Automatic testing of a domain specific implementation

Anonymous Author(s)

ABSTRACT

Domain specific languages (DSLs) can provide many advantages to the user, such as a high level of abstraction coupled with efficient code generation. Standard approaches used to test general-purpose compilers, such as differential testing, may not work or be sufficient to test the implementation of a DSL. Applications of differential testing on compilers depend on an outside source to serve as an oracle, such as another variation of the compiler. Due to the specialized nature of DSLs, there are typically not multiple variations of them. Therefore, the differential testing approach alone is insufficient to test the implementation of a DSL. Without sufficient testing, a DSL can be unreliable and detract users. Developers need to guarantee that the abstraction of their domain-specific languages are correct. To address this problem, we provide our strategy of *Domain-based* testing.

Domain-based testing builds on previous work in the software testing field, but leverages some knowledge of the problem domain to address the oracle problem. By understanding the problem domain test programs can be evaluated based on a ground truth and domain principles. Incorporating domain knowledge in a testing infrastructure tool enables the generation of effective and smart test programs.

This paper describes the automated testing of the implementation of a DSL and serves as an example of the importance of automated testing. Automated testing approaches should go hand-in hand with the implementation of a programming language. The methods describe in this paper were designed specifically for the compiler domain. As a result, the methods have created thousands of test programs and found bugs. This paper describes the approaches to test the implementation of a DSL and how the testing techniques might be used for other applications

1 INTRODUCTION

Domain specific languages (DSLs) can accelerate how researchers express their ideas in working code. When language syntax reflects the conceptual elements of some target domain, programming can be more intuitive and familiar. To support a high-level of abstraction, the language developer embeds domain-specific knowledge into the language, sometimes directly in the compiler. By their nature, then, DSL compilers must bridge a large semantic gap between the high-level surface language and the low-level compiler output, which increases the need to rigorously test the compiler and its many states of intermediate representation.

ISSTA 2018, Amsterdam, The Netherlands

978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 0000001.0000001

The paper demonstrates testing a new DSL compiler, called *Remy*. Various scientific computing packages produce *FE* (finite element) fields to store solutions to differential equations problems solved by the finite element method. The *Remy* compiler supports scientific visualization of *FE* fields by including specialized operators and types, such as tensor fields and their derivatives. For example, the common visualization method of *direct volume rendering* depends on gradients (first derivatives) of fields for surface shading [1]. Creating test *Remy* programs by hand is impractical, due to the number of supported operations, so automatic testing is required.

Automatic compiler testing is an area of ongoing research. Previous work has focused on the testing oracle problem and the generation of smart test cases [2, 3, 4, 5, 6, 7]. In the absence of a ground truth, previous testing approaches include randomized differential testing (RDT) [2, 7], different optimization levels (DOI) [3], and equivalence modulo inputs (EMI) [4]. Though existing work focused on the correctness of general purpose compilers, we have focused on testing a domain-specific compiler.

We present three approaches for testing *Remy*, which test different aspects of the compiler with a complementary mix of strategies. The first approach, T_{probe}, uses a differential testing approach to compare *Remy* with an existing tool (Firedrake) in evaluating random *FE* field at random locations. This approach was useful in early stages of *Remy* development, but it gave the developers a false sense of security by failing to uncover numerous bugs. Firedrake is designed to produce *FE* fields and *Remy* is for visualizing them, thus Firedrake can provide ground truth for only a small subset of *Remy* functionality, not the rich set of domain-specific field operators supported by *Remy*.

This paper describes a new domain-specific testing strategy, which we term *Domain-based* testing, for generating test programs that more explicitly represent the DSL's target domain, and can more rigorously test a domain-specific compiler. We introduce testing tool T_{math}, which implements the *Domain-based* testing approach by creating and evaluating domain-specific test programs that are compared against the results of the symbolic mathematics library Sympy [8]. Test fields are generated from randomly generated polynomials, which can be exactly represented within higher-order *FE* fields, as well as analytically represented within Sympy. The domain-specific operations in a *Remy* program on *FE* field data are also applied to the Sympy representation of the same field, to supply a ground truth for field evaluations in *Remy*.

While T_{math} can test a large set of domain-specific operations supported by *Remy*, few non-trivial *Remy programs* produce output with known ground truth, because scientific visualization algorithms often involve numerical integrations with no analytic solution. This paper thus also introduces method T_{vis}, which uses full visualization programs as tests, computed on a class of fields (polynomials) in which the representational abilities of *Remy* overlap with those of a related domain-specific language, Diderot [9, 10, 11].

To summarize, we describe three approaches to testing *Remy*, all of which finish with an equality tests between *Remy* output and

that of another system. The testing programs differ in the type of *Remy* program generated, and the reference system for evaluation.

(1) T_{probe}: randomly probing random *FE* field, compared with Firedrake.
(2) T_{math}: randomly applying mathematical operations (such as differentiation) on random polynomial fields, compared with Sympy.
(3) T_{vis}: running full visualization programs on specific polynomial fields, compared with Diderot.

The rest of paper is organized as follows. The background (Section 2) describes the DSL domain and relevant syntax. The DSL is tested with three approaches T_{probe} (Section 3), T_{math} (Section 4), and T_{vis} (Section 5). The results (Section 6) describe the state of the compiler over time and report examples of bugs found. Related work described in Section 7. The paper concludes in Section 8.

2 BACKGROUND

Effective testing of a DSL like *Remy* necessarily involves the elements of domain knowledge embedded in the compiler. This section describes the target domain of *Remy* (Section 2.1) and its syntax (Section 2.2).

2.1 Domain

Scientists use PDEs (Partial Differential Equations) to represent a wide range of problems. They often solve PDEs with the finite element method, a numerical technique. The finite element method discretizes the domain into small regions, collectively called the mesh, and then places a set of polynomial basis functions of degree k on each region, called the finite element of degree k. The finite elements on each region of the mesh are used to numerically approximate a PDE solution [12].

Solving PDEs on with finite elements is itself an active research area of computational science, producing tools like *Firedrake* [13] and *Dolfin* [14], which is part of the *FEniCS Project* [15]. Upon computing an approximate PDE solution, these tools output an *FE* field, which is specified by the mesh geometry, the finite element, and a vector of real numbers giving the coefficients for the basis functions over the mesh. *Remy* is designed to take such an *FE* field and facilitate subsequent visual exploration via various techniques of scientific visualization. Visualization algorithms in this domain often use tensors and tensor fields to represent the data and features extraced from it. Computing spatial derivatives, for example as needed for volume rendering [1] or computing ridges and valleys [16] increments tensor order (e.g. gradient vectors or Hessian matrices from scalars), which increases implementation complexity. The Diderot DSL [9, 10, 11] was created to support the implementation of tensor-based visualization methods, but only for fields created from regularly sampled image or volume data, not finite elements.

We leveraged the open source capabilities of Diderot to create *Remy*. A *Remy* programmer can then apply the capabilities of Diderot to a *FE* field to create interesting visualizations that are otherwise difficult or impossible to create with existing tools. The *Remy* compiler handles applying operations of *FE* field data. The evaluation is a nontrivial implementation that includes the translation between coordinate spaces, weights assigned to basis functions, and manipulation of *FE* field data.

τ	::=	**tensor**$[\sigma]$	tensor with shape σ
		field $(d)[\sigma]$	tensor field with shape σ, and dimension d (2 or 3)
υ_n	::=	2 | 3 | 4	dimension
σ	::=	nil' | $\upsilon_1, \cdots, \upsilon_n$	$n \leq 3$ Tensor shape
ops	::=	$\|\| \bullet \|\|$,−, normalize	*generic*
	|	∇, $\nabla\otimes\nabla$, sin, arccos $\sqrt{\bullet}$	*scalars*
	|	$\nabla\times$, $\nabla\bullet$, $\nabla\otimes$, [1]	*first-order*
	|	inverse, trace, determinant	*second-order*
	|	transpose [:,0], [1,:]	
	|	+, −, ∗, /, •, :, ×, ⊗, max, min	*binary*
	|	probe, concat, ∘, modulate	

Figure 1: Remy syntax: types (τ) and operators (ops)

2.2 *Remy*

The testing approaches in this paper are designed to test the *Remy* compiler. Using the language to visualize *FE* fields includes three steps (1) a python program that creates *FE* field data, (2) a *Remy* program that provides the framework to visualize *FE* fields, and (3) connecting the two. The third step is completed by compiling the *Remy* program to a library. The program is then initiated by the python program via a python to C foreign function interface. In the following section we describe the syntax used in the python and *Remy* program in more detail.

The *Remy* language supports tensors and *FE* field (τ in Figure 1). Tensors include scalars (order 0), vectors (order 1), and matrices (order 2). The tensor *shape* σ is a list of integers (with length equal to tensor order) giving the integral range each tensor index can take. For instance, a tensor represented with (**tensor**$[\upsilon]$), indicates a vector with length v. *FE* fields (femfield $\#k(d)[\sigma]$) are d-dimensional functions from vectors of length d to tensors with shape σ.

To support the implementation of visualization ideas, the language offers various mathematical operators on tensor fields. Operators specific to fields include composition (∘),differentiation ($\nabla\otimes\nabla, \nabla\cdot, \nabla\times$), and probing (probe). Probing includes evaluating *FE* field at points. Supported unary operators on tensors include slicing ([:,0]), matrix manipulations (inverse, trace), negation (−), and norm ($\|\cdot\|$). Binary operators include arithmetic operators (+,∗) and comparative operators (max).

The syntax supported to define *FE* field in *Remy* is similar to that supported by FE tools (Figure 2). A *FE* field is composed of a function space, element, and mesh. The *FE* field notation is used in a *Remy* program to let the compiler know about the structure of the data so it can correctly sample and apply computation to it. An FE tool to creates the *FE* field data by solving a PDE or by interpolating a polynomial expression.

3 T_{PROBE}: TESTING SIMPLE PROGRAMS

The simplest *Remy* program consists of defining an *FE* field and then probing it. A probe operation $probe(F, x)$ evaluates an *FE* field F at a position x. The probe operation is the most basic yet critical operation and all operations on *FE* fields build on it. While simple

$$
\begin{array}{rcl}
m & ::= & \text{UnitSquareMesh} \mid \text{UnitCubeMesh} \\
e & ::= & \text{Lagrange} \mid \text{P} \\
s & ::= & \text{FunctionSpace(m,e,k) where k} \in \mathbb{N} \\
 & \mid & \text{TensorFunctionSpace(m,e,k,}\tilde{n}\text{)}
\end{array}
$$

Figure 2: Function Space (s) are defined with a mesh (m) and element (e)

on the surface, the probe operation is a complex algorithm with a nontrivial implementation.

The initial stages of *Remy* development focused on correctly implementing probing. Fortunately, Firedrake supports the same operation. Our T_{probe} method, described here, compares how *Remy* and Firedrake probe randomly generated *FE* field.

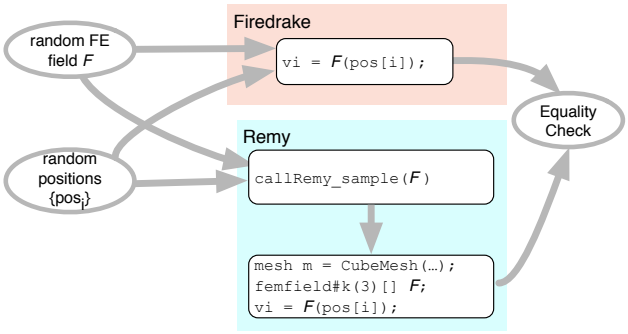

Figure 3: Pipeline for T_{probe}. The data is created by Firedrake. Then the data is sampled by Firedrake (top) and *Remy* (bottom). The results are compared based on numerical equality.

3.1 Testing approach concept

Like differential testing, *Domain-based* testing, can compare the result of a test program against another implementation. More specifically, one program written in one language is not being run by two compilers of the same nature; rather, two different programs that ought to produce the same results are sent to their respective compilers. T_{probe} combines various ideas and tools into a testing pipeline: a list of the various types of *FE* fields, a mechanism for generating a random *FE* field of a particular type, and the use of Firedrake as a testing oracle to evaluate *Remy*'s probe of a random *FE* field. Before generating tests, T_{probe} must look over various types of *FE* fields indexed by mesh, element, and degree so it can make a test for each type. Then, for each type, T_{probe} uses the representation of an *FE* field as a vector of real numbers to randomly generate *FE* fields that can be used by Firedrake [17].

3.2 Testing tool implementation

A simple test program involves probing an *FE* field at a list of positions. T_{probe} creates random *FE* field data and positions. The field is sampled in two ways. In one way (top of Figure 3) Firedrake evaluates the field at positions. In the second way (bottom of Figure 3) Firedrake passes the field to a *Remy* program, which samples the field. The output is expected to be equal.

3.3 Advantages and Limitations

The current implementation of T_{probe} covers one *Remy* field type, but it could be extended to a richer variety of fields. T_{probe} found numerical bugs in early versions of the *Remy* compiler, which more than justified its modest (230 lines of code) implementation effort.

The problem of developing a testing tool that relies on an outside source, is that the testing scope is limited by that outside source. Test programs that can be written with *Remy* can not be written and evaluated with Firedrake. Firedrake can create and probe *FE* field at positions, but it can not extend beyond a certain point. As a result of the limited testing scope, T_{probe} could not rigorously test the *Remy* language. It did not detect any bugs with T_{probe} that we would not have detected with other approaches (Section 6.6). It was unable to find bugs that were uncovered by other approaches with more complicated test programs.

4 T_{MATH}: TESTING DOMAIN-SPECIFIC OPERATIONS

Firedrake can probe a given *FE* field, but it lacks the *FE* field operations (like field differentiation) that could make *Remy* a useful new language for visualizing *FE* fields. This section describes an application of the *Domain-based* testing approach and presents the automated testing tool,T_{math}.

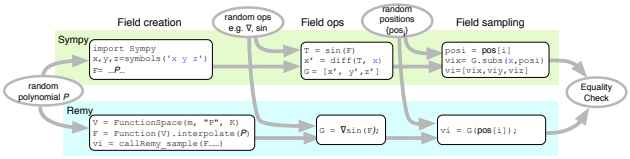

Figure 4: Pipeline for T_{math}. The test uses two comparable ways to represent data and apply computations to it with *Domain-based* testing: Sympy (top) and *Remy* (bottom). With *Remy* the data is created with Firedrake and a computation is applied to it in a *Remy* program. Sympy also represents the data and evaluates it. The output is compared based on numerical equality.

4.1 Testing approach concept

Even if there is no other language with the same exact capabilities as *Remy*, there are other software systems which support some of the same mathematical ingredients. Our *Domain-based* testing approach isolates a domain-specific abstraction in the DSL, such as that of the tensor field and mathematical operations on it, and finds a trusted alternate representation and implementation of the same abstraction. Testing then involves generating instances of the abstraction that can be exactly represented in both systems. For T_{math}, illustrated in Figure 4, we chose polynomial functions over a vector space as test fields, and compared results with Sympy [8]. Sympy supports symbolic mathematics, but not *FE* fields.

4.2 Testing tool implementation

A developer can specify input to define several key factors for T_{math} such as what is being tested and how to search for test cases. The testing input serves as input for the test generation process.

A test descriptions synthesize a single test case that is created, executed, and evaluated.

4.2.1 Testing input. The testing input is defined by various variable settings and with command line arguments. The settings include *FE* field data details, such as the ability to choose a reference element (or randomize it), and mesh size. More generally the settings indicate data creation factors, test program details, computation details, and type of search to find test cases. The command line arguments describe the subset of types and operators that are being tested (the scope). The testing scope can be limited to a single test or a group of tests. By limiting the testing scope, it is possible to re-run a single test, instead of all tests, and enable debugging.

4.2.2 Test case generator. Test generation is parameterized by the testing input. A core computation involves the application of mathematical operators to arguments. An internal testing type-checker checks the core computation and the generator only creates test cases for computations that make mathematical sense. As a result, T_{math} does not create test programs with type checking errors.

4.2.3 Single test case. A single test case is created by running the following steps. The settings are pulled from the testing frame, coefficients for polynomials and positions are randomly created. Synthetic data is created for each argument and is represented both by a data file and a symbolic expression. A Sympy package is used to create a numerically correct solution. Firedrake test program is generated and used to create data. *Remy* test program is created and compiled to a library to sample the data. The output is reduced to a list of numbers and compared. The expected behavior is numerical equality between the output. If the output is within some error tolerance then the test passes. An example of a test case is provided in Section 4.3 and measured in Section 6.5.

4.3 Test program example

The following illustrates an example of a single test case. In the heart of the T_{math} test program is a core mathematical computation. The section describes the test that is generated for the following test description.

$$G = (\nabla F_0) \bullet F_1$$

where F_0 is a 3-d scalar field and F_1 is a 3-d vector field.

The description describes the application of a binary operator (inner product[1]) between the result of a unary operator (gradient of $\mathtt{F}_0$) and another field argument ($\mathtt{F}_1$). The tensor field ($\mathtt{G}$) is sampled at positions $(pos_1, pos_2 \ldots pos_{n-1})$.

For each test case T_{math} creates synthetic data, a list of positions, a ground truth, and a *Remy* test program. Synthetic data is created for each argument and is represented both by a data file and a symbolic expression. The symbolic expression is used to define the ground truth for each test case.

4.3.1 Data Representation. Each argument has a list of randomly generated numbers (data) that represent coefficients to a polynomial expression. Variables (x, y, z) are generated for each field dimension. Given the running example, two polynomials are generated. A polynomial for a 3-d scalar field

$\mathtt{P}_0\mathtt{(x,y,z)} = 2 + 4x + 3y^2 + z$

and 3-d vector field:

$\mathtt{P}_1\mathtt{(x,y,z)} = [7x, x + y, 9]$

The same polynomials are used to create expressions in the *Numerically correct output* and the *FE* field data in the test program.

4.3.2 Numerically correct output. The test description is translated to a symbolic expression that can be evaluated. The operators being tested are tensor calculus-based operators. Since the operators are based on mathematics the Python sympy package can be used to analytically derive the correct solution.

Each test description includes the operators that are used in the test case. Each operator is matched to a function that will apply a manipulation on the polynomial expression based on the argument types. The following walks through the core computation in T_{math} from a mathematical standpoint.

The expression ($\mathtt{P}_0$) can be symbolically differentiated.

$\mathtt{grad\_P}_0\mathtt{(x,y,z)} = [4, 6y, 1]$// the derivative of $\mathtt{P}_0$.

and it can be manipulated with a series of tensor operators on and between them:

$\mathtt{G(x,y,z)} = 28x + 6xy + 6y^2 + 9$// $\mathtt{grad\_P}_0 \bullet \mathtt{P}_1$

After the operators are applied the output expression is evaluated at positions.

$\mathtt{pos}_i$ `= [1,0,10]` $\mathtt{G(pos}_i\mathtt{)} = 37$

The computation reduces to a series of number that represent the ground truth for a given test case.

4.3.3 Test programs. As discussed in Section 2.2, a single test program actually involves two programs; a Firedrake and *Remy* program. The test description automatically generates the Firedrake program. The field arguments are created by interpolating an expression over a function space. For instance, a unit cube mesh is created for the 3-d scalar field $\mathtt{F}_0$.

```
m = UnitCubeMesh( 4, 4, 4)
V= FunctionSpace(m,"Lagrange",degree=3)
```

$\mathtt{F}_0$ `= Function(V).interpolate(`$\mathtt{P}_0$`)`

The second field argument ($\mathtt{F}_1$) is created in a similar way.

$\mathtt{F}_1$ `= Function(V).interpolate(`$\mathtt{P}_1$`)`

The python program initializes visualization parameters and provide a pointer to the field arguments.

```
_call.callRemy(F0,F1)
```

T_{math} supports the automatic creation of test programs by translating the test description into *Remy* code and compiling it to the a library. The *Remy* program has a representation for a mesh, element, and function space for each field argument (Figure 2). The core computation being tested is generated in the *Remy* program.

$\mathtt{femfield\#k(3)[]\ G} = (\nabla\ \mathtt{F}_0)\bullet\mathtt{F}_1\mathtt{;}$

The core of a test program is a tensor field sampled at a position $(pos_1, pos_2 \ldots pos_{n-1})$. A strand in a *Remy* program initiates the position value (*pos*), while the tensor field computation remains the same. An inside test is imposed to be sure the position is in the field domain.

$\mathtt{if(inside(F,pos}_i\mathtt{))\{\ldots\}}$

The result of computation is the observed value.

$\mathtt{tensor[]\ observed} = \mathtt{G(pos}_i\mathtt{);\}}$

Once the *Remy* program is written, it is compiled and executed.

[1]The inner product operation involves component-wise multiplication followed with a summation.

4.4 Advantages and Limitations

4.4.1 Checking limitations of the Remy programs. In the past, compiler testing tools have evaluated tests by comparing the output of different versions of a compiler [2], or have asked the human user to supply a criterion that can be checked automatically [18]. The evaluation is based on a ground truth, but still, the evaluation is comparing the output of floating point arithmetic done by the *Remy* compiler with an analytically derived solution. The potential rounding errors that can occur in floating point arithmetic are well-known. In T_{math} it is possible that numerical errors will result in a false positive.

T_{math} does take some precautions against doing operations with undefined results. To automate this process, each operator is created with certain restrictions. The argument(s) to the operation might have to be within a certain range of values. If so, the test program generates if statements to check that these conditions are met. If the conditions are not met then that strand is invalid. If not enough strands pass this condition then the test is thrown out.

For example, the composition operation ∘ between two fields

$G = F_0 \circ F_1$

is tagged by a condition that limits its arguments. Each field must generate it's own inside test. For each strand, if the expression evaluated for composition is valid then the computation is evaluated and added to the output.

```
//check limitations
if(inside(F1, posi)){
  if(inside(F0, probe(F1,posi)))
   {observed = probe(G, posi);//include}
 }
else{observed=null//do not include}
```

Otherwise the strand is not "included" in the output.

4.4.2 Features. T_{math} supports two different types of searches; exhaustive and random. Every possible combination of operators and arguments types are generated in the exhaustive setting. It is not always practical to run an exhaustive test and create tens of thousands of programs. In the random mode, the test generator will randomly choose the test cases to create. Random testing does not ensure coverage, but it makes it feasible to explore a larger set of complicated programs (with a varying number of nested operators) in a more manageable amount of time. Experiments that use an exhaustive or random search for test cases are presented in Section 6.

In order to enable more interesting visualization programs the *Remy* project continues to add new operators and features. Due to the intricate rewriting and optimizations done in the compiler it is necessary to rigorously test each additional operator. It is easy to add new operators to T_{math} and then generate hundreds of test programs that involve those new operators.

T_{math} is designed to find bugs in the *Remy* compiler but it can also aid in the process of debugging. T_{math} can track failed tests and enables reproducibility. It is possible to make copies of every program that fails, but that would be unmanageable. In lieu of that, labels are used to apply targeted testing.

Targeting testing is a way to limit the testing scope to a subset of test cases with the use of a tag. Each operator has an identifier. A user can use that identifier (with others) as a tag to limit the testing scope. The scope of the testing is one of three modes:

(1) Run a single case, *e.g.*, the addition of two 2-d scalar fields.
(2) Target a group of test cases, *e.g.*, the addition operator.
(3) Run all possible test cases define, *e.g.*, addition, outer product, negation, ...on and between d-dimensinonal tensor fields.

The labeling aids the developer in the process of debugging by identifying tests that fail and enables the developer to test them again after making changes to the compiler. Targeting testing can be helpful when testing a new operator added to the language and only creating test programs that use that operator.

4.4.3 Domain-based testing. There are numerous domain-specific tools, languages, and packages that are built for the scientific computing and scientific visualization domain that use a high level of math. As far as one is aware, there is a lack of publications on automatically testing the software. T_{math} demonstrates a way to automatically create test programs for a mathematically focused domain and evaluate the tests based on a ground truth.

The idea of incorporating domain-specific knowledge in the testing approach can be extended to other test programs without a ground truth. For example, T_{math} creates two versions of a test program. Similarly, T_{vis} changes the data representation in order to enable the use of a test program and evaluates the results by comparing images.

5 T_{VIS}: TESTING VISUALIZATION PROGRAMS

Remy was created for the scientific visualization and image analysis domain. Programs in this domain can be complicated, involving the tensor calculus operators, comparative operators, and other language structures (conditionals and loops). Sophisticated programs are more likely to use untested parts of the compiler, which complicates knowing whether the results are correct. In addition, visualization methods such as volume rendering [1] and flow streamlines [19] are forms of numerical integration, with no known analytical solution except in trivial cases. The testing approaches described above involve a mix of operators and types supported by *Remy*, but do not reflect the programs that a real user might write in *Remy*. This section presents T_{vis}, an idea for testing the compiler on full visualization programs.

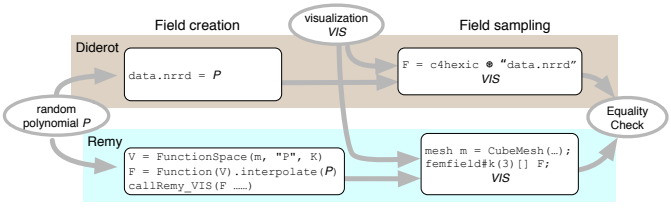

Figure 5: Pipeline for T_{vis}. The test uses two comparable ways to create the same data and the same program to sample it:Diderot(top) and *Remy*(bottom). The data is created by Diderot and *Remy*. The data is sampled by a visualization program. The resulting images are compared with a quick eyeball check.

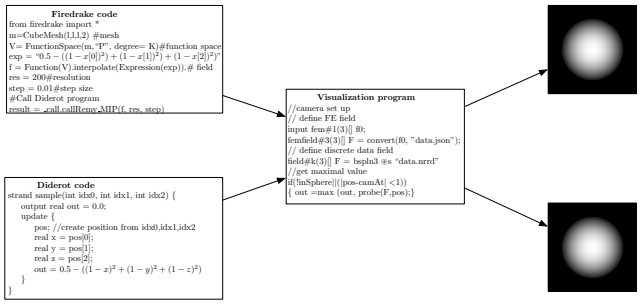

Figure 6: This figure compares data created by two different sources and visualized with the same tool. The fields are created by Firedrake and Diderot. The expression creates a sphere and the field is defined by interpolating the expression given the function space.We make a call to the *Remy* library by passing the field and initializing the resolution and step size.

5.1 Testing approach and examples

The T_{vis} method uses a visualization program to test the compiler. Like, T_{math} the data is being tested has two different data representations. Afterwards, the data is sampled, manipulated, or rendered by the same visualization program. The output is expected to be comparable.

This section provides two examples of using the T_{vis}approach. In both cases the test program data is represented in two ways; an *FE* field and a synthetic field. *Remy* creates the *FE* field by interpolating the expression (*exp*) over a function space. Diderot creates the discrete image file and uses a convolution kernel to create a continuous field. Both representations of the data are visualized by a *Remy* program. As can be expected, the images created from the visualization output are comparable.

Examples are demonstrated in Figure 6 and Figure 7. In Figure 6, the *Remy* program does a Maximal intensity projection[2] of both representations of the data. Figure 7 visualizes the Cayley Cubic surface with a shaded volume rendering program.

5.2 Advantages and Limitations

Evaluating the output to a realistic visualization programs is a difficult problem. The T_{vis} method demonstrates a way to use a visualization program in a way so that the results can be evaluated. The visualization program is something a domain user might write to visualize their data. As a result, this paper offered an approach to evaluate the visualization of *FE* field data.

The significant limitation is that T_{vis} is not an automated testing tool. Both examples demonstrated one type of *Remy* type (3-d scalar field). Each examples uses just one visualization program, which tests a particular visualization algorithm and uses a specific set of operators applied to a specific type of data. The T_{vis} method is not designed to rigorously test the richness available in the *Remy* language, like the T_{math} tool. To expand, the number of examples the visualization programs would need to be chosen and written by hand.

[2]MIP or maximum intensity projection is a minimal volume visualization tool for 3-d scalar images.

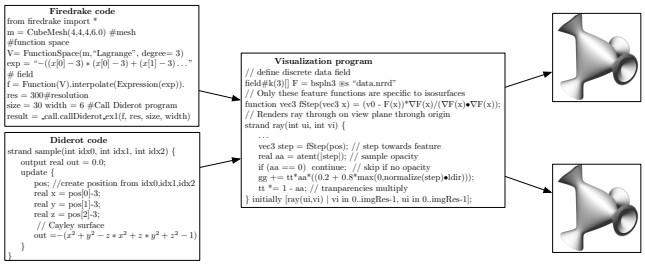

Figure 7: The implementation is similar to Figure 6, except that the expression represent the Cayley Cubic [20]**.**

6 RESULTS

In this section, we present four sets of results from using T_{math}. The first set applies T_{math} at different snapshots of the compiler to measure the state of the *Remy* compiler. The second set is a description of the bugs that were found in the compiler. The third set is measures the impact of changing the type of search (exhaustive or random). The fourth set is a measurement of how the time is spent in a single test case. Lastly, there is a discussion of the different approaches.

6.1 Experimental Framework and Definitions

The experiments were run on an Apple iMac with a 2.93 GHz Intel core i7 processor, 4GB memory, and OS X Capitan (10.11.6) operating system. Factors in the testing frame are constant unless otherwise indicated. A simple program includes one operator, while a nested computation includes two, and complicated includes three operators.

6.1.1 Testing output. The test results are organized by six different descriptions: unsupported, compilation issue, run-time, NA, results incorrect, and pass. Test programs that were not supported by the language syntax at the time are filtered as "unsupported". Test programs with a numerically correct "pass". The rest of the labels are different failure modes: compilation, run-time, and results-incorrect.

The first description, "unsupported" refers to a test program that uses syntax (operator, type, mesh) that was not yet enabled on *FE* fields. If created these test programs would result in a type checking error. The test results could be correctly categorize as a"compilation issue" but that can give a false impression for the number of test programs that were expected to work at a particular time. Instead each test program that can be generated but is not expected to work, due to language support is filtered into the "unsupported" category.

A "compilation issue" results in a program that can not be successfully compiled with *Remy*. A compilation error could be caused by a mistake in a rewriting step that halted compilation of the program, in the generated code created to evaluate a *FE* field, or a type error. Most rewriting issues were due to internal representation of *FE* field. A type error indicates an issue with the *Remy* type checker. A type error can occur because the *Remy* implementation did not have the language support that was expected.

A "run-time issue" describes a program that can compile but can not produce a result. This issue can be caused by code generation

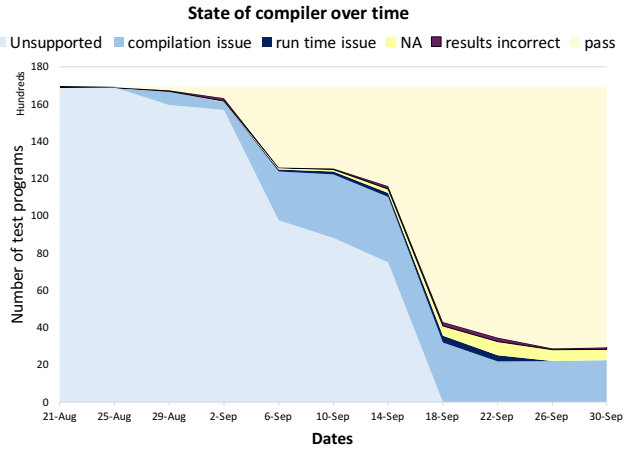

Figure 8: Progress of the implementation of *Remy* as measured by T_{math}.

or issues with memory allocation. An "NA" marks a test program that was discarded. This can be because the program was doing something mathematically invalid, such as taking the square root of a negative.

A numerical error indicates that the test program did compile and execute, but the *Remy* output is not within the error tolerance of the analytically derived result. These results are particularly interesting. These bugs would be difficult to find if comparing *Remy* to another version of the compiler.

6.2 Progress of *Remy*

This section reports the detailed results of running T_{math}. T_{math} was run on eleven different snapshots of the compiler. The snapshots of the compiler where taken off the *Remy* repository over four days apart, starting in August 2017. Each snapshot of the compiler measured the results for 16,917 test programs. The test programs were created to represent the ideal version of the language with a rich support of operators and types on *FE* fields. Each generated test program was initialized by the same testing frame. The core computation consisted of two nested operators for a variety of different types.

Figure 8 provides the results from the experiment with T_{math}. Over the course of development, more operators and types were introduced to the compiler, creating the opportunity to write and test more programs. A large number of test programs fail due to compilation issues and numerical issues. The number of test programs that do pass mostly increase, which shows progress in the development of T_{math} but it is imperfect.

At the last measured data point *Remy* still has bugs the developers need to fixed. The development in the compiler is ongoing. As discussed the compiler representation is complicated, dense, and obtuse. Testing shows the developer where to target their effort. The testing infrastructure has been crucial to compiler development.

6.3 Bugs

To date, T_{math} produced over 2000 programs that failed in some manner. We analyzed less than 10% of these to find the exact cause of failure and found 25 bugs: 15 compile-time issues, 3 run-time errors, and 7 numerically incorrect results. Since such a small portion of the failures have been analyzed, we expect to find many more bugs in the future despite the possibility that a single bug might explain multiple failures.

Errors in a single phase of the compiler can lead to different types of bugs and testing results. Therefore, this section describes bugs found in *Remy* organized by the source of error. The main sources of error include type checking (Section 6.3.1), internal representation of *FE* fields (Section 6.3.2) , and code generation (Section 6.3.3). Lastly, we provide some additional examples of bugs highlighted during run-time (Section 6.3.4).

6.3.1 Missing support. T_{math} can offer a full coverage of a set of operators. The testing coverage includes common computations that the user is expected to use and uncommon ones that a compiler writer is not likely to test. As a result it has found gaps in language expressivity. That is, programs that are expected to work do not, because of the missing support in the typechecker.

6.3.2 Internal representation. The bug illustrates how important it is to create test cases for the different combination of operators. Programs with nested operators offer a more rigorous test of the compiler than simple programs. The bigger computations are optimized inside the compiler. Creating bigger computations with nested operators is a way to test the optimization steps inside the compiler.

An error occurs when using two particular operators in a particular order. The application of the first operator is okay. It is the use of the second operator on the first that creates a specialized form. The computation raised a compilation error because the specialized form was not handled correctly by a rewriting step.

6.3.3 Misinterpretation of FEM types. The evaluation of *FE* field data requires several steps not previously supported in *Remy*. A single field is represented inside the compiler by a lower level construct that is translated into several functions in the generated code. Each function depends on some characteristics associated with the field, such as the mesh or differentiation level. If a program used two fields with the same mesh, then it should use just one call to mesh-dependent functions. When the process was not implemented correctly it lead to the following program failures:

(1) a compilation issue when the program duplicated the same function definition
(2) a numerically incorrect result when the same functions were used when they should not have been.

These issues can arise with a unique combination of fields types and function spaces. It illustrates the importance to creating test programs with multiple fields that have different function spaces.

6.3.4 Runtime Discovery. The following are examples of bugs that are highlighted during the run-time process.

Sampling outside the mesh: *FE* fields are well defined in a set region of space. If not careful, the probing process could sample outside this region creating an unsafe operation. When taking the derivative this property was not checked resulting in segfault.

Deleted memory: Computing the derivative of a first-order or higher order tensor resulted in a run-time error. It generates multiple calls to certain helper function to compute values. These helper functions free memory, and calling it multiple times lead to repeatedly deleting memory. This issue would not arise with other computations because scalar field helper functions did not delete memory.

Memory related Numerical Bug: Computing the derivative of a higher order tensor field resulted in an incorrect numerical result. Due to a misunderstanding of the tensor reference data structures used by *Remy*, helper functions incorrectly pass results to the evaluation function. This issue is only highlighted with higher-order tensor fields. Its shows the importance to using a variety of types while creating tests.

6.4 Timing and number of test programs

This experiment demonstrates the number of test programs that can be created with T_{math}. The testing input is the same as described previously. T_{math} is run with an exhaustive or random search setting. The test programs are run in parallel.

The timing experiment reported the following.

- It takes about twenty one minutes to run 100% of simple test programs (450 programs). The first step of testing the compiler included creating simple test programs.
- It takes about twelve minutes to run 1% of nested test programs (169 programs). It is useful to quickly check a diverse number of tests.
- It is possible to create 100% of nested test programs in about 20 hours (16,917). The ability to crate test programs by iterating over different operators and types is essential to catching bugs that are only highlighted with specific computations.
- It is possible to create hundred of thousands of complicated test programs (558,815). It takes about five hours to run 1% of complicated test programs. T_{math} can be used to create a vast number of programs.

T_{math} can be used to create a large number of simple to complicated test programs automatically and do faster regression testing.

6.5 Breakdown of a single test case

This experiment shows how the time in a single test produce by T_{math} is spent. The experiment runs a random search for 1% of test cases and generates nested test programs. The average time is measured for the different steps in the generation of the execution of a single test case.

There are five main steps involved in executing a single test case: setting up (1%), writing test programs (2%), making the *Remy* library (1%), creating and sampling *FE* field (84%), and evaluating the test program (2%). Figure 9 shows the average breakdown of time spent for a single test program. The implementation steps were described in more detail in Section 4.3.

The experiment measures the time spent generating a single test case in T_{math}. On average, 95% of the time is spent on the compiling and executing the test programs and the remaining 5% is spent on the creation and evaluation of the test programs. The

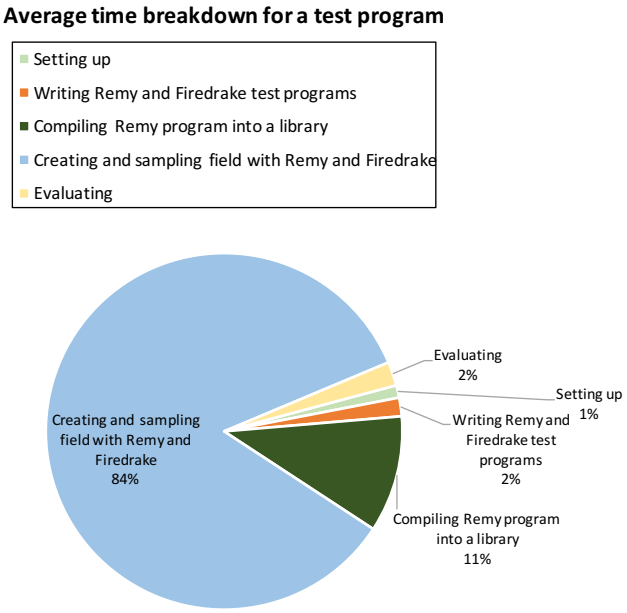

Figure 9: The average percentage of time spent in various areas in the course of creating and executing a single test case. The data comes from a random search of 1% of test cases and nested test programs.

time spent on the testing infrastructure (5%) is not substantial but it could indicate room for improvement.

6.6 Different approaches

Testing tools are created to find bugs but they can also provide some assessment for the state of the software. When a testing tool reports no bugs it can lead to a sense of confidence. If there are actual bugs in the software, the lack of reported bugs can lead to a false sense of security and frustrated end-users.

In this section we describe the results for using the two automated testing tools, T_{probe} and T_{math}, at different stages of development. The conditions for the experiment were described in Section 6.2. The results are measured on twelve different snapshots of the compiler over time. T_{probe} uses three meshes, two elements and four different polynomial degrees and in total it creates twenty-four tests.

While no testing tool can prove the absence of bugs, missing bugs that do exists does not help development. T_{probe} reports that all test pass at an early stage of development which can lead to a false sense of security. T_{math} found bugs that can not be found with T_{probe}. Therefore, it was crucial for compiler development to create T_{math} to find bugs that were not easily or possible to find with traditional testing approaches.

7 RELATED WORK

7.1 Types of Testing

Randomized differential testing (RDT) is way of testing by examining two comparable systems [21, 2, 3, 22]. When the results differ (or one crashes), then there is a test case for a potential bug. Many

applications of differential testing, use different variations of a compiler or system to test the same program. A setback to differential testing is if there is a common issue among systems then the bugs can go undetected.

RDT is a widely used method for testing compilers in practice. *Csmith* [2] is a tool that can generate random C programs with the goal of finding deep optimization bugs. The programs are expressive and contain complex code. Similar to T_{math}, Csmith effectively looks for deep optimization bugs in atypical combinations of language features. Unlike T_{math}, Csmith evaluates results by comparing various C compilers, and so has no ground truth.

*Metamorphic testing*is another way to test without an oracle [23]. Metamorphic testing [23], uses a metamorphic relation or property to test a family of test programs where the solution is unknown. The testing outputs are compared based on the relationship between the two programs [24]. The relation is typically referred to as the metamorphic relation and defined by the problem domain [25]. Donaldson [22] *et al.* applies metamorphic testing to shader-language compilers by using value injection. In the future, it would be interesting to do a type of metamorphic testing on *Remy*.

Property-based testing libraries and tools make it possible for a user to specify property-based tests [18, 26]. *QuickCheck* [18] is a widely used testing tool that allows Haskell programmers to test properties of a program. It is an embedded language for writing properties and can creates test cases that satisfy a condition. The test case generator is limited to a number of candidate test cases. *Equivalence modulo inputs*(EMI) creates a family of like programs [4].

There is work on testing specific properties of general purpose languages. *Herbie* is a tool can be used automatically to improve accuracy in floating point arithmetic [27]. The work introduces a method to evaluate the average accuracy of a floating-point expression and to localize the source of rounding error. It randomly samples inputs, generates rewriting candidates, and discovers rewrites to improve accuracy.

Different optimization levels (DOL) can be used to compare the output for comparable compilers at different optimization levels for the same program. *Palka et al.* generates random and type-correct programs for the Glasgow Haskell compiler [28]. The output of optimized and unoptimized versions of the compiler are compared. Chen *et al.* [3] compares various compiler testing techniques. They found DOL it was effective at finding optimization-related type bugs. The work in *Remy* is at the early stages. The code implemented is necessary to derive the most basic computations. There aren't different optimization levels that can be turned off and on to test our implementation of *Remy*.

7.2 Verification and Validation

T_{math} uses ground truth to evaluate test programs. To directly quote Etiene *et al.* "Verification is the process of assessing software correctness and numerical accuracy of the solution to a given mathematical model." [29]. The measure of correctness for computations written in *Remy* is based on how accurately the output of *Remy* program represents the mathematical equivalent of the computations.

Verifiable visualization allows us to apply a verification process to visualization algorithms [30, 29]. Instead of real-world datasets, one uses test cases with *manufactured solutions*. The manufactured solutions could be created in a way to predict result of algorithm with its Implementation when evaluating a known model problem. Etiene *et al.* derives formulas for the expected order of accuracy of isosurface features and compares them to experimentally observed results in order to provide confidence in behavior [31].

7.3 Domain-Specific Testing

There are a variety of domain-specific languages and frameworks that provide features similar to the features of *Remy*, such as Vivaldi [32] and ViSlang [33]. There is no published work on testing (automated or otherwise) for these DSLs. There are also programs that provide visualizations of *FE* fields, such as ELVis [34], and Gmsh [35]. There is no published work on testing these visualization programs although Gmsh does automatically tests its portability [35]. Also, they can not produce the range of visualization programs and operators that *Remy* is designed to support and can not serve as oracles. To our knowledge ELVis does not support exact evaluation of higher derivatives or allow easy communication with outside visualization software [34, 36] while Gmsh technically does not even support the same type of point-wise evaluation as *Remy* [35].

The importance of testing domain-specific languages has been discussed previously [37, 38]. Wu *et al.* introduces a framework, *DUFT*, to generate unit tests engines for DSLs [38] by adding a layer of DSL unit testing on top of existing general-purpose language tools and debugging services. DUFT tests DSL programs, but not the DSL implementation.

Ratiu *et al.* tested *mbeddr*, a set domain-specific languages on top of C built with HetBrains MPS language workbench [37]. Language developers define assertions from the specification of the DSL and defines a translation into the DSL.

DATm is a tool designed to test the mathematical operators available in the Diderot language[39]. Both *DATm* and T_{math} are embedded with domain specific knowledge about the compiler they are designed to test and share steps in the test generation pipeline. A tool user specifies the test settings, a test generator iterates over the different constructs available in the compiler, and creates individual test cases for valid computations. For each test case, Sympy is used to evaluate a core math computation.

The tools are largely different because T_{math} has embedded knowledge about the scientific computing domain. T_{math} not only tests the variety of operators and types available in Diderot but also includes constructs such as meshes, elements, function spaces, and new field definitions that are available in *Remy*. A test case in *DATm* is simple, while T_{math} generates multiple test programs for a single test and their execution is more complicated. In T_{math} the developer can initiate various settings to augment domain-specific properties in the test program, such as the mesh size, which can factor into how rigorous the test will be. Lastly, T_{math} can be run in parallel, which allows for a deeper search for test cases in a shorter amount of time.

7.4 Compiler testing and choosing test cases

There is extensive work on how to create and choose test cases. McKeeman [21] describes test case reduction and test quality with differential testing. When generating test cases *EasyCheck* [40] focuses on traversal strategy. Bernardy, *et al.* [41] present a scheme that leads to a reduction in the number of needed test cases. In addition to fixing types, they also fix some arguments passed to functions, effectively avoiding meaningless tests. A *Test set diversity metric* [42] is applied to ensure a diverse set of test cases by using information distance (regardless of datatype) when automatically creating tests. *LET* [6] is a model created for efficient compiler testing. The model learns characteristics of programs that reveal bugs and then prioritize test programs based on that.

There is an extensive amount of compiler testing for general purpose languages [2, 3, 4, 5, 6, 7]. Sun *et al.* finds and analyzes compiler warning defects. [7] Lindig *et al.* [5] automatically tests consistency of C compilers, specifically C calling conventions, using randomly generated programs. The tool evaluates if the functions (that are being tested) correctly receive parameters. T_{math} creates programs that are semantically more expressive than the programs created by this tool but as a default T_{math} does find issues with the calling conventions in the generated code created the evaluate *FE* fields.

8 FUTURE WORK AND CONCLUSION

Our work developing the *Remy* compiler has been guided by the testing approaches we created, presented here, to increase confidence in its correctness. These approaches evaluate simple programs (T_{probe}), computationally complicated programs (T_{math}), and visualization programs (T_{vis}). T_{probe} was used in the earliest stages of implementation, T_{math} uncovered bugs that would have been hard to discover otherwise, and T_{vis} will become more important as the complexity of real-world *Remy* programs increase. We describe here other possible testing approaches that may prove useful: property-based testing in the absence of mathematical ground truth, smarter generation of test programs, and better coordination between the testing tool and the *Remy* compiler.

8.1 Property-based testing

The visualization algorithms enabled by the *Remy* language are independent of the source of data, but the data itself can come in many forms and create new sources of error. A *Remy* program can involve a variety of *FE* field data and computations on that data. A subset of *FE* field data can be represented with polynomials, and we can use a *Domain-based* testing approach (Section 4) to effectively find bugs. For another much larger subset of *FE* field data, an alternate representation is more complicated. Furthermore, the alternative subsets of *FE* field data have different qualitative properties. To solve this potential coverage gap, it may be reasonable to use property-based testing.

Automatic creation of test programs will rely on being able to create random data for this subset of *FE* field data (PDEs) and choosing the right property. The drawback in randomly creating *FE* field data is in solving a PDE. The PDE solution process is outside the testing scope and can be both flawed and inflexible. The variety of randomly generated PDEs can be difficult if not impossible to solve. Additionally, the property selected must be flexible enough that it can be applied to various test programs, practical enough it can be measured by *Remy*, and rigorous enough to find bugs.

The canonical example of a testable property that a wide class of PDEs have is a maximum principle, which indicate that a PDE solution or some linear combination of its derivatives has a certain maximum that they obtain at a certain place [43]. This proved to be a challenging approach. The tool generated to check this property was not as effective at finding bugs that could be found with T_{math}. It was a step towards covering more testing ground. In the future, it can be useful to think of different properties for a property-based testing tool that can compliment our testing of the DSL.

8.2 Smarter test generation

Since *Remy* is a relatively new compiler, with bugs that are triggered with a specific combination of operators, it was necessary to take an exhaustive testing approach with a randomization option. T_{math} generates small *Remy* programs and there is not much need for minimization or reducing the program size. However, T_{math} could benefit from a better distribution of test cases when doing a random search. T_{math} could evaluate test cases based on previous test cases and by applying some size metric to potential new ones [42, 41].

Besides, the choice of test programs, the individual test programs can be more effective at testing evaluation. The positions used in a test programs are chosen at random. Smart positions could make it more likely to find bugs, due to the complicated mesh geometry. In practice, T_{math} lets the user specify the number of positions. In the future it may be more efficient to choose a small number of smart positions rather than a large number of random ones.

8.3 Coordination with *Remy* compiler

There is currently no way to automatically distinguish between the various types of compile-time issues. A single bug could cause multiple test cases to fail. Instead, *Remy* could generate error messages with labels that T_{math} could read. The bug log would then be organized by errors and produce a more informative report.

The subset of the *Remy* language that is being tested is clearly described. T_{math} is testing the fundamental computations and types of the language. T_{math} does not currently measure the how many lines of the *Remy* compiler are being tested. In the future it might be possible to use existing tools to mark and measure unused paths in the compiler.

There is also the possibility that the *Remy* compiler could itself by used as debugging tool. The work on visualizing FE data opens the door to more useful applications. *Remy* could potentially be used to visually debug and validate fields created by FEM and possibly reveal hidden details in data created with higher-order elements.

REFERENCES

[1] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics & Applications*, vol. 8, no. 5, pp. 29–37, 1988.

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 46, no. 6, pp. 283–294, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1993316.1993532

[3] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 180–190. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884878

[4] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 216–226. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594334

[5] C. Lindig, "Random testing of C calling conventions," in *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*. New York, NY, USA: ACM, 2005, pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/1085130.1085132

[6] B. Busjaeger and T. Xie, "Learning for test prioritization: An industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 975–980. [Online]. Available: http://doi.acm.org/10.1145/2950290.2983954

[7] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 203–213. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884879

[8] *SymPy website at http://www.sympy.org/en/index.html*, Sympy is a python library.

[9] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer, "Diderot: A parallel dsl for image analysis and visualization," *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 47, no. 6, pp. 111–120, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2345156.2254079

[10] G. Kindlmann, C. Chiw, N. Seltzer, L. Samuels, and J. Reppy, "Diderot: a domain-specific language for portable parallel scientific visualization and image analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 1, pp. 867–876, Jan. 2016.

[11] C. Chiw and J. H. Reppy, "Properties of normalization for a math based intermediate representation," *CoRR*, vol. abs/1705.08801, 2017. [Online]. Available: http://arxiv.org/abs/1705.08801

[12] S. C. Brenner and R. Scott, *The mathematical theory of finite element methods*. Springer, 2007, vol. 15.

[13] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake: Automating the finite element method by composing abstractions," *ACM Trans. Math. Softw.*, vol. 43, no. 3, pp. 24:1–24:27, Dec. 2016. [Online]. Available: http://doi.acm.org/10.1145/2998441

[14] *DOLFIN*, DOLFIN, available from https://bitbucket.org/fenics-project/dolfin.

[15] T. Dupont, J. Hoffman, C. Johnson, R. C. Kirby, M. G. Larson, A. Logg, and R. Scott, *The FEniCS project*. Chalmers Finite Element Centre, Chalmers University of Technology, 2003.

[16] D. Eberly, *Ridges in Image and Data Analysis*. Kluwer Academic Publishers, 1996.

[17] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake: Automating the finite element method by composing abstractions," *ACM Transactions on Mathematical Software*, vol. 43, no. 3, pp. 24:1–24:27, Dec. 2016. [Online]. Available: http://doi.acm.org/10.1145/2998441

[18] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of Haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM, 2000, pp. 268–279. [Online]. Available: http://doi.acm.org/10.1145/351240.351266

[19] D. Weiskopf and G. Erlebacher, "Overview of flow visualization," in *The Visualization Handbook*, C. Hansen and C. R. Johnson, Eds. Academic Press, 2004, ch. 12, pp. 261–278.

[20] "Cayley cubic," http://mathworld.wolfram.com/CayleyCubic.html.

[21] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998. [Online]. Available: http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[22] A. F. Donaldson and A. Lascu, "Metamorphic testing for (graphics) compilers," in *Proceedings of the 1st International Workshop on Metamorphic Testing, International Conference on Software Engineering 2016, Austin, Texas, USA, May 16, 2016*, 2016, pp. 44–47. [Online]. Available: http://doi.acm.org/10.1145/2896971.2896978

[23] T. Y. Chen, "Metamorphic testing: A simple method for alleviating the test oracle problem," in *Proceedings of the 10th International Workshop on Automation of Software Test*, ser. AST '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 53–54. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819261.2819278

[24] Z. Q. Zhou, D. H. Huang, T. H. Tse, Z. Yang, H. Huang, and T. Y. Chen, "Metamorphic testing and its applications," in *PROCEEDINGS OF THE 8TH INTERNATIONAL SYMPOSIUM ON FUTURE SOFTWARE TECHNOLOGY (ISFST 2004)*. Software Engineers Association, 2004.

[25] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[26] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values," in *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, ser. Haskell '08. New York, NY, USA: ACM, 2008, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/1411286.1411292

[27] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2015, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737959

[28] M. H. Palka, K. Claessen, A. Russo, and J. Hughes, "Testing an optimising compiler by generating random lambda terms," in *Proceedings of the 6th International Workshop on Automation of Software Test*. New York, NY, USA: ACM, 2011, pp. 91–97. [Online]. Available: http://doi.acm.org/10.1145/1982595.1982615

[29] C. T. S. Tiago Etiene, Robert M Kirby, *An Introduction to Verification of Visualization Techniques*, 3rd ed. Clifton Park, New York: Morgan and Claypool Publishers, Dec. 2015, vol. 7.

[30] I. Babuska and J. Oden, "Verification and validation in computational engineering and science: Basic concepts," *Computer Methods in Applied Mechanics and Engineering*, vol. 193, no. 36-38, pp. 4057–4066, 9 2004.

[31] T. Etiene, C. Scheidegger, L. G. Nonato, R. M. Kirby, and C. T. Silva, "Verifiable visualization for isosurface extraction," *Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1227–1234, 2009.

[32] H. Choi, W. Choi, T. Quan, D. Hildebrand, H. Pfister, and W. Jeong, "Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2407–2416, Dec. 2014.

[33] P. Rautek, S. Bruckner, M. E. Gr"oller, and M. Hadwiger, "Vislang: A system for interpreted domain-specific languages for scientific visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2388–2396, 2014. [Online]. Available: https://www.cg.tuwien.ac.at/research/publications/2014/Rautek_Peter_2014_VSA/

[34] B. Nelson, E. Liu, R. M. Kirby, and R. Haimes, "ElVis: A system for the accurate and interactive visualization of high-order finite element solutions," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2325–2334, dec 2012. [Online]. Available: https://doi.org/10.1109%2Ftvcg.2012.218

[35] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities," *International Journal for Numerical Methods in Engineering*, vol. 79, no. 11, pp. 1309–1331, sep 2009. [Online]. Available: https://doi.org/10.1002%2Fnme.2579

[36] "GLVis: Accurate finite element visualization," glvis.org.

[37] D. Ratiu and M. Voelter, "Automated testing of DSL implementations: Experiences from building mbeddr," in *Proceedings of the 11th International Workshop on Automation of Software Test*. New York, NY, USA: ACM, 2016, pp. 15–21. [Online]. Available: http://doi.acm.org/10.1145/2896921.2896922

[38] H. Wu, J. Gray, and M. Mernik, *Unit Testing for Domain-Specific Languages*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 125–147. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03034-5_7

[39] C. Chiw, G. Kindlmann, and J. Reppy, "Datm: Diderot's automated testing model," in *Proceedings of the 12th International Workshop on Automation of Software Testing*, ser. AST '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 45–51. [Online]. Available: https://doi.org/10.1109/AST.2017.5

[40] J. Christiansen and S. Fischer, "Easycheck: Test data for free," in *Proceedings of the 9th International Conference on Functional and Logic Programming*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 322–336. [Online]. Available: http://dl.acm.org/citation.cfm?id=1788446.1788479

[41] J.-P. Bernardy, P. Jansson, and K. Claessen, "Testing polymorphic properties," in *Proceedings of the 19th European Conference on Programming Languages and Systems*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 125–144. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11957-6_8

[42] R. Feldt, S. Poulding, D. Clark, and S. Yoo, "Test Set Diameter: Quantifying the Diversity of Sets of Test Cases," *ArXiv e-prints*, Jun. 2015.

[43] L. C. Evans, *Partial Differential Equations: Second Edition (Graduate Studies in Mathematics)*. American Mathematical Society, 2010. [Online]. Available: https://www.amazon.com/Partial-Differential-Equations-Graduate-Mathematics/dp/0821849743?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0821849743