2017 IEEE/ACM 12th International Workshop on Automation of Software Testing

DATm: Diderot's Automated Testing Model

Charisee Chiw, Gordon Kindlmann, and John Reppy
Department of Computer Science, University of Chicago Email:{cchiw,glk,jhr}@cs.uchicago.edu

***Abstract*—Diderot is a parallel domain-specific language for the analysis and visualization of multidimensional scientific images, such as those produced by CT and MRI scanners [6], [14], [5]. Diderot is designed to support algorithms that are based on differential tensor calculus and produces a higher-order mathematical model which allows direct manipulation of tensor fields. One of the main challenges of the Diderot implementation is bridging this semantic gap by effectively translating high-level mathematical notation of tensor calculus into efficient low-level code in the target language.**

A key question for a high-level language, such as Diderot, is how do we know that the implementation is correct. We have previously presented and defended a core set of rewriting rules, but the full translation from source to executable requires much more work. In this paper, we present DATm, Diderot's automated testing model to check the correctness of the core operations in the programming language. DATm can automatically create test programs, and predict what the outcome should be. We measure the accuracy of the computations written in the Diderot language, based on how accurately the output of the program represents the mathematical equivalent of the computations.

This paper describes a model for testing a high-level language based on correctness. It introduces the pipeline for DATm, a tool that can automatically create and test tens of thousands of Diderot test programs and that has found numerous bugs. We make a case for the necessity of extensive testing by describing bugs that are deep in the compiler, and only could be found with a unique application of operations. Lastly, we demonstrate that the model can be used to create other types of tests by visual verification.

I. Introduction

To discover the structure and dynamics of physical and biological specimens, scientists employ imaging methods like computed tomography (CT) scanning and microscopy to generate detailed 3D datasets. To understand this data requires advanced image analysis and visualization methods. Because Diderot implements such a high-level programming model, it is a significant compilation effort to bridge the gap between surface-language operations and the hundreds of lines of C ++ code that is required to implement these operations. With such a large semantic gap, the question of correctness of the compilation process is a serious concern.

Testing a compiler for a high-level mathematical programming-language poses a number of challenges not found in previous work on testing compilers. While it is easy to write down complicated mathematical expressions to feed to the compiler, it is difficult to predict what the correct answer should be. For this reason, manual construction of tests for the Diderot compiler was time consuming and prone to biases (*i.e.*, combinations of operations that were easy for the test author to understand). Furthermore, Diderot is a rich language with many operators, so the space of possible combinations is too large for manual exploration. Thus, as in previous work, it is vital that we build a testing tool that can automatically generate test cases that provide good coverage of the features of the language [16], [8].

There is extensive research in compiler testing. Differential testing relies on comparing various versions and implementations of the compiler [17], [25]. Equivalence Modulo Inputs [15] creates a family of programs that can be used for differential testing. Alone, these approaches did not seem sufficient in the case of Diderot. Most of the substantial transformations that occur during compilation are necessary and not reasonable to disable. Additionally, earlier versions of the compiler are much less expressive than the current implementation. Our goal is to validate the correct answer for the new operations in the Diderot language. We chose to develop a ground truth for our test cases by careful construction of data sets and careful choices of points of evaluation.

We present Diderot's automated testing model. It is designed to rigorously test the core mathematical parts of the Diderot implementation. **DATm** combines generation of test programs with generation of synthetic data for which the correct values and properties of Diderot program output is known. For each test program, synthetic data is used to synthesize tensors and tensor fields that are being used in the Diderot program. The solution can then be derived analytically as an operation on polynomials. The generated Diderot program is compiled and run on the test data. **DATm** compares the result of the Diderot program with the analytically-derived correct answer and if the answers are within an error tolerance, the test passes.

We also demonstrate how the testing model can be extended to automatically create and test different Diderot programs. We apply a metamorphic testing technique to do visual verification on an important class of algorithms. Metamorphic testing is used to evaluate unknown solutions based on some property.

DATm can offer a full coverage of a set of operators. This includes common computations that the user is expected to use and uncommon ones that a compiler writer is not likely to test. It has found various bugs in the Diderot compiler and has enabled quick debugging of new operators. It is designed to aid development by supporting quick reproducibility of test cases, providing exhaustive testing , and random testing. It has provided other unexpected benefits, such as identifying mathematically valid programs that were unnecessarily rejected by the compiler because of artificial limits in the typing rules.

The remainder of the paper is organized as follows. We first introduce Diderot's Automated Testing model and describe its implementation. We describe how the model can be extended to automatically test a class of visualization algorithms. We

978-1-5386-1548-5/17 $31.00 © 2017 IEEE
DOI 10.1109/AST.2017.5

then present results about both the effectiveness of **DATm** in finding bugs and about **DATm**'s efficiency in generating and running tests. We discuss future improvements and conclude in Section VI.

II. Background

We rely on the expressivity of the Diderot language to support the implementation of visualization ideas. Visualization algorithms involve computing certain properties from a dataset. The mathematical core of these ideas are ingrained in tensor calculus. Central to them are operations on and between tensors fields. For a scientist or mathematician, it might be natural to think of these concepts first in a mathematical notation rather than in lower-level code. The actual implementation of visualization algorithms can be laborious, and inefficient. Diderot eases the transformation of ideas into workable intuitive code by allowing the math-like notation to be written directly into the language [4].

In Diderot, algorithms can be directly expressed on tensor and tensor fields. The type **`tensor`**`[`ς`]` is a tensor with shape ς. Tensors refer to scalars, vectors, and matrices. The type **`image`**`(`d`)[`κ`]` represents image data that is sampled on a regular grid. A kernel type **`kernel`**`#k` is represented as a C^k kernel. The language defines a group of built in kernels. The type **`field`**`#`k`(`d`)[`κ`]` is a tensor field with continuity k, shape κ, and dimension d ($1 \leq d \leq 3$).

Diderot's notation is similar to that used in tensor calculus but with Unicode characters. This includes arithmetic operations such as the inner product $\bullet$ and cross product $\times$. Fields are created for convolution between an image data and kernels $\circledast$. Diderot also supports differentiation of tensor fields ∇.

III. DATm

This paper describes the technique we used to test the new operators added to the Diderot language by introducing two models (Figure III-E2): **DATm** a model that evaluates based on equality and **DAVm** a model that evaluates based on symmetry. We will give an overview of **DATm** in the following section and then provide further details.

A. Overview

This section will introduce **DATm** as illustrated in Figure III-E2 and relate it to common terminology. The *test requirements* are described in the frame and serve as input to the test generator. The testing frame defines several key factors for **DATm** such as what is being tested and how to search for test cases. The generator creates a test description for each test case. The test description describes an application of operators to arguments with different Diderot types. The test description can also be considered the *test input*.

The testing process involves creating synthetic data and a test program given the test description. Synthetic data is created for each argument and is represented with a data file (test.nrrd) and symbolic expression. A Diderot program is generated by using the corresponding synthetic data and list of points. The output of the executed Diderot program is a data file (out.nrrd). The analytically derived solution is found by evaluating the symbolic expression at the same series of points as the Diderot test program.

We want to ensure that the Diderot program compiled. The *test output* is a nrrd file created by Diderot and a sympy expression. We expect that both output reduces to a series of numbers. The *expected behavior* is numerical equality between the output.

τ	::=	**`tensor`**`[`ς`]`	tensor with shape ς
		`field``#`k`(`d`)[`κ`]`	tensor field with continuity k, shape κ, and dim d
		`image``(`d`)[`κ`]`	image data with shape κ, and dim (d)
d_n	::=	2 | 3	dimension
v_n	::=	2 | 3 | 4	extended dimension
κ	::=	nil' | v_1 | $d_1 d_1$	Field shape

Figure 1. Subset of Diderot types that can be tested with **DATm**

B. Generating test cases

Automatically building test cases is one of the goals of **DATm**. Test cases are based on a subset of the Diderot language types τ (Figure 1) and operations (Figure 2). The core of a test case is an application of an operator to expressions. An exhaustive generation of test cases can be found by iterating over the various types and operators in the scope, then internally typechecking it to filter out the applications that do not make mathematical sense. Test generation is parameterized by the testing frame which is explained in the next section.

C. Testing Frame

The testing frame is defined by the settings and scope. The *settings* indicates how to initialize various variables. The *scope* describes the subset of types and operators included.

1) Settings: The settings initialize the following:

- *Data Creation factors* when creating synthetic data. This includes order of coefficients linear, quadratic, and cubic, and the quantity and orientation of samples taken (by randomizing shear and angle). Using fewer samples and creating a non-isotropic grid creates a stronger field reconstruction (a core operation for Diderot) test.
- *Test program details* and restrictions to the scope of test cases. Generated test programs probe a tensor field at a set number of positions and use a specified reconstruction kernel (Figure 2). The restriction of the test cases include the number of operators (1-3), and limitations to argument types (such as only tensor fields).

ops	::=	$-$, $\lVert\cdot\rVert$, `@` , $\surd$, ∇, $\nabla\otimes$, $\nabla\times$, $\nabla\bullet$, `inverse`,
	|	**`normalize`** , `trace` , `transpose` , `det`,
	|	`sin` , `cos` , `tan` , `arccos` , `arcsin` ,
	|	`[:,0]` , `[1]` , `[:,1,:]` , `[1,0,:]` , ...
	::=	`+` , `-` , `*` , `/` , $\bullet$, `:` , $\times$, $\otimes$, $\circledast$, `modulate`
kernel	::=	`c4hexic` , `tent` , `ctmr` , `bspln3`

Figure 2. Subset of Diderot operators and kernels that can be tested with **DATm**

- *Random or exhaustive exploration* of test cases. It is not always practical to run an exhaustive test and create tens of thousands of programs. The developer specifies the probability of a single test program being executed in the testing frame. 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.

2) Scope: The scope is the set of possible programs that can be tested. It is possible to specify a single test case or target a subset of test cases with command line arguments. This can aid the process of debugging and enable more targeted testing. The scope of the testing falls into three modes:

1) Run all possible test cases described under the testing frame. In the exhaustive case, this can be every possible combination of operators and arguments types supported by **DATm**. In the random mode, the test generator will randomly choose the test cases to create.
2) Targeting a group allows us to limit the testing scope to some argument(s). This can be helpful when testing a new operator added to the language, by only creating tests that involve that operator, *i.e.* the inverse of an argument.
3) Run a single case, *i.e.* the addition of two 2-d scalar fields.

D. Generate Diderot program

DATm supports a scripted instantiation of test program templates for each test program. Type description of arguments τ are converted to Diderot types. Tensor field values are initialized by loading a nrrd file.

```
field#k(2)[2] F = load("F.nrrd") ⊛ c4hexic;
tensor[2] T = [expT1,expT2];
```

Each operator is defined by attributes that allow its scripted printing in Diderot programs. The outer product operator :

$\text{op}_{outer} = \{out : \otimes,\ placement : \text{middle},\ arity : 2,\ limit : \text{None}\}$

indicates the placement and arity of unicode symbol $\otimes$ in

```
field#k(2)[2,2] G = inv (F⊗T);
```

The field is probed at multiple points in the field domain with the application of the inside test.

```
if(inside(F,pos)){... }
```

The result of computation is the observed value.

```
tensor[2,2] observed = G(pos);
```

Once the Diderot program is written, it is compiled and executed. The resulting nrrd file is read as observed data.

E. Data Creation

DATm offers a scripted generation of synthetic data used in the testing process. This is done by defining the values to tensor and tensor field arguments as polynomials. Random numbers serve as coefficients to polynomial expressions. For each argument in a test case, we create a representation for Diderot and Python.

1) Nrrd Format: In a Diderot program, a tensor field is instantiated with a data file (in a `F.nrrd` format [22]) and a reconstruction kernel. The file maintains sampling orientation and discrete data points. A Diderot template can be synthesized with arguments that represent a polynomial expression.

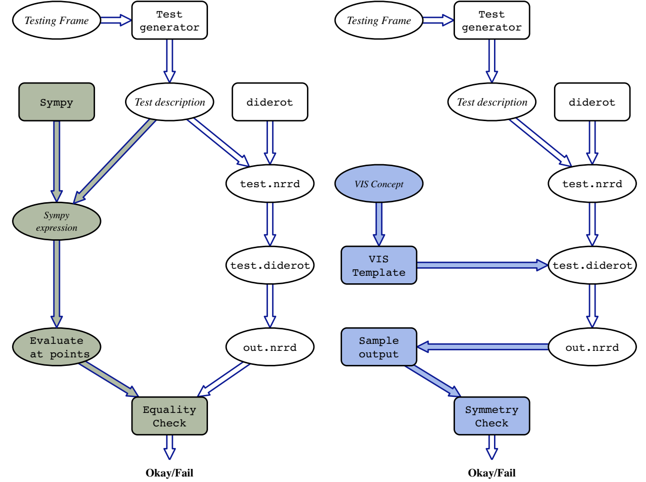

Figure 3. Pipeline of Diderot's Automated Testing model (left) and an extension of it **DAVm** (right). Highlighted are the different portions to the pipeline. **DATm**, and **DAVm** evaluate the correctness of Diderot test programs based on equality and symmetry, using Sympy, and a visualization concept, respectfully.

The template takes samples from the function created from the polynomial expression and saves it to a Diderot nrrd file (test.nrrd on left in Figure III-E2). The number of samples and their orientation are based on parameters in the frame.

2) Sympy Expression: The operators we are testing are tensor calculus-based operators. This gave us the opportunity to use Python sympy package [20] to analytically derive the correct solutions independently of Diderot. A symbolic expression can be created from a list of coefficients, and handled easily ($exp_1 = x^2 + 3x + 4$). It can be differentiated, manipulated with a series of tensor operators, and evaluated at points.

F. Evaluation

The same coefficients that are used to synthesize a Diderot tensor field are also used to create a Sympy expression. The operators that applied in the generated Diderot test program can also be applied between expressions in Python. The expected test output is then the result of running the Diderot test program and the evaluated Sympy expression.

We expect to be able to compare the test output based on equality. If the output are within some error tolerance then the test passes. There are three different possible failure modes. A type error indicates an issue with the Diderot type checker. A compilation error could be caused by a mistake in a rewriting step that halted compilation of the program. A numerical error indicates that the test program did compile and execute, but the Diderot output is not comparable to the analytically derived result.

G. Numerical Instability

We choose to evaluate test results based on a ground truth. Still, the evaluation is comparing the output of floating point arithmetic. The potential rounding errors that can occur

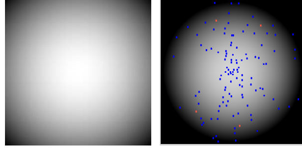

Figure 4. Volume rendering of a symmetric 3-d field. Summation projection (left), Maximal projection (right) with positions that are sampled to evaluate correctness. We sampled 30 groups of 4 points that were equal distance from the center. Highlighted in red are one of the groups.

from floating point arithmetic are well-known and experienced [10]. It is possible that the numerical instability of floating point arithmetic results in a false positive. **DATm** does take some precautions against doing operations with undefined results. Each operation is tagged with a condition attribute. For instance, the square root operation $\sqrt{e}$ is tagged by a condition that limits the argument to only positive numbers.

IV. Visualization Verification

This section demonstrates a modest way to do automated visual verification of the Diderot language by using metamorphic testing. Programs written for scientific visualization or image analysis can be more mathematically complicated than what we have tested. It is possible that the complexity could bring to light more bugs. Evaluating results of a visualization program based on the numerical accuracy can be difficult (if not impossible), because sometimes we can't easily know what the algebraic solution should be. Therefore, each visualization program had required an "eyeball test" in addition to already discussed drawbacks of writing tests by hand. That is, just visually looking at the resulting image and seeing if was as expected. In an effort to include more types of testing, we created Diderot's Automated Visualization model **DAVm** and use a known property to evaluate an unknown result.

A. Concept

To test Diderot, we need to construct a visualization program that can be checked in an automated way. We choose to compute a simple volume renderings of synthetic 3-D fields created by the new operators. Projecting a rotationally 3-D symmetric field, restricted to a spherical domain, should produce a rotationally symmetric 2-D image, regardless of the view point and the field operations involved. **DAVm** generates Diderot programs to (1) do a volume rendering and (2) a sampling of the output of the volume rendering. Figure 4 provide examples of the output from these programs. In the following we describe aspects to **DAVm** that depend on the visualization concept.

B. Pipeline

DAVm, an extension of **DATm**, uses much of the same basic code. It was created to generate meaningful tests while keeping the benefits of automated and random testing. Figure III-E2 provides the pipeline for **DAVm**. **DAVm**, like **DATm**, is used to generate test cases, create synthetic data, write Diderot programs, and evaluate each test program. In **DAVm**, the evaluation of those programs do not use sympy and instead is grounded in symmetry checks.

1) Generate Test cases: A Diderot program is used as a template for creating test programs. We use a MIP program [9] as the base code for a basic volume rendering program in Diderot. We expect that the field being probed `G` is a symmetrical 3-d scalar field. The core of the MIP program gets the maximum value along the ray and stores it `out = max(out, G(pos));` Another template can be used to do a summation projection volume rendering by using: `out += out;` Figure 4 presents example images from these programs.

2) Evaluation: The correct numerical answers for these visualization programs are unknown (the projections would involve potentially unwieldy symbolic integrations) but we can use symmetry arguments to detect major bugs. We test the symmetry of the output by sampling the field at groups of points, at the same distance from the center of the image, and comparing them. Each point in a group is at the same distance from the center. For each point, the x-coordinate is chosen randomly, and the corresponding y-coordinate is chosen using the distance formula. This maintains some notion of randomization while probing the field at meaningful positions.

V. Performance and Results

In this section, we present four sets of results from using **DATm**. The first set is from an experiment that measures the impact from changing the type of search (exhaustive or random). **DATm** can be used to create a vast number of programs automatically and do faster regression testing. The second set is from an experiment that applies **DATm** at different snapshots of the compiler. This experiment demonstrates that a lot of bugs were not being caught until we developed **DATm**. The third set is a description of the bugs that were found in the compiler. We provide examples of bugs that would have been especially difficult to find without **DATm**. The fourth set is from a simple experiment that runs **DAVm**. It demonstrates that it is possible to do other types of tests with an extension of the testing model.

A. Experimental Framework

The experiments were run on an Apple iMac with a 2.7 GHz Intel core i5 processor, 8GB memory, and OS X Yosemite (10.10.5) operating system. The experiments may run different sets of tests by changing some settings in the testing frame, but the following factors in the testing frame are constant. To create synthetic data, quadratic coefficients are used. To create nrrd files for a tensor fields 70 samples are taken and the sampling orientation is not randomized. The generated test program used c4hexic kernels to reconstruct tensor fields and 7 positions to probe the fields. We omit times for doing exhaustive testing with a high number of nested operators in favor of doing random testing in this large search space.

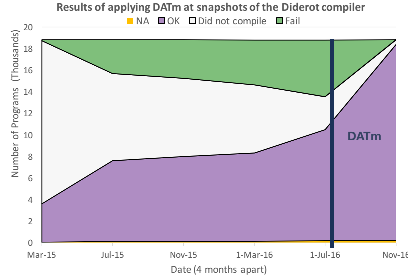

Figure 5. Results when running **DATm** over time. Categories"OK", "Fail", "Did not compiler", and "NA" indicates a program passed, failed, it did not compile, or was thrown out, because conditions set by the operators where not met, respectfully. The vertical line marks when **DATm** was introduced.

No. of Ops	No. of Programs	Time given probability.			
		0 %	1%	10%	100%
1	695	0.25	0.55	4.65	32.53
2	18,819	7.62	18.35	121.23	1099.16
3	495,626	58.83	393.2	-	-

Figure 6. The following offers measurements from executing **DATm** with different settings. The settings are (1) the number of operators, and (2) the probability to run a single test case. A 0% probability refers to iterating test cases only, a 100% is an exhaustive test, and the range in-between refer to a random search with a set probability to execute each test case. This figure records the number of test programs that are created with an exhaustive search, and the timing measurement (in minutes) for running **DATm**

B. Exhaustive vs. Random Testing

This experiment demonstrates the ability to do exhaustive and random testing with **DATm**. It executes the model by initializing the testing frame with different settings. The changes vary the number of test programs that are created and how they are found. We report the impact of changing these variables by recording how long testing takes and number of test cases available.

Our experiment does a single exhaustive search and several random searches for up to 3 nested operators. Table 6 records the timing measurement from doing these different searches and varying the number of nested operators. The measurements range from seconds to hours. Table 6 records the number of test programs that are created with an exhaustive search.

The experiment demonstrates that **DATm** can create a vast number of test programs. This capability has made it possible to find hidden and unique bugs. **DATm** can create thousands of programs, but it is not feasible to do exhaustive search each time there is a change to the compiler. To enable quicker regression testing it is necessary to also do random testing. The developer can choose to parametrize **DATm** to do random testing or exhaustive testing while targeting a single operator.

C. Progress

To evaluate the effectiveness of **DATm** we ran the same set of programs on six different snapshots of the compiler. The snapshots of the compiler where taken off the Diderot repository over four months apart, starting from March 2015. To be clear, we did a post evaluation of the state of the compiler using **DATm**. The testing frame was set to the same constants as the previous experiment, and we did an exhaustive search with 2 nested operators. The set of programs included almost 19 thousand tests.

The test results are organized by three different categories, "failed", "compilation error", or "passed". The "failed" description means that there was a clear error because the numerical result was not correct, or there was an error when executing the program. The "compilation error" are the programs that did not compile. This can include type errors from testing operations that were not part of the language syntax at the time. The experiment measures the number of tests programs that fall into these different categories. Figure 5 provides the results from the experiment.

Over time, the number of passing tests is inconsistent, rather than constantly improving. This indicates that new errors may have been introduced and not caught. The test does not indicate the number of bugs, and we expect that many of tests fails due to the same compiler bug. Various compilation errors may be due to earlier versions of the language not fully supporting all the functionality. The experiment shows that development of Diderot rapidly changed once **DATm** was introduced. At the latest data point, Diderot did not fail any of the tests. Previously, the bugs were unknown and unfixed.

D. Bugs

DATm has found various bugs in the Diderot compiler when it was developed. **DATm** found seven compile time errors, three numerical errors, and eight type errors. While there might not be many bugs, some of the compile and type errors were caused by complicated mistakes in the compiler. The numerical bugs would have been difficult to find without **DATm**. The type error bugs account for missing features and unexpected restrictions in the language. The following provides a description of some of the bugs and provides motivation for extensive testing.

1) Unique Bugs: **DATm** discovered bugs that could only arise with a unique combination of operators. Bugs of this nature are unlikely to be found by a Diderot user and difficult to identify in the code alone do to its complexity. Space does not permit describing a full list of bugs, but a few examples follow.

a) Ex1. Numerical error caused by complicated transformation: This bug was an example of a rare numerical error in the output, which was exposed by computing:

```
field#k(3)[3,3] G = transpose(∇⊗(V));
```

The Diderot compiler must generate code to map derivatives from image-index space to world space. A subtle error arises in tracking the variable index that represents the shape of the tensor field. Because of the use of `transpose`, the index is swapped with another, which gives rise to the error. It is the unique combination of two operators (transpose of a differentiated vector field) that triggers the bug.

b) Ex2. Bug exposed by testing nested operators: Programs with nested operators reach deeper parts of the Diderot compiler. This is because some algorithms are designed to optimize complicated expressions. The trace of the modulate between a negation of A and B can be computed as

No. of Ops	No. of Programs	Time given probability.	
		1 %	100 %
1	15	.01	16
2	216	3.13	344
3	3,151	81.63	-

Figure 7. Results from running **DAVm**. The time is given in minutes from an exhaustive and random search.

```
G = trace(modulate(-A, B));
```

The application of the trace operator on modulate is okay. It is the use of a third operator that triggers an algorithm to be applied. The computation raised a compilation error because the case was not handled correctly by that algorithm.

2) Language expressivity: An unexpected benefit of implementing **DATm** was the gut-check to Diderot's advertised expressivity. Previous work offered a type specification for the Diderot language. In actuality, the implementation provided the wrong type. This error did not cause numerically incorrect results, but instead restricted the use of the operator. Fixing it makes Diderot a more complete language.

E. Visualization Results

In this section, we present the results of running **DAVm**. The experiment created test programs based on the MIP template. To sample the result, we created 30 groups of 4 points equal distance from the center. The right-most image in Figure 4 is an example of the output from this type of testing. In the image, the points sampled are imposed on the volume rendering of the test program. This experiment in Figure 7 measures the the number of programs and time.

Testing with **DAVm** is limited. It takes longer to execute and it is limited to subset of Diderot programs (3-d scalar fields). It did not find errors, but perhaps because it was created after **DATm**. This experiment does demonstrate that it is possible to extend **DATm** to evaluate tests with unknown numerical answers.

VI. Discussion

a) Verification: Verifiable visualization allows us to apply a verification process to visualization algorithms. 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. We use the idea of verification [1], [23] as a guiding metric for testing. 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." [23]. The measure of correctness for computations written in the Diderot language, is based on how accurately the output of the Diderot program represents the mathematical equivalent of the computations.

b) Types of Testing: Randomized differential testing (RDT) is way of testing by examining two comparable systems [17], [25], [3], [10]. When the results differ (or one crashes), then there is a test case for a potential bug. This is a widely used method for testing compilers in practice. **Csmith** [25] 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 **DATm**, Csmith effectively looks for deep optimization bugs in an atypical combination of language features. Donaldson [10] *et al.* applies metamorphic testing to graphic compilers by using value injections. When comparing images, they note that small differences in rendered images can occur even when there is no compiler bug. **Mettoc** creates a family of programs and compares them using an equivalence relation [21]. Palka *et al.* generates random and type correct programs for the Glasgow Haskell compiler [18]. The output of optimized and unoptimized versions of the compiler are compared. **QuickCheck** [8] is a widely used testing tool that allows Haskell programmers to test properties of a program. It is an embedded language for writing properties. The type checker creates test cases that satisfy a condition.

It has been valuable to evaluate our test programs based on a ground truth, and an application of different types of testing could complement our testing process. While it is not helpful to test against earlier versions of Diderot (due to extensive language developments), we could possibly create a family of programs and do some variation of differential testing [17]. This paper introduced a way to do visual verification for programs with unknown algebraic solution. Our application of **DAVm** demonstrated how volume rendering templates can be used to generate test programs. We believe that it is possible to build other types of visualization test programs in this way.

In the future, it would be interesting to evaluate the effectiveness of different testing approaches on the Diderot compiler. Chen *et al.* [3] compares various compiler testing techniques. Besides RDT, they use "different optimization levels" (DOL) where they compare the output for comparable compilers at different optimization levels for the same program. They found it was effective at finding optimization-related type bugs.

c) Domain-Specific Testing: There are a variety of domain-specific languages that provide similar features to Diderot, **Vivaldi** and **ViSlang**. There is no published work on testing (automated or otherwise) for these DSLs. Wu *et al.* introduces a framework, **DUFT**, to generate unit tests engines for DSLs [24] by adding a layer of DSL unit testing on top of existing general-purpose language tools and debugging services. Unlike **DATm**, 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 [19]. Language developers define assertions from the specification of the DSL and how it would be translated to the target language.

d) Test case generation: There is extensive work on how to create and choose test cases. **Csmith** [25] generates C programs minimizing unknown behavior. Palka randomly generated lambda terms [18]. Lindig [16] tested C calling conventions. The tool generates the types of functions and checked that the parameters were received. McKeeman[17] describes test case reduction and test quality with differential

testing. When generating test cases **EasyCheck** [7] focuses on traversal strategy. Bernardy, et. al [2] present a scheme that leads to reduction in the number of needed test cases. It exhaustively enumerates on possible data types. **Test set diversity metric** [13] is applied to ensure a diverse set of test cases. **DATm** generates small Diderot programs (for hefty tensor computations), and there is not much need for minimization or reducing the program size. There is a need to evaluate test case distribution when implementing random search in a deep test space. To make this feasible, **DATm** could do smarter test generation by evaluating based on existing cases or applying some size metric.

e) Coverage: The subset of the Diderot language that is being tested is clearly described. **DATm** is testing the fundamental computations and types of the language, while **DAVm** puts those computations in the core part of a visualization program. What is not clear yet is how many lines of the Diderot compiler are being tested.

f) Testing in Parallel: The time it takes to run a large numbers of tests is a limiting factor in the usefulness of the tool. **DATm** takes from 2–5 seconds per test (depending on the test's complexity), which limits its use to either long runs or very sparse random testing. Fortunately, it should be fairly easy to run multiple tests in parallel using the parallelism of Unix processes on multicore servers or workstations.

VII. Conclusion

We have presented **DATm**, an automated testing tool for a high level language. **DATm** provides a practical way to test Diderot by creating thousands of tests and evaluates the results based on a ground truth. We have described the details in implementing **DATm** as well as an extension to create other types of tests. We have provided examples of bugs **DATm** was able to find in Diderot.

Acknowledgments

Portions of this research were supported by National Science Foundation under award CCF-1564298. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

References

[1] Ivo Babuska and J. Tinsley Oden. Verification and validation in computational engineering and science: Basic concepts. *Computer Methods in Applied Mechanics and Engineering*, 193(36-38):4057–4066, 9 2004.

[2] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 125–144, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 180–190, 2016.

[4] Charisee Chiw. Ein notation in the diderot compiler. http://diderot-language.cs.uchicago.edu/papers/chiw-masters.pdf.

[5] Charisee Chiw, Gordon L Kindlman, and John Reppy. EIN: An intermediate representation for compiling tensor calculus. In *Proceedings of the 19th Workshop on Compilers for Parallel Computing (CPC 2019)*, July 2016.

[6] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 111–120, June 2012. (alphabetical author order).

[7] Jan Christiansen and Sebastian Fischer. Easycheck: Test data for free. In *Proceedings of the 9th International Conference on Functional and Logic Programming*, FLOPS'08, pages 322–336, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[9] Diderot. website at https://github.com/Diderot-Language/examples.

[10] Alastair F. Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2016, Austin, Texas, USA, May 16, 2016*, pages 44–47, 2016.

[11] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: Functional enumeration of algebraic types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 61–72, New York, NY, USA, 2012. ACM.

[12] Tiago Etiene, Carlos Scheidegger, L Gustavo Nonato, Robert M Kirby, and Cláudio T Silva. Verifiable visualization for isosurface extraction. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1227–1234, 2009.

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

[14] Gordon Kindlmann, Charisee Chiw, Nicholas Seltzer, Lamont Samuels, and John Reppy. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. *IEEE Transactions on Visualization and Computer Graphics (Proceedings VIS 2015)*, 22(1):867–876, January 2016.

[15] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226, 2014.

[16] Christian Lindig. Random testing of C calling conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 3–12, New York, NY, USA, 2005. ACM.

[17] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[18] Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM.

[19] Daniel Ratiu and Markus Voelter. Automated testing of DSL implementations: Experiences from building mbeddr. In *Proceedings of the 11th International Workshop on Automation of Software Test*, AST '16, pages 15–21, New York, NY, USA, 2016. ACM.

[20] Sympy is a python library. *http://www.sympy.org/en/index.html*.

[21] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*, pages 270–279, 2010.

[22] Teem Library. *Teem website at http://teem.sf.net*.

[23] Cludio T Silva Tiago Etiene, Robert M Kirby. *An Introduction to Verification of Visualization Techniques*, volume 7. Morgan and Claypool Publishers, Clifton Park, New York, 3rd edition, December 2015.

[24] Hui Wu, Jeff Gray, and Marjan Mernik. *Unit Testing for Domain-Specific Languages*, pages 125–147. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.