Noname manuscript No.
(will be inserted by the editor)

Lightweight, Semi-Automatic Variability Extraction: A Case Study on Scientific Computing

Alexander Grebhahn · Christian Kaltenecker · Christian Engwer · Norbert Siegmund · Sven Apel

Received: date / Accepted: date

Abstract In scientific computing, researchers often use feature-rich software frameworks to simulate physical, chemical, and biological processes. Commonly, researchers follow a *clone-and-own approach*: Copying the code of an existing, similar simulation and adapting it to the new simulation scenario. In this process, a user has to select suitable artifacts (e.g., classes) from the given framework and replaces the existing artifacts from the cloned simulation. This manual process incurs substantial effort and cost as scientific frameworks are complex and provide large numbers of artifacts. To support researchers in this area, we propose a lightweight API-based analysis approach, called VORM, that recommends appropriate artifacts as possible alternatives for replacing given artifacts. Such alternative artifacts can speed up performance of the simulation or make it amenable to other use cases, without modifying the overall structure of the simulation. We evaluate the practicality of VORM—especially, as it is very lightweight but possibly imprecise—by means of a case study on the DUNE numerics framework and two simulations from the realm of physical simulations. Specifically, we compare the recommendations

Alexander Grebhahn
adesso SE
E-mail: Alexander.Grebhahn@adesso.de

Christian Kaltenecker
Saarland University, Saarland Informatics Campus
E-mail: kaltenec@cs.uni-saarland.de

Christian Engwer
University of Münster
E-mail: christian.engwer@uni-muenster.de

Norbert Siegmund
Leipzig University
E-mail: norbert.siegmund@uni-leipzig.de

Sven Apel
Saarland University, Saarland Informatics Campus
E-mail: apel@cs.uni-saarland.de

by VORM with recommendations by a domain expert (a developer of DUNE). VORM recommended 34 out of the 37 artifacts proposed by the expert. In addition, it recommended 2 artifacts that are applicable but have been missed by the expert and 32 artifacts not recommended by the expert, which however are still applicable in the simulation scenario with slight modifications. Diving deeper into the results, we identified an undiscovered bug and an inconsistency in DUNE, which corroborates the usefulness of VORM.

Keywords software variability; configuration; variability extraction; variability analysis

1 Introduction

In scientific computing, natural and artificial processes are simulated in a computer-aided way. Examples of these processes include physical, geological, and biological processes, such as convection, blood flow, root–soil interaction, and the large deformation contact problem. One efficient way to simulate these processes is by modeling them as partial differential equations and by applying grid-based methods to solve these equations. The idea is to discretize the typically continuous domain into a finite grid of elements, volumes, or differences. For example, instead of computing or keeping track of the temperature in a solid at any real point, one considers only a set of points arranged by a grid structure. At this general stage, we refer the reader to standard text books on numerical simulations, such as these by Braess (2007) or Knabner and Angerman (2003).

Mathematical properties of the processes to simulate constrain the choices of the discretization techniques and, in turn, the choice of the discretization strategy constrains the choices of the grid-based method one can use for solving the equations (Grebhahn et al., 2017). To support researchers in selecting optimal data structures and algorithms when writing their simulation code, there are different frameworks and toolboxes available, such as DUNE (Bastian et al., 2008b,a), UG4 (Vogel et al., 2013), HYPRE (Falgout and Yang, 2002), and FEniCS (Alnæs et al., 2015). The key benefit of these frameworks is that they provide reusable implementation artifacts, which substantially reduce the implementation effort of new simulation code (Remmel, 2014). These implementation artifacts can be classes, structs, or enum values. For example, there might be different implementations of a grid, describing the geometric properties of the data or different solvers that offer methods for solving the system of equations represented by the elements of a grid.

For illustration, we show an excerpt of a simulation code for the convection–diffusion problem in Listing 1, which can be used to simulate the transport in a fluid or porous media, for instance. In Lines 3–6, a two-dimensional grid is initialized, which is used later to discretize the partial differential equation and its solution on entities of the grid. The LeafGridView object is necessary for iterating over all entities to perform calculations on each of them. In Line 9, LeafGridView is passed as a template parameter to ProblemA containing the

```
...
// basic declaration and configuration
const int dim = 2;
using Grid = Dune::YaspGrid<dim>;
using GV = Grid::LeafGridView;
const GV& gv = grid->leafGridView();
...
// construct model problem
using Problem = Dune::SPL::ProblemA<GV, double>;
...
```

Listing 1 Excerpt from a simulation code solving the convection-diffusion problem.

convection–diffusion model problem. Different implementation choices represent options and alternatives to accomplish similar objectives in a simulation. Depending on the mathematical properties and assumptions of the simulated process, some of these choices are invalid and some are more suitable than others. For example, Grebhahn et al. (2017) have shown that, depending on the discretization and the characteristics of the problem to solve, only a subset of existing strategies can be used to simulate the process efficiently.

The main obstacle of using scientific frameworks is their complexity and high variability: They provide a large number of artifacts that are applicable only for certain scenarios (e.g., DUNE provides 11 artifacts to describe the geometry of the computational domain). Identifying suitable artifacts for a specific scenario requires knowledge of (i) the mathematical properties of the numerical methods, (ii) the physical properties of the process to be simulated, and (iii) the set of available artifacts provided by the framework that can be used for the specific purpose. For example, Grebhahn et al. (2017) discuss in detail the variability and constraints when solving partial differential equations arising in the domain of porous media flows. In the example of Listing 1, the structured grid as implemented by `YaspGrid<dim>` is usually a good choice to simulate flow and transport in porous media, while for industrial applications with complicated geometries, an unstructured grid such as `UGGrid<dim>` is more appropriate.

Typically, researchers who conduct simulations often have no professional background in software engineering (Remmel, 2014). Hence, they fall back to a *clone-and-own approach*, in which they copy code of existing simulations and adapt it to the new simulation scenario. However, the artifacts that are used in the copied code might not be the best choice for the newly considered simulation. To support researchers in adopting real-world scientific simulations, we propose a lightweight, semi-automatic, API-based analysis approach, VORM*. VORM analyzes the code and the documentation of the given framework to identify artifacts of the framework that can be used in an application (a scientific simulation, in our case) instead of a currently used artifact. In our example

*VORM stands for extracting the **V**ariability based on the d**O**cumentation to identify **R**elations among **M**odules.

of Listing 1, VORM might suggest other grids that might be more suitable for the considered simulation (e.g., UGGrid<dim> instead of YaspGrid<dim>, for complicated geometries). In general, the set of alternative artifacts can be reviewed by a user to remove false positives or to reduce the number of considered artifacts. This procedure can be repeated if multiple artifacts need to be replaced.

After selecting a set of suitable alternative artifacts, VORM can generate one running configuration of the simulation for each alternative artifact or combinations thereof. These configurations can then, later on, be used to assess resource usage, find errors, or to identify the best combination of artifacts with respect to execution time. For example, a researcher who is interested in an alternative solver implementation for a given problem can generate a set of configurations of the simulation, where each configuration uses a different solver type (e.g., Jacobi and Richardson), and then benchmark the different variants to identify the performance-optimal solver for the specific use case.

It is important to note that VORM is a *lightweight* approach that uses only information provided by the API of the framework for identifying alternative artifacts and thus does not consider information from program analysis or runtime. Without this information at hand, the recommendations of VORM are necessarily imprecise, which is a prerequisite for making VORM scale to real-world code.

In a real-world case study, we apply VORM to DUNE, which is a scientific computing framework offering a large number of strategies to solve partial differential equations using grid-based methods.[†] Our overarching question is:

Is a lightweight, semi-automatic approach, as pursued in VORM, able to identify meaningful alternative artifacts to a given artifact for specific application scenarios that are written using a real-world scientific framework, such as DUNE?

In more detail, we aim at answering the following research objectives:

RQ$_1$: Can VORM identify the same alternative artifacts to a given artifact that have been proposed by a domain expert?

RQ$_2$: Can VORM discover relations among artifacts of the framework or inconsistencies within the framework not previously known to the domain expert?

To answer RQ$_1$ and RQ$_2$, we consider two simulations written using DUNE: LINEARSOLVER and ELLIPTICPROBLEM. From each simulation, we select 8 artifacts used from DUNE for which alternatives shall be found. Then, a framework developer (domain expert) of DUNE[‡] proposes a set of alternative artifacts for each of the 8 artifacts. We compare these manually selected alternative artifacts with the ones automatically identified by VORM.

[†] https://www.dune-project.org/

[‡] Who is our domain expert and also one of our co-authors.

Our case study shows that VORM is able to automatically identify a large number of suitable alternatives that have also been proposed by the domain expert. Note that researchers using the framework might not be as knowledgeable as a developer of DUNE, which provides us already with a tough baseline. Overall, VORM identifies 94 artifacts to be alternatives for the 8 original artifacts used in the two subject simulations. Of these alternative artifacts, 34 were also proposed by the domain expert. From the remaining 60 alternative methods, 31 are alternatives that can be used in the simulation with only slight modifications on the simulation (as confirmed by the domain expert); 2 artifacts were missed by the domain expert but are still applicable; 28 artifacts are false positives. After having a deeper look at the artifacts not identified by VORM but proposed by the domain expert, we identified a bug and an inconsistency in DUNE, which were not known beforehand. These findings underline the usefulness of VORM in practice.

Based on these results, we conclude that VORM's lightweight, API-based approach (1) can identify meaningful alternatives to given artifacts, (2) can identify artifacts that can be used in a slightly different scenario, and (3) can unveil bugs or inconsistencies in the framework to a developer. While a more sophisticated approach that also considers information from program analysis or runtime might increase accuracy of the recommendations, it is certainly much more heavy-weight.

2 Background

In this section, we explain the fundamental concepts underlying VORM. In Section 2.1, we give an introduction into the DUNE framework, which we use as case study for the evaluation of VORM. In Section 2.2, we introduce the basic terminology that we use throughout the article.

2.1 DUNE

DUNE, which stands for "**D**istributed and **U**nified **N**umerics **E**nvironment", is a scientific framework, written in C++, which is being developed to solve partial differential equations with grid-based methods. DUNE provides reusable implementation artifacts for a large number of different mathematical concepts. They can be combined in various ways to solve discretized partial differential equations from different domains. For example, DUNE provides 11 grid implementations to describe the geometry of structured and unstructured computational domains and 34 implementations to describe the finite element space basis of a simulation.

The DUNE development process started in 2002. It has a modular structure (see Figure 1) with five core modules providing basic functionality. A range of specialized features and higher-level functionality are available as extension modules, such as the flexible Finite Element Method abstraction of

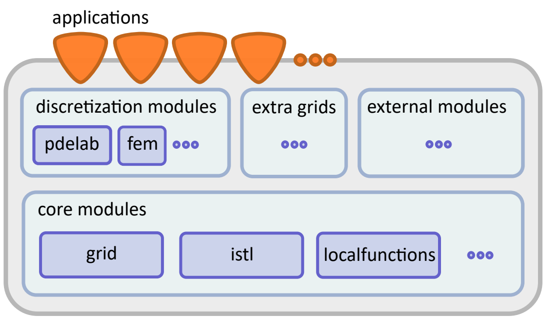

Fig. 1 Structure of the DUNE framework. Five core modules build the foundation, a range of higher level modules provide a rich body of functionality.

DUNE-PDELAB. The core modules have a combined size of approximately 160 thousand lines of C++ code; the discretization module DUNE-PDELAB, including all ten dependent modules, has a size of over 340 thousand lines. The modules we consider offer 3 022 different artifacts that can be used in simulations to solve or specify certain parts of the simulation.[§]

Overall, DUNE implements the following basic concepts (Bastian et al., 2010): The (i) separation of data structures and algorithms, (ii) use of generic programming techniques, and (iii) reuse of existing finite element software.

Similar to the Standard Template Library of C++, DUNE separates data structures and algorithms, offering developers the possibility of using a suitable combination for the considered simulation. A developer might simply change the used algorithm without modifying the whole simulation. For each choice, DUNE offers different implementation artifacts.

DUNE makes heavy use of polymorphism and template meta-programming offering the possibility to specialize classes with template parameters without sacrificing performance (Bastian et al., 2008a). The arguments passed to a template class can again be artifacts provided by DUNE. Last, DUNE offers lightweight interfaces to support the use of existing libraries. For further information about the framework, we refer to Bastian et al. (2008a,b).

2.2 Terminology and Formalization

Artifacts. As stated previously, mathematical frameworks provide various artifacts for different mathematical concepts. This includes, for example, different implementations of a grid, which is used to represent the data of a simulation, or different solver implementations, which can be used to solve a system of equations. We denote the artifacts that a framework offers with the set $\mathcal{A}$. Typically, a simulation $s \in \mathcal{S}$ uses only a subset $\mathcal{A}_s \subset \mathcal{A}$ of artifacts provided by the framework. The simulation code is entirely controlled by the framework and has no further configuration options or runtime inputs. While technically

[§]We use version 2.6 of the DUNE framework.

```
1 ...
2 // VarPoint(Finite element map; using FEM;
3 //           Dune::PDELab::QkLocalFiniteElementMap<GV, GV::ctype, Real, degree>)
4 using FEM = Dune::PDELab::QkLocalFiniteElementMap<GV, GV::ctype, Real, degree>;
5 ...
6 // VarPoint(ISTLSolverBackend; using LS; Dune::PDELab::ISTLBackend_SEQ_CG_ILU0)
7 using LS = DUNE::PDELab::ISTLBackend_SEQ_CG_ILU0;
8 ...
```

Listing 2 Excerpt from a simulation with two variation points.

artifacts can include class definitions, struct definitions, structs, enumeration values, etc., in practice (in DUNE) most are modelled as classes using either static (parametric) or dynamic (subtype) polymorphism.

Variation points. The artifacts provided by a framework can be used, instantiated, and combined to create a simulation. Typically, a developer is interested in replacing only a subset of the artifacts (e.g., data structures or algorithms) used in her simulation code, which we call *variation points*. A variation point $v \in \mathcal{V}$ is a triple $(id, decl, a_0)$ consisting of (1) an identifier (id), (2) the declaration used in the variation point $(decl)$, and (3) the original artifact to which the variation point refers $(a_0 \in \mathcal{A}_s)$. For simplicity, we assume that the set of artifacts of a simulation s that are variable equals $\mathcal{A}_s$.

For illustration, we show a small excerpt of a simulation written based on DUNE in Listing 2. The excerpt contains two variation points (Line 4 and Line 7). The first variation point is

$$\big(\text{"Finite element map"}, \mathsf{using\ FEM}, \mathsf{Dune::PDELab::QkLocalFiniteElementMap}\big).$$

For brevity, we write Qk referring to its artifact. The artifact has four template parameters: <GV, GV::ctype, Real, degree>, which we simplify for further analysis (see Section 3, for details): <GV, D, R, k>.

Alternatives. Determining the alternatives of a given artifact relies on the subtype relationship (<:) between artifacts as defined in the framework. Two artifacts are in subtype relation if (1) there is a declared subclass relation in the framework, (2) they are identical, or (3) they are transitively related:

$$a <: a \qquad \frac{a <: a' \quad a' <: a''}{a <: a''} \qquad \frac{\mathsf{subclass}(a, a')}{a <: a'}$$

Based on the subtype relation, function alt determines the set of artifacts that are alternative to a given artifact. An artifact a is an alternative to another artifact a' if the two artifacts have a common superclass or if a provides, at least, all public methods of a'.

$$\begin{aligned}\mathsf{alt}(a_0) = \Big\{ a \mid\ & \big(\exists\, a_s \in \mathcal{A}.\ a_0 <: a_s \wedge a <: a_s\big) \\ \vee\ & \big(\mathsf{pub}(a_0) \subseteq \mathsf{pub}(a)\big) \\ \vee\ & \big(\exists\, a_e \in \mathcal{A}.\ a_0 \in \mathsf{enumval}(a_e) \wedge a \in \mathsf{enumval}(a_e)\big) \Big\}\end{aligned}$$

where pub returns a set of all public methods of an artifact and enumval returns the set of all enum values of an enum (empty set if the given artifact is no enum). Note that the first condition of the set comprehension relies on nominal subtyping as defined by the inheritance hierarchy of the framework. The second condition implements structural subtyping by matching the public interfaces of two artifacts.

For illustration, consider the results for the two variation points of Listing 2. For the first variation point, we would obtain:

$$\mathsf{alt}\big(\,\mathsf{Qk{<}GV,D,R,k{>}}\,\big) = \big\{\,\mathsf{Qk{<}GV,D,R,k{>}},\ \mathsf{Pk{<}GV,D,R,k{>}}\,\big\}$$

The alternative PkLocalFiniteElementMap (not shown in the code), abbreviated with Pk, is selected because it provides all public methods of Qk. Pk and Qk do not have a common superclass, though. In this example, the original artifact as well as its alternative provide a set of template parameters. The (names of) template parameters of Pk were adjusted such that they match, which we describe in Section 3.

For the second variation point, we would obtain:

$$\begin{aligned}&\mathsf{alt}\big(\,\mathsf{ISTLBackend\_SEQ\_CG\_ILU0}\,\big) = \\ &\quad \big\{\,\mathsf{ISTLBackend\_SEQ\_CG\_ILU0},\ \mathsf{ISTLBackend\_SEQ\_LOOP\_Jac}\,\big\}\end{aligned}$$

In this example, ISTLBackend_SEQ_LOOP_Jac is selected as alternative for ISTLBackend_SEQ_CG_ILU0 because both classes have a common superclass.

Configurations. Based on the identified alternatives, our approach generates a set of *configurations* for a given simulation, for further analysis. A configuration is a fully functional variant of the original simulation using a unique combination of alternative artifacts identified by function alt.

To obtain the set of all possible configurations of a simulation s, we compute the cartesian product over all sets of alternative artifacts:

$$\mathsf{configs}(s) = \big\{\, \{a_1, \ldots, a_k\} \mid a_1 \in \mathsf{alt}(a^1), \ldots, a_k \in \mathsf{alt}(a^k) \wedge a^1, \ldots, a^k \in \mathcal{A}_s \big\}$$

Applying function configs to the example of Listing 2 yields a set of four configurations:

$$\begin{aligned}\mathsf{configs}(s) = \Big\{ & \big\{\,\mathsf{Qk{<}GV,D,R,k{>}},\ \mathsf{ISTLBackend\_SEQ\_CG\_ILU0}\,\big\}, \\ & \big\{\,\mathsf{Qk{<}GV,D,R,k{>}},\ \mathsf{ISTLBackend\_SEQ\_LOOP\_Jac}\,\big\}, \\ & \big\{\,\mathsf{Pk{<}GV,D,R,k{>}},\ \mathsf{ISTLBackend\_SEQ\_CG\_ILU0}\,\big\}, \\ & \big\{\,\mathsf{Pk{<}GV,D,R,k{>}},\ \mathsf{ISTLBackend\_SEQ\_LOOP\_Jac}\,\big\} \Big\}\end{aligned}$$

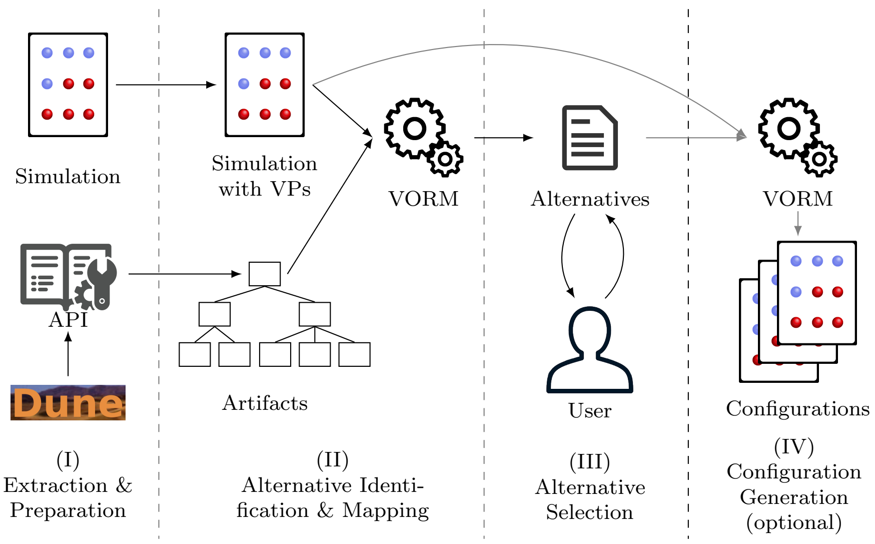
Simulation
Simulation with VPs
VORM
Alternatives
VORM
API
Dune
Artifacts
User
Configurations
(I) Extraction & Preparation
(II) Alternative Identification & Mapping
(III) Alternative Selection
(IV) Configuration Generation (optional)

Fig. 2 Workflow of VORM. In Step I, all relevant information from the framework is extracted by parsing its API and the simulation is extended with one variation point for each artifact the developer is interested in alternatives for. In Step II, the variation points including their original artifacts are extracted from the simulation and alternative artifacts are identified in the framework. In Step III, the developer of the simulation can select a subset of the identified alternatives and extend the alternatives with additional information, if necessary. In Step IV, which is optional, one configuration for each combination of the alternative artifacts is generated.

Since both of the variation points, Finite Element Map and ISTLSolverBackend, have two alternative artifacts each, the cartesian product of these sets of alternatives results in four configurations.

3 Approach

We present the workflow of VORM in Figure 2. Overall, VORM consists of three main steps and optionally a fourth one: (I) extraction and preparation, (II) identification of alternatives and mapping of parameters, (III) selection of alternatives, and (IV) generation of configurations. Next, we describe these four steps in detail.

In the extraction of the artifacts in Step I, we are interested in the following information: (I) the methods provided by the artifacts, (II) the inheritance hierarchy among the artifacts, and (III) template parameters of the artifacts. To derive this information, we include the classes and the signatures (i.e., return value, name, parameter types) of their public methods by analyzing the API of the framework (in our case study, we use Doxygen for this task; cf. Section 6). The simulation that uses the framework has to be extended manually by annotating the one variation points, for which alternatives are desired. The annotations of variation points are given in the form of comments,

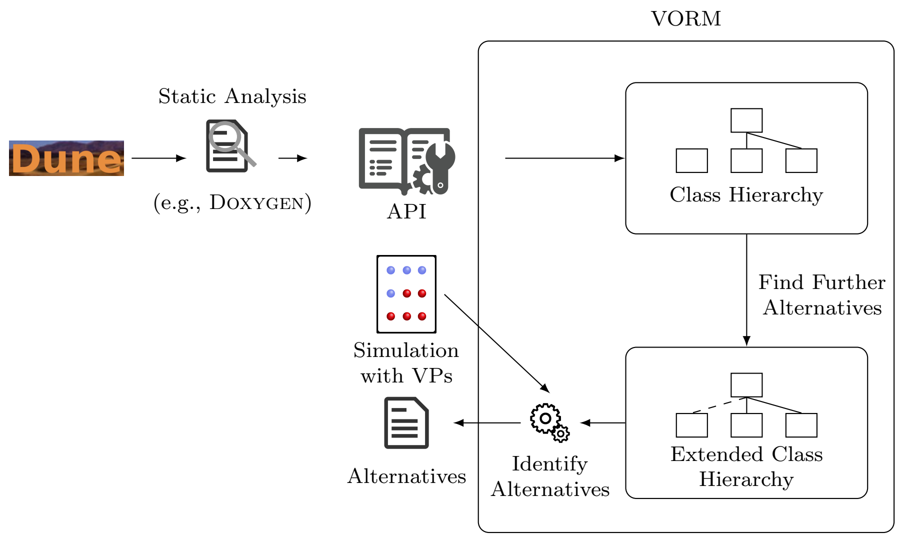

Fig. 3 A more detailed overview of Step I and Step II of the workflow. VORM receives an XML file with a description of DUNE's API and constructs the corresponding class hierarchy. Therein, VORM searches for further classes that may be alternatives to each other and extends the class hierarchy accordingly. Finally, VORM uses information on variability points from the given simulation to find suitable alternatives in the extended class hierarchy.

as illustrated in Lines 2–3 and 6 of Listing 2. Overall, we aim at keeping the extraction process as general as possible to be able to support different frameworks and different programming languages.

In Step II, VORM identifies the set of suitable alternative artifacts for every given original artifact declared as variation point. To this end, we extract the original artifacts used in the simulation by matching the variation points, and we use the extracted information to find suitable alternatives by considering the rules describes in Section 2.2.

In Figure 3, we zoom into the workflow of Step I and II: VORM receives an XML file with API information (which can be generated by Doxygen or any other API generator, such as CastXML) and parses the class hierarchy from the API description. Then, VORM searches for suitable artifacts in the class hierarchy to identify possible alternatives for existing artifacts, as described in Section 2.2. After identifying alternative artifacts, VORM uses the annotated variation points from the simulation code to identify specific alternatives for the requested artifacts in the extended class hierarchy.

It might be necessary to incorporate simulation-specific information from the original artifacts in their alternatives. For example, DUNE makes extensive use of template meta-programming by offering the user the possibility of parameterizing artifacts (see Line 4 in Listing 2, where the used artifact, Finite element map, has 4 template parameters). To support template parameters with VORM, we have to use heuristics to replace the template parameters of the alternative artifacts with the template parameters of the original artifact.

To this end, VORM establishes a mapping between parameters of the original artifacts and parameters of the alternative artifacts. Pursuing a lightweight approach, we consider information about the labels of the parameters. For example, the labels in the API of DUNE for the template parameters of `Qk` are <`GV`, `D`, `R`, `k`>. For one of the alternative artifacts, `Pk`, the labels are the same <`GV`, `D`, `R`, `k`>. Since the template parameters of these two artifacts have the same labels, VORM can use parameter values of `Qk` for `Pk`. For template parameters that do not have a counterpart in the original artifact (i.e., a template parameter with the same label), we leave the label untouched.

In Step III, we present a list of all alternative artifacts in a compact and human-readable representation to the user. This way, developers are able to remove false positives based on domain knowledge or can replace template parameters of the alternative artifacts with suitable ones, if necessary. This is necessary when, for example, an alternative artifact has more template parameters than the original artifact.

Last, in the optional Step IV, one can generate configurations of the simulation code; each configuration applies a different set of alternative artifacts revised in Step III. That is, in each configuration, the original artifacts of the variation points have been replaced with alternative artifacts. Therefore, we consider the list of all alternative artifacts and create all combinations of artifacts proposed for different variation points (i.e., the Cartesian product of all alternatives). Then, for each combination, we generate the according source code by replacing the original artifact used in the variation points of the simulation.

4 Methodology

In this section, we describe our experiment setup and methodology.

4.1 Research Method

Due to the exploratory nature of our study, we use the *case study research method* as proposed by Shull et al. (2007), which is an initial investigation on the considered phenomena. To obtain a realistic picture, we use existing simulations implemented by domain experts that represent real-world use cases, instead of considering a large number of variation points without considering their context. Since this is the first application of VORM, we study two simulation codes in depth to explore the correctness and meaningfulness of the proposed alternative artifacts. That is, we discuss reasons why our approach is not able to identify some of the artifacts specified by the domain expert and why some artifacts proposed by our approach are not valid although they follow from the inference rules presented in Section 2.2. This way, we increase internal validity to identify confounding factors that may influence the results of our approach.

4.2 Measurement

To answer our research questions, we compare for each variation point the set of alternative artifacts proposed by VORM with the set of alternatives provided by the domain expert. We treat the artifacts provided by the domain expert as ground truth and, hence, categorize the artifacts proposed by our approach in: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).

After a discussion with the domain expert, we further categorize the false positives in: (i) artifacts that are true false positives (TFP) and (ii) artifacts that are generally applicable alternative artifacts (GAA) of the original artifact but do not work for the specific problem of the considered simulation without slight modifications. The artifacts of GAA are also suitable alternative artifacts when considering other simulations.

4.3 Subject Simulations

In our experiments, we consider two simulations implemented on top of DUNE. These simulations are based on system tests that are included in the modules DUNE-ISTL and DUNE-PDELAB deployed with DUNE. First, we extend the two simulations with a set of variation points, where each point specifies one artifact provided by DUNE. We selected the variation points based on domain knowledge. To obtain a ground truth of alternative artifacts, a domain expert of DUNE suggests feasible replacements based on his experience. In Table 1, we show the variation points and the number of alternative artifacts suggested by the domain expert for the variation points of the two considered simulations. We provide the source code of the simulations on our supplementary Web site¶ along with a Docker container‖.

The first subject simulation, LINEARSOLVER, explores the variability of an iterative linear solver in DUNE. The DUNE-ISTL module provides a range of different iterative *solvers* and *preconditioners*. The preconditioners improve performance of the solvers by reducing the condition number of the system. The simulation attempts to solve a simple elliptic finite-difference matrix using an iterative solver. Since the problem is symmetric, all iterative solvers can, in principle, be applied, given suitable parameters. Also, most preconditioners can be applied. It offers also parallel preconditioners, which work in an MPI parallel setup and require additional information about parallel data decomposition. As the original simulation runs sequentially, we are only interested in sequential solvers and preconditioners and do not consider parallel preconditioners. There are also a few further special-purpose implementations, which are not usable without individual configuration, therefore we consider these implementations as non-suitable.

¶https://github.com/se-passau/VariabilityExtraction-SupplementaryWebsite/

‖https://hub.docker.com/r/christiankaltenecker/vorm/

The second subject simulation, ELLIPTICPROBLEM, is built on DUNE's module DUNE-PDELAB. The module offers various functionalities to define different models and assemble the associated system matrices, given a particular *grid* implementation and a *finite element space basis*. We consider an elliptic test problem and vary the used discretization and the solver. A discretization is determined by a particular mesh, given a *geometry type* (e.g., simplex or cube), a *grid* implementation, and by the finite element space basis, given as a *finite element map*. Here, additional constraints on the computational domain can be specified, if necessary. As the problem is linear, it is possible to employ a linear or a non-linear solver and a particular iterative linear solver, given as *problem solver*.

These two subject systems were recommended by the domain expert for illustrating the capabilities of VORM. They cover typical use cases in the context of numerical solutions of partial differential equations, and they touch different parts of the DUNE framework following different paradigms. For example, while, in the first simulation code, solvers and preconditioners use inheritance, the second makes heavy use of duck typing, so relations among artifacts are only implicit. Although while using VORM on another framework would (slightly) increase external validity, we focus on one framework as a case study. This way, we can have a more in-depth look into each of the variation points and can match the results of VORM to the expectations of the domain expert. This enables us to assess the quality of the proposed approach and increase internal validity.

5 Results

Answering RQ_1 and RQ_2, we show the results of VORM for the two simulations and compare the recommended alternatives with the artifacts proposed by the domain expert in Table 2. The table shows the number of alternative artifacts proposed by the domain expert (DE) and the number of artifacts identified in our extraction (Positives), which we divided in the categories described in Section 4.2.

5.1 Subject Simulation: LINEARSOLVER

The LINEARSOLVER simulation has two variation points. For the first variation point, one can choose different solver implementations. VORM is able to identify all 7 alternatives proposed by the domain expert (TP). In addition, it identifies two other artifacts that, however, are not suitable because they represent only interfaces for the definition of iterative solvers (TFP).

For the second variation point (Preconditioner), VORM identified 8 alternatives that are applicable in the simulation (TP). 7 out of 8 alternatives are alternatives proposed by the expert (DE) and 1 additional alternative proposed by VORM was missed by the expert since it was recently added (MA).

Table 1 Subject simulations of our case study including their variation points, the list of proposed alternative artifacts in DUNE provided by our domain expert. Note that the first alternative is always the original artifact.

Simulation	Variation point	Proposed alternatives
LINEARSOLVER		
	Solver	LoopSolver, BiCGSTABSolver, CGSolver, GeneralizedPCGSolver, GradientSolver, MINResSolver, RestartedGMResSolver
	Preconditioner	SeqJac, SeqGS, SeqILU, SeqILU0, SeqILUn, SeqSOR, SeqSSOR
ELLIPTICPROBLEM		
	Geometry type	cube, simplex
	Grid	YaspGrid, OneDGrid, ALUGrid, UGGrid
	Finite element map	QkLocalFiniteElementMap, PkLocalFiniteElementMap, QkDGLocalFiniteElementMap, OPBLocalFiniteElementMap<Qk>, OPBLocalFiniteElementMap<Pk>
	Constraints	NoConstraints, ConformingDirichletConstraints
	ISTLSolverBackend	SEQ_CG_ILU0, SEQ_SUPERLU, SEQ_LOOP_Jac, SEQ_CG_Jac, SEQ_CG_SSOR, SEQ_CG_AMG_SSOR, SEQ_BCSG_Jac, SEQ_BCSG_SSOR, SEQ_MINRES_SSOR
	Solver type	StationaryLinearProblemSolver, Newton

In addition, it proposes 21 further alternatives. A closer look reveals that 11 of them are, in general valid (GAA), but need more parameters (e.g., AMG) or would slow down converging to a solution (e.g., Richardson). The remaining 10 alternatives are not suitable because they are parallel preconditioners (or components), and our main application considers only sequential execution. For example, class NonoverlappingRichardson implements a parallel preconditioner or ILUSubdomainSolver computes updates in a single subdomain (e.g., on a single MPI rank) inside a parallel overlapping Schwarz preconditioner (TFP). Parallel preconditioners only work with parallel scalar products, which does not apply to our case. Since the parallel preconditioners fulfill the subtype relation from Section 2.2, they are falsely identified as proper alternatives by VORM.

Table 2 Results of our case study for the two simulations LINEARSOLVER and ELLIPTICPROBLEM when using the original artifacts as starting points for the alternatives search. Variation Point: name of the variation point; DE: # of alternative artifacts suggested by the domain expert; TP: # of alternative artifacts suggested by the domain expert that are also proposed by VORM (true positives); FN: # of alternative artifacts suggested by the expert but not proposed by VORM (false negatives); MA: # of alternative artifacts suggested by VORM but missed by the expert; TN: # of artifacts provided by the framework, which are neither proposed by VORM nor suggested by the domain experts as alternatives; FP: # of artifacts identified as alternatives not provided by the domain expert but proposed by VORM (false positives); TFP: # of artifacts that are true false positives; GAA: # of artifacts identified as alternatives that are alternatives for the given artifact but do not work in the considered simulation but possibly in others.

Variation Point	DE	TP	FN	MA	TN	FP	
						TFP	GAA
LINEARSOLVER							
Solver	7	7	0	0	3 014	2	0
Preconditioner	7	8	0	1	2 994	10	11
ELLIPTICPROBLEM							
Geometry type	2	2	0	0	3 016	4	0
Grid	4	4	0	0	3 016	1	1
Finite element map	4	5	0	1	2 992	6	13
Constraints	2	1	1	0	3 021	0	0
ISTLSolverBackend	9	8	1	0	3 003	5	6
Solver type	2	1	1	0	3 021	0	0

5.2 Subject Simulation: ELLIPTICPROBLEM

In the second simulation, we consider six variatixon points. For the first variation point, Geometry type, VORM is able to identify all 2 alternatives proposed by the domain expert (TP). In addition, VORM identified 4 alternatives, true false positives (TFP). Having a closer look, we see that the original artifact is a value of an enumeration, and all proposed alternatives are other values of the same enumeration, from which some are valid and others are not. Differentiating between these two groups of values is not possible without further domain knowledge.

Second, we are interested in different Grid implementations, for which the domain expert proposes 4. VORM is able to identify these 4 alternative artifacts (TP) and 2 additional artifacts (FP). From the 2 false positives one is an abstract class used to define the interface of grid implementations, whereas the other could be used as an alternative, but does not work in the considered problem (GAA). This artifact is a meta-grid, which is used to host another grid, enriching it with further functionality. This is not used in the considered simulation.

For the variation point Finite element map, 4 alternative artifacts were proposed by the domain expert, all of which have been found by VORM. VORM identifies 20 more artifacts, from which 13 can be generally used as an alternative for the original artifact, but not in the specific scenario that we considered (GAA). For example, class RT0Cube2DLocalFEM provides lowest-order Raviar-Thomas shape functions, which are vector valued, while the subject simulation is working with a scalar partial differential equation; the same holds for BDM1Simplex2DLocalFiniteElementMap. Another 6 cases are true false positives (TFP). All false positives are template specializations of a mix-in class for particular finite element maps. They cannot be directly instantiated and should be considered as implementation details. The last alternative (RannacherTurekLocalFiniteElementMap) was missed by the expert (MA), as it is reportedly a valid but unusual choice.

The fourth variation point, Constraints, requires 2 artifacts to define constraints on the solution, for example, the type of boundary conditions (DE). For this variation point, VORM was not able to identify the alternative artifact proposed by the domain expert. So, we have 1 false negative (FN). The reason is that different methods are called, depending on the presence of boolean flags. The original artifact defines all of them as empty methods for documentation purposes, which is also explicitly stated in the documentation of the code. However, the proposed alternative by the domain expert offers only required methods. In contrast, the alternative artifact, which has been proposed by the domain expert, offers only required methods and not the additional set of empty methods, which violates the subtype relation since the desired artifact has less public methods than the original artifact. Such cases cannot be detected by an automated approach.

The fifth variation point concerns ISTLSolverBackend. VORM identified 8 out of the 9 alternative artifacts proposed by the domain expert (TP). For the one alternative not found by VORM, the artifact provides a different return type for a method, which precludes matching (we refer to this alternative when answering RQ_2). VORM identifies 11 more alternatives, of which 6 are general alternatives but cannot be used in the context of the simulation (GAA) and 5 are true false positives (TFP). Out of these 5 true false positives, 3 are abstract classes defining interfaces of specific types of solver backends and 2 shall be used for completely different parts in a simulation although they provide a matching interface to the original artifact.

Last, for the sixth variation point (Linear problem solver), we were not able to identify the alternative proposed by the domain expert (FN). This is because the alternative artifact focuses on solving linear or non-linear problems, whereas the original artifact focuses on solving linear problems only. So, both artifacts have different interfaces, which hinders VORM identifying the proposed alternative.

RQ_1: VORM is able to identify valid alternatives for a large number of the considered variation points. Overall, it identified 34 out of the 37 alternative artifacts proposed by the domain expert. However, for 2 of the 8 variation points, VORM were not able to identify alternatives. The reason is that the alternatives suggested by the domain expert exhibit a substantially different interface. Generally, VORM provides meaningful alternative artifacts as long as the artifacts provide a common interface and the alternative artifacts can be used for the same purpose.

As illustrated in the variation point Preconditioner in LINEARSOLVER, VORM was able to identify two artifacts that were missed by the domain expert. One artifact that was missed is the preconditioner SeqILDL, which was added in a recent version of DUNE and, thus, was missed by the domain expert. The other missed artifact is the finite element map RannacherTurekLocalFiniteElementMap, which is a rather unusual choice for a finite element map. This finite element map is a non-conform approach and works for 2-dimensional simulations, which is why this approach is used only in few cases.

The process of identifying alternatives is influenced by the presence of bugs and inconsistencies in a framework. As explained before, we were not able to identify a certain alternative artifact in ISTLSolverBackend proposed by the domain expert. This was due to a previously unknown bug in the interface of the alternative method, which was confirmed by the domain expert. Due to this bug, the alternative artifacts do not fulfill the subtype rule in Section 2.2.

Moreover, VORM identified an inconsistency in the template parameters of one artifact of the framework. There, the assumption was made that the input matrix and the solution vector have to be of the same precision and thus have to be from the same type, which is defined by the template parameter. Even with this assumption, we were able to identify the alternative artifact, but we were not able to replace all template parameters with appropriate artifacts.

RQ$_2$: With VORM, we were able to unveil new knowledge about 2 artifacts of DUNE to the domain expert. Moreover, VORM proposed 31 artifacts not recommended by the expert that are still applicable in the simulation scenario with slight modifications on the source code. Thus, VORM can be used to filter the artifacts provided by the framework to ease the burden of understanding the whole framework in detail. Notably, we were also able to identify a bug and an inconsistency in DUNE, which were unknown beforehand.

6 Discussion

In this section, we discuss the lessons learned from applying VORM to the two subject simulations and put them in context with the limitations of our approach.

6.1 Applicability

To identify alternative artifacts, we use a lightweight approach that only uses information from the API of the framework. Although, it would be possible to use domain knowledge to define a set of alternative artifacts for each artifact provided by the framework, gathering and maintaining these sets of alternatives might become time consuming, especially in the context of the evolution of the framework. These sets also highly depend on the knowledge of domain experts and on the use case they are familiar with. However, even without domain knowledge, VORM was able to identify meaningful sets of alternative artifacts for a large number of the variation points of the two simulations.

Moreover, we were able to find a bug and an inconsistency while comparing the identified artifacts with the artifacts proposed by the domain expert. This illustrates the potential of VORM as a lightweight means to find implementation errors and inconsistencies in scientific frameworks.

Not to restrict VORM to C++ frameworks, we generate and later parse the API using DOXYGEN, which supports a large number of programming languages, such as C, C#, Java, Fortran, or Python.[**] In principle, VORM can be applied to frameworks written in programming languages that provide inheritance as well as subtype and parametric polymorphism, such as C++, C#, or Java. However, for different programming languages, different inference rules have to be defined, depending on the semantics of the programming language. If a desired programming language is not supported by DOXYGEN, a different extraction mechanism has to be defined, which, however, does not hinder the general applicability of VORM, because VORM uses an internal language-agnostic representation to identify the artifacts. It is possible to extend VORM with additional frontends to parse APIs of other representations, such as CASTXML. Furthermore, VORM does not require templates for finding alternatives. The information provided by templates is used only to further aid the user by inserting the right template parameter. The same generality holds for the simulation. The simulation can be a file written in any programming language so far as the variability points are provided as depicted in Listing 1.

6.2 Limitations

As VORM is a very lightweight approach, it relies only on the original artifact applied in the variation points. As a consequence, the original artifact has a strong influence on the computed set of alternative artifacts. To illustrate this, we show the number of alternative artifacts proposed by our approach for the artifact Finite element map of ELLIPTICPROBLEM in Table 3. When using QK as original artifact, our approach proposes 23 alternative artifacts, whereas when using Pk as original artifact, our approach finds only 2 alternatives. The reason is that Pk provides one more method compared to the other artifacts, so VORM cannot propose the other artifacts to be suitable alternative artifacts. Again, VORM does not know whether a method of the artifacts is used in the simulation. Such methods could be identified when considering the call graph of the simulation., but, creating and analyzing an exact call graph can become

[**] http://www.doxygen.nl/

Table 3 Alternative artifact identified in our extraction process for different finite element map classes (Dune version 2.6). A checkmark (✓) states that the output artifact was identified to be an alternative to the input artifact. A dash (–) states that the output artifact was not identified to be an alternative to the input artifact.

Original/Alternative	Positives	Qk	Pk	QkDG	OPB
Qk	23	✓	✓	✓	✓
Pk	2	–	✓	–	–
QkDG	19	✓	–	✓	✓
OPB	19	✓	✓	✓	✓

very complex and would result in a heavyweight approach. Besides, function alt, as presented in Section 2.2, is not symmetric because of the subset relation. This means that an artifact might be an alternative to another artifact but not the other way around.

Another limitation arises from the mapping process, where we use information on the template parameters of the original artifact to replace the template parameters of the alternative artifacts with suitable labels. Using the heuristics mentioned in Section 3, we are able to map only parameters that have the same label in the original artifact and the alternative artifact. Although we are aware that these heuristics make strong assumptions on the naming of template parameters and that heuristics considering the usage of the parameters are more accurate, we refrain from using more complex heuristics, because they would make the whole approach more heavyweight. Both limitations could be addressed with interfaces in Java or concepts in C++ 20. Similar issues arise in Python2.4, because the programming language makes extensive use of duck typing.

A further limitation of VORM becomes apparent when an alternative artifact needs more or different template parameters than the original artifact. For example, the original artifact of ISTLSolverBackend provides zero template parameters, whereas three identified alternatives offer one template parameter. Despite these limitations, VORM was able to replace the template parameters for a large portion of the alternatives (for 60 out of the 78 alternatives that provide template parameters) identified for the two simulations. For the remaining 18 alternatives, VORM was not able to replace, at least, one template parameter and thus the user has to specify a suitable value for, at least, one template parameter. Here, a more complex heuristic might lead to more accurate results at the cost of increasing complexity and analysis cost. Furthermore, the set of identified alternative artifacts is not affected by template parameters of the original artifact, so identifying alternative artifacts and replacing the template parameters with suitable values are independent.

In Step IV, the combinatorial explosion of the number of configurations to explore is a another limitation. This is because we generate one configuration for each combination of the identified alternative artifacts. As a consequence, the number of configurations grows with the number of identified alternatives and the number of variation points of the simulation. To mitigate this limitation, VORM can be applied multiple times on a simulation for different sets of variation points or by removing proposed alternatives using domain knowledge.

6.3 Threats to Validity

Internal. By construction, we can not make sure that VORM identifies all theoretically valid alternatives to a artifact that is used in a variation point. This is in line with the observation that VORM is able to identify alternatives that were not proposed by the domain expert (see Section 5). However, this is an open problem that can not be solved in general. Clearly, such a lightweight approach focuses not on completeness, but on practicality. We were able to show that VORM finds a large portion of alternatives that also have been suggested by the domain expert. Besides, the statement about whether an artifact can be applied to a specific scenario also slightly depends on the domain expert and the knowledge about how the artifacts can be tailored to specific purposes by enriching them with further information. We mitigated this threat by using suggestions from a developer of the framework, who has in-depth knowledge about its functions and artifacts. During the implementation of VORM, we performed several tests using different artifacts provided by Dune as input and compared the artifacts identified by VORM with our expectations.

External. Although we demonstrated that our approach can identify a large number of alternative artifacts of DUNE, we cannot state that the approach also proposes meaningful alternatives on other case studies even if DUNE is used. This can also be seen for the two variation points Constraints and Solver type, for which VORM was not able to identify the alternative proposed by the domain expert. As we discussed in Section 5, the alternative artifacts proposed for the Solver type variation point work for different kinds of problems (e.g., linear vs. non-linear problems).

To be able to apply VORM also on applications or simulations that are not written in C++, we refrained from analyzing the source code of the framework but consider the API. The API provides all user-relevant information about the structure and the public interface of the classes of the framework. For the extraction and modification of the simulation, we also refrain from parsing the simulation using a C++ parser but pursue a line-based approach, which requires us to insert the variation point definition before the use of the original method (as seen in Lines 2–3 of Listing 2). As a consequence, VORM is not limited to C++, but also for other frameworks being written in other programming languages, such as Java or C#. However, when considering framework written in other programming languages, different inference rules might be necessary for the identification process of alternative artifacts.

7 Related Work

In this work, we focus on extracting the variability in a framework by considering the API of the framework. Thus, work on extracting variability and work that performs API recommendations is related.

Variability Extraction. To extract the hidden variability in software systems, there are different approaches considering different information sources.

Extracting the variability of an application or system is done using different information sources. For example, Zhang and Becker (2012) extract a variability model from the source code of an application by considering relations among preprocessor directives; Dietrich et al. (2012) extract the variability of the Linux system by considering the build system of Linux. There are approaches that extract variability information from natural language documents such as functional requirements, see for example, Mefteh et al. (2016) or Li et al. (2020). For a representative overview of approaches using natural language techniques, we refer to Li et al. (2017).

API Recommendation. Prior research that analyzes the API of a library to improve the code of an application is done at different levels of granularity.

At the most coarse-grained level, there are approaches that recommend using a library instead of another library offering similar functionality. For example, Thung et al. (2013) use association rule mining on a training set that contains information of the usage of third-party libraries to recommend libraries that are already used in software systems together with other libraries.

At a more fine-grained level that considers the methods provided by an API, Nguyen et al. (2016) developed a tool that recommends methods of an API to developers based on a statistical learning approach. For this purpose, they consider a large corpus of fine-grained code changes. Overall, their approach relies on the regularity and repetitiveness of code changes during software evolution, and thus also consider the context of the changes to propose methods to a developer that are already used in a similar context (Negara et al., 2014). For our use case, however, DUNE does not provide a large corpus of regular and repetitive code changes of applications.

Kawrykow and Robillard (2009) aim at identifying patterns in an application where an API is not used efficiently to replace code with API calls offering the same functionality. They focus on parts of the code of an application where the API is used to mimic the functionality of other methods provided by the same API. To this end, they first create an abstraction of the code of the library and consider the byte code of the application (they focus on Java application in their work). Then, the application code is compared against the library to identify patterns where the code of an API method is imitated without calling the method from the API.

To recommend method parameters for API method calls, Zhang et al. (2012) analyze parameter usage and their context in a code base. This is done by generating API method parameter candidates using type information of the method parameters and a corpus of API calls. Although this is similar to our template parameter replacement heuristic, their approach is not applicable to our setting due to incomplete type information in DUNE artifact template parameters.

Code Generation. Other approaches focus on domain-specific code generation and optimization based on an abstract definition of the problem at hand.

For example, Püschel et al. (2004) developed Spiral to generate highly performant signal-processing algorithms, such as fast Fourier transformations, based on an abstract domain-specific specification. Using this specification as starting point, they apply a set of rewriting rules considering domain-specific knowledge about the considered problem. Different sequences and alternatives of rewriting rules may be used leading to different implementations for the same problem. Since some transformations produce more performant code than others, they rely on a feedback-loop mechanism to generate high-performance code.

Another example of domain-specific code generation is developed in the ExaStencils project (Lengauer et al., 2020). ExaStencils focuses on generating multigrid solvers to efficiently solve partial differential equations. It comprises four domain-specific languages, at different levels of abstraction (from mathematical equations to hardware-specific details), which can be used as a starting point for code generation. Based on the specification of a problem, the ExaStencils generator transforms the provided domain-specific code into high performant C++ code.

In general, code-generation approaches usually focus on one specific domain and exploit domain knowledge during code generation. Although these approaches demonstrate the benefits of generating high performant code without requiring deeper knowledge on the required implementation (which needs to be built into the code generator, though), they are applicable only for specific domains. VORM is domain independent and does not rely on domain-specific information. Instead, it uses only information provided by the API of the considered framework.

8 Conclusion

To support application engineers in the development of scientific simulation code, there are several frameworks available that provide reusable implementational artifacts. When using a given framework, the most suitable set of artifacts provided by the framework has to be identified for a given simulation, which is a non-trivial task, because global knowledge about all existing artifacts provided by the framework is required. To support developers in this task, we present and validate the usability of VORM, a lightweight, semi-automatic API-based approach to identify artifacts that can be used instead of an already applied artifact in a simulation.

To demonstrate the usefulness of VORM and to validate whether its recommendations are accurate, we compared the artifacts proposed by VORM with artifacts recommended by a domain expert of the framework for two subject simulations written using the scientific framework DUNE.

Our results show that VORM is able to identify 34 out of the 37 alternative artifacts proposed by a domain expert. VORM was even able to identify two artifacts that were applicable but missed by the domain expert. Additionally, VORM proposes 31 further artifacts that can also be used as alternative artifacts to the used artifacts but not in the considered simulation scenario. Notably, when comparing the results of VORM with domain knowledge provided by a developer of DUNE, we were able to identify a bug and an inconsistency in DUNE that were unknown beforehand. Based on these results, we conclude that VORM's lightweight, semi-automatic approach can be used to identify suitable alternatives to given artifacts.

There are some open issues that we leave to further research. Applying VORM to multiple different frameworks is one topic for future research to assess the general applicability of a lightweight approach. Another avenue of further work is to analyze inaccuracies presented in RQ_1. It is unclear whether the inaccuracies are mainly due to an undisciplined use of interfaces or a lack of expressiveness.

For future work, we see two further steps to assess the applicability and usefulness of VORM. First, VORM shall be evaluated using a larger set of applications on top of DUNE. Second, the approach of VORM shall be applied to other frameworks, possibly written in different programming languages. Additionally, different API documentations, such as CLDOC[††] should be considered in the evaluation. However, for each considered framework, at least one domain expert of the respective framework is required for evaluation, which is well outside the scope of this study.

References

Alnæs M, Blechta J, Hake J, Johansson A, Kehlet B, Logg A, Richardson C, Ring J, Rognes M, Wells G (2015) The FEniCS Project Version 1.5. Archive of Numerical Software 3(100)

Bastian P, Blatt M, Dedner A, Engwer C, Klöfkorn R, Kornhuber R, Ohlberger M, Sander O (2008a) A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE. Computing 82(2–3):121–138

Bastian P, Blatt M, Dedner A, Engwer C, Klöfkorn R, Ohlberger M, Sander O (2008b) A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. Computing 82(2–3):103–119

Bastian P, Heimann F, Marnach S (2010) Generic Implementation of Finite Element Methods in the Distributed and Unified Numerics Environment (DUNE). Kybernetika 46:294–315

[††]https://jessevdk.github.io/cldoc/

Braess D (2007) Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics. Cambridge University Press

Dietrich C, Tartler R, Schröder-Preikschat W, Lohmann D (2012) A Robust Approach for Variability Extraction from the Linux Build System. In: Proceedings of the International Software Product Line Conference, ACM, SPLC, pp 21–30

Falgout RD, Yang UM (2002) Hypre: A library of high performance preconditioners. In: Proceedings of the International Conference on Computational Science-Part III, Springer, ICCS, pp 632–641

Grebhahn A, Engwer C, Bolten M, Apel S (2017) Variability of Stencil Computations for Porous Media. Concurrency and Computation: Practice and Experience 29(17):4119:1–4119:14

Kawrykow D, Robillard MP (2009) Improving API Usage through Automatic Detection of Redundant Code. In: Proceedings of the International Conference on Automated Software Engineering, IEEE, ASE, pp 111–122

Knabner P, Angerman L (2003) Numerical Methods for Elliptic and Parabolic Partial Differential Equations. Texts in Applied Mathematics, Springer

Lengauer C, Apel S, Bolten M, Chiba S, Rüde U, Teich J, Größlinger A, Hannig F, Köstler H, Claus L, Grebhahn A, Groth S, Kronawitter S, Kuckuk S, Rittich H, Schmitt C, Schmitt J (2020) ExaStencils–Advanced Multigrid Solver Generation. In: Software for Exascale Computing – SPPEXA 2016-2019, Lecture Notes in Computer Science and Engineering, Springer

Li Y, Schulze S, Saake G (2017) Reverse Engineering Variability from Natural Language Documents: A Systematic Literature Review. In: Proceedings of the International Systems and Software Product Line Conference, ACM, SPLC, pp 133–142

Li Y, Schulze S, Xu J (2020) Feature Terms Prediction: A Feasible Way to Indicate the Notion of Features in Software Product Line. In: Proceedings of the Evaluation and Assessment in Software Engineering, ACM, EASE, pp 90–99

Mefteh M, Bouassida N, Ben-Abdallah H (2016) Mining Feature Models from Functional Requirements. The Computer Journal 59(12):1784–1804

Negara S, Codoban M, Dig D, Johnson RE (2014) Mining Fine-grained Code Changes to Detect Unknown Change Patterns. In: Proceedings of the International Conference on Software Engineering, ACM, ICSE, pp 803–813

Nguyen AT, Hilton M, Codoban M, Nguyen HA, Mast L, Rademacher E, Nguyen TN, Dig D (2016) API Code Recommendation Using Statistical Learning from Fine-grained Changes. In: Proceedings of the International Symposium on Foundations of Software Engineering, ACM, FSE, pp 511–522

Püschel M, Moura JMF, Singer B, Xiong J, Johnson J, Padua D, Veloso M, Johnson RW (2004) Spiral: A generator for platform-adapted libraries of signal processing alogorithms. The International Journal of High Performance Computing Applications 18(1):21–45

Remmel H (2014) Supporting the Quality Assurance of a Scientific Framework. Dissertation, Ruperto-Carola University of Heidelberg

Shull F, Singer J, Sjøberg DI (2007) Guide to Advanced Empirical Software Engineering. Springer

Thung F, Lo D, Lawall J (2013) Automated Library Recommendation. In: Proceedings of the Working Conference on Reverse Engineering, IEEE, WCRE, pp 182–191

Vogel A, Reiter S, Rupp M, Nägel A, Wittum G (2013) UG 4: A Novel Flexible Software System for Simulating PDE Based Models on High Performance Computers. Computing and Visualization in Science 16(4):165–179

Zhang B, Becker M (2012) Code-based Variability Model Extraction for Software Product Line Improvement. In: Proceedings of the International Software Product Line Conference, ACM, SPLC, pp 91–98

Zhang C, Yang J, Zhang Y, Fan J, Zhang X, Zhao J, Ou P (2012) Automatic Parameter Recommendation for Practical API Usage. In: Proceedings of the International Conference on Software Engineering, IEEE, ICSE, pp 826–836