GenCoG: A DSL-Based Approach to Generating Computation Graphs for TVM Testing

Zihan Wang
Shanghai Jiao Tong University
Shanghai, China
wangzh99@sjtu.edu.cn

Pengbo Nie
Shanghai Jiao Tong University
Shanghai, China
yuemonangong@sjtu.edu.cn

Xinyuan Miao
Shanghai Jiao Tong University
Shanghai, China
mxinyuan@sjtu.edu.cn

Yuting Chen*
Shanghai Jiao Tong University
Shanghai, China
chenyt@sjtu.edu.cn

Chengcheng Wan
East China Normal University
Shanghai, China
wancc1995@gmail.com

Lei Bu
State Key Laboratory for Novel Software Technology
Nanjing University
China
bulei@nju.edu.cn

Jianjun Zhao
Kyushu University
Fukuoka, Japan
zhao@ait.kyushu-u.ac.jp

ABSTRACT

TVM is a popular deep learning (DL) compiler. It is designed for compiling DL models, which are naturally computation graphs, and as well promoting the efficiency of DL computation. State-of-the-art methods, such as Muffin and NNSmith, allow developers to generate computation graphs for testing DL compilers. However, these techniques are inefficient — their generated computation graphs are either type-invalid or inexpressive, and hence not able to test the core functionalities of a DL compiler.

To tackle this problem, we propose GenCoG, a DSL-based approach to generating computation graphs for TVM testing. GenCoG is composed of (1) *GenCoGL*, a domain-specific language for specifying type constraints of operators, and (2) an *approach* that concolically solves type constraints and incrementally generates computation graphs of high expressivity. We implement and evaluate GenCoG on TVM releases. Our results show that GenCoG is effective in generating valid and expressive computation graphs — all of the GenCoG-generated graphs pass type checking, a critical graph validation stage; letting the graphs' expressivity be measured by their vertex and edge diversities, GenCoG outperforms state-of-the-arts by achieving 1.65~6.93× in vertex diversity and 1.06~7.08× in edge diversity, respectively. Furthermore, GenCoG has detected 16 bugs in TVM v0.8 and v0.9, with 14 confirmed and 12 fixed.

*Yuting Chen is the corresponding author.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598105

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Compilers**; *Domain specific languages*; • **Computing methodologies** → *Neural networks.*

KEYWORDS

Deep Learning Compiler, Computation Graph Generation, Constraint Solving

ACM Reference Format:
Zihan Wang, Pengbo Nie, Xinyuan Miao, Yuting Chen, Chengcheng Wan, Lei Bu, and Jianjun Zhao. 2023. GenCoG: A DSL-Based Approach to Generating Computation Graphs for TVM Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598105

1 INTRODUCTION

In recent years, a variety of deep learning (DL) models have been proposed [11, 14, 33] to solve the challenging problems in different domains. To alleviate the heavy burden of deploying DL models on various hardware devices, DL compilers have been developed [1, 30]; they automatically import models from DL frameworks and generate optimized code for target platforms and devices. Among them, TVM [4] is a popular one — it was designed as a cornerstone of DL, and has been integrated into various machine learning frameworks for high performance [19, 39].

Similar to traditional compilers [41], TVM is also subject to bugs [32]. As Figure 1 shows, the software architecture of TVM consists of several lowering- and optimization-oriented components. A bug in any of the components may either interrupt model deployment or lead to incorrect results produced by the deployed models. Correspondingly, developers can generate DL models, which are naturally computation graphs, and feed them to TVM. Many efforts do exist in automatically generating DL models for testing

DL compilers, such as LEMON [35], Muffin [12], Luo et al. [22], NNSmith [20]. In this way, they are able to test core functionalities in the graph compilation flow of TVM.

Despite the above efforts, due to the complex software architecture of TVM, it is still not easy to sufficiently test the TVM implementation, due to the two main challenges.

Challenge 1: Validity. A computation graph consists of many operators. Each operator takes one or more tensors as input and contains *attributes* that determine its actual behavior. The operator imposes constraints on the tensor shapes and data types (collectively referred to as *tensor types*) of inputs, as well as attributes. The type constraints of operators are non-trivial, due to the possibly large number of tensor shape dimensions and attribute elements, as well as their intricate relations.

It is necessary to generate type-valid computation graphs, as any violation of the constraints leads to an immediate rejection of the graph during type checking of the TVM's compilation flow, and thus the successive components cannot be exercised. Let 10k graphs be generated through *pure random sampling*. As Figure 2(a) shows, only 5.7% of graphs having 4 vertices can successively pass type checking, and less than 1% of graphs having 6 vertices are type-valid.

Challenge 2: Expressivity. We use the term "expressivity" to indicate how much useful information (operators, wirings, tensors, attributes, etc.) computation graphs carry for DL compiler testing. Expressive computation graphs are presumed to be effective in testing the TVM — as what we will show in §3, some compiler bugs can only be revealed by computation graphs with specific operator calls and/or wiring combinations. More specifically, there are two levels of requirements for test computation graphs. At the *operator* level, there should be as many as possible unique combinations of input tensor types and attributes in calls to each kind of operator. During compilation, only certain combination(s) can trigger TVM bugs. At the *graph* level, the topologies of the graphs, as well as the wiring combinations between operators should be as diverse as possible. Since TVM performs graph-level optimizations, it needs to guarantee that these optimizations correctly handle graphs of various possible topologies and operator combinations. Let operator coverage [22] be used to measure the expressivity of the graphs at the operator level. As Figure 2(b) shows, type-valid graphs having 8+ vertices can only cover less than 50% of operators, and thus are inexpressive in TVM testing.

Few techniques successfully tackle both challenges at the same time. For example, LEMON [35] suggests the idea of mutating real-world DL models to generate more complicated ones, while it restricts the mutation to operators whose input and output tensor shapes are identical. Muffin [12] and Luo et al. [22] leverage predefined structure templates or random graph models to generate graph structures, followed by filling in tensor types and operator attributes. However, they do not fully support type constraints of common operators and their graphs miss certain wiring combinations. NNSmith [20] incrementally generates computation graph for testing DL compilers. However, it requires large human efforts to write low-level SMT constraints for each operator; it also fails to guarantee the expressivity of computation graphs. To the best of our knowledge, few techniques can generate both type-valid and expressive computation graphs for sufficiently testing TVM.

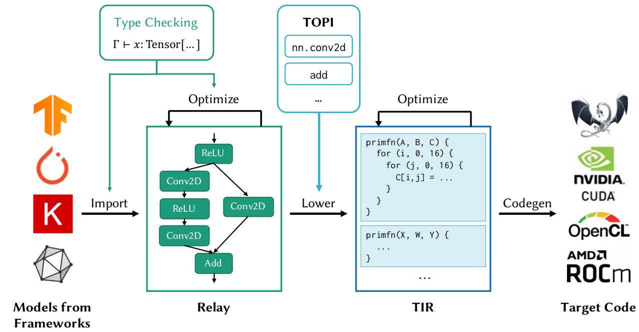
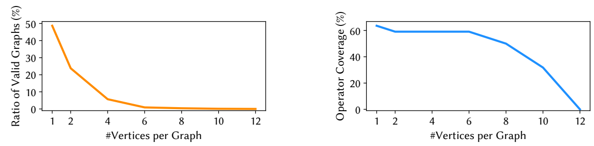


Figure 1: Compilation flow of TVM.



(a) Validity (measured by ratio of valid graphs). **(b) Expressivity (measured by operator coverage).**

Figure 2: Validity and expressivity of 10k graphs generated through *pure random sampling* with increasing graph size.

Our Work. To tackle the challenges in DL compiler testing and overcome the limitations of existing methods, we specify the constraints on operators using a domain-specific language, measure a graph's expressivity by its vertex and edge diversities, and then present GenCoG (Generator of Compution Graphs), a DSL-based approach to computation graph generation for TVM testing. Our core insight is that solving type constraints helps generate computation graphs that are both *type-valid* and *expressive*. On validity, we take type checking as the critical graph validation stage in TVM. As long as a computation graph meets the type constraints of operators and passes type checking, it is able to exercise the subsequent core functionalities of the compilation flow. On expressivity, we believe that the type constraints define the space of feasible combinations of input tensor types and attributes of an operator. By adequately exploring this space, along with trying different wiring combinations, GenCoG generates expressive computation graphs which can reveal deep bugs in TVM.

We make the following contributions in this paper:

(1) **Language.** We design GenCoGL, a domain-specific language (DSL) for specifying type constraints of operators. GenCoGL is expressive in representing constraints on tensor types and operator attributes. GenCoGL also takes a classified approach to specification, making its constraints concise and comprehensible.
(2) **Algorithm.** We propose two algorithms for computation graph generation. The first algorithm incrementally generates computation graphs with an expressivity-directed strategy, allowing rich operator calls and wiring combinations

to be constructed. The second concolically solves type constraints in GenCoGL, producing type-valid and expressive operator calls. With both algorithms, GenCoG promotes expressivity of generated graphs at operator and graph levels.

(3) **Evaluation.** We implement and evaluate GenCoG on TVM v0.8 and v0.9[1]. Our results show that GenCoG is effective in generating type-valid and expressive computation graphs — all of the GenCoG-generated graphs pass type checking, a critical graph validation stage; GenCoG outperforms state-of-the-arts by achieving 1.65~6.93× in vertex diversity and 1.06~7.08× in edge diversity, respectively. Furthermore, GenCoG has detected 16 bugs in TVM v0.8 and v0.9, with 14 confirmed and 12 fixed.

The rest of this paper is organized as follows: §2 and §3 present the background and an illustrative example, respectively. §4 introduces GenCoGL, our DSL for specifying type constraints. §5 provides the details of the graph generation approach. §6 evaluates GenCoG. §7 presents related work and §8 concludes.

2 BACKGROUND

2.1 TVM and Relay IR

TVM is an end-to-end DL compiler stack. It compiles DL models to executable programs by a compilation flow shown in Figure 1. The input to TVM is a model defined in any supported DL framework. The compiler frontend imports the model and converts it to Relay Intermediate Representation (IR). Relay is the common high-level IR for models imported from different DL frameworks, and it keeps the structure and constraints of these models.

In this paper, GenCoG generates computation graphs in the form of Relay IR. The Relay IR goes through several high-level optimizations including graph substitution [9, 16] and operator fusion [27, 43]. Then the operators in Relay IR are lowered to their specific implementations in low-level Tensor IR (TIR). Such implementations are predefined in the Tensor Operator Inventory (TOPI). At the TIR level, many implementation-specific optimizations are performed. The code generator finally translates TIR to target code, which is eventually compiled into an executable program deployed on a device.

For Relay IR, type checking is a critical graph validation stage. Type checking checks whether the input tensor types and attributes meet the type constraints of each operator, and computes the output tensor types. Type checking is frequently called in TVM: TVM runs this stage both after converting a model from DL framework to Relay IR and after applying any optimization on Relay IR. If a computation graph passes type checking, it is then processed by the successive compilation flow and is able to exercise other key components in TVM.

2.2 Preliminaries

The core of GenCoG is to specify and solve type constraints of operators for graph generation. We formally define several concepts related to this topic.

Definition 2.1 (Computation Graph). A computation graph $G = (V, E, X)$ is a direct acyclic graph (DAG) with tensor data flow. V is the vertex set and each vertex $v \in V$ represents a call to an operator. E is the edge set and each directed edge $(u, v) \in E$ represents a dependency of v on u. X is the tensor set and each tensor $x \in X$ is a symbolic value produced by an operator. Each $(u, v) \in E$ maps to a tensor $x \in X$, which is the specific value for which v depends on u.

Definition 2.2 (Tensor Type). For each $x \in X$, its tensor type $type(x)$ has the following form:

$$\texttt{Tensor}[(d_1, d_2, \ldots, d_k), dt]. \tag{1}$$

$(d_1, d_2, \ldots, d_k)$ is the *shape* of the tensor and k is the *rank*. dt is the *data type* of the tensor.

Definition 2.3 (Operator). An operator op is a function that performs primitive computation or transformation on input tensors. op is called in the following form:

$$y_1, \ldots, y_n = op(x_1, \ldots, x_m)\{l_1 = a_1, \ldots, l_p = a_p\}. \tag{2}$$

$x_1, \ldots, x_m$ are inputs and $y_1, \ldots, y_n$ are outputs of the operator. $\{l_1 = a_1, \ldots, l_p = a_p\}$ is a dictionary of *attributes* of the operator call. The attributes are *named* compile-time constants that determine the behavior of the operator.

Definition 2.4 (Type Constraint). The type constraint on an operator op is a predicate on the types of its inputs and outputs, as well as its attributes:

$$\begin{aligned} &P_{op}(type(x_1), \ldots, type(x_m); type(y_1), \ldots, type(y_n); \\ &\quad l_1 = a_1, \ldots, l_p = a_p). \end{aligned} \tag{3}$$

3 AN ILLUSTRATIVE EXAMPLE

GenCoG consists of (1) GenCoGL, a DSL for specifying type constraints of operators in TVM, and (2) an incremental graph generation approach that concolically solve type constraints to generate expressive graphs. We present an example, as Figure 3 shows, to illustrate how GenCoG generates computation graphs.

Operators and Type Constraints. Operators are the building blocks of a computation graph, and each call to an operator should satisfy the constraints imposed by the operator. There are three operators involved in this example, including `dense`, `expand_dims`, and `add`. Figure 3 shows the definitions of the attributes, input and output tensor shapes of these operators. Their definitions already contain several in-place constraints, such as the attribute `axis` $\in [0, k]$ in the operator `expand_dims`. Tensor data types are trivial in this example and we assume that they are all `float32`. *Extra constraints* are the constraints on the operator attributes and input tensor types in addition to their original definitions. The operator `add` contains such an extra constraint, which is named the *broadcasting rule.*

Type constraints of operators are varied in their forms. From Figure 3 we can see that these constraints can be classified into three categories:

(1) *Numerical constraints*, including equalities or inequalities involving integer or floating point arithmetics. The constraint `axis` $\in [0, k]$ of `expand_dims` is a numerical constraint.
(2) *Logical constraints*, which are propositions joint by logical connectives. Usually, the clauses in a logical constraint are numerical constraints. The above-mentioned broadcasting rule of `add` can serve as an example.

[1] Source code is available at https://github.com/wzh99/GenCoG.

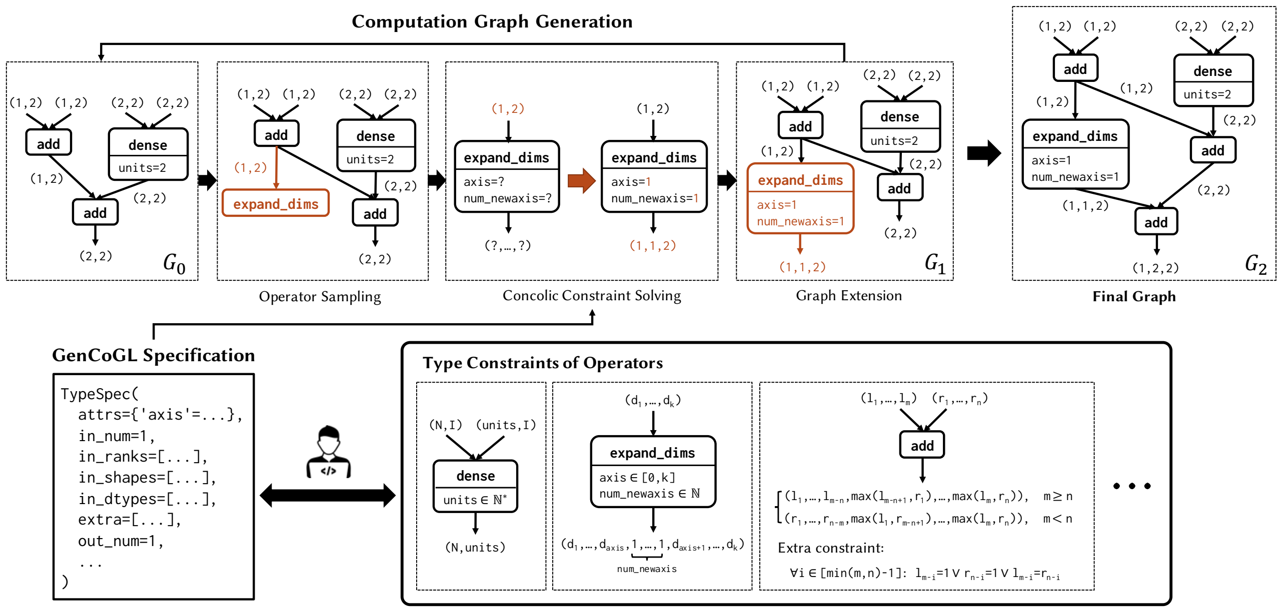


Figure 3: An illustrative example of computation graphs generated by GenCoG.

(3) *Structural constraints*, which define the length and elements of a list. Structural constraints are prevalent because tensor shapes and some attributes are represented as lists.

In GenCoG, users are expected to specify the type constraints of operators in GenCoGL, a domain-specific language. As shown in Figure 2, a GenCoGL specification specifies the type constraints on the operators such as `add`, `dense`, `expand_dims`. The specification is further used in computation graph generation, guaranteeing the validity of the graph during each iteration. The details of GenCoGL will be introduced in §4.

Computation Graph Generation. The general process of computation graph generation is also illustrated in Figure 3. GenCoG applies an incremental generation process, which extends the graph with one vertex during each iteration. Given a computation graph G_0, GenCoG grows the graph to G_1 by adding an operator call to `expand_dims`, connecting it with `add`, and solving the attributes and tensor types of the call to `expand_dims`. With an expressivity-directed strategy and a concolic constraint solving technique, GenCoG incrementally generates type-valid and expressive computation graphs.

The computation graph G_1 is further grown to G_2. As Figure 3 shows, G_2 owns five operator calls, each having a unique combination of tensor shapes. There are four wiring combinations between operators as well. This computation graph triggers an operator fusion bug in TVM, which will be further explained in §6.5.

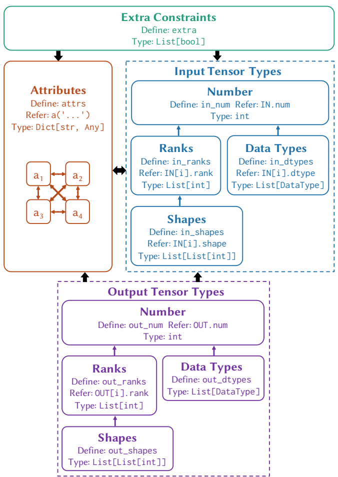
4 GENCOGL: A DSL FOR SPECIFYING TYPE CONSTRAINTS OF OPERATORS

We design GenCoGL, a Python-embedded DSL, for specifying type constraints in an expressive and comprehensible way. GenCoGL is fundamental to GenCoG in generating valid and diverse computation graphs.

e	::=	c	(constant)
	|	`var` $*$? τ? r?	(variable)
	|	s	(symbol)
	|	e *bop* e	(binary operation)
	|	e *cop* e	(comparison operation)
	|	`not` e	(negation)
	|	`and` $e*$	(conjunction)
	|	`or` $e*$	(disjunction)
	|	`forall` r $\lambda s.e$	(universal quantification)
	|	`cond` e e e	(conditional)
	|	`[` $e*$ `]`	(list literal)
	|	`list` e $\lambda s.e$	(list construction)
	|	e `[` e `]`	(element access)
	|	e `[` e `:` e `]`	(slice)
	|	`len` e	(list length)
	|	`concat` $e*$	(concatenation)
	|	`map` e $\lambda s.e$	(transformation)
	|	`reduce` e *bop* c	(reduction)
	|	`filter` e $\lambda s.e$	(filter)
	|	`in` e e	(set membership)
	|	`subset` e e	(subset relation)
	|	`a` l	(attribute reference)
	|	`(IN | OUT).num`	(arity reference)
	|	`(IN | OUT)` `[` e `]` `.(rank | shape | dtype)`	(tensor type reference)
c	::=	`true` | `false` | $\mathbb{Z}$ | $\mathbb{R}$ | l | dt	(constant)
τ	::=	`bool` | `int` | `float` | `str` | `DataType`	(type annotation)
r	::=	`range` e e	(range)
bop	::=	$+ \mid - \mid \times \mid / \mid$ mod | min | max	(binary operator)
cop	::=	$= \mid \neq \mid < \mid \leq \mid > \mid \geq$	(comparison operator)

Figure 4: Abstract syntax of GenCoGL expressions.

Rich Constraint Forms. GenCoGL supports various constraint forms, including numerical, logical and structural constraints. Each constraint is expressed by a GenCoGL expression, and the expressivity of GenCoGL stems from the richness of its expressions. Figure 4 shows the abstract syntax of GenCoGL expressions. The expression forms involve constants, variables, arithmetics, logics, lists, sets, and domain-specific references. Developers write these expressions



(a) Complete structure of a constraint specification.

```
TypeSpec(
  # Requirements on attributes and
  # input tensor types
  attrs={
    'newshape': List(
      Var(ran=Range(1, 6)),
      lambda _: Var(int, tmpl=True)
    )
  },
  in_num=1,
  in_ranks=[Var()],
  in_shapes=[
    List(
      IN[0].rank,
      lambda _: Var(tmpl=True)
    )
  ],
  in_dtypes=[Var()],
  extra=[
    Reduce(IN[0].shape, '*', 1) ==
    Reduce(a('newshape'), '*', 1)
  ],

  # Guarantees on output tensor types
  out_num=1,
  out_ranks=[Len(a('newshape'))],
  out_shapes=[a('newshape')],
  out_dtypes=[IN[0].dtype]
)
```

(b) Constraint specification of the reshape operator.

Figure 5: The structure of a constraint specification and a corresponding example in GenCoGL.

by calling their corresponding Python APIs. With these expression forms, GenCoGL supports type constraints for almost all of the common operators.

Classified Specification. In GenCoGL, a constraint specification contains several expressions. To organize these expressions into a well-formed specification, GenCoGL takes a classified approach, where each class focuses on one aspect of the specification.

As Figure 5(a) shows, attributes are defined as a dictionary in field `attrs`, and an attribute named l can be referred to by an expression `a` l. For both input and output tensor types, GenCoG further splits the specification into four classes (number, ranks, data types, and shapes, respectively). This allows the constraints of tensor types to be specified at different granularities. Any class for input tensor types can refer to any attribute item, and vice versa. Any class for output tensor types can refer to any attribute item and any class for input tensor types. Extra constraints are a list of Boolean expressions, each of which explicitly defines a predicate on the elements of attributes and input tensor types.

With its classified approach, GenCoGL guides developers to organize type constraints in a hierarchy. It helps developers write correct, concise and comprehensible constraint specifications.

Case Study. Figure 5(b) shows the constraint specification of the `reshape` operator. *First*, there is one attribute `newshape`, defining the target shape. `newshape` is provided with a list constructor, presenting that its structure is a list whose length is within the range $[1, 6)$. The body of the lambda expression is a template variable, indicating that each element is a unique integer variable. *Second*, `reshape` has only one input tensor and it can be of arbitrary rank and shape. We define one variable as the rank of the input tensor in `in_ranks`, which is referred to by `in_shapes` for defining the shape of the input tensor. *Third*, the number of elements in the output tensor must be equal to the one of the input tensor. We define this constraint in the `extra` field of the specification, which computes the numbers of elements in input and output tensors and equate both numbers. *Fourth*, `reshape` has one output tensor, which has the same data type as the input. The rank and shape of the output is provided by `newshape`.

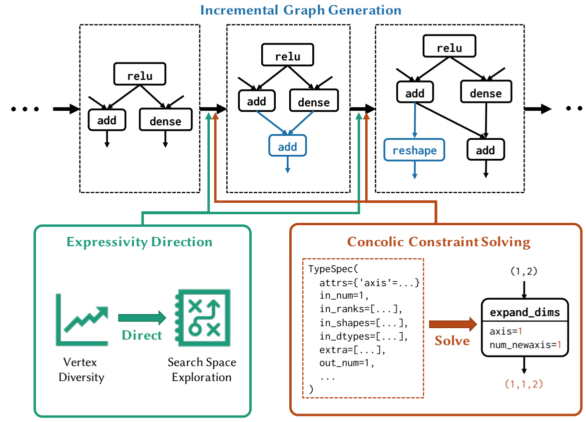


Figure 6: The graph generation process of GenCoG.

5 APPROACH

As shown in Figure 6, GenCoG follows a general process presented in NNSmith [20], which incrementally generates computation graphs. Furthermore, GenCoG adopts an *expressivity-directed* strategy and a *concolic constraint solving* method, which guarantees the type-validity and expressivity of the graphs. The generated computation graphs are then used to reveal deep bugs that interrupt the deployment of DL models or lead to incorrect results.

5.1 Expressivity-Directed Graph Generation

GenCoG is an expressivity-directed approach to generating expressive graphs, aiming at triggering core compilation and optimization functionalities of the TVM during testing.

Expressivity Metric. Given a set of computation graphs, we simply measure their *expressivity* by their *vertex and edge diversities* (a.k.a., vertex and edge coverage in [22]).

Definition 5.1 (Vertex Diversity and Edge Diversity). Given an operator set O and a test suite $\mathcal{G}$ of computation graphs, we define vertex diversity $div_V(\mathcal{G})$ and edge diversity $div_E(\mathcal{G})$ as follows:

$$div_V(\mathcal{G}) = \frac{1}{\#O} \sum_{o \in O} \frac{\#\{(v.\text{in\_types}, v.\text{attrs}) \mid v.\text{op} = o, v \in G.V, G \in \mathcal{G}\}}{N_{est}(o)}, \quad (4)$$

$$div_E(\mathcal{G}) = \frac{\#\{(o_1, o_2) \mid v_1.\text{op} = o_1, v_2.\text{op} = o_2, (v_1, v_2) \in G.E, G \in \mathcal{G}\}}{(\#O)^2}, \quad (5)$$

where `in_types`, `attrs`, and `op` are the input tensor types, the attribute dictionary, and the operator of a vertex v, respectively; N_{est} is a rough estimate of the size of an operator o's call space.

Vertex diversity evaluates the variety in combinations of input tensor types and attributes of operator calls, corresponding to the expressivity at the operator level. Edge diversity evaluates the variety of wiring combinations between any two operators, approximating the expressivity at the graph level.

Algorithm 1: Generating a computation graph.

Input: Operator set O, vertex limit n_v, rejection probability p_{rej}.
Output: Computation graph G.

1 $x_0 \leftarrow$ Create tensor with randomly generated tensor type τ;
2 $X \leftarrow \{x_0\}, V \leftarrow \emptyset$;
3 $\mathcal{R} \leftarrow \{o \mapsto \{\} \mid o \in O\}$;
4 **while** $\#V < n_v$ **do**
  ▷ *Sample the first input tensor and operator*
5   $x_1^\iota \leftarrow$ Sample a value in X;
6   $O_{\text{cand}} \leftarrow \{$the first input of o is compatible with $x_1^\iota \mid o \in O\}$;
7   $op \leftarrow$ Sample an operator in O_{cand} with Equation 6;
  ▷ *Sample attributes and the rest input tensors through constraint solving*
8   Try to solve specification S for op given $[type(x_1^\iota)]$ with Algorithm 2;
9   **if** *solving failure is reported* **then continue**;
10   $X^\iota \leftarrow$ Try to match tensors in X according to input tensor types specified by S;
11   **if** X^ι *cannot be constructed* **then continue**;
12   Try to solve S again given $[type(x_i^\iota) \mid x_i^\iota \in X^\iota]$;
13   **if** *solving failure is reported* **then continue**;
  ▷ *Create vertex*
14   Evaluate output tensor types in S;
15   $X^o \leftarrow$ Create output tensors with types in S;
16   $v \leftarrow$ Create a call to op with $S.\texttt{attrs}, X^\iota$ and X^o;
  ▷ *Possibly reject the vertex if it is not unique*
17   **if** $(v.\texttt{in\_types}, v.\texttt{attrs}) \in \mathcal{R}[op]$ **then**
18   **if** $\texttt{rand()} < p_{\text{rej}}$ **then continue**;
19   **else**
20     $\mathcal{R}[op] \leftarrow \mathcal{R}[op] \cup \{(v.\texttt{in\_types}, v.\texttt{attrs})\}$;
  ▷ *Extend the graph with the new vertex*
21   $V \leftarrow V \cup \{v\}, X \leftarrow X \cup X^o$;
22 Create computation graph G with V and X;
23 **return** G;

Algorithm 2: Solving type constraints of an operator.

Input: Constraint specification S, known types of input tensors $T^\iota = [\tau_1^\iota, \dots, \tau_m^\iota]$.
Output: Solved specification S.

Specialize S given T^ι;
while $\texttt{true}$ **do**
  ▷ *Simplify expressions until a fixed point is reached*
  repeat
    $alt \leftarrow \texttt{false}$;
    ▷ *Simplify attributes and input tensor types*
    foreach $l \mapsto e \in S.\texttt{attrs}$ **do**
      $alt \leftarrow alt \vee \texttt{SimplifyAttr}(l, S)$;
    $alt \leftarrow alt \vee \texttt{SimplifyTypes}(S)$;
    ▷ *Simplify extra constraints*
    $C \leftarrow \emptyset$;
    foreach $e \in S.\texttt{extra}$ **do**
      $e' \leftarrow \texttt{SimplifyExpr}(e, S)$;
      $alt \leftarrow alt \vee (e \neq e')$;
      if $e = \texttt{false}$ **then** report solving failure;
      else if $e \neq \texttt{true}$ **then** $C \leftarrow C \cup \{e'\}$;
    $S.\texttt{extra} \leftarrow C$;
  until $\neg alt$;
  ▷ *Constrainedly sample variables*
  $C_s \leftarrow \texttt{IdentifySolvable}(S)$;
  if $\neg\texttt{ConstrainedSample}(C_s, S)$ **then break**;
if *S has unsolved constraints* **then** report solving failure;
return S;

Algorithm. GenCoG takes an expressivity-directed strategy in generating computation graphs — The higher expressive a computation graph, the more diverse the vertices/edges (and the wiring combinations), and thus the more compiling/optimization logic of a DL compiler be triggered during testing. [2]

Correspondingly, GenCoG uses vertex diversity to direct graph generation. The direction of vertex diversity is applied in two steps of the generation process: (1) selecting an operator from candidates, and (2) deciding to add a new vertex to the graph. When sampling an operator, GenCoG first computes a normalized score $score(o_i)$ for each operator o_i in the candidate operator set O_{cand}, according to its contribution to increase in vertex diversity in the recent generation process. Then, the candidate operators are sampled from the following Softmax probability:

$$Prob(o_i) = \frac{\exp(score(o_i))}{\sum_{o_j \in O_{\text{cand}}} \exp(score(o_j))}. \tag{6}$$

If an operator o_i has a large search space, it will continuously contribute to the increase in vertex diversity, thus achieving higher $score(o_i)$ and higher probability of being chosen.

After a vertex is successfully constructed through constraint solving, GenCoG checks whether it increases the expressivity. If the expressivity is increased, the vertex can definitely be added to the graph. Otherwise, GenCoG rejects the vertex with probability p_{rej}. A vertex may still be kept in this case as it would be helpful for increasing edge diversity.

Algorithm 1 shows the procedure of generating a computation graph. At first, an initial tensor x_0 is created to serve as the input of the whole graph (Line 1). Then the algorithm iteratively generates the graph until the vertex limit n_v is reached (Line 4). During each iteration, a tensor is firstly chosen (Line 5), and a matching operator is sampled with probability in Equation 6 (Line 7). The algorithm determines operator attributes and the other input tensors by calling Algorithm 2 (Lines 8 and 12). If there is a failure during constraint solving, the operator is abandoned. Otherwise, the algorithm creates a new vertex v and determines whether this new vertex should be added to the vertex set V (Lines 17~20). At the end of the algorithm, the complete graph is created (Line 22), consisting of all of the vertices and tensors kept during incremental generation.

5.2 Concolic Constraint Solving

GenCoG employs a concolic algorithm to solve type constraints specified in GenCoGL for each operator. The key idea of this algorithm is to combine *symbolic* constraint solving with *concrete* tensor types that are already known in the computation graph. In this way, the exploration of the call space of operators is decomposed to a series of sampling in a lower dimensional space, which makes the exploration more adequate. This section details the constraint solving algorithm, which is an iterative algorithm based on expression simplification and constrained random sampling.

Expression Simplification. The constraints in a specification written in GenCoGL may contain several structural (list-related) and domain-specific (attributes and tensor types) expression forms. These high-level forms cannot be processed by mainstream SMT

[2] We do not use TVM's code coverage (e.g., statement coverage or branch coverage) as a metric to direct graph generation, since graphs of high expressivity may not achieve high code coverage during testing.

solvers. We convert these expressions to numerical and logical expressions by specializing them with known values from the specification. Our implication strategies are summarized as follows:

- *Logical expression.* For `and` and `or`, we either eliminate the expression with short-circuit evaluation or reduce the number of clauses in the expression. `forall` is expanded to an `and` expression. For `cond`, the corresponding branch is chosen according to the predicate.
- *List*. We try to simplify the list expression to a list literal and select or transform the expression elements in it. Nondeterminism is involved in the simplification of `in` and `subset`, where one or more elements are arbitrarily selected from a list-represented set. A random number generator [28] is used for sampling elements.
- *Domain-specific reference.* The reference is replaced with the corresponding expression in the specification.

Solvable Variables and Constraints. GenCoG may perform several iterations such that a specification can be completely solved. The reason is that there are *definition dependencies* among variables in a specification, where a variable can only be defined after another variable is solved. As Figure 5(b) shows, the variables of elements in `newshape` are defined only after the length of `newshape` is determined. The extra constraint in the specification is not solvable during the first iteration. Therefore, GenCoG needs to identify solvable variables and constraints within one iteration. A solvable variable is a concrete variable that is only bounded by solvable constraints. A solvable constraint is a constraint that only contains solvable variables as well as numerical and logical operations. We adopt the union-find algorithm [15] to identify these variables and constraints.

Constrained Random Sampling. After the solvable variables and constraints are identified, GenCoG determines the values of the solvable variables with constrained random sampling. If a variable is only bounded by a constant range, GenCoG directly samples the specified range because the variable is orthogonal to the others. For the other variables and constraints, GenCoG resorts to an SMT solver (Z3 [5] in our implementation). We enable the SMT solver to produce randomized solutions by encoding the constraints in bit vector arithmetics and setting random phase selection. Based on the randomized results of constrained sampling, GenCoG is able to explore the call space of operators adequately and promote diversity at the operator level.

Algorithm. Algorithm 2 shows the detailed procedure of constraint solving. In the beginning, the algorithm specializes the specification by assigning the variables related to known tensor types (Line 1). Then it iteratively solves the constraints. One iteration contains two main stages. During the first stage, the algorithm simplifies the expressions in the specification repeatedly until a fixed point (Lines 3~15). Either `SimplifyAttr` or `SimplifyTypes` updates the simplified expression in-place, returning a Boolean indicating whether the expression is altered (Lines 5~7). The extra constraints are also simplified (Lines 8~14). During the second stage, the algorithm finds solvable variables and constraints (Line 16), and leverages constrained random sampling to solve these constraints (Line 17). The loop terminates if no variable is solved during one iteration. The algorithm reports a solving failure if any expressions are left unsolved (Line 18).

Table 1: List of the 62 operators supported by GenCoG.

Category	Operators
Element-wise	negative, abs, ceil, floor, round, trunc, exp, sin, cos, tan, sigmoid, tanh, relu, leaky_relu
Broadcasting	add, subtract, multiply, divide, maximum, minimum
Reduction	sum, mean, min, max
Tensor Transformation	expand_dims, squeeze, reshape, transpose, concatenate, split, strided_slice
Convolution & Pooling	conv1d, conv2d, conv3d, conv1d_transpose, conv2d_transpose, conv3d_transpose, max_pool1d, max_pool2d, max_pool3d, avg_pool1d, avg_pool2d, avg_pool3d, adaptive_max_pool1d, adaptive_max_pool2d, adaptive_max_pool3d, adaptive_avg_pool1d, adaptive_avg_pool2d, adaptive_avg_pool3d
Other Neural Network Operators	dense, bias_add, prelu, softmax, batch_flatten, pad, batch_norm, layer_norm, instance_norm, group_norm, upsampling, upsampling3d

6 EVALUATION

This section evaluates GenCoG. The evaluation is designed to answer the following research questions:

- **RQ1 (Validity)** How much of the computation graphs generated by GenCoG are valid?
- **RQ2 (Expressivity)** How expressive are the graphs generated by GenCoG?
- **RQ3 (Effort)** How much human effort is required to support graph generation of GenCoG?
- **RQ4 (Bug)** What bugs are detected by GenCoG in TVM?

6.1 Setup

Implementation. The core functionality of GenCoG is implemented in around 4.8k lines of Python code. As is shown in Table 1, we provide type specifications in GenCoGL for 62 commonly-used operators in TVM, which are chosen based on their popularity in mainstream DL models and pervasiveness in computational patterns. These specifications can easily be extended to support additional operators.

Baselines. Table 2 shows the methods for DL infrastructure testing which are compared with GenCoG in the evaluation:

- LEMON [35] is a mutation-based model generator for DL library testing. LEMON only generates Keras[3] models, and we convert these models to Relay IR with TVM's Keras frontend.
- Muffin [12] is a DL library testing technique via neural architecture fuzzing. Muffin proposes two graph structures: chain structure and cell-based structure, which are marked in our evaluation as Muffin-Chain and Muffin-Cell, respectively. Muffin also generates Keras models, and we apply the same conversion process as LEMON to Muffin's models.
- Luo et al. [22] propose a graph-based fuzz testing method for DL inference engines. They apply two random graph models to graph generation: Watts-Strogatz (WS) model [36] and

[3] https://keras.io/

Table 2: Summary of the techniques under comparison.

Technique	Graph Generation Method	Original Representation	Validity-Keeping Strategy	Expressivity-Improving Strategy
LEMON [35]	Mutation-based	Keras[1]	Allowing for shape-preserving operators only	MCMC-guided model mutation
Muffin-Chain/Cell [12]	Generation-based	Keras[1]	Using adapting operators	Using graph structure templates
Luo-WS/RN [22]	Generation-based	Relay[2]	Using adapting operators	Using random graph models & performing MCTS-based operator sampling
NNSmith [20]	Generation-based	PyTorch[1]	Symbolic solving of SMT constraints	Incremental graph generation
GenCoG	Generation-based	Relay	Concolic solving of GenCoGL specifications	Expressivity-directed graph generation

1. The Keras and PyTorch models need to be converted into the corresponding Relay IR for TVM testing.
2. Luo's technique is reimplemented since its source code is not publicly available.

Residual Network (RN) model, which are marked as Luo-WS and Luo-RN, respectively, in our evaluation.

- NNSmith [20] is a graph generation technique for deep learning compiler testing. It generates DL models in PyTorch [30] originally and converts the model to Relay.

For RQ1 and RQ2, we set the tensor rank range as $[1, 5]$ and the shape dimension range as $[1, 4]$ for quantitatively comparing the techniques. For RQ4, we do not impose such restrictions. For GenCoG, we set vertex limit $n_v = 32$ and rejection probability $p_{\text{rej}} = 0.9$.

Metrics. We evaluate GenCoG on the release versions of TVM, including v0.8 and v0.9.[4] We use the following metrics to evaluate GenCoG:

– *Pass rate of type checking.* We use *pass rate* of Relay's type checking to evaluate the validity of generated graphs. For all the compared methods, we generate 1k graphs and count the number of graphs that pass type checking of TVM v0.9. Since graphs generated by LEMON, Muffin, and NNSmith are not originally in Relay, we report pass rates of computation graphs in both their original representations and Relay.

– *Vertex and edge diversity.* We use the vertex and the edge diversities in §5.1 for expressivity measurement. [5] For each method, its generated graphs collectively contain at most 20k vertices.

– *Code length of constraint specifications.* As GenCoG requires developers to provide constraint specifications, it is necessary to evaluate how much human effort is involved in this work. We measure the code lengths of the specifications written in GenCoGL and compare them against the ones in NNSmith.

6.2 RQ1: Validity

Figure 7 shows the type checking pass rate of graphs generated by the methods. In their original representations, LEMON, Luo et al., NNSmith, and GenCoG all achieve 100% pass rate. Muffin, however, fails to guarantee the validity of the computation graphs even in its original Keras representation. For the two graph structures, 3.0% (Muffin-Chain) and 9.1% (Muffin-Cell) of the graphs are invalid. The rejections of the graphs by Keras are mainly caused by invalid tensor shapes (such as $(1, 4, 0)$) in the graphs generated by Muffin.

[4] v0.8 and v0.9 are the most recent TVM versions when we prepared this paper.
[5] To fairly compare the methods, we limit the generation to a set of 22 operators which are commonly supported by all methods.

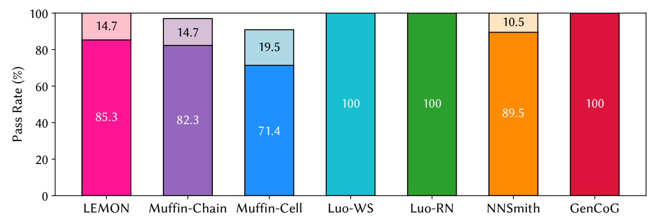


Figure 7: Type checking pass rate of 1k graphs generated by the graph generation methods. For LEMON, Muffin-Chain, Muffin-Cell, and NNSmith, the upper parts of the stacked bars represent the graphs accepted in their original representations but rejected by Relay after conversion.

The invalidity of the graphs is also raised by the conversion process. For LEMON, Muffin, and NNSmith which rely on TVM's frontend to convert the computation graphs, about 10%~20% graphs that are valid in the original representations become invalid in Relay. The reason is that TVM's frontend cannot perfectly align the semantics of operators in the original representations and the ones in Relay. Comparatively, GenCoG directly generates computation graphs in Relay and 100% of the graphs are accepted by Relay. GenCoG effectively guarantees the type-validity of generated graphs.

Answer to RQ1. GenCoG achieves 100% pass rate of type checking. GenCoG guarantees the validity of generated graphs.

6.3 RQ2: Expressivity

We measure the expressivity of generated computation graphs by their vertex/edge diversities. The diversities of the compared methods are listed in Table 3. LEMON have extremely low vertex and edge diversity. The two structures proposed by Muffin perform better, followed by NNSmith and two graph models from Luo et al., and GenCoG has the highest diversities.

We take a further investigation of the diversities achieved by these techniques. LEMON only handles sequential graph structure and shape-preserving operators, and thus it is not able to generate expressive computation graphs. Muffin and Luo et al. generate operator calls with hand-written sampling procedures, which cannot adequately explore the call space of some operators. Besides,

Table 3: Expressivity of the compared methods with a limit of 20k vertices.

Method	Vertex	Edge
LEMON	0.087	0.136
Muffin-Chain	0.252	0.446
Muffin-Cell	0.254	0.455
Luo-WS	0.325	0.901
Luo-RN	0.365	0.907
NNSmith	0.293	0.707
GenCoG	**0.603**	**0.963**
Outperformance	1.65~6.93×	1.06~7.08×

both methods generate graph structures and operator calls separately, and use auxiliary operators (e.g., `pad`, `slice`, and `reshape`) to align input tensor shapes for operators with multiple inputs. This strategy hinders the exploration of various wiring combinations between operators. NNSmith employs symbolic constraint solving, which only guarantees validity and does not improve expressivity at operator level. We observe that the operator attributes in graphs generated by NNSmith are usually boundary values. Though NNSmith does not use auxiliary operators when generating graphs in its original representation, it still does not achieve high edge diversity because some adapting operators are introduced when converting PyTorch models to Relay IR.

Comparatively, GenCoG achieves the highest diversities among the compared methods, outperforming these baselines by 1.65~6.93× in vertex diversity and 1.06~7.08× in edge diversity. The high vertex diversity of GenCoG comes from our expressivity-directed generation strategy and concolic constraint solving algorithm, which explore space of valid computation graphs more adequately, and provide a stronger guarantee on the expressivity of graphs. Note that the edge diversity of GenCoG is also the highest. Though GenCoG does not take edge diversity as guidance, it does not compromise edge diversity and still promotes variety of wiring combinations between operators.

Answer to RQ2. GenCoG is effective in generating expressive computation graphs. GenCoG outperforms existing graph generation techniques by 1.65~6.93× in vertex diversity and 1.06~7.08× in edge diversity.

6.4 RQ3: Effort

We compare GenCoG against NNSmith by the code length for specifying the type constraints of 10 representative operators.[6] The results are shown in Figure 8. It can be observed that GenCoG always requires fewer lines to write specifications than NNSmith. For operators with simpler constraints like `abs`, the code length of GenCoGL specifications is comparable with NNSmith. However, for operators with more complex constraints including `add`, `concatenate`, and `conv2d`, the degree to which the code length is reduced becomes more significant. In total, GenCoG reduces lines of code for specifying constraints by 44.1% compared with

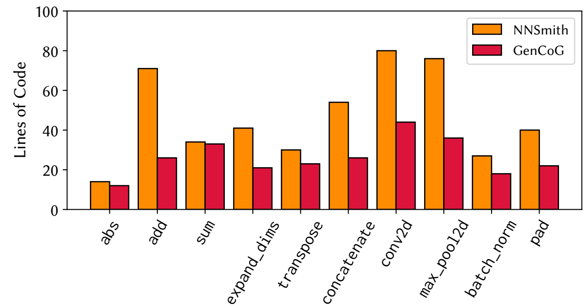


Figure 8: Lines of code for specifying type constraints of 10 representative operators in NNSmith and GenCoG.

Table 4: Statistics of the constraints of the 62 operators.

Kind	Mean	Min	Max
Numerical	6.14	3	15
Logical	0.37	0	3
Structural	3.60	2	8
Total	10.11	5	22

NNSmith. This means that GenCoGL is concise and well-organized, which effectively reduces human efforts for ensuring validity of computation graphs.

Let the broadcasting rule mentioned in Figure 3 be taken as an example. This constraint can be specified in GenCoGL with only five lines:

```
ForAll(Range(end=IN[0].rank.min(IN[1].rank)),
  lambda i: Or(
    IN[0].shape[m-i-1] == IN[1].shape[n-i-1],
    IN[0].shape[m-i-1] == 1, IN[1].shape[n-i-1] == 1
  ))
```

Comparatively, NNSmith needs 28 lines of code[7] for specifying the rule. NNSmith needs to manually handle low-level implementation details of the constraints, including aligning the input shapes, defining numerical constraints for each shape dimension, and simplifying the constraints algebraically. In contrast, GenCoG only requires a high-level definition and handles all the implementation details automatically in concolic constraint solving.

We conduct a simple study on the type constraints involved in the operator specifications. We count the constraints in the specifications for the 62 operators and list the statistics in Table 4. From the total number of constraints in each specification, we can see that the complexity of type constraints is greatly varied, ranging from 5 (for elementwise operators such as `relu`) to 22 (for `conv3d`). On average, one operator has more than 10 constraints. This indicates that GenCoGL is able to express complex type constraints of operators, and GenCoG can also correctly solve these constraints. From the detailed statistics on different kinds of constraints, we can observe that both numerical and structural constraints are very common, while logical constraints appear less frequently.

[6] All the specifications are formatted with PEP8 (https://peps.python.org/pep-0008/) with a line length limit of 99 characters.

[7] https://github.com/ise-uiuc/nnsmith/blob/v0.1.0/nnsmith/abstract/op.py#L187-L216

Table 5: Details of the bugs detected by GenCoG in TVM v0.8 and v0.9.

Index	Symptom	Occurring Component	Causal Component	Root Cause[1]	Status	Previously Unknown
			TVM v0.8			
1	Crash	Relay	Relay	IMA	Fixed	✓
2	Crash	Relay	Relay	IMA	Fixed	✓
3	Crash	TIR	LLVM Target	TDTP	Fixed	
4	Exception	Relay	Relay	TSP	Fixed	✓
5	Exception	Relay	Relay	OAP	Fixed	
6	Exception	Relay	Relay	TDTP	Fixed	
7	Exception	Relay	Relay	IMA	Confirmed	✓
8	Exception	TOPI	Relay	OAP	Fixed	✓
9	Exception	TOPI	Relay	IOL	Confirmed	✓
10	Exception	Runtime	TOPI	TSP	Fixed	✓
11	Inconsistency	Runtime	TOPI	IMA	Reported	✓
12	Inconsistency	Runtime	TOPI	IOL	Fixed	✓
13	Inconsistency	Runtime	TOPI	OAP	Fixed	
			TVM v0.9			
14	Inconsistency	Runtime	TOPI	TSP	Fixed	
15	Inconsistency	Runtime	TIR	IOL	Fixed	✓
16	Inconsistency	Runtime	Relay	IOL	Reported	✓

1. Abbreviations of root causes: **IMA** – Invalid memory access; **TSP** – Tensor shape problem; **TDTP** – Tensor data type problem; **OAP** – Operator attribute problem; **IOL** – Incorrect optimization logic.

Answer to RQ3. GenCoGL is concise and expressive in specifying type constraints of operators. GenCoG effectively reduces human efforts—compared with NNSmith, GenCoGL saves 44.1% lines of code in specifying type constraints on operators.

6.5 RQ4: Bug

We use a two-step process to detect the TVM bugs. *First*, we run all testing techniques for 24 hours, including generating and compiling graphs. Specifically, we compile each generated graph at five optimization levels and feed these five compiled graphs with the same tensor inputs. *Second*, we check crashes and inconsistencies among the outputs to detect bugs. We then manually identify the root causes of the bugs.

GenCoG detects 13 bugs in TVM v0.8 and 3 additional bugs in TVM 0.9. We check issues and pull requests in the GitHub repository of TVM and report previously unknown bugs. To date, 14 TVM bugs have been confirmed by developers and 12 have been fixed.

Among the 16 bugs detected by GenCoG, LEMON, Muffin, Luo et al., and NNSmith are able to detect 3, 7, 8, and 9 of them. There are 7 bugs that are only detectable by GenCoG. GenCoG detects these unique bugs by generating more expressive computation graphs, with increased vertex and edge diversities.

Table 5 shows the details of the 16 TVM bugs detected by GenCoG, including their symptoms, occurring components, causal components, root causes, statuses, and whether they are previously unknown. The occurring component is the TVM component whose API is called by user code when the symptom occurs, while the causal component is the one that has a defect causing the bug.

Root Cause and Case Study. We identify the root causes of all the bugs detected by GenCoG and present a case study of representative bugs for each category of the root cause.

Tensor shape problem (TSP). Tensor shape determines the structure of a tensor, and it is critical for lowering and optimization in TVM. As the computation of tensor shape is sometimes complex, developers are liable to make mistakes in this aspect. The bugs in this category mainly occur as unexpected exceptions in shape-related checking or computation in several compiler components.

Bug 4 in Table 5 is an exception bug in type checking of `conv3d` in TVM v0.8. When both `kernel_size` and `channels` (output channel number) are provided, weight shape is inferred from input data shape and attributes. However, the weight shape is wrong in some cases of group convolution. One case is shown as follows:

```
def @main(%x: Tensor[(1, 16, 32, 32, 32), float32], %w:
    Tensor[(4, 4, 1, 1, 1), float32]) {
  nn.conv3d(%x, %w, kernel_size=[1, 1, 1], channels=4,
    groups=4)
}
```

The correct shape of weight `%w` is annotated in the code, which is $(4,4,1,1,1)$. However, the actual weight shape inferred by Relay is $(16,0,1,1,1)$, which is obviously wrong. The root cause of this bug is that the type checking implementation falsely takes the operator call as a depthwise convolution and computes the wrong weight shape for this case. This issue is fixed by removing the special handling of depthwise convolution and replacing it with more general shape computation code.

Incorrect optimization logic (IOL). TVM performs a number of optimizations on both Relay and TIR. If the optimization logic is incorrect, the optimized program may be invalid and the other optimization passes or compiler components may raise exceptions. It is also possible that an optimization alters the semantics of IR, and causes inconsistencies before/after optimizations.

Bug 9 in Table 5 is an operator fusion bug in Relay. When compiling the computation graph in Figure 3, TOPI reports an error that it cannot schedule the tensor program whose output rank is 3. The root cause is described as follows: During operator fusion, all of the five vertices in the graph are fused to a single group. Since `dense` is the only computation-intensive operator in the group, TOPI leverages the predefined schedule of `dense` to handle the whole group. A rank mismatch occurs here because `dense` always outputs a tensor of rank 2 but the whole group outputs a tensor of rank 3. In fact, `dense` should not be fused with the `add` operator at the bottom of the graph, because the latter is a broadcasting operator that increases the rank. The operator fusion optimization does not properly handle broadcasting operators in this case.

Tensor data type problem (TDTP). Tensor data type is the primitive type for each element in a tensor. TVM supports several floating point and integer data types, and there is possibly conversion between data types during compilation. If the data types are not carefully handled, bugs caused by type mismatches can occur.

Operator attribute problem (OAP). An operator may contain several attributes that determine its behavior. Attributes are stored in each operator call in Relay IR, and they affect the low-level implementation of these operators in TOPI. Mishandling of operator attributes may lead to bugs occurring in either component.

Invalid memory access (IMA). An invalid memory access occurs when a program accesses a memory address that is not physically or logically valid. If the address is physically invalid (i.e., not allocated to the process), there is a segmentation fault and the program crashes. If the address is logically invalid (i.e., allocated to the process but out of the expected bound), the program produces incorrect results. For the element access methods of some data structures such as `std::vector::at` where the index is bound-checked, out-of-bound access leads to an exception.

Answer to RQ4. GenCoG detects 16 bugs in TVM v0.8 and v0.9, with 14 confirmed and 12 fixed. 7 bugs are only detectable by GenCoG, due to the increased expressivity of computation graphs generated by GenCoG.

7 RELATED WORK

DL Infrastructure Testing. There are a growing number of researches on improving the reliability of DL infrastructures, including libraries, frameworks, and compilers.

One line of work proposes graph-level testing techniques for DL frameworks. CRADLE [31] and Audee [13] use existing neural network models for testing. CRADLE [31] feeds existing models with real-world datasets to detect inconsistencies across different DL frameworks, while Audee [13] mutates inputs and parameters to reveal more inconsistencies. LEMON [35] applies search-based mutation on the shape-preserving operators in existing models for higher coverage. Muffin [12] and Luo et al. [22] generate computation graphs from scratch, with hand-coded sampling procedures for operators and predefined graph structures or models. Therefore, the computation graphs generated by those techniques lack expressivity at both graph and operator levels. In addition, they require enormous human efforts to ensure validity for each supported operator. On the contrary, GenCoG generates computation graphs to achieve high expressivity at both levels, with expressivity-directed graph generation and concolic constraint solving. The DSL-based approach of GenCoG also reduces human efforts in ensuring the validity of the graphs.

NNSmith [20] is a very recent graph generation technique which employs incremental graph generation and symbolic constraint solving. However, it requires large human efforts to write low-level SMT constraints for each operator; it also fails to guarantee the expressivity of computation graphs, as the results of constraint solving are usually boundary values. Compared with NNSmith, GenCoG makes three major improvements: (1) GenCoG takes a DSL-based approach to constraint specification, which significantly reduces human efforts for writing specifications of operators; (2) The graph generation process of GenCoG is guided by expressivity metrics, which provides stronger guarantees on the expressivity of the generated graphs; (3) GenCoG proposes a concolic constraint solving algorithm, which improves operator-level expressivity.

Researches also propose methods for testing DL frameworks at the API or operator level. Predoo [42] is only able to change the parameter values inside tensors. EAGLE [34] adopts differential testing by trying to construct an equivalent graph for each API under test. It focuses on inconsistency bugs and also has low test diversity. FreeFuzz [37] and DocTer [40] infer specifications by mining open-source code or documentation, which is likely to introduce false alarms. In addition, these methods are only able to generate test cases that only call one operator, while computation graphs call multiple. Comparatively, GenCoG relies on exact constraint specification by developers which ensures validity. It generates tests with multiple operator calls to reveal more defects.

Another work, Tzer [21], adopts joint IR-pass mutation to generate code and pass sequences in low-level TIR. Unfortunately, it cannot apply graph-level mutation for testing graph-level optimization in TVM. It is complementary to our GenCoG, targeting bugs in different TVM components.

Constrained Random Sampling. It is an important problem to generate a number of random solutions satisfying a set of Boolean constraints for software testing [10, 29] and hardware verification [3, 25, 26]. Researchers have proposed several techniques for solving this problem, utilizing universal hashing [2, 7, 23], Markov Chain Monte Carlo algorithms [17, 18, 38], and SMT-solvers [6, 8, 24]. They aim to improve the scalability or uniformity of sampling solutions for Boolean constraints, which is not the main goal of DL operator call generation. Instead, DL compiler testing needs to handle structural constraints on tensor shapes and certain attributes, and hierarchical constraints which involve definition dependencies. Therefore, the traditional constrained sampling approach is not directly applicable to TVM testing. Comparatively, GenCoG provides a DSL for specifying type constraints of operators and proposes an concolic algorithm to solve these constraints.

8 CONCLUSION

GenCoG is a DSL-based approach to computation graph generation for TVM testing. It specifies, using GenCoGL, type constraints of operators. It employs an expressivity-directed strategy and a concolic constraint solving approach in incrementally generating computation graphs. Our evaluation shows high validity and expressivity of GenCoG-generated computation graphs, as well as their effectiveness in bug detection.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback. We would also like to thank the TVM community for analyzing our reported issues. This research is supported by National Natural Science Foundation of China (Grant No. 62272296 and 62032004). The author from Nanjing University is supported in part by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001), the National Natural Science Foundation of China (No. 62232008 and 62172200), and the Fundamental Research Funds for the Central Universities (No.020214380101). The role of Jianjun Zhao is also supported by JSPS KAKENHI (Grant No. JP23H03372) and JST-Mirai Program (Grant No. JPMJMI20B8). This work was also supported by CCF-Huawei Innovative Research programme (TC20210701006/CCF2021-admin-270).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Supratik Chakraborty, Daniel J Fremont, Kuldeep S Meel, Sanjit A Seshia, and Moshe Y Vardi. 2015. On parallel scalable uniform SAT witness generation. In

International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 304–319.

[3] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2014. Balancing Scalability and Uniformity in SAT Witness Generator. In *Proceedings of the 51st Annual Design Automation Conference (DAC '14)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2593069.2593097

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[5] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science, Vol. 4963)*. Springer, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[6] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 549–559. https://doi.org/10.1145/3180155.3180248

[7] Stefano Ermon, Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2013. Embed and project: Discrete sampling with universal hashing. *Advances in Neural Information Processing Systems* 26 (2013).

[8] Stefano Ermon, Carla P Gomes, and Bart Selman. 2012. Uniform solution sampling using a constraint solver as an oracle. *arXiv preprint arXiv:1210.4861* (2012).

[9] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2020. Optimizing DNN Computation Graph using Graph Substitutions. *Proc. VLDB Endow.* 13, 11 (2020), 2734–2746. http://www.vldb.org/pvldb/vol13/p2734-fang.pdf

[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[11] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative Adversarial Networks. *Commun. ACM* 63, 11 (oct 2020), 139–144. https://doi.org/10.1145/3422622

[12] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, USA, 1418–1430. https://doi.org/10.1145/3510003.3510092

[13] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2021. Audee: Automated Testing for Deep Learning Frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. ACM, New York, NY, USA, 486–498. https://doi.org/10.1145/3324884.3416571

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 770–778. https://doi.org/10.1109/CVPR.2016.90

[15] J. E. Hopcroft and J. D. Ullman. 1973. Set Merging Algorithms. *SIAM J. Comput.* 2, 4 (1973), 294–303. https://doi.org/10.1137/0202024

[16] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, New York, NY, USA, 47–62. https://doi.org/10.1145/3341301.3359630

[17] Nathan Kitchen and Andreas Kuehlmann. 2007. Stimulus generation for constrained random simulation. In *2007 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 258–265.

[18] Nathan Boyd Kitchen. 2010. *Markov Chain Monte Carlo stimulus generation for constrained random simulation*. University of California, Berkeley.

[19] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[20] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS 2023)*. ACM, New York, NY, USA, 530–543. https://doi.org/10.1145/3575693.3575707

[21] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 73 (apr 2022), 26 pages. https://doi.org/10.1145/3527317

[22] Weisi Luo, Dong Chai, Xiaoyue Run, Jiang Wang, Chunrong Fang, and Zhenyu Chen. 2021. Graph-based Fuzz Testing for Deep Learning Inference Engines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 288–299. https://doi.org/10.1109/ICSE43902.2021.00037

[23] Kuldeep S Meel, Moshe Y Vardi, Supratik Chakraborty, Daniel J Fremont, Sanjit A Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. 2016. Constrained sampling and counting: Universal hashing meets SAT solving. In *Workshops at the thirtieth AAAI conference on artificial intelligence*. AAAI Press.

[24] Alexander Nadel. 2011. Generating diverse solutions in SAT. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 287–301.

[25] Reuven Naveh and Amit Metodi. 2013. Beyond Feasibility: CP Usage in Constrained-Random Functional Hardware Verification. In *Principles and Practice of Constraint Programming*. Springer, Berlin, Heidelberg, 823–831.

[26] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. 2006. Constraint-Based Random Stimuli Generation for Hardware Verification. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*. AAAI Press, 1720–1727. http://www.aaai.org/Library/AAAI/2006/aaai06-287.php

[27] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: Accelerating Deep Neural Networks Execution with Advanced Operator Fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. ACM, New York, NY, USA, 883–898. https://doi.org/10.1145/3453483.3454083

[28] Melissa E. O'Neill. 2014. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Technical Report HMC-CS-2014-0905. Harvey Mudd College, Claremont, CA.

[29] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 75–84. https://doi.org/10.1109/ICSE.2007.37

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[31] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[32] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. ACM, New York, NY, USA, 968–980. https://doi.org/10.1145/3468264.3468591

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[34] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: Creating Equivalent Graphs to Test Deep Learning Libraries. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, USA, 798–810. https://doi.org/10.1145/3510003.3510165

[35] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 788–799. https://doi.org/10.1145/3368089.3409761

[36] Duncan J. Watts and Steven H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393 (1998), 440–442.

[37] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, USA, 995–1007. https://doi.org/10.1145/3510003.3510041

[38] Wei Wei, Jordan Erenrich, and Bart Selman. 2004. Towards efficient sampling: Exploiting random walk strategies. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, Vol. 4. AAAI Press / The MIT Press, 670–676.

[39] Xiongfei Wu, Jinqiu Yang, Lei Ma, Yinxing Xue, and Jianjun Zhao. 2022. On the usage and development of deep learning compilers: an empirical study on TVM. *Empirical Software Engineering* 27, 7 (2022), 1–35.

[40] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael Godfrey. 2022. DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534220

[41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[42] Xufan Zhang, Ning Sun, Chunrong Fang, Jiawei Liu, Jia Liu, Dong Chai, Jiang Wang, and Zhenyu Chen. 2021. Predoo: Precision Testing of Deep Learning Operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. ACM, New York, NY, USA, 400–412. https://doi.org/10.1145/3460319.3464843

[43] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: Enabling a New Multi-Dimensional Optimization Space for Memory-Intensive ML Training and Inference on Modern SIMT Architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. ACM, New York, NY, USA, 359–373. https://doi.org/10.1145/3503222.3507723

Received 2023-02-16; accepted 2023-05-03