked for their opinion on the methods using a questionnaire. The first part of the questionnaire only includes open questions about the advantages and disadvantages of the methods to capture the developers' opinion without influencing their thoughts with predetermined options. In the second part of the questionnaire the developers are asked to give their opinion on possible predetermined advantages and disadvantages of the methods using closed questions. The entire questionnaire can be found in Appendix A.

Research Question	**Hypothesis**	**Data Source**
F_RQ_VM1: How do developers perform variability modeling?		Observation
F_RQ_VM2: What do the developers believe are the advantages of variability modeling for the DUNE development?	The advantages of variability modeling include: a systematic way to model different possibilities to solve a mathematical problem, a support for system test program development.	Observation, open and closed questionnaire questions
F_RQ_VM3: What do the developers believe are the disadvantages of variability modeling for the DUNE development?	Variability model creation is a complex task and requires deep domain knowledge.	Observation, open and closed questionnaire questions
F_RQ_VM4: Can variability modeling be used to capture the variability of mathematical problems solved by the framework?	Yes, variability modeling can be used to capture the variability of mathematical problems.	Observation

Table 6.1: Feasibility of Variability Modeling

Research Question	Hypothesis	Data Source
F_RQ_DC1: How do developers perform desk-checking?		Observation
F_RQ_DC2: What do the developers believe are the advantages of desk-check for the DUNE development?	The advantages of desk-checking include: finding software failures even before testing, a reminder of creating tests and documentation, increase in software quality, in particular readability and maintainability.	Observation, open and closed questionnaire questions
F_RQ_DC3: What do the developers believe are the disadvantages of desk-check for the DUNE development?	Desk-checking leads to minor overhead.	Observation, open and closed questionnaire questions

Table 6.2: Feasibility of Desk-Checking

In addition to the research questions about feasibility, we wanted to find out if the DUNE developers think that the advantages of these methods outbalance the effort required. They could proclaim their opinion on a scale from "strongly agree" to "strongly disagree" with closed questionnaire questions. The according research questions are in Table 6.3.

Research Question	Hypothesis	Data Source
E_RQ_VM: Do the advantages of a variability model outbalance the effort of creating it?	The advantages of a variability model outbalance the effort of creating it.	Closed questionnaire questions
E_RQ_DC: Do the advantages of desk-checking outbalance the effort needed for it?	The advantages of a desk-checking outbalance the effort for it.	Closed questionnaire questions

Table 6.3: Advantages Versus Effort

6.1.1.2 Acceptance

The acceptance part of our case study is based upon the Technology Acceptance Model (TAM) [26]. This method was actually developed for software systems, but in our case we use it for software engineering methods. Davis et al. found out, that during a one-hour hands-on introduction, people form a perception of a system's (method's) usefulness that is strongly linked to their usage intention. Furthermore, the intention of use is significantly correlated with the future acceptance of the system (method). According to TAM, perceived usefulness and perceived ease of use are of primary relevance for acceptance behavior.

Strictly following the instructions of TAM, we ask the developers for their opinion on the usefulness, ease of use and intention of use of the methods in closed questionnaire questions. The corresponding research questions for acceptance are listed in Table 6.4.

Research Question	Hypothesis	Data Source
A_RQ_VM/DC1: Do the developers think variability modeling/desk-checking is useful for them?	The developers find variability modeling/desk-checking useful for them.	Closed questionnaire questions
A_RQ_VM/DC2: Do the developers think variability modeling/desk-checking is easy to use?	The developers think variability modeling/desk-checking is easy to use.	Closed questionnaire questions
A_RQ_VM/DC3: Do the developers intend to use the variability modeling/desk-checking in DUNE development?	The developers intend to use variability modeling/desk-checking in DUNE development.	Closed questionnaire questions

Table 6.4: Acceptance of Variability Modeling/Desk-Checking

6.1.2 Research Methods

For the case study, we had a two hour meeting with the DUNE developers with the following agenda:

- Introduction to variability modeling (10 minutes)
- Hands-on example together with the researcher (10 minutes)
- Task 1: Creating a variability model (40 minutes)
- Discussion about variability modeling (25 minutes)
- Break (5 minutes)
- Introduction to desk-checking (10 minutes)
- Task 2: Adjustments on desk-checking checklist (15 minutes)
- Discussion about desk-checking (10 minutes)

In addition to the six DUNE developers, one researcher (the author of this thesis) took part in the meeting and moderated it. An external researcher attended the meeting for validity reasons. Her function was to make observation notes and to control the recording of the discussions. The DUNE developers performed the tasks together in a group. The researcher did not take part in the tasks, but questions of comprehension to the researcher were allowed. The case study meeting was recorded, transcribed and coded for analysis purpose.

Task 1 was to model different possibilities of how a grid can be defined (for details on grid definition, see [65]). The proposed approach was to first write down possible variation points on a flip chart and then go on with the variability modeling (using the orthogonal variability model by Pohl et al. presented in Section 2.4.2.1) on a poster board.

Task 2 was to adjust a proposed desk-checking checklist illustrated in Figure 6.1 for the needs of DUNE development. Which items are suitable and which are not? Which items are missing? The developers did not try out desk-checking directly since it was not possible to simulate a realistic application of desk-checking. The proposed desk-checking checklist was designed by the researcher based on the VAF-Pro QA process and knowledge about used software engineering methods in DUNE development. Reading through the source code and answering the first question ("Is the desired functionality or change

- Before checking in, please read through the source code one more time:
  - Is the desired functionality or change implemented correctly?
  - Is the source code sufficiently documented?
  - Does the source code follow the coding style?
  - Were unit tests created for new functionality? Were existing unit tests adjusted for the changed source code?
  - If new mathematical requirements were implemented, were system tests created?

Figure 6.1: Proposed Desk-Checking Checklist

implemented correctly?") is the main idea in desk-checking method as introduced by Kelly and Sanders in [44], which is our template for the method.

We used different data sources for the case study: observation, questionnaire, and discussion. Table 6.5 explains which kind of data was collected for each research question during the case study and how the data was aggregated. During observation and discussions, data was collected by the researcher using subjected notes and a recording. The recording was transcribed and coded afterwards. For the case study results, the notes and the coded transcription were summarized. Answers to open questionnaire questions were summarized by counting similar answers. Results to closed questionnaire questions were summarized using standard statistical means (Likert scale and median). The used questionnaire can be found in Appendix A.

Research Question	**Data Source**	**Metrics**	**Aggregation**
F_RQ_VM/DC1: Perform[1]	Observation	Subjective notes by the researcher	Summary of the notes
F_RQ_VM/DC2/ 3: Advantages/ disadvantages	OQ: What do you think are the benefits/disadvantages of VM/DC?	Subjective opinion of the DUNE developers	Sum-up[2]similar answers
	OQ: What did you (not) like about VM/DC?	Subjective opinion of the DUNE developers	Sum-up similar answers
	CQ: VM/DC will help me to develop source code in a higher quality.	Likert scale[3]	Median[4]

	CQ: I think VM helps in the development of system test applications.	Likert scale	Median
	CQ: DC leads to a higher rate of found failures.	Likert scale	Median
	CQ: DC leads to a better maintainability.	Likert scale	Median
	CQ: DC leads to a better readability.	Likert scale	Median
	Discussion	Subjective notes by the researcher	
F_RQ_VM4: Capture variability	Observation	Subjective notes by the researcher	
	Discussion	Subjective notes by the researcher	
E_RQ_VM/DC: Effort	CQ: the advantages of VM/ DC outbalance the effort.	Likert scale	Median
A_RQ_VM/DC1: Useful	CQ: Using VM/DC as a quality assurance method is important to me	Likert scale	Median
	CQ: VM/DC will help me to develop source code in a higher quality.	Likert scale	Median
A_RQ_VM/DC2: Easy to use	CQ: VM/DC is easy to learn	Likert scale	Median
	CQ: VM/DC is easy to use in practice	Likert scale	Median
A_RQ_VM/DC3: Intention of use	CQ: I intend to apply VM/ DC	Likert scale	Median

Table 6.5: Metrics and Data Aggregation

[1] Legend: VM = Variability Modeling, DC = Desk-Checking, OQ = Open questionnaire Question, CQ = Closed questionnaire Question

[2] With "sum-up" we mean that we count how many times similar answers regarding the content were given

[3] Likert Scale used: strongly agree, agree, rather agree, rather disagree, disagree, strongly disagree

[4] If median is between two values, choose the side with the higher dispersal in the answers

6.2 Results

In this section we report the results of the case study. For each research question, the data from different sources was subjectively summed up as explained in the subsections. The codes in parentheses indicate the handled research question listed in Tables 6.1, 6.2, 6.3, and 6.4. The results are discussed further in Section 6.3.

For each research question, we also provide tables with detailed answers to each open and close questionnaire question. As described in Table 6.5, for summarizing the answers for a closed questionnaire question we use a median. If the median is between two values, we choose the side with the higher dispersal in the answers.

6.2.1 Variability Modeling by Developers (F_RQ_VM1)

When the developers were working on their variability model, the working atmosphere was very open and everybody took part in the discussion about variability model details. The developers were motivated to learn the method and many questions of comprehension were asked.

First, the developers collected possible variation points on a flip chart. Every proposal was thoroughly discussed right away and at the end accepted or rejected by the group. Possible variants for the variation points were listed instantly for each variation point. Table 6.6 presents the notes made by the developers.

Before drawing the variability model, the developers thought about possible dependencies between the variation points and their variants. They wanted to draw the variation points with their dependencies close to each other. The variability model drawn by the developers is presented in Figure 6.2.

The developers thought of the model with *"levels"* or a *"hierarchy"*, although the proposed variability model does not have any levels. They considered which variation point should be put on the top, meaning which variation point is most essential for the variability model. They wanted to put variation points which are built in a similar way "on the same level" (e.g. variation points Parallel and Adaptivity in Figure 6.2). This seems to be an intuitive way of thinking of a variability model.

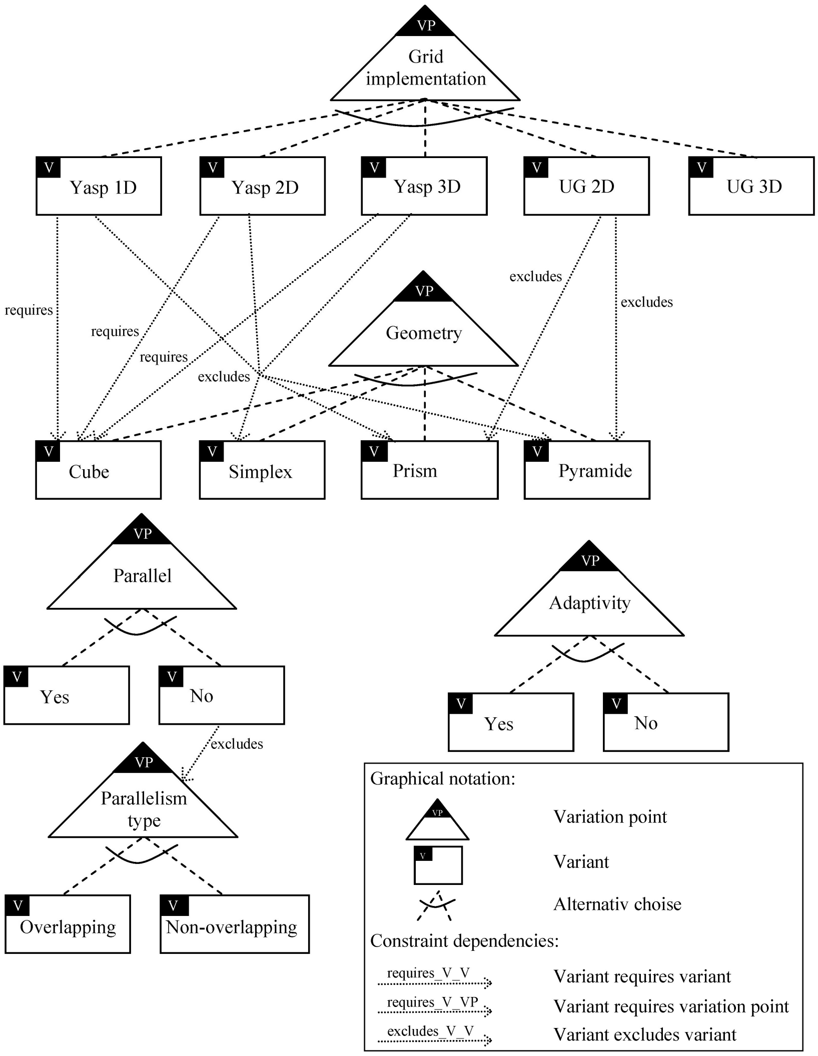


Figure 6.2: Variability model drawn by the developers

Dimension:	1,2,3
Geometry:	cube(d), simplex(d), pyramide(d), prism(d)
Parallel:	yes, no
Parallelism type:	overlapping, non-overlapping
Adaptivity:	yes, no
Closure type:	conform, non-conform
Refinement factor:	2, 2^d

Table 6.6: Possible variation points collected by the developers

In DUNE, there is a *technical constraint* when it comes to defining a grid: there are a handful of grid implementations a DUNE user can choose from when implementing a DUNE application. Each grid implementation has its own possible characteristics and constraints. It is only possible to use a grid with characteristics that suit to at least one of these grid implementations. The used grid implementation is actually not a characteristic of a grid, but technically highly essential. This is why the developers decided to select grid implementation as a central variation point. For keeping the variability model simple at first, they chose only two possible variants (Yasp[5] and UG[6], see Figure 6.2) for the grid implementation.

6.2.2 Advantages of Variability Modeling for the DUNE Development (F_RQ_VM2 and E_RQ_VM)

The following list collects the most often mentioned advantages that DUNE developers see in variability modeling. The numbers in parentheses indicate how many times each statement was mentioned in the open questionnaire questions about the advantages:

- Variability modeling offers a *systematic way* to cope with all different possible combinations of features and their dependencies (5)
- The process of variability modeling leads to a *deeper reflection* about the set of needed variants, concrete dependencies between the concepts in the software, or scope and goal of a test case (5)
- Variability modeling is the first step in the *automatic generation of test cases* for DUNE: every valid combination of variants is a test case (2)

The summarized results of closed questionnaire questions reveal that the developers

- agree that variability modeling would be helpful in developing system test applications,
- rather agree that variability modeling would help to develop DUNE source code in a higher quality, and

[5] http://www.dune-project.org/doc/doxygen/dune-grid-html/group_____yasp__grid.html

[6] http://atlas.gcsc.uni-frankfurt.de/˜ug/

Question	**Answer**	**Count**
What do you think are the benefits of VM?	Variability modeling offers a systematic way to cope with all different possible combinations of features and their dependencies.	5
	Variability modeling is the first step in the automatic generation of test cases for DUNE: every valid combination of variants is a test case.	2
	The process leads to a reflection about a detailed plan for test cases for a system test application.	1
What did you like about VM?	The process of variability modeling leads to a deeper reflection about the set of needed variants, concrete dependencies between the concepts in the software, or scope and goal of a test case.	5

Table 6.7: Detailed answers to open questionnaire questions to research question F_RQ_VM2: What do the developers believe are the advantages of variability modeling for the DUNE development

- rather agree that the advantages of a variability model outbalance the effort for creating it (E_RQ_VM).

Tables 6.7, 6.8, and 6.9 include detailed answers for each questionnaire question for the research questions F_RQ_VM2 and E_RQ_VM.

Question	Strongly agree	Agree	Rather agree	Rather disagree	Disagree	Strongly disagree	Median
VM will help me to develop source code in a higher quality.	1		3	1	1		Rather agree
I think VM helps in the development of system test applications.	2	2	2				Agree

Table 6.8: Detailed answers to closed questionnaire questions to research question F_RQ_VM2: What do the developers believe are the advantages of variability modeling for the DUNE development

Question	Strongly agree	Agree	Rather agree	Rather disagree	Disagree	Strongly disagree	Median
The advantages of VM outbalance the effort.		2	3		1		Rather agree

Table 6.9: Detailed answers to closed questionnaire questions to research question E_RQ_VM: Do the advantages of a variability model outbalance the effort of creating it

6.2.3 Disadvantages of Variability Modeling for the DUNE Development (F_RQ_VM3)

The disadvantages some developers see in variability modeling in general are:

- The creation of a variability modeling is costly because of the complexity of the mathematical problems (1)

- It will be difficult to implement the automatic creation of test cases, since each set of variants must be implemented in a different way (1)

These are the disadvantages the developers see in the proposed variability model:

- The modeling language is *not able to represent* some important aspects. Some dependencies are more complex than can be modeled: e.g. in some cases one variant should be excluded, if a combination of two other variants is chosen. To model this kind of situation, the developers combined two variation points, grid implementation and grid dimension (see Figure 6.2). This solution did not satisfy the developers (3).
- The presentation becomes *complex* easily and therefore unclear, unreadable, and hard to maintain. Many lines (dependencies) make the model unclear (5).
- The developers miss the possibility to define a *hierarchy* between variation points (4)

Table 6.10 includes detailed answers for each questionnaire question for the research question F_RQ_VM3.

6.2.4 Capturing the Variability of Mathematical Problems with Variability Modeling (F_RQ_VM4)

While working on the variability model, the developers did not come to an agreement about how *detailed* the model should be. Some developers repeatedly came up with special cases and other developers argued that these cases are not really relevant. One developer brought up that the variability model only needs to be as detailed as one wants to define the different test cases for a system test application. Some minor features could be implemented as arbitrary parameters without being part of the variability model.

All developers agreed that the *point of view* has a major influence on the created variability model. If one creates a variability model for a specific system test application, it will be different from a variability model for a general case. Many developers agreed that it makes sense to choose a "test application" point of view, since the variability model would be more precise and less complex.

Question	**Answer**	**Count**
What do you think are the disadvantages of VM?	The modeling language is not able to represent some important aspects.	3
	The developers miss the possibility to define a hierarchy between variation points.	4
	The modeling language should be closer related to the modeling problem.	1
	The creation of a variability modeling is costly because of the complexity of the mathematical problems.	1
	It will be difficult to implement the automatic creation of test cases, since each set of variants must be implemented in a different way.	1
What did you not like about VM?	The presentation becomes complex easily and therefore unclear, unreadable, and hard to maintain. Many lines (dependencies) make the model unclear.	5
	The exact goal of the modeling has to be clear first.	1

Table 6.10: Detailed answers to open questionnaire questions to research question F_RQ_VM3: What do the developers believe are the disadvantages of variability modeling for the DUNE development

6.2.5 Acceptance of Variability Modeling (A_RQ_VM1-3)

The developers answered in the questionnaire that they

- rather agree that variability modeling is useful for the DUNE development.
- rather agree that variability modeling is easy to learn and use.
- rather agree that they intend to use variability modeling for the DUNE development.

This means that variability modeling is rather accepted by the DUNE developers.

Tables 6.11, 6.12, and 6.13 include detailed answers for each questionnaire question for the research questions A_RQ_VM1-3.

Question	**Strongly agree**	**Agree**	**Rather agree**	**Rather disagree**	**Disagree**	**Strongly disagree**	**Median**
Using VM as a quality assurance method is important to me.	1	1	3		1		Rather agree
VM will help me to develop source code in a higher quality.	1		3	1	1		Rather agree

Table 6.11: Detailed answers to closed questionnaire questions to research question A_RQ_VM1: Do the developers think variability modeling is useful for them

Question	**Strongly agree**	**Agree**	**Rather agree**	**Rather disagree**	**Disagree**	**Strongly disagree**	**Median**
VM is easy to learn.		3	2	1			Rather agree
VM is easy to use in practice.		1	2	3			Rather agree

Table 6.12: Detailed answers to closed questionnaire questions to research question A_RQ_VM2: Do the developers think variability modeling is easy to use

Question	**Strongly agree**	**Agree**	**Rather agree**	**Rather disagree**	**Disagree**	**Strongly disagree**	**Median**
I intend to use VM when developing code for DUNE.		1	2	1	1		Rather agree

Table 6.13: Detailed answers to closed questionnaire questions to research question A_RQ_VM3: Do the developers intend to use the variability modeling in DUNE development

6.2.6 Desk-Checking by Developers (F_RQ_DC1)

The developers found many items in the desk-checking checklist important for the DUNE development. One developer noted that the items in the checklist are better suitable for the development in the DUNE *base classes* instead of source code for solutions of special mathematical problems. Another developer motivated the others to take time later for adjusting the checklist in detail for the DUNE development. He thought such a checklist is a good reminder for each developer.

These are the items that the developers found important in the proposed checklist:

- *Sufficiently documented:* the developers distinguished between source code documentation and commit messages in the version control system. They thought both of these were important. The commit messages were distributed over a mailing list which means that other developers could review the changes.

- *Creating and extending tests:* two developers argued for the importance of suitable tests in particular for changes in the DUNE base classes. One developer reminded that tests for created or changed source code are particularly important for other developers so that they also can check the functionality.

- Two developers thought that checking whether the source code follows the *coding style* is also important.

There were also some ideas for additional checklist items:

- Two developers found it important to check the *naming of variables* for suitability before checking in the source code.

- Another point mentioned was that each developer should check that her or his source code is written *comprehensibly.*

The item in the checklist that three of the developers found redundant was whether the functionality was correctly implemented or not. They found that it is self-evident that only source code that works will be checked in. They did not see any advantage in reading the source code one more time to review this. They rather check the functionality through testing or taking a look at the output of the software.

6.2.7 Advantages of Desk-Checking for the DUNE Development (F_RQ_DC2 and E_RQ_DC)

The main advantage the developers see in desk-checking is *quality improvement*, in particular of the documentation (5). Other quality improvements they expect are:

- a more careful *check* that an implementation is correct (2).

- a better chance that *proper tests* are developed (2).

- better *readable* code that follows the coding style (2).

In the closed questionnaire questions, the developers answered that they

- agree that desk-checking helps to develop source code of higher quality.
- rather agree that desk-checking leads to a higher detection rate of software failures.
- agree that desk-checking leads to better maintainability of the source code.
- agree that desk-checking leads to better readability of the source code.
- agree that the advantages of desk-checking outbalance the effort required (E_RQ_DC).

Tables 6.14, 6.15, and 6.16 include detailed answers for each questionnaire question for the research questions F_RQ_DC2 and E_RQ_DC.

Question	**Answer**	**Count**
What do you think are the benefits of DC?	Quality improvement, in particular of the documentation.	5
	A more careful check that an implementation is correct.	2
	A better chance that proper tests are developed.	2
	A better readable code that follows the coding style.	2
	Following the desk-checking steps can save a lot of time and energy.	1
What did you like about DC?	When using desk-checking, many failures can be avoided, the documentation won't be missing, and simple test problems are created.	1
	Low overhead.	1
	It is good to reflect on which criteria should be used when checking the source code before committing it.	1
	In my opinion, desk-checking is a basic requirement for each commit. Especially a suitable documentation is extremely important and should be demanded.	1

Table 6.14: Detailed answers to open questionnaire questions to research question F_RQ_DC2: What do the developers believe are the advantages of desk-checking for the DUNE development

Question	Strongly agree	Agree	Rather agree	Rather disagree	Disagree	Strongly disagree	Median
DC will help me to develop source code in a higher quality.		4	2				Agree
DC leads to a higher rate of found failures.		3	1	2			Rather agree
DC leads to a better maintainability.	2	4					Agree
DC leads to a better readability.	1	5					Agree

Table 6.15: Detailed answers to closed questionnaire questions to research question F_RQ_DC2: What do the developers believe are the advantages of desk-checking for the DUNE development

Question	Strongly agree	Agree	Rather agree	Rather disagree	Disagree	Strongly disagree	Median
The advantages of DC outbalance the effort.	1	2	2				Agree

Table 6.16: Detailed answers to closed questionnaire questions to research question E_RQ_DC: Do the advantages of desk-checking outbalance the effort needed for it

6.2.8 Disadvantages of Desk-Checking for the DUNE Development (F_RQ_DC3)

The disadvantages that the developers saw in desk-checking are:

- An *overhead* before checking in source code, mainly because of the creation of new tests (4). One developer pointed out that if there is no time for creating tests at once, the developer should create an issue in the issue tracking system as a reminder that the tests are still missing.

- Most of the items in the checklist are *subjective.* Every developer has her or his own opinion on, e.g what is "sufficiently documented". Minimum requirements must be defined for each issue (2).

Table 6.17 includes detailed answers for each questionnaire question for the research question F_RQ_DC3.

Question	**Answer**	**Count**
What do you think are the disadvantages of DC?	An overhead before checking in source code, mainly because of the creation of new tests.	4
What did you not like about DC?	Most of the items in the checklist are subjective. Every developer has her or his own opinion on, e.g what is "sufficiently documented". Minimum requirements must be defined for each issue.	2
	Correct implementation cannot be validated only by looking at the source code, especially not for mathematical source code.	1

Table 6.17: Detailed answers to open questionnaire questions to research question F_RQ_DC3: What do the developers believe are the disadvantages of desk-checking for the DUNE development

6.2.9 Acceptance of Desk-Checking (A_RQ_DC1-3)

The developers answered in the questionnaire that they

- rather agree that desk-checking is important to them.

- agree that desk-checking is useful for the DUNE development.

- agree that desk-checking is easy to learn.
- rather agree that desk-checking is easy to use.
- agree that they intend to use desk-checking for the DUNE development.

The developers clearly find desk-checking acceptable, although they only rather agree that it is easy to learn or personally important to them.

Tables 6.18, 6.19, and 6.20 include detailed answers for each questionnaire question for the research questions A_RQ_DC1-3.

Question	**Strongly agree**	**Agree**	**Rather agree**	**Rather disagree**	**Disagree**	**Strongly disagree**	**Median**
Using DC as a quality assurance method is important to me.		3	2	1			Rather agree
DC will help me to develop source code in a higher quality.		4	2				Agree

Table 6.18: Detailed answers to closed questionnaire questions to research question A_RQ_DC1: Do the developers think desk-checking is useful for them

Question	**Strongly agree**	**Agree**	**Rather agree**	**Rather disagree**	**Disagree**	**Strongly disagree**	**Median**
DC is easy to learn.		5		1			Agree
DC is easy to use in practice.		3	1	2			Rather agree

Table 6.19: Detailed answers to closed questionnaire questions to research question A_RQ_DC2: Do the developers think desk-checking is easy to use

Question	Strongly agree	Agree	Rather agree	Rather disagree	Disagree	Strongly disagree	Median
I intend to use DC when developing code for DUNE.		4	2				Agree

Table 6.20: Detailed answers to closed questionnaire questions to research question A_RQ_DC3: Do the developers intend to use the desk-checking in DUNE development

6.3 Discussion

In this section we discuss the results in the light of the goals listed in Section 6.1.1.

6.3.1 Goal 1: Feasibility of Variability Modeling

DUNE developers recognized important advantages of variability modeling, like having a systematic way to model variability and support for the development of system test applications. A surprising result was that almost every developer brought up the positive effect of a deeper reflection about the variability in the examined concept. The disadvantages found were almost all associated to the presented variability modeling language, not to variability modeling in general.

The results of the case study reveal that variability modeling can be used to capture the variability of mathematical problems if the point of view is fixed first and the modeling task is clearly defined.

This means that variability modeling is feasible for the DUNE QA, but we needed to find a different variability modeling language, which is able to represent all important aspects and enables the definition of a hierarchy. The variability modeling language used in FeatureIDE, the practical application of our approach introduced in Chapter 7 is a feature-tree that satisfies these requirements.

6.3.2 Goal 2: Feasibility of Desk-Checking

The case study convinced the developers that desk-checking helps to develop source code of a higher quality. They could see many advantages in desk-checking including a better documentation, a better chance that proper tests are developed and better readability and maintainability of the source code. The main disadvantage they see, an overhead, is mainly associated to the creation of tests. In fact, this is not an overhead in desk-checking itself, but in creating the tests. However, they are willing to accept this overhead, as they see it as an advantage that desk-checking reminds them of creating the tests.

Since the developers rather declined the item "Is the desired functionality /change implemented correctly?", the desk-checking method the DUNE developers prefer is somewhat different than the desk-checking method introduced by Kelly and Sanders in [44].

The case study results indicate that desk-checking is feasible for the DUNE development. As a next step the developers should adjust the desk-checking checklist to better suit their needs and define the minimum requirements for each item in the checklist.

6.3.3 Goal 3: Acceptance of Variability Modeling

The acceptance of variability modeling for the DUNE development was positive. We could see that the acceptance has even increased after introducing FeatereIDE, the practical application of our approach, to the developers, since FeatureIDE includes a more suitable variability modeling language.

6.3.4 Goal 4: Acceptance of Desk-Checking

The case study results in a clear acceptance of desk-checking. The developers see that desk-checking is useful for the DUNE development and intend to use it.

6.4 Threats of Validity

In this section we analyze the validity of the case study and its results. We distinguish between different aspects of the validity as presented by Runeson et al. in [68].

6.4.1 Construct Validity

Construct validity reflects to what extent the used research methods really represent what is investigated according to the research questions [68].

We used different methods to increase the construct validity of our case study. To achieve a methodological triangulation, we combined different types of data collection methods: observation, questionnaire, and discussion. The case study design includes a chain of evidence on how the data of the different data sources are used to answer the research questions. Observer triangulation was achieved by an external observer during the case study meeting.

The questions in the questionnaire were checked for understandability by several researchers. During the case study external influence on the developers was kept to a minimum. The moderator did not mention any advantages or disadvantages of the methods. The developers always answered the open questions about the advantages and disadvantages first before reading the closed question that mentioned some possible advantages. During the case study we found out that it was not clear to all developers if they should report on their opinion on variability modeling in general or on the proposed variability modeling language. Some problems with this specific variability modeling language may have influenced the results on the variability modeling part of the case study negatively.

The researcher who moderated the case study has been working with the DUNE developers regularly over the last years. This means there is a trustful relationship between the researcher and the developers. This is called a "prolonged involvement" by Runeson et al, [68]. One positive effect of this is that the researcher is able to understand how the developers interpret the terms that are used in the study.

As a further step to increase the construct validity, the results of the case study were sent to the participating DUNE developers. They confirmed that the results reflect their opinion correctly.

6.4.2 External Validity

The analysis of external validity seeks to find out to what extent the findings are of relevance for other cases [68].

Since variability modeling and desk-checking were found feasible and acceptable in this case study, this indicates that the methods could be found feasible and acceptable also for other cases in the context of scientific software development. Further examination is necessary to confirm this.

6.4.3 Reliability

The reliability aspect of validity relates to the extent the data and the analysis are dependent on a specific researcher [68].

During the design, data collection, and analysis of the case study, the researcher continuously documented every single step that was done. Each step was peer reviewed by a second researcher. Furthermore, the case study design was reviewed by an external researcher. This means there is a reproducible chain of evidence for the case study.

6.5 Related Work

In this subsection, we consider other empirical studies on QA processes for SPLE and variability modeling. We could not find any empirical studies for QA processes for scientific software in the literature. Independently of the domain, we also could not find any empical studies about desk-checking.

The unit testing part of RiPLE-TE, the QA process VAF-Pro is based on, was initially evaluated in an experimental study by Machado et al. [51]. The goal was to analyze the effectiveness of unit testing in this process and to find out which professional skills have impact on the test activity results. In the experiment, 30 undergraduate students tested the same source code with and without the process. The authors admitted that the results of this experiment were not very significant. This initial experiment serves as a baseline for future experiments.

Neto et al. [56] propose a very formal regression testing approach for the reference architecture of an 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 to system testing. The approach was evaluated in order to calibrate and improve it. Eight postgraduate students applied the approach in an experimental scenario. The approach showed its efficiency although it was not experimented in a real

SPL context.

One major advantage in our case study compared to the case studies mentioned above was that we could conduct the case study with developers of scientific software who are the actual target audience instead of under- or postgraduate students. This makes the results more significant.

We could find several studies comparing different variability modeling approaches ([22], [25], [39]), but these studies only describe technical issues like main concerns of the variability model, hierarchy, dependencies and constraints, tool support, or evolvement of the variability modeling approach.

In a survey of variability modeling in industrial practice, Berger et al. [13] asked the participants for their attitute towards variability modeling. Most of them found it either definitely useful (55%) or useful (35%). The main benefits the participants saw in variability modeling are the management of existing variability (77%) and product configuration (71%). 20% mentioned QA and testing. The most challenging tasks in variability modeling for the participants were visualization of models (59%), dependency management (59%) and model evolution (56%). Although the participants in our case study did not have any previous practical experience in variability modeling, our results are consistent with those in the survey of Berger et al.

6.6 Chapter Summary

This chapter reports on a case study analyzing the feasibility and acceptance by DUNE developers for two parts of the design of the VAF-Pro QA process: variability model creation and desk-checking.

The *case study design* is based on instructions by Runeson et al. in [68]. The goal was to find out, if DUNE developers are satisfied with the design of the VAF-Pro QA process and if they think it is useful for them. This way we still had the possibility to adjust the design before establishing the QA process.

Corresponding to the GQM approach, the *goals* for the case study are to assess the feasibility (acceptance) of variability modeling (desk-checking) in the QA from the developer's viewpoint. For the assessment of the *feasibility*, we first want to observe how the developers perform variability modeling (desk-checking). Afterwards, the developers are asked for their opinion on the methods using a questionnaire with two parts: first the developers are asked about advantages and disadvantages of the methods with open questions and then they are asked to give their opinion on possible predetermined advantages and disadvantages using closed questions. For the *acceptance* part of the case study we strictly follow the instructions of the TAM model and ask the developers for their opinion on the usefulness, ease of use and intention of use of the methods in closed questionnaire questions.

For the case study, we had a two hour meeting with six DUNE developers. The author of this thesis moderated the meeting and an external researcher attended for validity reasons. The DUNE developers performed the following two tasks together in a group: model different possibilities of how a grid can be defined and adjust a proposed desk-checking checklist for the needs of DUNE development.

DUNE developers found that variability modeling is a systematic way to model variability and that it supports the development of system test applications. The results of the case study reveal that variability modeling can be used to capture the variability of mathematical problems if the point of view and the exact modeling task are fixed first. Most of the found disadvantages were associated to the presented variability modeling language, not variability modeling in general. This means that variability modeling is feasible, but we needed to find a different variability modeling language. The acceptance of variability modeling was positive.

The developers were convinced that desk-checking helps to develop source code in a higher quality (e.g. with better documentation, a higher change to have proper tests, and better readability and maintainability of the source code). The main disadvantage, an overhead, is mainly associated to the creation of tests. This indicates that desk-checking is feasible for the DUNE development. The results also reveal a clear acceptance of desk-checking.

Chapter 7

System Testing with FeatureIDE and Automated Test Environment

This chapter introduces the practical implementation of the system testing part in the VAF-Pro QA process. We apply a tool called FeatureIDE[1] for the development of system test applications. For DUNE (see Section 2.6.1), the system tests are run in an automated test environment developed for DUNE.

Section 7.1 explains which requirements we had for tool support in system test development and introduces FeatureIDE and FeatureC++, a C++ language extension for feature-oriented programming, which fulfill these requirements. Section 7.2 then explaines how to develop system test applications using FeatureIDE and FeatureC++. Section 7.3 presents how the system test applications can be run in an automated test environment. Section 7.4 summarized the chapter.

7.1 Tool Support for the System Test Development

As we were planning the practical implementation of the system testing part of VAF-Pro QA process, we first looked for an existing tool that would fulfill our requirements. Such a tool should

- support variability modeling and the integration of the variability model in the

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

system test development.

- support several programming languages so that it would be applicable for several scientific frameworks.

- support the running of system test applications in an automated test environment.

- generate separate executables for the system test cases so that the system test applications could be executed independently and, if needed, using parallelism.

- The used variability model should fulfill the requirements defined by the developers in the case study (see Section 6.2.3).

- The tool should be continuously supported and in an ideal case have a large developer and user base.

We were able to find a tool that fulfills all our requirement: FeatureIDE. In this section we introduce FeatureIDE and its' possible applications in SPLE. As an example, we introduce how FeatureC++ that supports feature-oriented programming with C++ can be used within FeatureIDE.

7.1.1 FeatureIDE

FeatureIDE is an open-source Eclipse-based IDE that supports all phases of SPLE: domain analysis and implementation, requirements analysis, and software generation. It supports different SPL implementation techniques such as *feature-oriented programming*, *aspect-oriented programming*, *delta-oriented programming*, and *preprocessors* (for details about the different techniques see [76]).

FeatureIDE is developed since 2004, mainly at the University of Magdeburg in Germany. Currently, there are about 20 project members and contributors [32]. FeatureIDE is used in software engineering lectures at different universities all over the world [76].

This is a list of FeatureIDEs main features [32]:

- Full Eclipse Integration

- A graphical and text based variability model editor.

- A constraint editor with syntax and semantic checking.

- A configuration editor to create and edit configurations and with support for deriving valid configuration.

- Feature-oriented programming with AHEAD[2], FeatureC++[3] , and FeatureHouse[4] (with support for C, C#, Java, etc.)

- Aspect-oriented programming with AspectJ[5]

- Delta-oriented programming with DeltaJ[6]

- Support for different preprocessors (Colligens[7], TypeChef[8], etc.)

Example: FeatureIDE for DUNE framework For the DUNE framework, we apply FeatureIDE with FeatureC++, since DUNE is implemented in C++. This is why we use FeatureC++ as an example in this Chapter. According to Thüm et al. [76], the user interface for different implementation techniques is almost identical.

7.1.2 FeatureC++

FeatureC++ is a C++ language extension to support feature-oriented programming (FOP). FOP is an extension to object-oriented programming, where classes are decomposed into *feature modules* each implementing a certain feature. The feature modules can be composed to a program based on a given configuration [76].

FeatureC++ enables a programmer to express features in a modular way. FeatureC++ comes in form of a C++ preprocesser that transforms FeatureC++ code into native C++ code. FeatureC++ was developed at the Institute for Technical and Business Information Systems at the University at Magdeburg in Germany [31].

FeatureC++ uses *folders* to represent features. A hierarchy of features can be represented using a folder hierarchy. Classes implemented in plain C++ are distributed over multiple features. In contrast to plain C++, classes are completely implemented in header files. All header files with the same name represent the implementation of a particular feature of that class. There are two new keywords used in FeatureC++: **refines** is used to

[2] http://www.cs.utexas.edu/users/schwartz/ATS.html

[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/

[4] http://www.fosd.de/fh

[5] http://www.eclipse.org/aspectj/

[6] http://deltaj.sourceforge.net/

[7] https://sites.google.com/a/ic.ufal.br/colligens/

[8] http://ckaestne.github.io/TypeChef/

specify extension of an existing class and **super** is used to call an overridden method [31].

7.1.3 FeatureIDE and FeatureC++ example

Figure 7.1 describes, how the different parts of FeatureIDE and FeatureC++ relate to each others. A step by step instruction for system test application development with a concrete DUNE example can be found in Appendix C.

Depending on the variability model (see Section 2.4.2.2), FeatureC++ automatically creates a folder structure for the FeatureC++ source code. After the developers have written the source code and created a configuration file (called "HelloBeautifulWorld.equation" in this example), FeatureC++ automatically assembles the source code according to the feature configuration.

7.2 System Test Development with FeatureIDE

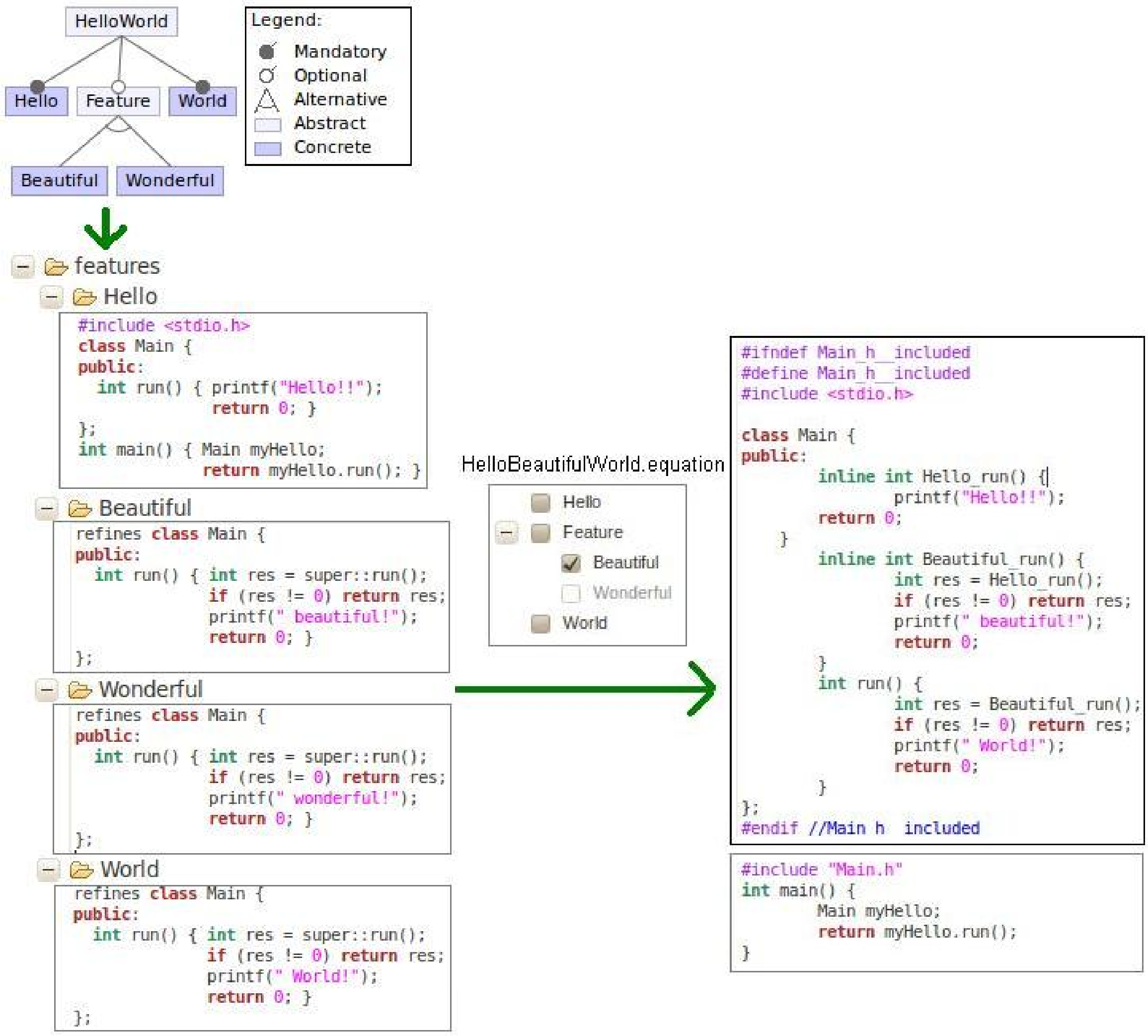
In the process of creating reengineering variability models and system test applications for a scientific framework (see Chapter 4), FeatureIDE and FeatureC++ can be applied as a supporting tool for the identification of features and their dependencies (Step 4.1.2.3), the identification of constraints between the features (Step 4.1.2.4), the derivation of test cases for a system test application (Section 4.1.3) and the development of system test applications (Section 4.2). Table 7.1 provides on overview of the support supplied by FeatureIDE and FeatureC++ for these steps.

Appendix C includes a detailed example of the use of FeatureIDE and FeatureC++.

7.2.1 Test Suite for a System Test Application

The set of configuration files, i.e. the test cases, build up a test suite for the system test application. In Section 4.1.3 we explained that often test suite selection methods are needed to reduce the test effort.

FeatureIDE has a functionality for building T-Wise configurations automatically. The functionality chooses, with a selected algorithm, a suitable subset of possible configurations for the test application. Unfortunately this functionality was not available in FeatureIDE for FeatureC++ in February 2014 when writing this thesis.



Hello:

```
#include <stdio.h>
class Main {
public:
  int run() { printf("Hello!!");
              return 0; }
};
int main() { Main myHello;
             return myHello.run(); }
```

Beautiful:

```
refines class Main {
public:
  int run() { int res = super::run();
              if (res != 0) return res;
              printf(" beautiful!");
              return 0; }
};
```

Wonderful:

```
refines class Main {
public:
  int run() { int res = super::run();
              if (res != 0) return res;
              printf(" wonderful!");
              return 0; }
};
```

World:

```
refines class Main {
public:
  int run() { int res = super::run();
              if (res != 0) return res;
              printf(" World!");
              return 0; }
};
```

HelloBeautifulWorld.equation:

- ☐ Hello
- ☐ Feature
  - ☑ Beautiful
  - ☐ Wonderful
- ☐ World

```
#ifndef Main_h__included
#define Main_h__included
#include <stdio.h>

class Main {
public:
        inline int Hello_run() {
                printf("Hello!!");
        return 0;
    }
        inline int Beautiful_run() {
                int res = Hello_run();
                if (res != 0) return res;
                printf(" beautiful!");
                return 0;
        }
        int run() {
                int res = Beautiful_run();
                if (res != 0) return res;
                printf(" World!");
                return 0;
        }
};
#endif //Main_h__included
```

```
#include "Main.h"
int main() {
        Main myHello;
        return myHello.run();
}
```

Figure 7.1: FeatureIDE and FeatureC++ example

7.3 Automated Test Environment

In the last section we explained how the system test applications can be developed with FeatureIDE. The part that is still missing is comparing the output for algorithm verification and scientific validation with previously defined reference values (see Sections 5.3.4 and 5.3.5). Furthermore, it is not practical to run each system test application

Process Step	**FeatureIDE and FeatureC++ support**
Variability Analysis, Step 2: Identifying Features and Their Dependencies	FeatureIDE supports the creation of a variability model with a graphical editor. The model is stored in an XML format and can also be edited textually simultaneously. Variability models can also be stored in several graphical formats and printed to a PDF file. Since the models tend to change with time, FeatureIDE supports refactoring variability models with a separate edit view [76]. An example of variability model creation in FeatureIDE can be found in Appendix C.4.
Variability Analysis, Step 3: Identifying Constraints Between the Features	FeatureIDE provides an editor for constraints between the features. The constraints can be expressed as logical formula over the set of existing features. An example of constraint definition in FeatureIDE can be found in C.5.
Deriving Test Cases for a System Test Application from the Variability Model	The alternative features in the variability model enable numerous possible feature combinations, i.e. different test cases for the system test application. To be able to compose one executable source code, one needs to choose a concrete set of features. In FeatureIDE, this is done by creating configuration files. See also Subsection 7.2.1. An example can be found in Appendix C.7.
Developing System Test Applications	The variability model and constraint configuration are used in feature-oriented programming with FeatureC++. Using the possibilities of feature-oriented programming the developers can systematically generate source code for different test cases for a system test application. An example of a DUNE system test application developed with FeatureIDE and FeatureC++ can be found in C.6.

Table 7.1: FeatureIDE and FeatureC++ support for the process of creating reengineering variability models and system test applications for a scientific framework

separately. What we need is an automated test environment that runs the system test applications automatically on a regular basis.

In this section we introduce one solution for an automated test environment that have been developed for DUNE. The environment does not just include the system tests, but also the unit and integration tests introduced in Section 5.3.3.

In the context of our research, the automated test environment for DUNE was extended by the system testing level. The foundations for this part, including the structure of the configuration files and the scripts running individual tests, were implemented by Felix Heimann from the DUNE development team for his own daily work. The original version of the script for the scientific validation was implemented by Jorrit Fahlke from the DUNE development team. Our main work was to adjust the test scripts for system testing and include them in the automated test environment.

Subsection 7.3.1 explains how the automated test environment works. In Subsection 7.3.2 we report on experiences we have made with the automated test environment during an implementation sprint in 2011. FeatureIDE was integrated to the system test development after this implementation sprint and we have not evaluated its' use in the daily development yet. First feedback from the DUNE developers after presenting the prototypical use of FeatureIDE is positive.

7.3.1 Running the Automated Test Environment

The automated test environment consists of scripts running the tests, files including the reference values for algorithm verification and scientific validation and a database where the results of test runs are saved. The result for the test runs are announced on an internet page.

A test run consists of the following steps:

- fetch the source code according to the test run configuration (e.g. current development version)
- compile the source code
- run unit and integration tests
- run system tests and compare the output for algorithm verification and scientific

validation with reference values

- insert the results to the database
- announce the results on the internet page

According to the configuration, the test environment runs automatically for example every night. It is also possible to run the environment manually.

For our research, we have included the system testing part to the automated test environment. For the system tests, FeatureC++ provides scripts that enable compiling the system test applications using the FeatureIDE configuration files outside the Eclipse environment. The system testing part of the automated test environment has the following steps:

- for each test case configuration of every system test application
    - compile the system test application using the test case configuration (FeatureIDE configuration file)
    - run the generated system test application
    - compare the output for algorithm verification and scientific validation with reference values

It is a great advantage for the DUNE developers that they can use the comfortable development environment of FeatureIDE for the system test development and then run the generated test cases in the automated test environment.

7.3.2 Experiences with the DUNE Automated Test Environment

During a DUNE implementation sprint of the dune-pdelab module in 2011 (FeatureIDE was not yet applied at that time) we applied the system testing part of the test environment to test the development changes and to evaluate how the test environment supports the scientists' work. The sprint took over one month and involved about ten DUNE developers. The changes consisted of over 400 commits with about 32000 new LOC and 19000 deleted LOC.

The implementation sprint included some major changes in the dune-pdelab module.

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

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

The unit tests were always run before the test application was executed. This way we observed that even though the single units were tested using unit tests, some faults could only be found through the system test environment. Using the test environment for such a major development sprint was challenging. Each test application had to be adjusted to the changed framework functionality which caused some extra work for the scientists. The test applications could only be changed after the development for a specific development change was completed. This sometimes led to a delay in testing these development changes.

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

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

7.4 Chapter Summary

This chapter introduces the prototypical implementation of the system testing part of VAF-Pro QA process. For the implementation, we were looking for a tool that should e.g. support variability modeling, several programming languages, and running the system test applications in an automated test environment. The used variability model should fulfill the requirements defined in the case study.

The tool that fulfills these requirements is *FeatureIDE*, an open-source Eclipse-based IDE supporting all phases of SPLE. It has graphical and text based editors for a variability model and its constraint and a configuration editor. Configuration files are used for the test case derivation. Furthermore, FeatureIDE supports different feature-oriented programming paradigm. *FeatureC++* is a C++ language extension to support feature-oriented programming. Depending on the variability model created in FeatureIDE, FeatureC++ automatically creates a folder structure for the feature-oriented source code.

FeatureIDE and FeatureC++ support the development of system test applications in the variability modeling, identification of constraints between the features, derivation of test cases, and the development of system test applications.

In the context of our research, an *automated test environment* developed for DUNE was extended by the system testing level. The test environment automatically compiles the system test applications based on the test case configuration defined in FeatureIDE, runs the tests on a regular basis and compares the output for algorithm verification and scientific validation with reference values.

The system testing level of the automated test environment was first applied during a DUNE implementation sprint in 2011. Even though the single software units were tested using unit tests, some faults could only be found through the system test environment. Using the test environment made the developers more confident about the good quality of the changed source code.

Part IV

Summary

Chapter 8

Conclusion and Future Work

This chapter concludes the thesis. A summary of the contributions is given in Section 8.1. Section 8.2 introduces some limitations to the contributions and Section 8.3 provides an overview of possible future work based on the contributions of our research.

8.1 Summary and Conclusion

In the QA of a scientific framework, we need to cope with special challenges of scientific software, as well as finding a way of dealing with the large variability of a framework.

In this thesis, we explain how SPLE supports the quality assurance of a scientific framework. We consider the framework as the product line platform. The applications developed by the users of the framework are the product line applications. The developers of a scientific framework only deal with domain engineering and, in particular, domain testing holds high importance for the developers. They need to test the functionality of the scientific framework without knowing exactly what kind of applications the users are going to develop.

In Chapter 3 we introduce VAF, a SPL test strategy for frameworks. In VAF, we create several variability models based on the mathematical requirements for the framework, i.e. the mathematical problems that the framework should solve. For each variability model, we develop as associated system test application that solves the mathematical problem presented with the variability model. The commonality, i.e. common characteristics for every application of the framework, is tested using unit and integration testing.

Chapter 4 concretizes VAF by introducing a process for creating reengineering variability models and system test applications for a scientific framework. In *reengineering product management*, we create product roadmaps that describe the scope and the goals of the scientific framework. The roadmaps determine major common and variable features of all possible applications for the framework. First, we define high level goals for the framework by describing mathematical problems and the approches for solving them with the framework. Second, we define a general mathematical model for each goal by describing the mathematical problem in detail. Third, we describe a general approach for solving the general mathematical problem. In *domain requirements engineering*, we establish a list of common requirements for all system test applications and create variability models that define the variable requirements for each system test application. For each variability model, we first choose a concrete mathematical model and subsequently identify its variable features, their dependencies and constraints by analyzing the previously created roadmap. We use the variability models for a model-based derivation of test cases for the system test applications. A framework's users can reuse the created artifacts for the development of their own applications.

For the implementation of the system test applications, we take into account the common requirements, the concrete mathematical model, and the variability model resulting from the aforementioned process. As described in Chapter 7, the tool FeatureIDE together with FeatureC++ support the system test application development with graphical editors for variability modeling and the identification of contraints. Furthermore, FeatureIDE supports the derivation of test cases for the system test applications.

Together with other QA activities, the test activities described above form VAF-Pro, the design of a QA process for scientific frameworks introduced in Chapter 5. In a manual literatur review, we collected the characteristics of scientific software development that were used as rationale for the design of the QA process. The QA process has three angles of view: the different testing levels (unit, integration, and system testing), different goals in V&V of scientific software (code verification, algorithm verification, and scientific validation), and regression testing. In the planning step, each developer creates unit and integration tests, and develops variability models with the associated system test applications whenever appropriate. In the review step, any changed development artifacts are reviewed by the developer based on a desk-checking checklist. Subsequently, unit, integration, and system tests are executed in a regression testing environment. The outcome of the system test applications include output for both algorithm verification and scientific validation.

Chapter 6 reports on a case study analyzing the feasibility and acceptance by DUNE developers for two parts of the VAF-Pro QA process design: variability model creation and desk-checking. In the case study, we observed how the developers perform variability modeling and desk-checking. Afterwards, a questionnaire ascertained developer's opinions on the advantages and disadvantages, usefulness, ease of use, and intention of use of the methods. The main results of the case study are that variability modeling is feasible for the development team, but we needed to find a different variability modeling language to represent all important aspects. The acceptance of variability modeling was positive. Desk-checking was also found feasible and was clearly accepted.

Chapter 7 introduces a prototypical implementation of the system testing part of VAF-Pro QA process using FeatureIDE and an automated test environment developed for DUNE. In the context of our research, the automated test environment was extended by the system testing level. The DUNE developers tried out the extension during an implementation sprint, after which they were more confident about the good quality of the changed source code.

8.2 Limitations

The process of creating reengineering variability models and system test applications is first used for the scientific framework DUNE. While we also expect the process to be applicable for other scientific frameworks, those in another domains could have some special characteristics that we could not foresee in developing the process. This may lead to necessary adaptions in the process and is a subject of future research.

VAF-Pro, the QA process for scientific frameworks, is designed for a scientific development team of an academic project. Such teams mostly consist of scientists developing the software, as well as possibly some technical staff. If the scientific framework in question is developed in a different environment (e.g. research center or industry), adaptions of the QA process are most likely necessary (e.g. in form of more extensive documentation). The design of VAF-Pro also assumes, that the developers test the software themselves. If there is a separate testing department, further test roles and activities (e.g. further test documentation for smooth transitions between the development and test teams) have to be inserted into the process.

If the developed scientific framework includes a high security risk (e.g. medical applications), this must also be taken into account in the QA process. In such a case, beside the desk-check review, structured inspections might be advisable, for example. The quality assurance should be carefully planned and documented.

8.3 Future Work

There are several possible research directions for the continuation of our research. One of the next planned steps is the adaption of the VAF-Pro quality assurance process for further scientific frameworks. Some questions that arise are: How does VAF-Pro (in particular, the process of creating reengineering variability models) fit other scientific frameworks, particularly other domains (e.g. biology or medical science)? What needs to be adapted on VAF-Pro? Furthermore, as discussed in the last section 8.2, it is an interesting research challenge to investigate how the VAF-Pro QA process needs to be adapted if the development environment changes (e.g. research center or industry instead of an academic project) or the framework in question includes a high securiry risk (e.g. medical applications). Both cases will likely lead to a more extensive documentation and for example to the adoption of structured inspections. Replicating the case study introduced in this thesis for other scientific frameworks could reveal such requirements for adaption mentioned above.

The list of special characteristics of scientific software that need to be taken into account when designing QA for scientific software reflects a unique effort in collecting such characteristics in the software engineering for the CSE research community. Further research in this direction could answer questions such as: Do we need to take other special characteristics into account, when designing QA for scientific frameworks in other domains (e.g. biology or medical science)? What kind of changes do such new special characteristics demand on the QA process design? How does the list of special characteristics change when we use it as rationale for the design of other software engineering methods, e.g. software design, requirements engineering, or software maintenance?

Teams developing scientific frameworks that have already existed for years or even decades often need to meet the challenge of re-structuring existing source code or even changing the programming paradigm used. Such re-structuring involves the risk of losing or breaking existing functionality and features. The process of creating reengineering variability models could be used to capture the functionality and feature of the scientific framework before the re-structuring. An interesting research question is: How can the developed variability models and system test cases be used to support re-structuring decisions and the QA during re-structuring?

Variability models capturing the features of a scientific framework can also be used as documentation for knowledge management, supporting the communication and decision making in the dvelopment team. It is also important for capturing knowledge about the framework's functionality so that it does not get lost when developers leave the development team. Interesting research questions include: What kind of management and development decisions for a scientific framework can be supported by variability modeling? What further information could be connected to a variability model to support knowledge management?

In further research, the model-based method for test case derivation based on a variability model developed in this thesis could be put under a large-scale practical test. An interesting research question would be: What kind of test suite selection methods are recommendable to be combined with our test case derivation method? In this area, a collaboration with the FeatureIDE development team could be possible to enable the use of test suite selection methods together with FeatureC++.

Correspondingly, a logical further research direction is a large-scale practical test for the use of VAF-Pro leaned on the prototypical implementation introduced in this thesis. This should include a more extensive use of feature-oriented programming in the development of the system test applications.

While the contributions of this thesis provide a basis for the adoption of SPL methods for the QA of scientific frameworks, but there are still many open questions concerning further possibilities for its application.

Appendix A

Case Study Questionnaire

The following questionnaire was used in the Case Study with DUNE developers.

DUNE Case Study: Variability modeling and desk-check

Variability modeling, open questions:

1. What do you believe are the benefits of variability modeling for the DUNE development?
2. What do you believe are the disadvantages of variability modeling for the DUNE development?
3. Furthermore, what did you like about creating the variability model?
4. Furthermore, what did you not like about creating the variability model? What did you find difficult?
5. Do you have any further comments on variability modeling?

Variability modeling, closed questions:

What is your opinion on the following arguments about variability modeling? Please mark the appropriate choice with a cross. When you answer the questions, please assume that variability modeling will be applied in DUNE development, and that you continue developing DUNE in the future.

	Strongly agree	Agree	Rather agree	Rather disagree	Disagree	Strongly disagree
I believe variability modeling is helpful in designing and developing system test applications.						
I believe the benefits of a variable model outbalance the effort for creating it.						
Using variability modeling as a quality assurance method for DUNE is important to me.						
Using variability modeling will help me to develop higher quality DUNE code.						
Variability modeling is easy to learn.						
I believe variability modeling is easy to use in practice.						
I intend to use variability modeling when developing code for DUNE.						

Desk-check, open questions:

1. What do you believe are the benefits of desk-checking for the DUNE development?
2. What do you believe are the disadvantages of desk-checking for the DUNE development?
3. Furthermore, what did you like about desk-checking?
4. Furthermore, what did you not like about desk-checking? What did you find difficult?
5. Do you have any further comments on desk-checking?

Desk-checking, closed questions:

What is your opinion on the following arguments about desk-checking? Please mark the appropriate choice with a cross. When you answer the questions, please assume that desk-checking will be applied in DUNE development, and that you continue developing DUNE in the future.

	Strongly agree	Agree	Rather agree	Rather disagree	Disagree	Strongly disagree
I believe that desk-checking leads to a higher rate of finding software defects.						
I believe that desk-checking leads to a better DUNE source code maintainability.						
I believe that desk-checking leads to a better DUNE source code readability.						
I believe the benefits of a desk-checking outbalance the effort for creating it.						
Using desk-checking as a quality assurance method for DUNE is important to me.						
Using desk-checking will help me to develop higher quality DUNE code.						
Desk-checking is easy to learn.						
I believe desk-checking is easy to use in practice.						
I intend to use desk-checking when developing code for DUNE.						

Appendix B

Source Code for Diffusion System Test Application

This appendix includes all relevant source code for the diffusion system test application: the feature-oriented source code in Section B.1, the main program in Section B.2 and the source code for the variability model in Section B.3.

B.1 Diffusion: Feature-Oriented Source Code

The following listings are the feature-oriented source code for the diffusion system test application.

Feature "diffusion":

```
class Features {
};
```

Feature "mesh":

```
refines class Features {
public:
    const std::string mesh() { return ""; }
};
```

Feature "cube":

```
refines class Features {
public:
    const std::string mesh() { return "CUBE"; }
};
```

Feature "simplex":

```
refines class Features {
public:
    const std::string mesh() { return "SIMPLEX"; }
};
```

Feature "dim":

```
refines class Features {
};
```

Feature "dim_2":

```
refines class Features {
public:
    const static int F_DIM = 2;
};
```

Feature "dim_3":

```
refines class Features {
public:
    const static int F_DIM = 3;
};
```

Feature "method":

```
refines class Features {
public:
    const std::string method() { return ""; }
};
```

Feature "SIPG":

```
refines class Features {
public:
    const std::string method() { return "SIPG"; }
};
```

Feature "FEM":

```
refines class Features {
public:
    const std::string method() { return "FEM"; }
};
```

Feature "maxlevel":

```
refines class Features {
public:
    int maxlevel() { return −1; }
};
```

Feature "ml_2" (ml_3 - ml_8 accordingly):

```
refines class Features {
public:
    int maxlevel() { return 2; }
};
```

Feature "degree":

```
refines class Features {
};
```

Feature "deg_1" (deg_2 - deg_4 accordingly):

```
refines class Features {
public:
    const static int F_DEGREE = 1;
};
```

B.2 Diffusion: Main Program

This is the "diffusion.cc" main program file:

```
/** \file
    \brief High−level test with Poisson equation
*/
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif
```

```
#include<iostream>
#include<vector>
#include<map>
#include<dune/common/parallel/mpihelper.hh>
#include<dune/common/exceptions.hh>
#include<dune/common/fvector.hh>
#include<dune/common/static_assert.hh>
#include<dune/common/timer.hh>
#include<dune/grid/yaspgrid.hh>
#include<dune/istl/bvector.hh>
#include<dune/istl/operators.hh>
#include<dune/istl/solvers.hh>
#include<dune/istl/preconditioners.hh>
#include<dune/istl/io.hh>
#include<dune/istl/paamg/amg.hh>
#include<dune/istl/superlu.hh>
#include<dune/grid/io/file/vtk/subsamplingvtkwriter.hh>

#include<dune/pdelab/finiteelementmap/monomfem.hh>
#include<dune/pdelab/finiteelementmap/opbfem.hh>
#include<dune/pdelab/finiteelementmap/qkdg.hh>
#include<dune/pdelab/finiteelementmap/pkfem.hh>
#include<dune/pdelab/constraints/conforming.hh>
#include<dune/pdelab/gridfunctionspace/gridfunctionspace.hh>
#include<dune/pdelab/gridfunctionspace/gridfunctionspaceutilities.hh>
#include<dune/pdelab/gridfunctionspace/interpolate.hh>
#include<dune/pdelab/constraints/common/constraints.hh>
#include<dune/pdelab/common/function.hh>
#include<dune/pdelab/common/functionutilities.hh>
#include<dune/pdelab/common/vtkexport.hh>
#include<dune/pdelab/gridoperator/gridoperator.hh>
#include<dune/pdelab/backend/istlvectorbackend.hh>
#include<dune/pdelab/backend/istlmatrixbackend.hh>
#include<dune/pdelab/backend/istlsolverbackend.hh>
#include<dune/pdelab/localoperator/diffusiondg.hh>
#include<dune/pdelab/localoperator/convectiondiffusionparameter.hh>
#include<dune/pdelab/localoperator/convectiondiffusiondg.hh>
#include<dune/pdelab/localoperator/convectiondiffusionfem.hh>
#include<dune/pdelab/stationary/linearproblem.hh>

// used here only for Dune::PDELab::LinearSolverResult
#include<dune/pdelab/newton/newton.hh>
```

```
#include "../utility/gridexamples.hh"
#include "../utility/utility.hh"
#include "src/Features.h"

const bool graphics = true;

template<typename GV, typename RF>
class Parameter
{
  typedef Dune::PDELab::ConvectionDiffusionBoundaryConditions::Type
          BCType;

public:
  typedef Dune::PDELab::ConvectionDiffusionParameterTraits<GV,RF> Traits;

  //! tensor diffusion coefficient
  typename Traits::PermTensorType
  A (const typename Traits::ElementType& e,
     const typename Traits::DomainType& x) const
  {
    typename Traits::PermTensorType I;
    for (std::size_t i=0; i<Traits::dimDomain; i++)
      for (std::size_t j=0; j<Traits::dimDomain; j++)
        I[i][j] = (i==j) ? 1 : 0;
    return I;
  }

  //! velocity field
  typename Traits::RangeType
  b (const typename Traits::ElementType& e,
     const typename Traits::DomainType& x) const
  {
    typename Traits::RangeType v(0.0);
    return v;
  }

  //! sink term
  typename Traits::RangeFieldType
  c (const typename Traits::ElementType& e,
     const typename Traits::DomainType& x) const
  {
    return 0.0;
  }

  //! source term
  typename Traits::RangeFieldType
  f (const typename Traits::ElementType& e,
     const typename Traits::DomainType& x) const
  {
    typename Traits::DomainType xglobal = e.geometry().global(x);
    typename Traits::RangeFieldType norm = xglobal.two_norm2();
    return (2.0*GV::dimension-4.0*norm)*exp(-norm);
  }

  //! boundary condition type function
  BCType
  bctype (const typename Traits::IntersectionType& is,
          const typename Traits::IntersectionDomainType& x) const
  {
    return
      Dune::PDELab::ConvectionDiffusionBoundaryConditions::Dirichlet;
  }

  //! Dirichlet boundary condition value
  typename Traits::RangeFieldType
  g (const typename Traits::ElementType& e,
     const typename Traits::DomainType& x) const
  {
    typename Traits::DomainType xglobal = e.geometry().global(x);
    typename Traits::RangeFieldType norm = xglobal.two_norm2();
    return exp(-norm);
  }

  //! Neumann boundary condition
  typename Traits::RangeFieldType
  j (const typename Traits::IntersectionType& is,
     const typename Traits::IntersectionDomainType& x) const
  {
    return 0.0;
  }

  //! outflow boundary condition
  typename Traits::RangeFieldType
  o (const typename Traits::IntersectionType& is,
     const typename Traits::IntersectionDomainType& x) const
  {
    return 0.0;
  }
};

/*! \brief Adapter returning ||f1(x)-f2(x)||^2 for two given
           grid functions

  \tparam T1  a grid function type
  \tparam T2  a grid function type
*/
template<typename T1, typename T2>
class DifferenceSquaredAdapter
  : public Dune::PDELab::GridFunctionBase<
  Dune::PDELab::GridFunctionTraits<typename T1::Traits::GridViewType,
           typename T1::Traits::RangeFieldType,
           1,Dune::FieldVector<typename T1::Traits::RangeFieldType,1> >
  ,DifferenceSquaredAdapter<T1,T2> >
{
public:
  typedef Dune::PDELab::GridFunctionTraits<
           typename T1::Traits::GridViewType,
           typename T1::Traits::RangeFieldType,
           1,Dune::FieldVector<typename T1::Traits::RangeFieldType,
           1> > Traits;

  //! constructor
  DifferenceSquaredAdapter (const T1& t1_, const T2& t2_) :
                            t1(t1_), t2(t2_) {}

  //! \copydoc GridFunctionBase::evaluate()
  inline void evaluate (const typename Traits::ElementType& e,
                        const typename Traits::DomainType& x,
                        typename Traits::RangeType& y) const
  {
    typename T1::Traits::RangeType y1;
    t1.evaluate(e,x,y1);
    typename T2::Traits::RangeType y2;
    t2.evaluate(e,x,y2);
    y1 -= y2;
    y = y1.two_norm2();
  }

  inline const typename Traits::GridViewType& getGridView () const
  {
    return t1.getGridView();
  }

private:
  const T1& t1;
  const T2& t2;
};

//! solve problem with DG method
template<class GV, class FEM, class PROBLEM, int degree,
         int blocksize>
void runDG (const GV& gv, const FEM& fem, PROBLEM& problem,
            std::string basename, int level, std::string method,
            std::string weights, double alpha)
{
  // coordinate and result type
  typedef typename GV::Grid::ctype Coord;
  typedef double Real;
  const int dim = GV::Grid::dimension;
  std::stringstream fullname;
  fullname << basename << "_" << method << "_w" << weights
           << "_k" << degree
           << "_dim" << dim << "_level" << level;

  // make grid function space
  typedef Dune::PDELab::NoConstraints CON;
  typedef Dune::PDELab::ISTLVectorBackend
          <Dune::PDELab::ISTLParameters::static_blocking,blocksize> VBE;
  typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
  GFS gfs(gv,fem);

  // make local operator
  Dune::PDELab::ConvectionDiffusionDGMethod::Type m;
  if (method=="SIPG")
      m = Dune::PDELab::ConvectionDiffusionDGMethod::SIPG;
  if (method=="NIPG")
      m = Dune::PDELab::ConvectionDiffusionDGMethod::NIPG;
```

```
Dune::PDELab::ConvectionDiffusionDGWeights::Type w;
if (weights=="ON")
    w = Dune::PDELab::ConvectionDiffusionDGWeights::weightsOn;
if (weights=="OFF")
    w = Dune::PDELab::ConvectionDiffusionDGWeights::weightsOff;
typedef Dune::PDELab::ConvectionDiffusionDG<PROBLEM,FEM> LOP;
LOP lop(problem,m,w,alpha);
typedef Dune::PDELab::ISTLMatrixBackend MBE;
typedef typename GFS::template ConstraintsContainer<Real>::Type CC;
CC cc;
typedef Dune::PDELab::GridOperator<GFS,GFS,LOP,MBE,Real,
                                   Real,Real,CC,CC> GO;
GO go(gfs,cc,gfs,cc,lop);

// make a vector of degree of freedom vectors and initialize it with
// Dirichlet extension
typedef typename GO::Traits::Domain U;
U u(gfs,0.0);
typedef Dune::PDELab
           ::ConvectionDiffusionDirichletExtensionAdapter<PROBLEM> G;
G g(gv,problem);

// make linear solver and solve problem
if (method=="SIPG")
  {
    typedef Dune::PDELab::ISTLBackend_SEQ_CG_ILU0 LS;
    LS ls(10000,1);
    typedef Dune::PDELab::StationaryLinearProblemSolver<GO,LS,U> SLP;
    SLP slp(go,u,ls,1e-12);
    slp.apply();
    Dune::PDELab::LinearSolverResult<double> ls_result = ls.result();
    TEST_OUTPUT("dG-Level=" << level << " IT", ls_result.iterations)
    TEST_OUTPUT("dG-Level=" << level << " rate of convergence",
                                        ls_result.conv_rate)
  }
else
  {
    typedef Dune::PDELab::ISTLBackend_SEQ_BCGS_ILU0 LS;
    LS ls(10000,1);
    typedef Dune::PDELab::StationaryLinearProblemSolver<GO,LS,U> SLP;
    SLP slp(go,u,ls,1e-12);
    slp.apply();
```

```
      Dune::PDELab::LinearSolverResult<double> ls_result ( ls.result() );
      TEST_OUTPUT("dG-Level=" << level << " IT", ls_result.iterations)
      TEST_OUTPUT("dG-Level=" << level << " rate of convergence",
                                           ls_result.conv_rate)
    }

  // compute L2 error
  typedef Dune::PDELab::DiscreteGridFunction<GFS,U> UDGF;
  UDGF udgf(gfs,u);
  typedef DifferenceSquaredAdapter<G,UDGF> DifferenceSquared;
  DifferenceSquared differencesquared(g,udgf);
  typename DifferenceSquared::Traits::RangeType l2errorsquared(0.0);
  Dune::PDELab::integrateGridFunction(differencesquared,
                                      l2errorsquared,12);

  TEST_OUTPUT("dG-Level=" << level << " gfs-globalsize", gfs.globalSize())
  TEST_OUTPUT("dG-Level=" << level << " L2ERROR", std::setw(11)
                                   << std::setprecision(7)
                                   << std::scientific << std::uppercase
                                   << sqrt(l2errorsquared[0]))

  // write vtk file
  if (graphics)
    {
      Dune::SubsamplingVTKWriter<GV> vtkwriter(gv,degree-1);
      vtkwriter.addVertexData(new Dune::PDELab::
                              VTKGridFunctionAdapter<UDGF>(udgf,"u_h"));
      vtkwriter.addVertexData(new Dune::PDELab::
                              VTKGridFunctionAdapter<G>(g,"u"));
      vtkwriter.write(fullname.str(),Dune::VTK::ascii);
    }
  std::cout << "========================================" << std::endl;
}


//! solve problem with DG method
template<class GV, class FEM, class PROBLEM, int degree>
void runFEM (const GV& gv, const FEM& fem, PROBLEM& problem,
             std::string basename, int level)
{
  // coordinate and result type
  typedef typename GV::Grid::ctype Coord;
typedef double Real;
const int dim = GV::Grid::dimension;
std::stringstream fullname;
fullname << basename << "_FEM" << "_k" << degree << "_dim"
         << dim << "_level" << level;

// make grid function space
typedef Dune::PDELab::ISTLVectorBackend<> VBE;
typedef Dune::PDELab::ConformingDirichletConstraints CON;
typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
GFS gfs(gv,fem);

// make constraints container and initialize it
typedef typename GFS::template ConstraintsContainer<Real>::Type CC;
CC cc;

// make local operator
typedef Dune::PDELab::ConvectionDiffusionFEM<PROBLEM,FEM> LOP;
LOP lop(problem);
typedef Dune::PDELab::ISTLMatrixBackend MBE;
typedef Dune::PDELab::GridOperator<GFS,GFS,LOP,MBE,Real,Real,Real,
                                   CC,CC> GO;
GO go(gfs,cc,gfs,cc,lop);

// make a vector of degree of freedom vectors and initialize it
// with Dirichlet extension
typedef typename GO::Traits::Domain U;
U u(gfs,0.0);
typedef Dune::PDELab::
        ConvectionDiffusionDirichletExtensionAdapter<PROBLEM> G;
G g(gv,problem);
Dune::PDELab::interpolate(g,gfs,u);

Dune::PDELab::ConvectionDiffusionBoundaryConditionAdapter<PROBLEM>
        bctype(gv,problem);
Dune::PDELab::constraints(bctype,gfs,cc);
Dune::PDELab::set_nonconstrained_dofs(cc,0.0,u);

// make linear solver and solve problem
typedef Dune::PDELab::ISTLBackend_SEQ_CG_ILU0 LS;
LS ls(10000,1);
typedef Dune::PDELab::StationaryLinearProblemSolver<GO,LS,U> SLP;
```

```
  SLP slp(go,u,ls,1e-12);
  slp.apply();
  Dune::PDELab::LinearSolverResult<double> ls_result ( ls.result() );
  TEST_OUTPUT("FEM-Level=" << level << " IT", ls_result.iterations)
  TEST_OUTPUT("FEM-Level=" << level << " rate of convergence",
                                          ls_result.conv_rate)

  // compute L2 error
  typedef Dune::PDELab::DiscreteGridFunction<GFS,U> UDGF;
  UDGF udgf(gfs,u);
  typedef DifferenceSquaredAdapter<G,UDGF> DifferenceSquared;
  DifferenceSquared differencesquared(g,udgf);
  typename DifferenceSquared::Traits::RangeType l2errorsquared(0.0);
  Dune::PDELab::integrateGridFunction(differencesquared,
                                      l2errorsquared,12);

  TEST_OUTPUT("FEM-Level=" << level << " gfs-globalsize",
                                          gfs.globalSize())
  TEST_OUTPUT("FEM-Level=" << level << " L2ERROR", std::setw(11)
                              << std::setprecision(7) << std::scientific
                              << std::uppercase
                              << sqrt(l2errorsquared[0]))

  // write vtk file
  if (graphics)
    {
      Dune::SubsamplingVTKWriter<GV> vtkwriter(gv,degree-1);
      vtkwriter.addVertexData(new Dune::PDELab::
                              VTKGridFunctionAdapter<UDGF>(udgf,"u_h"));
      vtkwriter.addVertexData(new Dune::PDELab::
                              VTKGridFunctionAdapter<G>(g,"u"));
      vtkwriter.write(fullname.str(),Dune::VTK::ascii);
    }
}

int main(int argc, char** argv)
{
  //Maybe initialize Mpi
  Dune::MPIHelper& helper = Dune::MPIHelper::instance(argc, argv);
  if(Dune::MPIHelper::isFake)
    std::cout<< "This is a sequential program." << std::endl;
  else
```

```
    {
      if(helper.rank()==0)
        std::cout << "parallel run on " << helper.size()
                  << " process(es)" << std::endl;
    }

  try
    {
      Features features;
      if (features.mesh()=="CUBE")
        {
          const int dim = Features::F_DIM;
          Dune::FieldVector<double,dim> L(1.0);
          Dune::FieldVector<int,dim> N(1);
          Dune::FieldVector<bool,dim> P(false);
          typedef Dune::YaspGrid<dim> Grid;
          Grid grid(L,N,P,0);
          typedef Grid::LeafGridView GV;
          for (int i=0; i<=features.maxlevel(); ++i)
            {
              const GV& gv=grid.leafView();
              typedef Parameter<GV,double> PROBLEM;
              PROBLEM problem;
              const int degree=Features::F_DEGREE;

              if (features.method()=="SIPG") {
                  typedef Dune::PDELab::
                          QkDGLocalFiniteElementMap<Grid::ctype,
                                                    double,degree,
                                                    dim> FEMDG;
                  FEMDG femdg;
                  const int blocksize =
                    Dune::QkStuff::QkSize<degree,dim>::value;
                  runDG<GV,FEMDG,PROBLEM,degree,blocksize>
                    (gv,femdg,problem,"CUBE",i,"SIPG","ON",2.0);
              }
              if (features.method()=="FEM") {
                  typedef Dune::PDELab::QkCGLocalFiniteElementMap
                    <Grid::ctype,double,degree,dim> FEMCG;
                  FEMCG femcg;
                  runFEM<GV,FEMCG,PROBLEM,degree>
                    (gv,femcg,problem,"CUBE",i);
              }
              // refine grid
              if (i<features.maxlevel()) grid.globalRefine(1);
            }
        }
#if HAVE_ALUGRID
BEGIN_BLOCK(HAVE_ALUGRID)
      if (features.mesh()=="SIMPLEX")
        {
          const int dim    = Features::F_DIM;

          // make grid
          ALUUnitCube<dim> unitcube;
          typedef ALUUnitCube<dim>::GridType Grid;
          typedef Grid::LeafGridView GV;

          for (int i=0; i<=features.maxlevel(); ++i)
            {
              const GV& gv=unitcube.grid().leafView();
              typedef Parameter<GV,double> PROBLEM;
              PROBLEM problem;
              const int degree = Features::F_DEGREE;

              if (features.method()=="SIPG") {
                  typedef Dune::PDELab::OPBLocalFiniteElementMap
                        <Grid::ctype,double,degree,dim,
                         Dune::GeometryType::simplex> FEMDG;
                  FEMDG femdg;
                  const int blocksize = Dune::PB::
                        PkSize<degree,dim>::value;
                  runDG<GV,FEMDG,PROBLEM,degree,blocksize>
                        (gv,femdg,problem,"SIMPLEX",i,"SIPG","ON",2.0);
              }

              if (features.method()=="FEM") {
                  typedef Dune::PDELab::PkLocalFiniteElementMap
                         <GV,Grid::ctype,double,degree,dim> FEMCG;
                  FEMCG femcg(gv);
                  runFEM<GV,FEMCG,PROBLEM,degree>(gv,femcg,problem,
                                                "SIMPLEX",i);
              }
              // refine grid
              if (i<features.maxlevel()) unitcube.grid().globalRefine(1);
            }
        }
END_BLOCK(HAVE_ALUGRID)
#endif
    }
  catch (Dune::Exception &e)
    {
      std::cerr << "Dune reported error: " << e << std::endl;
      return 1;
    }
  catch (...)
    {
      std::cerr << "Unknown exception thrown!" << std::endl;
      return 1;
    }
}
```

B.3 Diffusion: Variability Model

This is the source code for the diffusion variability model

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <featureModel chosenLayoutAlgorithm="1">
    <struct>
      <and mandatory="true" name="diffusion">
        <alt mandatory="true" name="mesh">
          <feature mandatory="true" name="cube"/>
          <feature mandatory="true" name="simplex"/>
        </alt>
        <alt mandatory="true" name="dim">
          <feature mandatory="true" name="dim_2"/>
          <feature mandatory="true" name="dim_3"/>
       </alt>
       <alt mandatory="true" name="method">
          <feature mandatory="true" name="SIPG"/>
          <feature mandatory="true" name="FEM"/>
        </alt>
        <alt mandatory="true" name="maxlevel">
          <feature mandatory="true" name="ml_2"/>
          <feature mandatory="true" name="ml_3"/>
          <feature mandatory="true" name="ml_4"/>
          <feature mandatory="true" name="ml_5"/>
          <feature mandatory="true" name="ml_6"/>
          <feature mandatory="true" name="ml_7"/>
          <feature mandatory="true" name="ml_8"/>
        </alt>
        <alt mandatory="true" name="degree">
          <feature mandatory="true" name="deg_1"/>
          <feature mandatory="true" name="deg_2"/>
          <feature mandatory="true" name="deg_3"/>
          <feature mandatory="true" name="deg_4"/>
        </alt>
      </and>
    </struct>
    <constraints>
      <rule>
        <imp>
          <conj>
            <var>cube</var>
            <conj>
              <var>dim_2</var>
              <var>SIPG</var>
            </conj>
          </conj>
          <not>
            <var>deg_4</var>
          </not>
        </imp>
      </rule>
      <rule>
        <imp>
          <conj>
            <var>cube</var>
            <conj>
              <var>dim_2</var>
              <var>FEM</var>
            </conj>
          </conj>
          <not>
            <disj>
              <var>deg_3</var>
```

```
              <var>deg_4</var>
            </disj>
          </not>
        </imp>
      </rule>
      <rule>
        <imp>
          <conj>
            <var>cube</var>
            <conj>
              <var>dim_3</var>
              <var>FEM</var>
            </conj>
          </conj>
          <not>
            <disj>
              <var>deg_3</var>
              <var>deg_4</var>
            </disj>
          </not>
        </imp>
      </rule>
      <rule>
        <imp>
          <conj>
            <var>simplex</var>
            <conj>
              <var>dim_2</var>
              <var>SIPG</var>
            </conj>
          </conj>
          <not>
            <var>deg_4</var>
          </not>
        </imp>
      </rule>
      <rule>
        <imp>
          <conj>
            <var>simplex</var>
            <conj>
              <var>dim_2</var>
              <var>FEM</var>
            </conj>
          </conj>
          <not>
            <var>deg_4</var>
          </not>
        </imp>
      </rule>
      <rule>
        <imp>
          <conj>
            <var>simplex</var>
            <conj>
              <var>dim_3</var>
              <var>SIPG</var>
            </conj>
          </conj>
          <not>
            <var>deg_4</var>
          </not>
        </imp>
      </rule>
    </constraints>
    <calculations Auto="true" Constraints="true"
                  Features="true" Redundant="true"/>
    <comments/>
    <featureOrder userDefined="false"/>
  </featureModel>
```

Appendix C

Developing System Test Applications with FeatureIDE and FeatureC++

This appendix explains step by step how to develop system test applications with FeatureIDE and FeatureC++.

The steps needed are

- installing FeatureIDE and FeatureC++,
- opening the FeatureIDE perspective in Eclipse,
- creating a FeatureIDE project,
- creating a variability model,
- defining constraints,
- developing source code,
- creating configurations for different test cases, and
- running the system test applications.

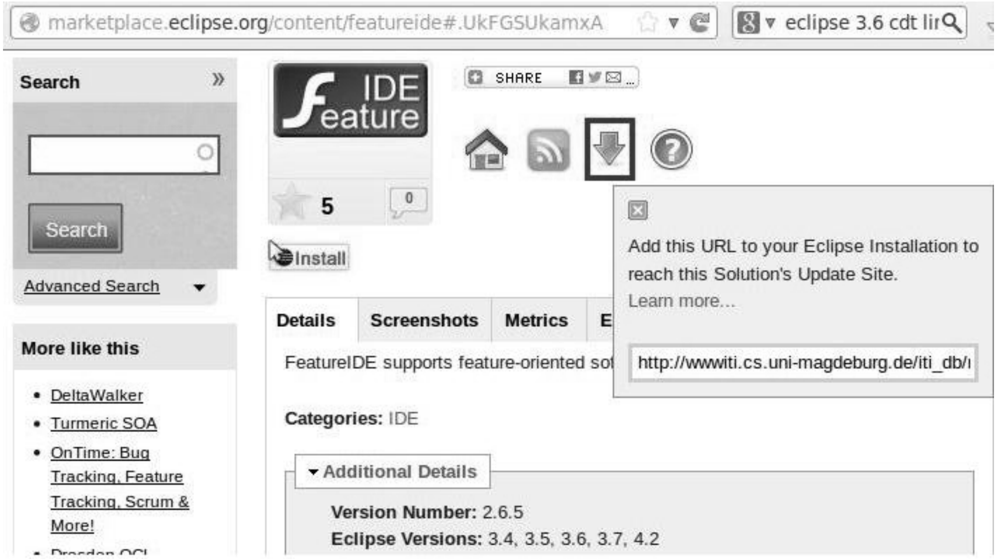
C.1 Install FeatureIDE and FeatureC++

This section explains how to install FeatureIDE and FeatureC++ for Eclipse, and FeatureC++ for the use in an automated test environment.

C.1.1 FeatureIDE and FeatureC++

FeatureIDE is an Eclipse plug-in. Please check at the FeatuteIDE internet page [31] first that there is a FeatureIDE version available for your Eclipse version. This documentation was created using Eclipse Galileo version 3.5.2.

Goto Eclipse Marketplace for FeatureIDE at marketplace.eclipse.org/content/featureide to get the correct URL for the Eclipse Installation. Click on the green arrow for download to see the URL (see Figure).



marketplace.eclipse.org/content/featureide#.UkFGSUkamxA

Add this URL to your Eclipse Installation to reach this Solution's Update Site.
Learn more...

http://wwwiti.cs.uni-magdeburg.de/iti_db/ı

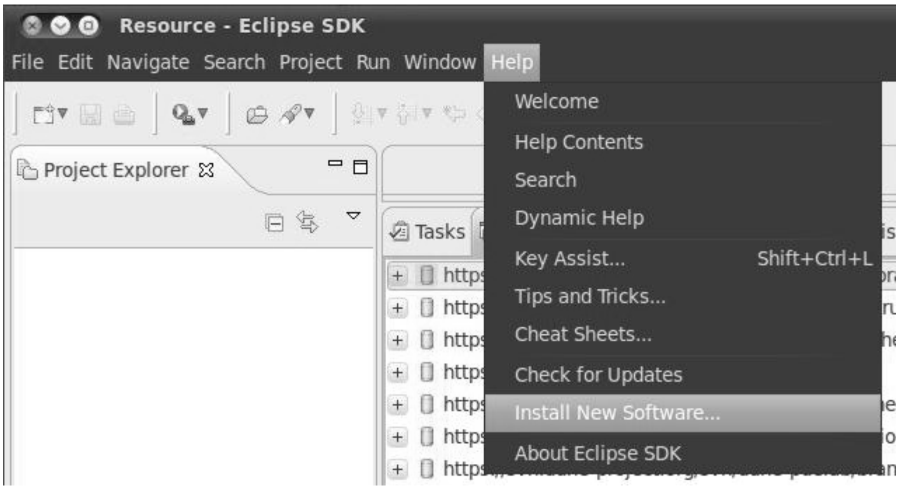
FeatureIDE supports feature-oriented so

Categories: IDE

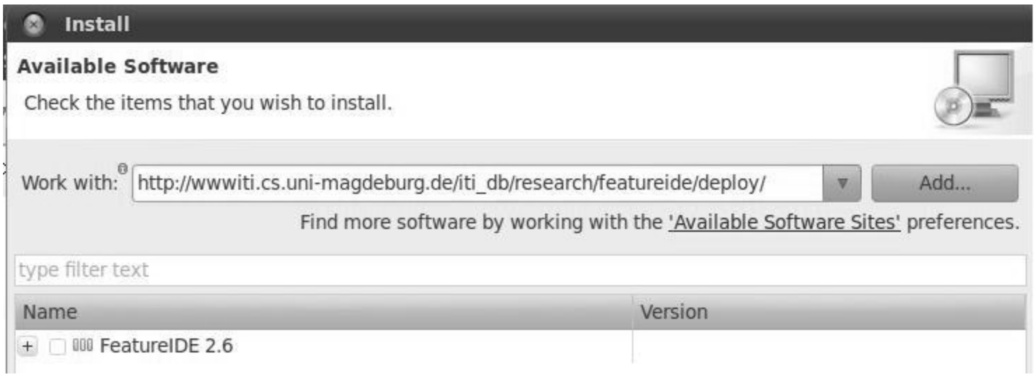
Additional Details

Version Number: 2.6.5
Eclipse Versions: 3.4, 3.5, 3.6, 3.7, 4.2

Start Eclipse and select menu Help/Install New Software...



In field "Work with:" insert the URL for Eclipse Marketplace for FeatureIDE.



Available Software

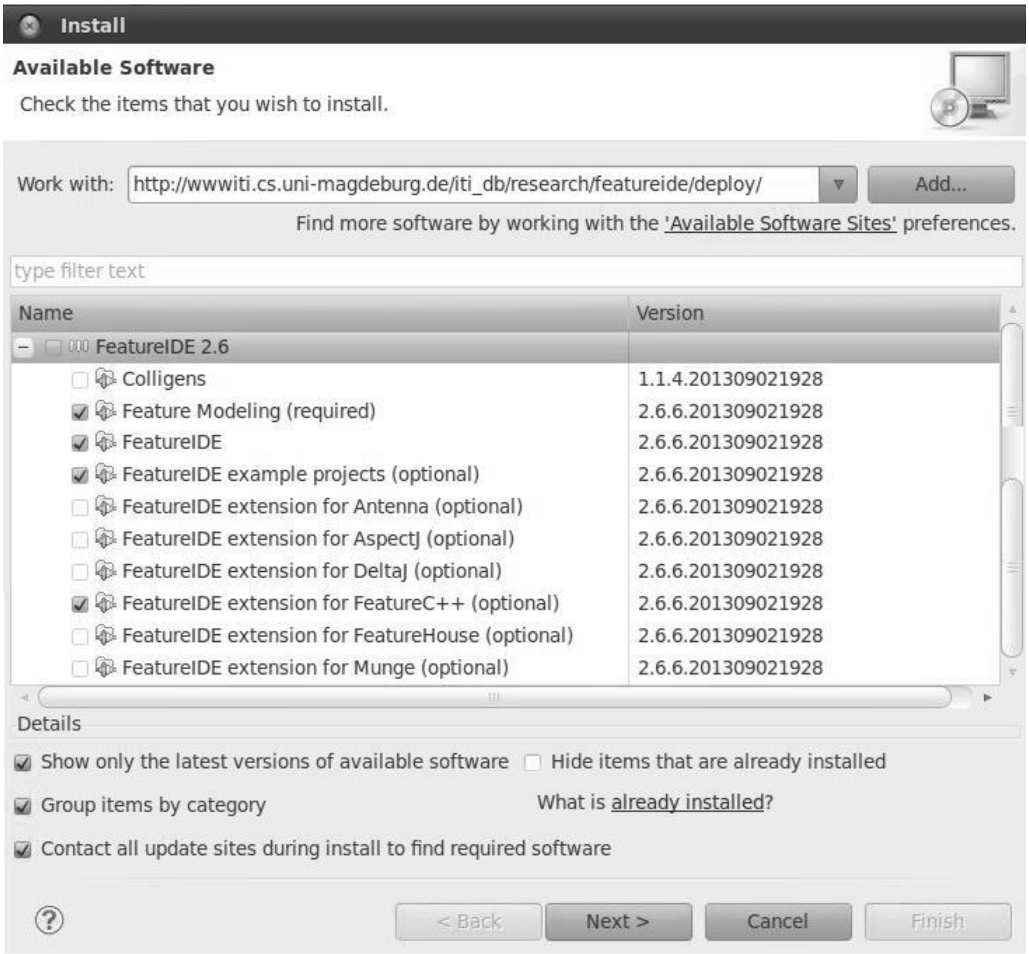
Check the items that you wish to install.

Work with: http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/deploy/

Find more software by working with the 'Available Software Sites' preferences.

Name	Version
□ FeatureIDE 2.6	

Select the FeatureIDE modules as shown in the figure below (at least variability modeling, FeatureIDE and FeatureIDE extension for FeatureC++).



Install

Available Software

Check the items that you wish to install.

Work with: http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/deploy/

Find more software by working with the 'Available Software Sites' preferences.

type filter text

Name	Version
□ FeatureIDE 2.6	
□ Colligens	1.1.4.201309021928
☑ Feature Modeling (required)	2.6.6.201309021928
☑ FeatureIDE	2.6.6.201309021928
☑ FeatureIDE example projects (optional)	2.6.6.201309021928
□ FeatureIDE extension for Antenna (optional)	2.6.6.201309021928
□ FeatureIDE extension for AspectJ (optional)	2.6.6.201309021928
□ FeatureIDE extension for DeltaJ (optional)	2.6.6.201309021928
☑ FeatureIDE extension for FeatureC++ (optional)	2.6.6.201309021928
□ FeatureIDE extension for FeatureHouse (optional)	2.6.6.201309021928
□ FeatureIDE extension for Munge (optional)	2.6.6.201309021928

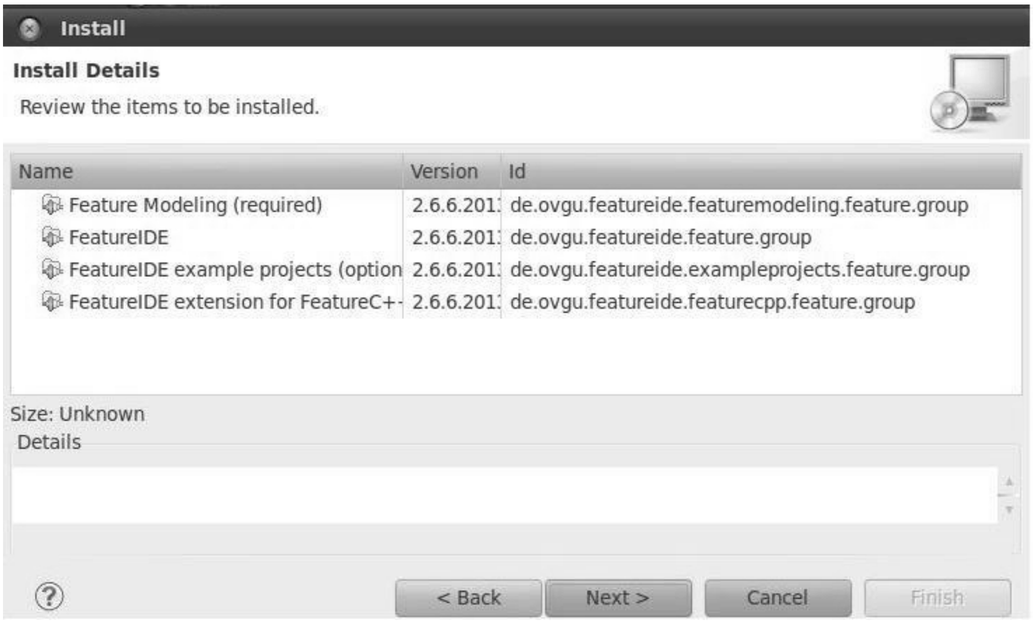
Details

☑ Show only the latest versions of available software □ Hide items that are already installed

☑ Group items by category What is already installed?

☑ Contact all update sites during install to find required software

Click next to install.



Install

Install Details

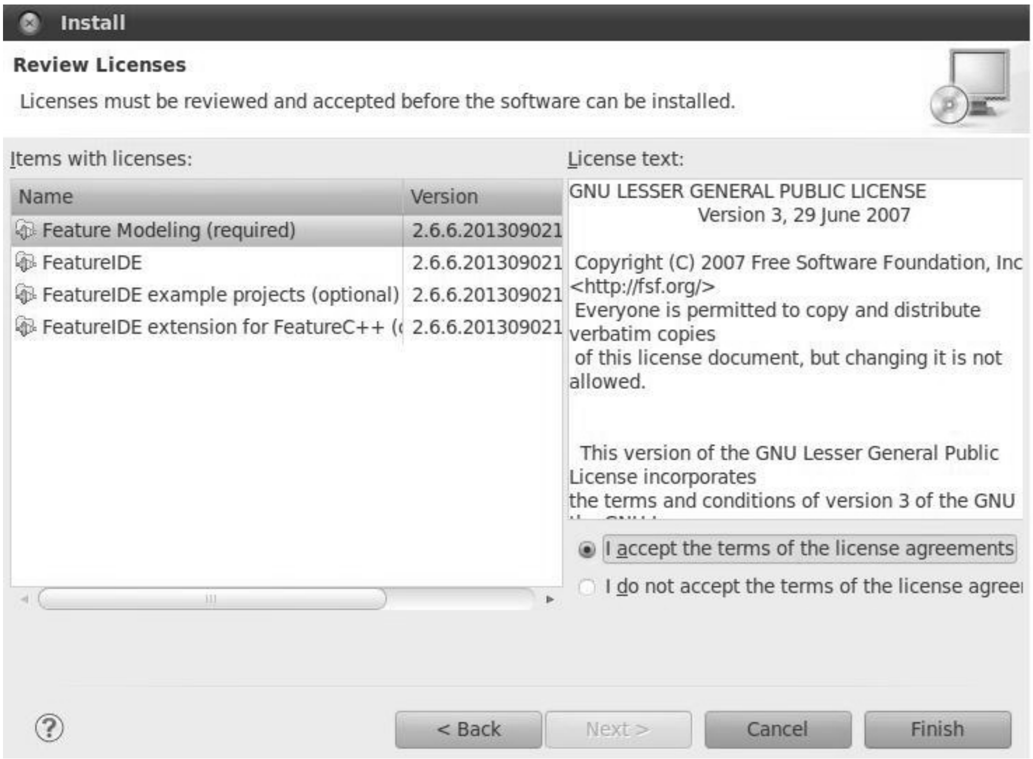
Review the items to be installed.

Name	Version	Id
Feature Modeling (required)	2.6.6.201	de.ovgu.featureide.featuremodeling.feature.group
FeatureIDE	2.6.6.201	de.ovgu.featureide.feature.group
FeatureIDE example projects (option	2.6.6.201	de.ovgu.featureide.exampleprojects.feature.group
FeatureIDE extension for FeatureC+-	2.6.6.201	de.ovgu.featureide.featurecpp.feature.group

Size: Unknown

Details

Accept the license agreement.



Install

Review Licenses

Licenses must be reviewed and accepted before the software can be installed.

Items with licenses:

Name	Version
Feature Modeling (required)	2.6.6.201309021
FeatureIDE	2.6.6.201309021
FeatureIDE example projects (optional)	2.6.6.201309021
FeatureIDE extension for FeatureC++ ((	2.6.6.201309021

License text:

GNU LESSER GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

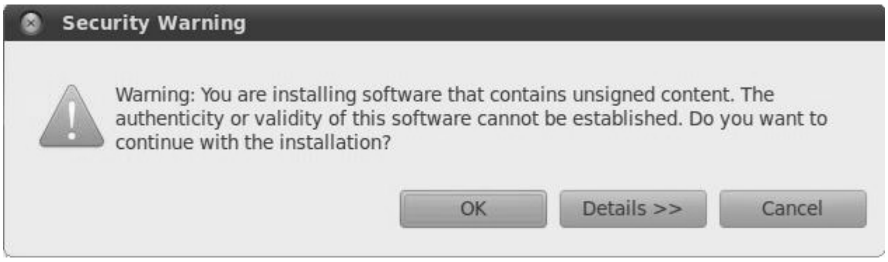
Copyright (C) 2007 Free Software Foundation, Inc
<http://fsf.org/>
Everyone is permitted to copy and distribute
verbatim copies
of this license document, but changing it is not
allowed.

This version of the GNU Lesser General Public
License incorporates
the terms and conditions of version 3 of the GNU

● I accept the terms of the license agreements

○ I do not accept the terms of the license agreei

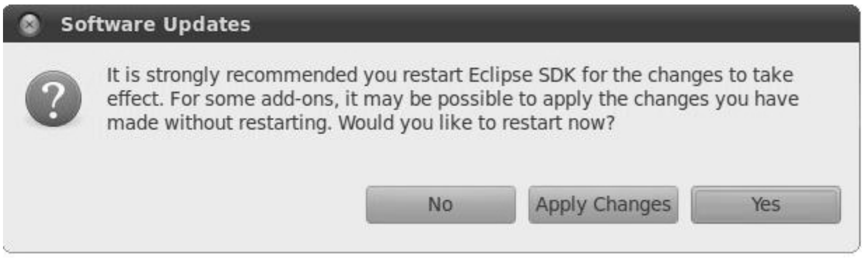
Accept the warning with OK.



Security Warning

Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?

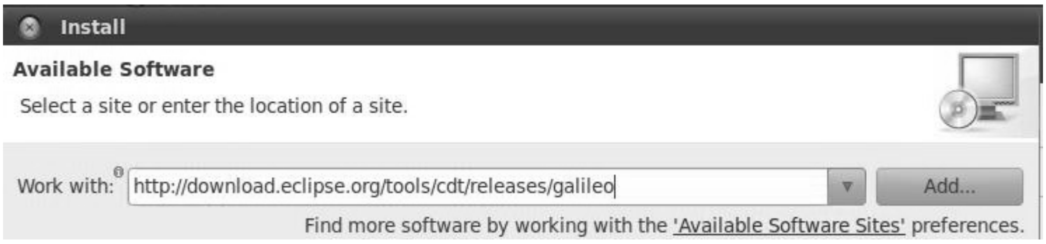
Restart Eclipse.



Software Updates

It is strongly recommended you restart Eclipse SDK for the changes to take effect. For some add-ons, it may be possible to apply the changes you have made without restarting. Would you like to restart now?

FeatureC++ needs the CDT plug-in. You can check, if it is install by trying to install it in the same way as FeatureIDE.



Install

Available Software

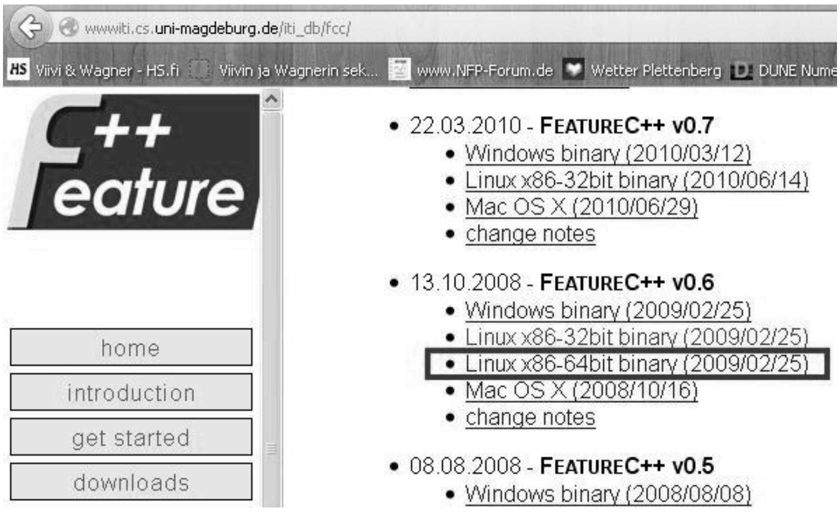
Select a site or enter the location of a site.

Work with: http://download.eclipse.org/tools/cdt/releases/galileo

Find more software by working with the 'Available Software Sites' preferences.

C.1.2 FeatureC++ for the Automated Test Environment

To be able to use FeatureC++ also without Eclipse for an automated test environment scripts, you need to install a FeatureC++ binary. Instruction for the installation can be found on the FeatureIDE internet page http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/under Get Started/Installation. In our case we needed the "Linux x86-64bit binary (2009/02/25)" binary.

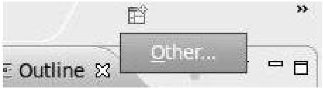


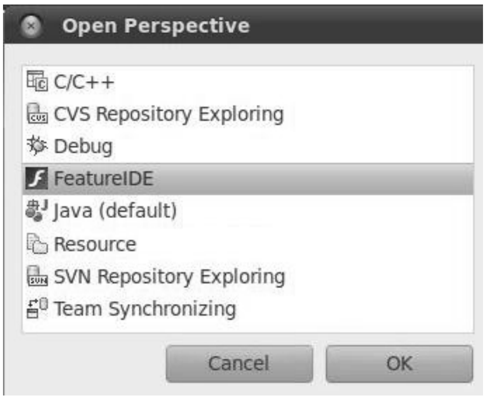
wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

- 22.03.2010 - FEATUREC++ v0.7
  - Windows binary (2010/03/12)
  - Linux x86-32bit binary (2010/06/14)
  - Mac OS X (2010/06/29)
  - change notes
- 13.10.2008 - FEATUREC++ v0.6
  - Windows binary (2009/02/25)
  - Linux x86-32bit binary (2009/02/25)
  - Linux x86-64bit binary (2009/02/25)
  - Mac OS X (2008/10/16)
  - change notes
- 08.08.2008 - FEATUREC++ v0.5
  - Windows binary (2008/08/08)

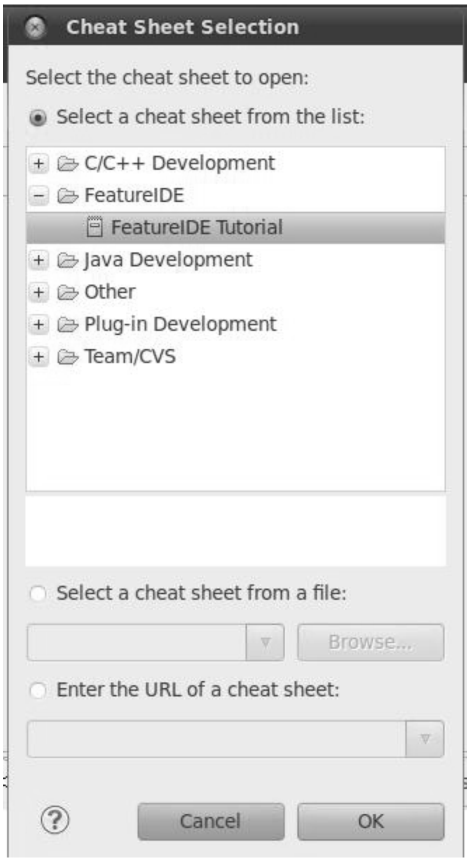
C.2 Open the FeatureIDE Perspective

After installing FeatureIDE, the FeatureIDE perspective can be opened using the context menu of the icon on the right upper corner of the Eclipse window.





For a short introduction to FeatureIDE and FeatureC++, see the FeatureIDE Cheet Sheet in menu Help/Cheat Sheets.

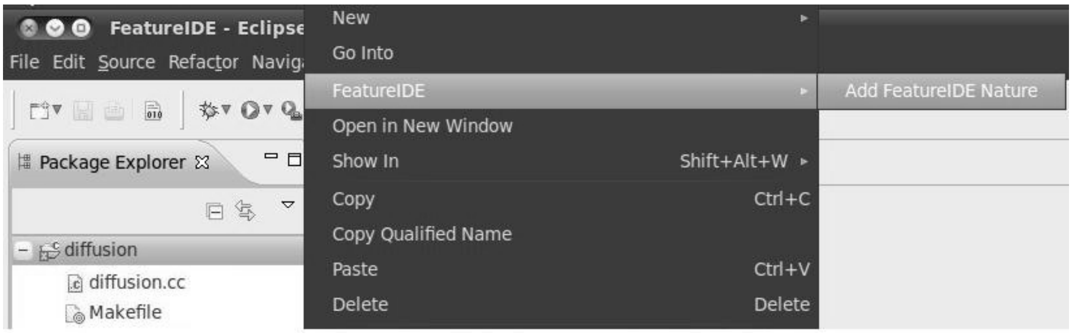


C.3 Create a FeatureIDE Project

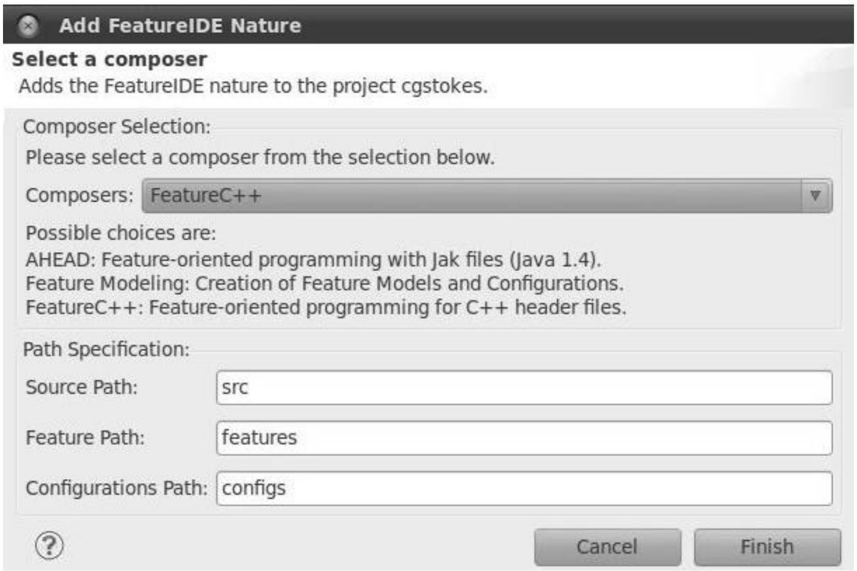
There are two possibilities for creating a FeatureIDE project: it can be created directly using the menu item New/FeatureIDE Project or the FeatureIDE nature can be included to an existing project, as described below.

During the project creation the developer can choose between the different implementation techniques for FeatureIDE. In the example below we choose FeatureC++.

For adding the FeatureIDE nature to a project, select the project in the Package Explorer and choose FeatureIDE/Add FeatureIDE Nature in the context menu.



Choose the Composer FeatureC++. Leave the path specification as it is.



Add FeatureIDE Nature

Select a composer

Adds the FeatureIDE nature to the project cgstokes.

Composer Selection:

Please select a composer from the selection below.

Composers: FeatureC++

Possible choices are:
AHEAD: Feature-oriented programming with Jak files (Java 1.4).
Feature Modeling: Creation of Feature Models and Configurations.
FeatureC++: Feature-oriented programming for C++ header files.

Path Specification:

Source Path: src

Feature Path: features

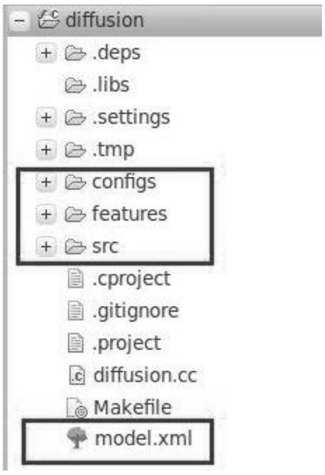
Configurations Path: configs

FeatureIDE includes three new folders and one new file to the project:

- model.xml file will include the variability model for the system test application
- configs folder will include configuration files with different feature combinations based on the variability model
- features folder will include the FeatureC++ source code for each concrete feature

in the variability model

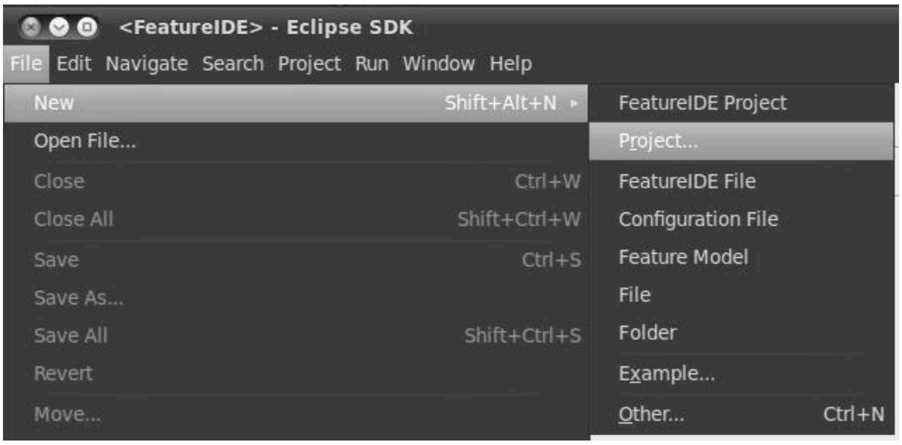
- src folder will include the executable source code created by FeatureC++ based on the source code in the features folder and the chosen configuration



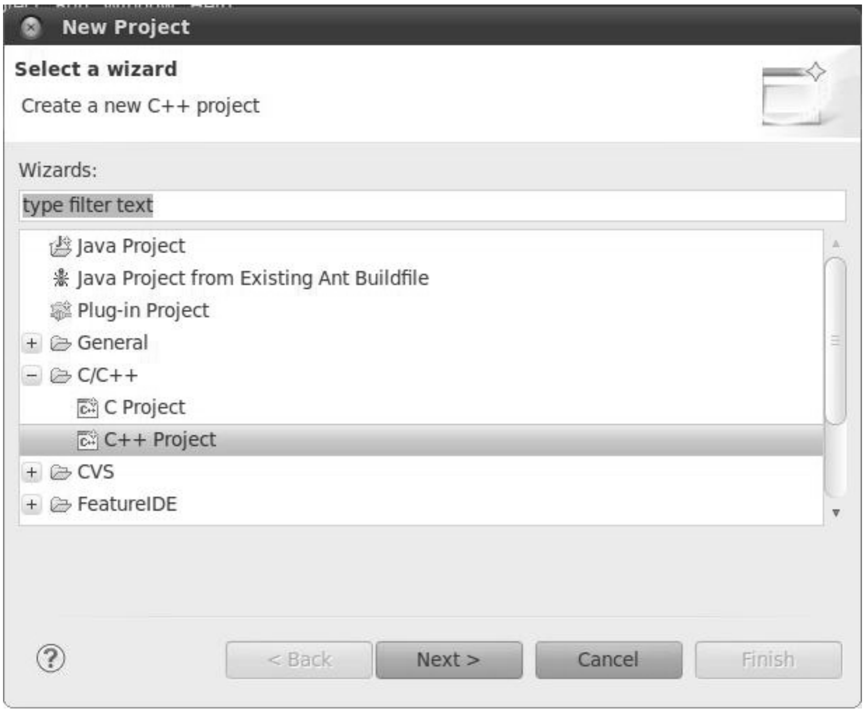
Example: Create a DUNE Makefile Project with FeatureIDE

DUNE applications are build using makefiles. This is why we want to import an existing makefile project in FeatureIDE for the diffusion system test application.

Select menu File/New/Project...



Select C++ Project and click Next.



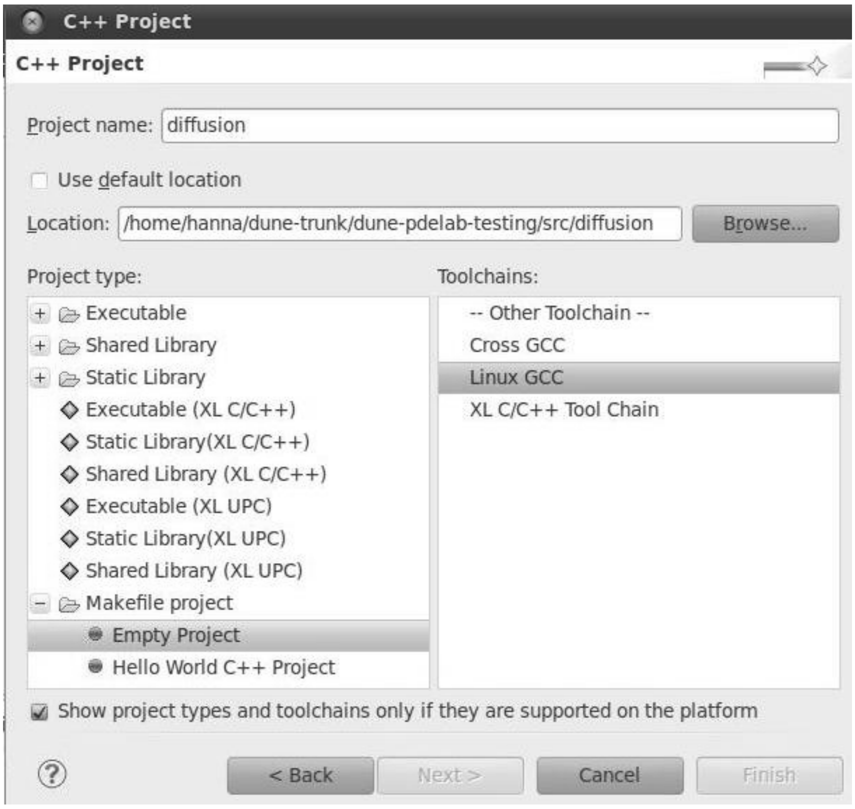
New Project

Select a wizard

Create a new C++ project

Wizards: type filter text

Give the project a suitable project name. As location select the folder where the makefile is located. Select Empty Project for the Project Type and Linux GCC for the Toolchain. Click next.



C++ Project

C++ Project

Project name: diffusion

□ Use default location

Location: /home/hanna/dune-trunk/dune-pdelab-testing/src/diffusion [Browse...]

Project type:

- + Executable
- + Shared Library
- + Static Library
- Executable (XL C/C++)
- Static Library(XL C/C++)
- Shared Library (XL C/C++)
- Executable (XL UPC)
- Static Library(XL UPC)
- Shared Library (XL UPC)
- − Makefile project
  - Empty Project
  - Hello World C++ Project

Toolchains:

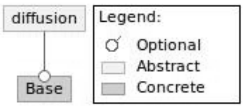
- -- Other Toolchain --
- Cross GCC
- Linux GCC
- XL C/C++ Tool Chain

☑ Show project types and toolchains only if they are supported on the platform

Now one can already see the makefile and other sources located in the same folder in the Package Explorer. Add the FeatureIDE nature to the project as explained in the beginning of the section.

C.4 Create a Variability Model and Define Constraints

FeatureIDE automatically creates a file named model.xml in the project folder. It can be opened with a double click. At first, the model will only include a feature named in the same way as the project and a dummy feature Base.



By selecting a feature and using the context menu one can

- create a feature above or below the selected feature
- rename or delete a feature

- make a feature abstract or concrete. For every concrete feature FeatureC++ creates a subfolder in features folder for the FeatureC++ source code.

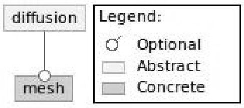
- set the relationship between a parent and a child feature mandatory or optional. Mandatory means that for a valid feature configuration, this child feature must be selected.

- select the type of the relationship between a parent and its child features: and-, or-, or alternative-group.

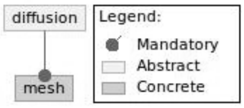
- create constraints that restrict possible feature combinations

Example: Create a variability model for the diffusion system test application

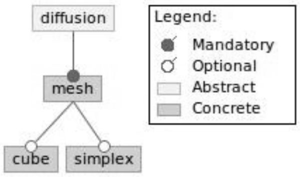
We demostrate the creation of a variability model with the DUNE system test application "diffusion". As described in Section 4.1.2.3, the test application diffusion has the following variable features: mesh, dimension, method, maximum level and degree. In the variability model, select the feature Base and use the context menu to rename Base to mesh.



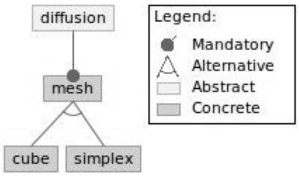
For mesh (like for each of the features listed above), a value must be selected for a test case. Select feature mesh and use the context menu to change the feature to mandatory.



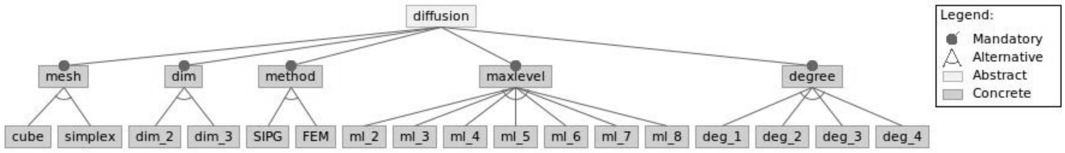
In the diffusion system test application, possible values for mesh are cube and simplex (see Section 4.1.2.3). Select feature mesh and add two new feature using the context menu item Create Feature Below.



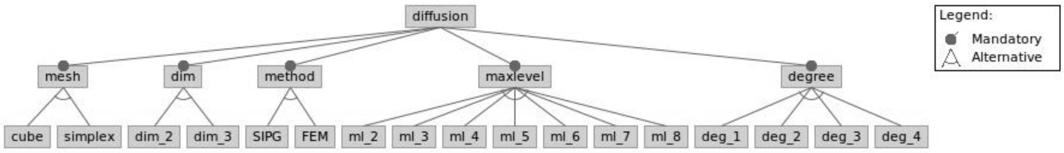
For mesh, one needs to choose exactly one value, cube or simplex. This means the relationship between parent feature mesh and child features cube and simplex is an alternative choice. Select feature mesh and choose Alternative in the context menu.



Insert other features in the same way. Note: feature names cannot be pure numerical.

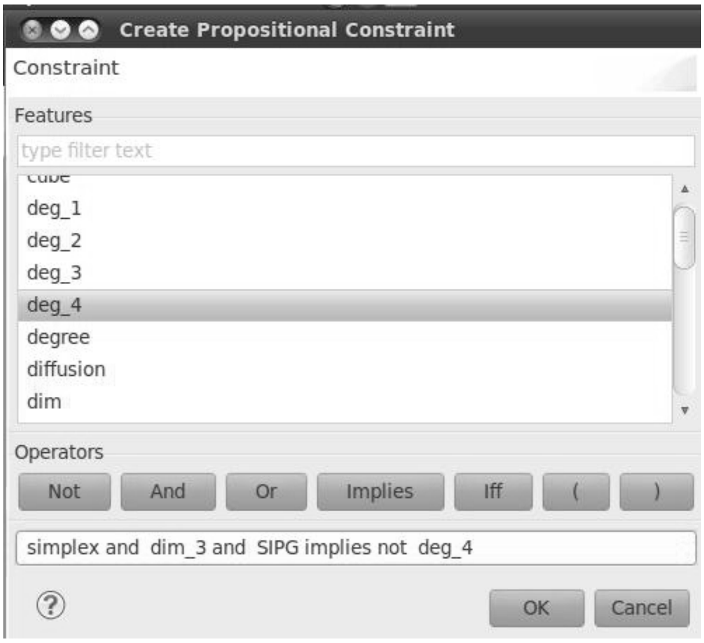


Since we also want to add FeatureC++ source code for the root feature diffusion, select the feature and use the context menu to change it to concrete (remove Abstract flag).



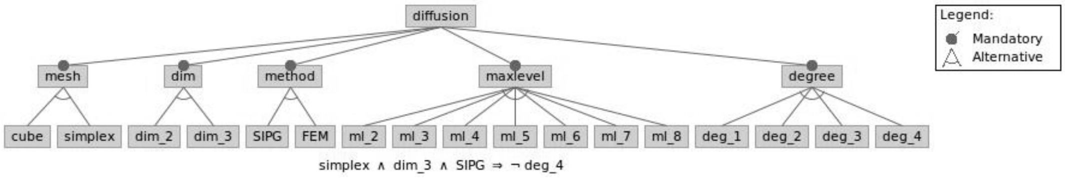
C.5 Define Constraints

Now we can define some constraints between the features, if necessary. For example we could define that for a simplex mesh in 3D and method SIPG it is not possible to choose degree four. Choose Create Constraint in the context menu (none of the features need to be selected) and define the constraint by clicking on the feature names and operators.

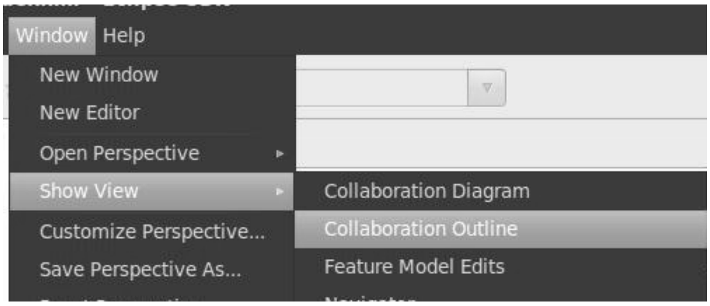


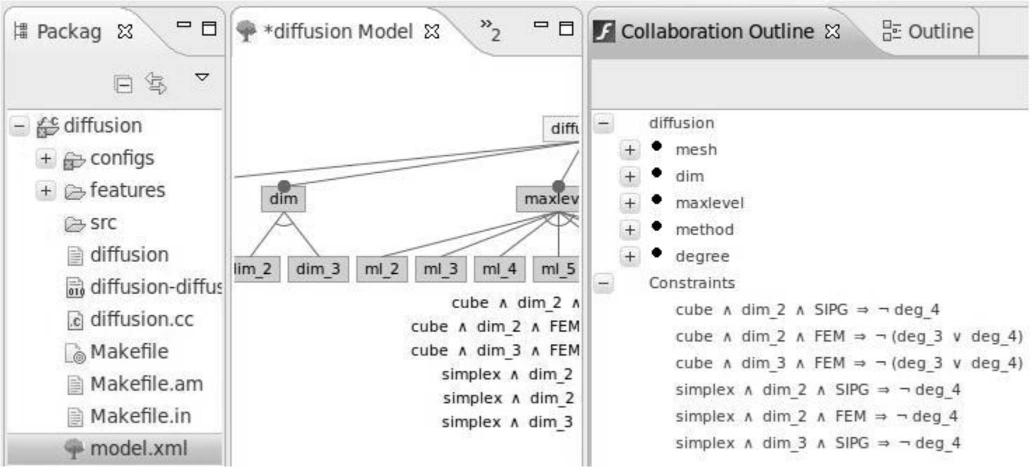
simplex and dim_3 and SIPG implies not deg_4

The constraint is displayed in the variability model.



The variability model can also be edited in a very similar way using the Collaboration Outline view.

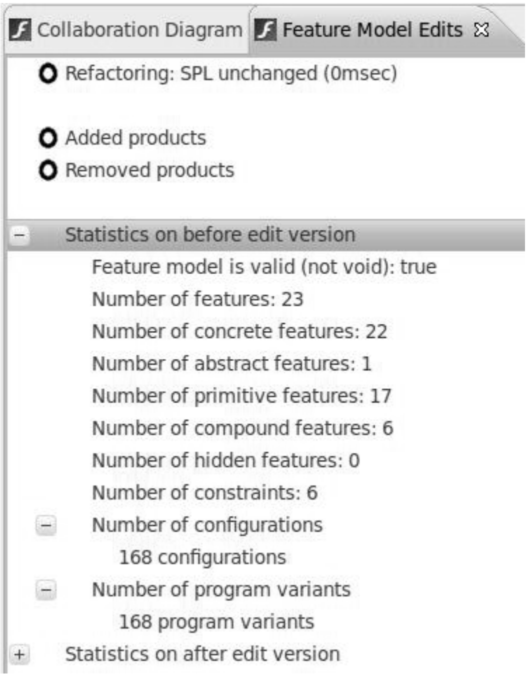




```
diffusion
  configs
  features
  src
  diffusion
  diffusion-diffus
  diffusion.cc
  Makefile
  Makefile.am
  Makefile.in
  model.xml
```

```
diffusion
  • mesh
  • dim
  • maxlevel
  • method
  • degree
Constraints
  cube ∧ dim_2 ∧ SIPG ⇒ ¬ deg_4
  cube ∧ dim_2 ∧ FEM ⇒ ¬ (deg_3 ∨ deg_4)
  cube ∧ dim_3 ∧ FEM ⇒ ¬ (deg_3 ∨ deg_4)
  simplex ∧ dim_2 ∧ SIPG ⇒ ¬ deg_4
  simplex ∧ dim_2 ∧ FEM ⇒ ¬ deg_4
  simplex ∧ dim_3 ∧ SIPG ⇒ ¬ deg_4
```

FeatureIDE view variability model Edits provides some statistics of the variability model.



```
○ Refactoring: SPL unchanged (0msec)

○ Added products
○ Removed products

− Statistics on before edit version
    Feature model is valid (not void): true
    Number of features: 23
    Number of concrete features: 22
    Number of abstract features: 1
    Number of primitive features: 17
    Number of compound features: 6
    Number of hidden features: 0
    Number of constraints: 6
  − Number of configurations
      168 configurations
  − Number of program variants
      168 program variants
+ Statistics on after edit version
```

C.6 Develop Source Code

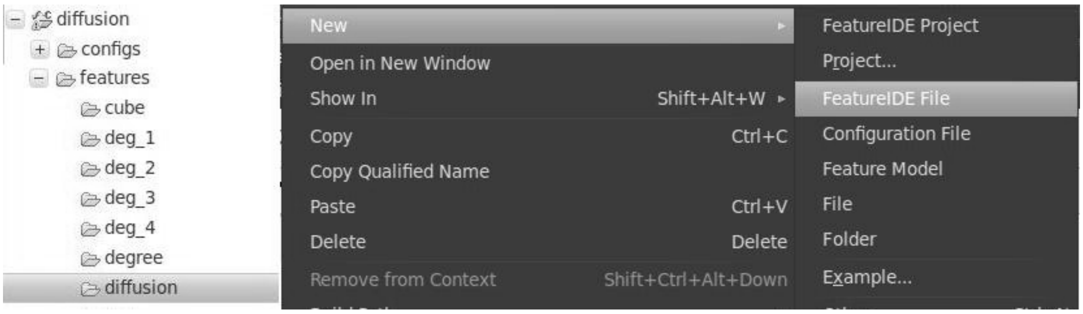
Depending on the used implementation tool, FeatureIDE supports the source code development with editors, syntax highlighting, on-the-fly error checking etc [76]. In the following we concentrate on the implementation tool FeatureC++.

In features folder, FeatureC++ automatically creates a subfolder for each concrete feature in the feature tree. In these folders one can include the FeatureC++ source code. For details about FeatureC++, see Section 7.1.2. Note: all files for one class need to be named the same way. FeatureC++ source code is completely included in header files, there are no cpp files.

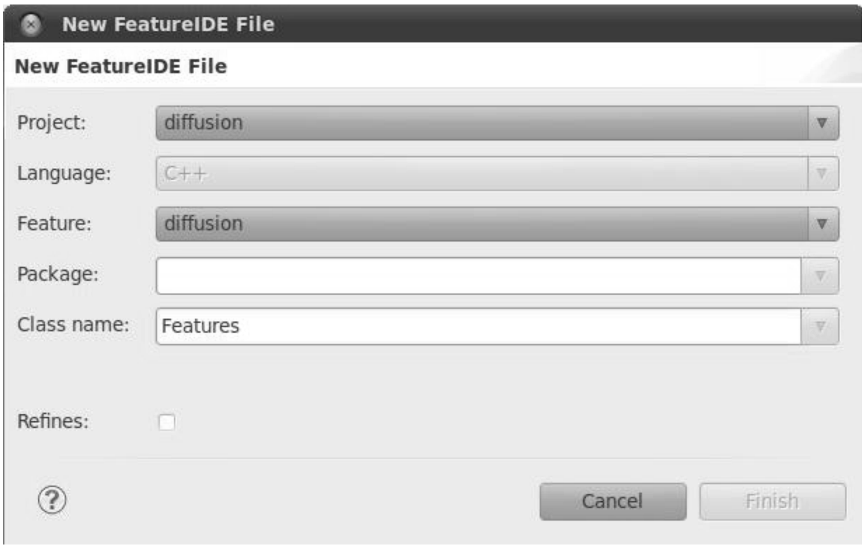
Example: Add FeatureC++ source code for the diffusion system test application

In this example we add FeatureC++ source code for the root feature diffusion and the features mesh, cube and simplex for the DUNE system test application diffusion.

In the package explorer, select subfolder diffusion in the folder features and select New/FeatureIDE File in the context menu.



Add class name Features and click Finish.


New FeatureIDE File

New FeatureIDE File

Project: diffusion

Language: C++

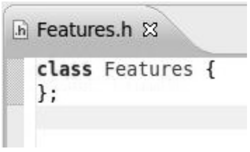
Feature: diffusion

Package:

Class name: Features

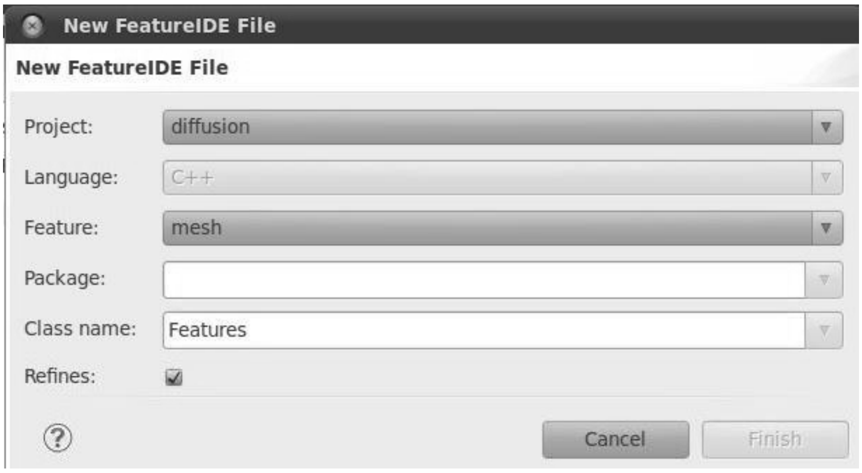
Refines: □

For the root Feature diffusion we leave the class Features empty. We will extend this class in the child features.



```
class Features {
};
```

Add class Features for the subfolder mesh, too. This time, we note that we refine the class Features.



Project: diffusion

Language: C++

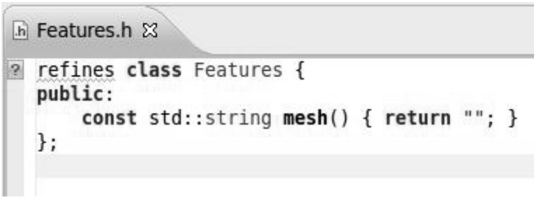
Feature: mesh

Package:

Class name: Features

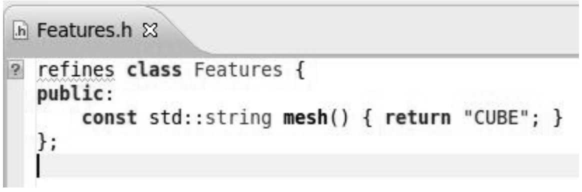
Refines: ☑

For the feature mesh, we extend the class Features to include a method mesh(). Since the value for mesh is first defined by the child features for mesh, this method return an empty string.



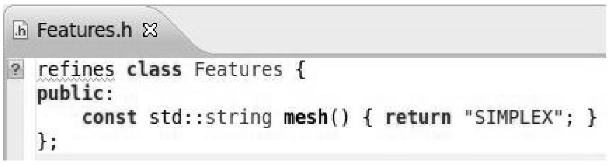
```
refines class Features {
public:
    const std::string mesh() { return ""; }
};
```

Insert the class Features for subfeature cube (refines class Features). For the feature cube, we override the method mesh(). The FeatureC++ source code for cube looks almost exactly the same as for parent feature mesh, but the method mesh() returns "CUBE".



```
refines class Features {
public:
    const std::string mesh() { return "CUBE"; }
};
```

In the same way, insert source code for the subfeature simplex.



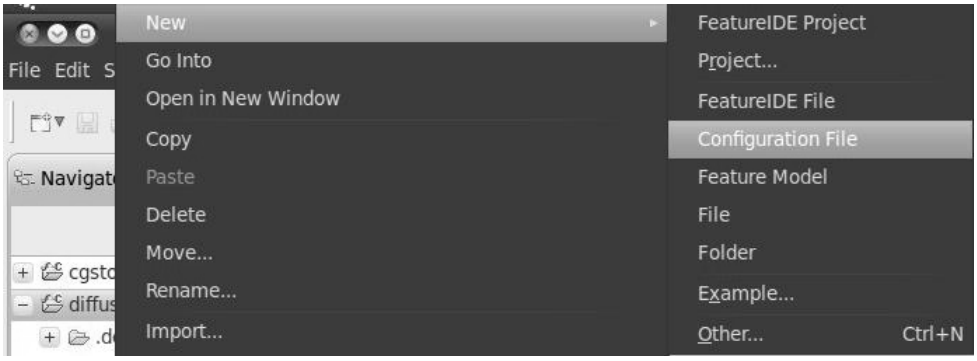
```
refines class Features {
public:
    const std::string mesh() { return "SIMPLEX"; }
};
```

The complete FeatureC++ source code for the DUNE system test application diffusion can be found in Appendix B.

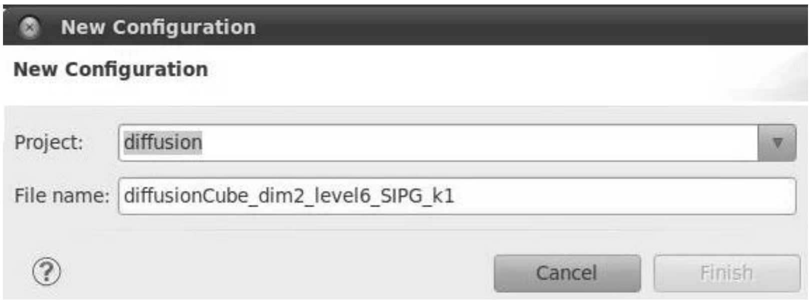
C.7 Create Configurations

The configuration editor in FeatureIDE gets the variability model as input and offers configuration choices. The developer can choose a set of features and save the selection in a configuration file. It is possible to create multiple configurations and one configuration is marked to be the current configuration for which FeatureIDE compiles the feature-oriented source code [76].

Select New/Configuration File in the context menu (it does not make a difference, which file or folder of the project is selected in the project explorer).



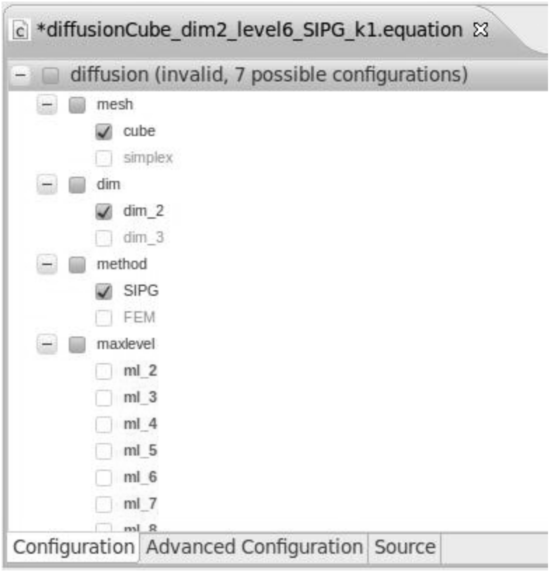
Give a name for the configuration file.



Project: diffusion

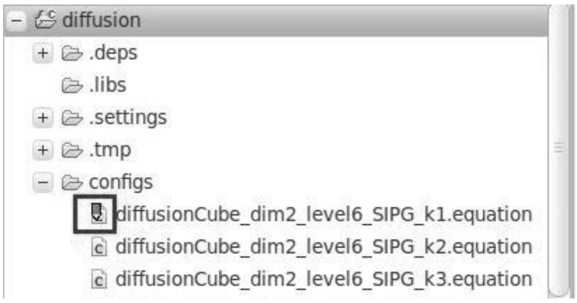
File name: diffusionCube_dim2_level6_SIPG_k1

Now the feature values for the configuration can be chosen. FeatureIDE highlights all possible values and informs the developer if the configuration is not valid.



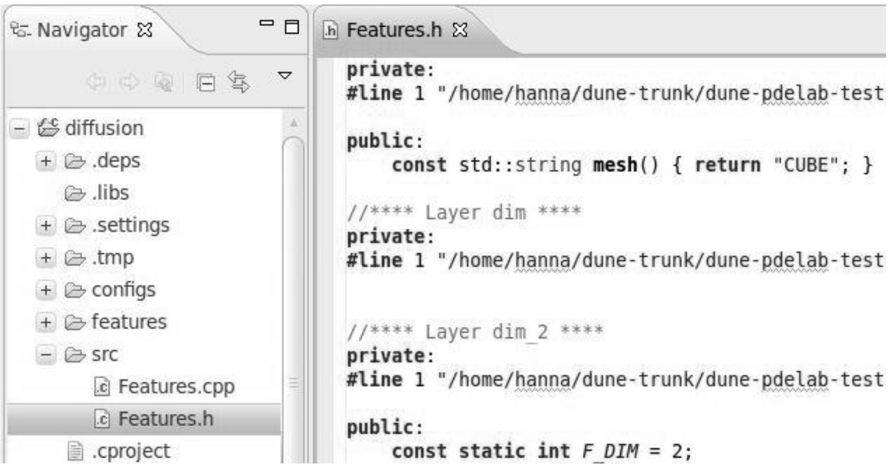
```
*diffusionCube_dim2_level6_SIPG_k1.equation
− □ diffusion (invalid, 7 possible configurations)
    − □ mesh
          ☑ cube
          □ simplex
    − □ dim
          ☑ dim_2
          □ dim_3
    − □ method
          ☑ SIPG
          □ FEM
    − □ maxlevel
          □ ml_2
          □ ml_3
          □ ml_4
          □ ml_5
          □ ml_6
          □ ml_7
```

Different configuration files for one system test application can be created. Currently active configuration is highlighted in the Project Explorer.



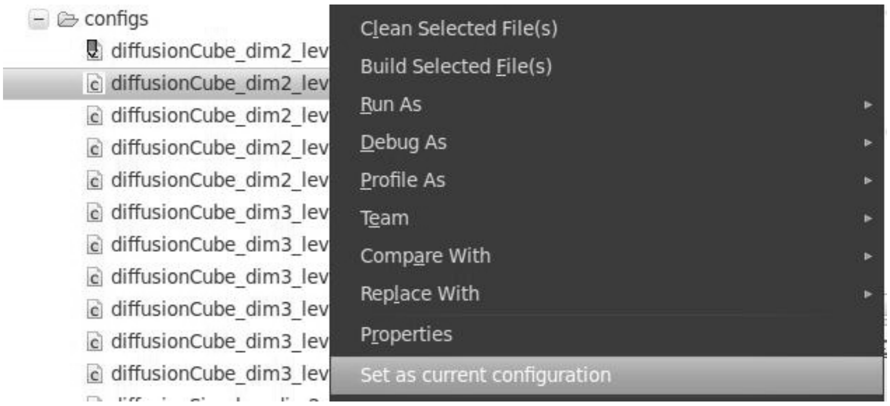
```
− diffusion
    + .deps
      .libs
    + .settings
    + .tmp
    − configs
          diffusionCube_dim2_level6_SIPG_k1.equation
          diffusionCube_dim2_level6_SIPG_k2.equation
          diffusionCube_dim2_level6_SIPG_k3.equation
```

Active configuration means, that FeatureC++ creates executable source code out of the FeatureC++ source code corresponding to the active configuration. After creating one valid configuration, the executable source code can be found in the src folder in Project Explorer.



```
private:
#line 1 "/home/hanna/dune-trunk/dune-pdelab-test

public:
    const std::string mesh() { return "CUBE"; }

//**** Layer dim ****
private:
#line 1 "/home/hanna/dune-trunk/dune-pdelab-test


//**** Layer dim_2 ****
private:
#line 1 "/home/hanna/dune-trunk/dune-pdelab-test

public:
    const static int F_DIM = 2;
```

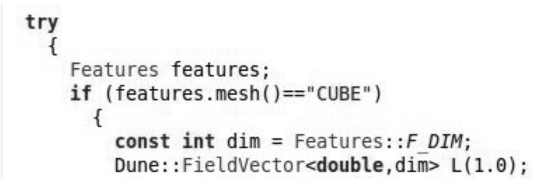
The active configuration can be changed by selecting the favored configuration and selecting Set as current configuration in the context menu.



C.8 Example DUNE: Adjust diffusion.cc Source Code

Now we want to use the FeatureC++ source code for the DUNE system test application diffusion. We need to adjust the diffusion.cc source file in the following way:

- include src/Features.h which is the executable FeatureC++ source code for the currently active feature configuration
- Create an object Features and use it to access the feature values (see code example below).

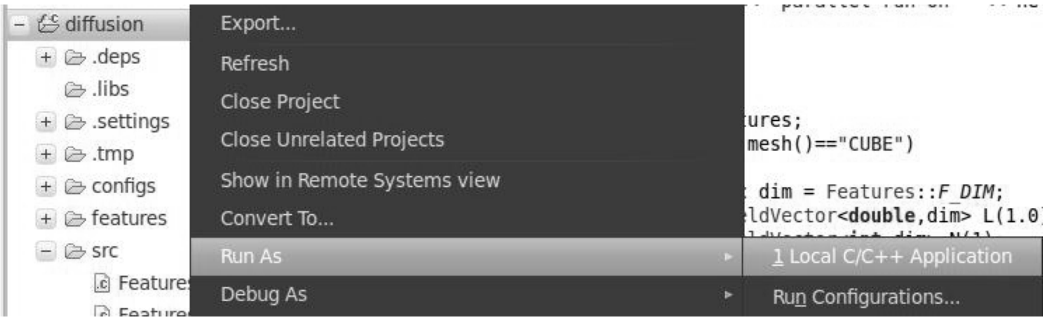


```
try
  {
    Features features;
    if (features.mesh()=="CUBE")
      {
        const int dim = Features::F_DIM;
        Dune::FieldVector<double,dim> L(1.0);
```

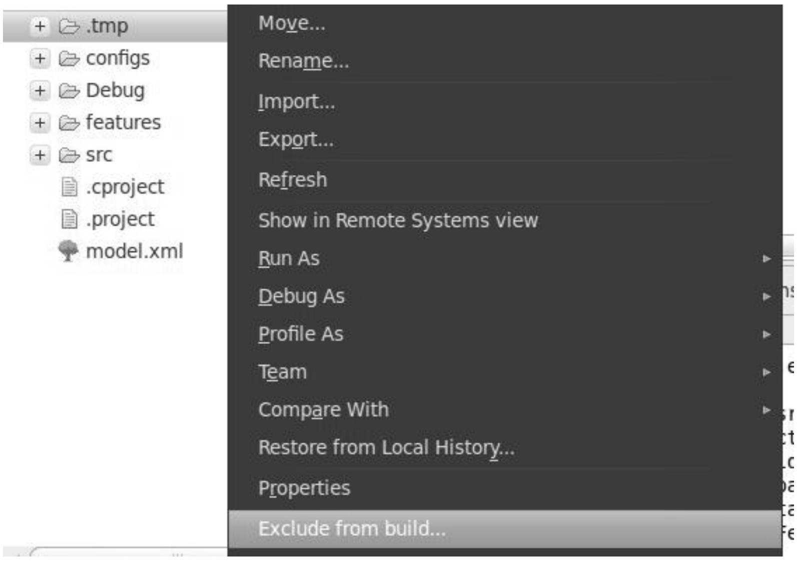
The complete source code for diffusion.cc can be found in Appendix B.

C.9 Run Program

When we set a new active configuration, FeatureIDE automatically compiles the system test application with the corresponding feature configuration (i.e. test case). The system test application can be executed by selecting the project in the Project Explorer and choosing Run As/Local C,C++ Application in the context menu.



Before this works, it may be necessary to exclude some files or folders (like the folder .tmp) from the build, first.



In FeatureIDE, it is possible to trigger the automatic compilation of all possible applications (i.e. all valid configurations). This is only applicable for small variability models. It is also possible to trigger the automatic compilation of all manually created configurations [76].

Bibliography

[1] Ackroyd, K.S., Kinder, S.H., Mant, G.R., Miller, M.C., Ramsdale, C.A., and Stephenson, P.C., "Scientific Software Development at a Research Facility," IEEE Software 25, pp. 44-51, 2008. 4, 15

[2] Arens, T., Hettlich, F., Karpfinger, Ch., Kockelkorn, U., Lichtenegger, K., and Stachel, H., "Mathematik," Spektrum Akademischer Verlag Heidelberg, 2008. 11

[3] Arnord, R.S., "Software Reengineering," IEEE Compuer Society Press, 1994. 44

[4] Avrunin, G.S., Siegel, S.F., and Siegel, A.R., "FiniteState Verification for High Performance Computing," In Proceeding of the 2nd Workshop on Software Engineering for High Performance Systems Applications (SE-HPCS '05), ACM New York, NY, USA, pp. 68-72, 2005.

[5] Babuška, I., "The finite element method with Lagrangian multipliers", In Numerische Mathematik, Vol. 20, Is. 3, pp. 179-192, Springer-Verlag, 1973. 52

[6] Basili, V.R., Caldiera, G., and Rombach, H.D., "The Goal Question Metric Approach," Encyclopedia of Software Engineering, Wiley and Son, pp. 528-532, 1994. 81

[7] Basili, V.R., Carver, J.C., Cruzes, D., Hochstein, L.M., Hollingsworth, J.K., Shull, F., and Zelkowitz, M.V., "Understanding the High-Performance-Computing Community: A Software Engineer's Perspective," Software, IEEE, vol. 25, no. 4, pp. 29-36, 2008. 2, 64

[8] Bastian, P., Blatt, M., Dedner, A., Engwer, C., Klöfkorn, R., Ohlberger, M., and

Sander, O., "A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework," Computing 82, pp. 103-119, 2008. 25, 28

[9] Bastian, P., Blatt, M., Dedner, A., Engwer, C., Klöfkorn, R., Kornhuber, R., Ohlberger, M., and Sander, O., "A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE," Computing 82, pp. 121-138, 2008. 25

[10] Batory, D., "Feature Models, Grammars, and Propositional Formulas," Proc. Int'l Software Product Line Conference (SPLC), Springer, Berlin, Heidelberg, New York, London, pp. 7-20, 2005. 21

[11] Baxter, R., "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. 3, 16

[12] Beck, K., "Test-Driven Development," Addison-Wesley Longman, Amsterdam, 2002. 70

[13] Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., and Wąsowski, A., "A survey of variability modeling in industrial practice," In VaMoS '13 Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, No. 7, ACM New York, NY, USA, 2013. 107

[14] Bertolino, A., Fantechi, A., Gnesi, S. and Lami, G., "Product Line Use Cases: Scenario-Based Specification and Testing of Requirements," In Software Product Lines, Springer Berlin Heidelberg, pp. 425-445, 2006. 24

[15] Blatt, M., and Bastian, P., "The Iterative Solver Template Library," In Applied Parallel Computing, State of the Art in Scientific Computing 4699, pp. 666-675, Springer, 2007. 26

[16] Booth, S., and Henty, D., "Verification strategies for High Performance Computing Software," In Proceeding of the 1st Workshop on Software Engineering for High Performance System (HPCS) Applications, pp. 24-26, 2004. 14

[17] Carnegie Mellon Software Engineering Institute, http://www.sei.cmu.edu/. 11

[18] Bühne, S., and Pohl, K., "Domain Requirement Engineering," In Software Product Line Engineering, Springer Berlin Heidelberg, pp. 193-216, 2005. 49, 50, 52

[19] Carver, J.C., Hochstein, L., Kendall, R.P., Nakamura, T., Zelkowitz, M.V., Basili, V.R., and Post, D.E., "Observations about Software Development for High End Computing," In CTWatch, vol. 2, no. 4A, pp. 33-38, 2006. 3, 13, 64

[20] Carver, J.C., Kendall, R.P., Squires, S.E., and Post, D.E., "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. 3, 64, 66

[21] Carver, J.C., "Report: The Second International Workshop on Software Engineering for CSE," Computing in Science & Engineering, vol. 11, no. 6, pp.14-19, 2009. 3, 26

[22] Chen, L., Babar, M.A., and Ali, N., "Variability management in software product lines: a systematic review," In SPLC '09 Proceedings of the 13th International Software Product Line Conference, Carnegie Mellon University Pittsburgh, PA, USA, pp. 81-90, 2009. 107

[23] Crispin, L., and Gregory, J., "Agile Testing: A practical Guide for Testers and Agile Teams," Addison-Wesley Professional, 2008. 16

[24] Czarnecki, K., and Eisenecker, U.W., "Generative Programming: Methods, Tools, and Applications," ACM/Addison-Wesley, New York, NY, USA, 2000. 21

[25] Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., and Wąsowski, A., "Cool features and tough decisions: a comparison of variability modeling approaches," In VaMoS '12 Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, ACM New York, NY, USA, pp. 173-182, 2012. 107

[26] Davis, F.D., Bagozzi, R.P., and Warshaw, P.R., "User Acceptance of Computer Technology: A Comparison of two Theoretical Models," Management Science, vol. 35, no. 8, pp. 982-1003, 1989. 85

[27] Douglas, J., and Dupont, T., "Interior penalty procedures for elliptic and parabolic Galerkin methods," In Lecture Notes in Physics, Vol. 58, Springer-Verlag, 1976. 53

[28] Dubois, P.F., "Maintaining Correctness in Scientific Programs," Computing in Science & Engineering, vol. 7, no. 3, pp. 80-85, 2005. 13, 15, 16, 64, 177

[29] Engström, E., and Runeson, P., "Decision Support for Test Management and Scope Selection in a Software Product Line Context," Fourth International Conference

on Software Testing, Verification and Validation Workshops (ICSTW), pp.262-265, 2011.

[30] Eriksson, K., Estep, D., Hansbo, P., and Johnson C., "Computational Differential Equations," Campridge University Press, 1996. 27

[31] FeatureC++ internet page http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/. 111, 112, 148

[32] FeatureIDE internet page http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/. 110

[33] Freeman, S.M., Clune, T.L., and Burns III, R.W., "Latent Risks and Dangers in the State of Climate Model Software Develoment," Spektrum Akademischer Verlag Heidelberg, 2008. 2

[34] Hannay, J.E., Langtangen, H.P., MacLeod, C., Pfahl, D., Singer, J., and Wilson, G., "How Do Scientists Develop and Use Scientific Software?," SECSE '09 ICSE Workshop on Software Engineering for Computational Science and Engineering, pp. 1-8, 2009. 15

[35] Heider, W., Rabiser, R., Grünbacher, P., and Lettner, D., "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. 22

[36] Heroux, M.A., "Improving the Development Process for CSE Software," PDP '07, 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing, pp. 11-17, 2007. 16

[37] Heroux, M.A. and Willenbring, J.M., "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. 16

[38] Hook, D., and Kelly, D., "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. 3, 13, 64, 177

[39] Hubaux, A., Tun, T.T., and Heymans, P., "Separation of concerns in feature diagram languages: A systematic survey," ACM Computing Surveys (CSUR), vol. 45, no. 4,

ACM New York, NY, USA, 2013. 107

[40] IEEE Standard for Software Reviews, IEEE Std 1028-1997. 15

[41] International Workshop on Software Engineering for Computational Science and Engineering, http://secse13.cs.ua.edu/ICSE/index.htm 10

[42] Johansen, M.F., Haugen, Ø., and Fleurey, F., "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. 25

[43] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., and Peterson, A.S., "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, 1990. 20

[44] Kelly, D., and Sanders, R., "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. 15, 72, 80, 87, 104

[45] Kelly, D., and Sanders, R., "The Challenge of Testing Scientific Software," CAST 2008, Proc Conference of the Association of Software Testing, Toronto, Canada, 2008. 3, 64

[46] Kelly, D. and Smith, S., "2nd CASCON Workshop on Software Engineering for Science," CASCON 2009, pp. 345-347, 2009. 64

[47] Kelly, D., Smith, S., and Meng, N. "Software Engineering for Scientists," Computing in Science and Engineering, vol. 13, no. 5, pp. 7-11, 2011. 3, 64

[48] Kim, C.H.P., Batory, D., and Khurshid, S., "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. 24

[49] Kuhn, D.R., Wallace, D.R., and Gallo, A.M., "Software fault interactions and implications for software testing," IEEE Transactions on Software Engineering, vol. 30, no. 6, pp. 418-421, 2004. 25

[50] Machado, I.C., "RiPLE-TE: A software product lines testing process," M.Sc. Dissertation, CIn - Informatics Center, UFPE - Federal University of Pernambuco, Recife-PE, Brazil, 2010. 23, 69

[51] Machado, I.C., da M.S. Neto, P.A., Almeida, E.S., and de Lemos Meira, S.R., "RiPLE-TE: A Process for Testing Software Product Lines," SEKE, Knowledge Systems Institute Graduate School, pp. 711-716, 2011. 22, 23, 31, 106, 177

[52] McGregor, J., "Testing a Software Product Line," Technical Report, CMU/SEI-2001-TR-022, ESC-TR-2001-022, 2001. 25

[53] Morris, C., "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. 72

[54] Nebut, C., Le Traon, Y., and Jezequel, J.-M., "System Testing of Product Lines: From Requirements to Test Cases," In Software Product Lines, Springer Berlin Heidelberg, pp. 447-477, 2006. 24

[55] Neely, R., "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. 16

[56] da M. S. Neto, P.A., Machado, I.C., Cavalcanti, Y.C., Almeida, E.S., Garcia, V.C., and de Lemos Meira, S.R., "A Regression Testing Approach for Software Product Lines Architectures," Fourth Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), pp. 41-50, 2010. 22, 24, 106

[57] Obbink, H., Pohl, K., Kang, K.C., Kim, M., Lee, J. and Kim, B., "Feature-Oriented Re-engineering of Legacy Systems into Product Line Assets - a Case Study," In Software Product Lines Springer Berlin / Heidelberg, pp. 45-56, 2005. 35

[58] Oberkampf, W.L., Trucano, T.G., and Hirsch, C., "Verification, Validation, and Predictive Capability in Computational Engineering and Physics", Applied Mechanics Rev., pp. 345-384, 2004. 13, 14, 16

[59] Pasetti, A., "Software frameworks and embedded control systems," Springer-Verlag, 2002. 3, 10

[60] Pohl, K., Böckle, G., and Linden, F., "A Framework for Software Product Line Engineering," In Software Product Line Engineering, Springer Berlin Heidelberg, pp. 19-38, 2005. 17, 18, 45

[61] Pohl, K., Böckle, G., Linden, F., and Lauenroth, K., "Principles of Variability," In Software Product Line Engineering, Springer Berlin Heidelberg, pp. 57-88, 2005. 19, 177

[62] Pohl, K., Böckle, G., Linden, F., and Lauenroth, K., "Software Product Line Engineering - Foundations, Principles, and Techniques," Springer Berlin Heidelberg, 2005. 4, 11, 17, 18

[63] Pohl, K., and Reuys, A., "Application Testing," In Software Product Line Engineering, Springer Berlin Heidelberg, pp. 355-370, 2005. 17, 18, 37

[64] Pohl, K. and Reuys, A., "Domain Testing," In Software Product Line Engineering, Springer Berlin Heidelberg, pp. 257-284, 2005. 11, 36, 37, 38, 41, 42

[65] Remmel, H., Paech, B., Engwer, C. and Bastian, P., "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. 86

[66] Reuys, A., Reis, S., Kamsties, E., and Pohl, K., "The ScenTED Method for Testing Software Product Lines", in Software Product Lines, Springer Berlin Heidelberg, pp. 479-520, 2006. 24, 37

[67] Roache, P.J., "Building PDE codes to be verifiable and validatable," Computing in Science and Engineering, Vol. 6, No. 5, pp. 30-38, 2004. 14

[68] Runeson, P., Höst, M., Rainer, A., and Regnell, B., "Case Study Research in Software Engineering," John Wiley & Sons, Inc., Hoboken, NJ, USA, 2012. 80, 81, 104, 105, 106, 107

[69] Sayre, K. and Poore, J., "Automated Testing of Generic Computational Science Libraries," In Proceedings of the 40th International COnference on System Sciences (HICSS), Waikoloa, Hawaii, USA, pp. 277c, 2007.

[70] Segal, J., "Models of scientific software development," Proceeding of the 2008 Workshop Software Engineering In Computational Science and Engineering, 2008. 16

[71] Society for Industrial and Applied Mathematics, www.siam.org/. 9

[72] Software Engineering Body of Knowledge, IEEE Computer Society, www.swebok.org/ . 9, 10, 15

[73] Stricker, V., Metzger, A., and Pohl, K., "Avoiding redundant testing in application engineering," In Software Product Lines: Going Beyond, Springer Berlin Heidelberg, pp. 226-240, 2010. 24

[74] Tevanlinna, A., Taina, and Kauppinen, R., "Product family testing: a survay," ACM SIGSOFT Software Engineering Notes, vol. 29, no. 2, pp. 1-6, 2004. 37

[75] Thüm, T., Kästner, C., Erdweg, S. and Siegmund, N., "Abstract Features in Feature Modeling," Proc. Int'l Software Product Line Conference (SPLC), IEEE, Washington, DC, USA, pp. 191-200, 2011. 21

[76] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T., "FeatureIDE: An Extensible Framework for Feature-Oriented Software Development," Science of Computer Programming, vol. 79, no. 0, pp. 70-85, 2014. 20, 21, 52, 110, 111, 114, 161, 164, 168, 177

[77] Vigder, M., "End-User Software Development in a Scientific Organization," In Proceedings of the 2009 ICSE Workshop on Software Engineering Foundations for End User Programming, IEEE Computer Society, pp. 15-19, 2009. 2

[78] Xie, T., and Notkin, D., "Checking inside the black box: regression testing by comparing value spectra," IEEE Transactions on Software Engineering, vol.31, no.10, pp. 869- 883, 2005. 10, 16

[79] Yin, R.K., "Case Study Research: Design and Methods," 3rd edition. SAGE Publications, 2003. 12

[80] Yoshimura, K., Ganesan, D. and Muthig, D., "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, 2006. 35, 42

[81] Yu, W., and Smith, S., "Reusability of FEA software: A program family approach," In Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering, IEEE Computer Society, pp. 43-50, 2009. 34, 42

List of Figures

List of Tables