Taking Back Control in an Intermediate Representation for GPU Computing

VASILEIOS KLIMIS, Imperial College London, UK
JACK CLARK, Imperial College London, UK
ALAN BAKER, Google, Canada
DAVID NETO, Google, Canada
JOHN WICKERSON, Imperial College London, UK
ALASTAIR F. DONALDSON, Imperial College London, UK and Google, UK

We describe our experiences successfully applying lightweight formal methods to substantially improve and reformulate an important part of Standard Portable Intermediate Representation (SPIR-V), an industry-standard language for GPU computing. The formal model that we present has allowed us to (1) identify several ambiguities and needless complexities in the way that *structured control flow* was defined in the SPIR-V specification; (2) interact with the authors of the SPIR-V specification to rectify these problems; (3) validate the developer tools and conformance test suites that support the SPIR-V language by cross-checking them against our formal model, improving the tools, test suites, and our models in the process; and (4) develop a novel method for fuzzing SPIR-V compilers to detect miscompilation bugs that leverages our formal model. The latest release of the SPIR-V specification incorporates the revised set of control-flow definitions that have arisen from our work. Furthermore, our novel compiler-fuzzing technique has led to the discovery of twenty distinct, previously unknown bugs in SPIR-V compilers from Google, the Khronos Group, Intel, and Mozilla. Our work showcases the practical impact that formal modelling and analysis techniques can have on the design and implementation of industry-standard programming languages.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: shader/kernel language compilers, control flow, GPUs, SPIR-V, fuzz testing

ACM Reference Format:
Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. 2023. Taking Back Control in an Intermediate Representation for GPU Computing. *Proc. ACM Program. Lang.* 7, POPL, Article 60 (January 2023), 30 pages. https://doi.org/10.1145/3571253

1 INTRODUCTION

When it comes to control flow, two conventional options for the designer of an imperative language or intermediate representation (IR) are *unstructured control flow*, which consists of labelled instructions and goto statements, and *structured control flow*, consisting of block-structured constructs such as if-statements and while-loops. Unstructured control flow affords maximal freedom of expression, but imposing some structure can allow compilers more scope for optimisation.

Authors' addresses: Vasileios Klimis, Department of Electrical and Electronic Engineering, Imperial College London, UK, v.klimis@imperial.ac.uk; Jack Clark, Department of Computing, Imperial College London, UK, jack.clark1@imperial.ac.uk; Alan Baker, Google, Canada, alanbaker@google.com; David Neto, Google, Canada, dneto@google.com; John Wickerson, Department of Electrical and Electronic Engineering, Imperial College London, UK, j.wickerson@imperial.ac.uk; Alastair F. Donaldson, Department of Computing, Imperial College London, UK and Google, UK, alastair.donaldson@imperial.ac.uk.

2475-1421/2023/1-ART60
https://doi.org/10.1145/3571253

This tension is particularly interesting in the context of GPU programming. GPUs offer a *single program, multiple data* (SPMD) concurrency model that provides useful heterogeneity between threads: it allows different threads to access different data elements and to follow different control-flow paths through the same program code. In return for these benefits, GPU programmers, and tools that generate GPU code, must respect certain control flow-related properties. For instance, in current GPU programming models, barriers used to synchronise threads must be placed such that when any thread executes a barrier, all threads must execute the same barrier, otherwise behaviour is undefined (usually resulting in an execution hang).[1] Furthermore, many GPUs feature *lock-step* execution, where threads are subdivided into *subgroups* that share a program counter. Execution of conditional code is computationally expensive when not all threads in a subgroup follow the same path, and compilers for GPU languages can generate more efficient GPU machine code if they can transform the program to increase convergence, or exploit guarantees of convergence provided by the programmer.

In short, control flow in GPU programming languages is a complicated business. This has led to the development of IRs for GPU compilers with peculiar requirements on control flow. One of the most widely used of these IRs is SPIR-V [Khronos Group 2022a],[2] an open standard maintained by the Khronos Group. SPIR-V is the GPU programming language associated with the Vulkan 3D graphics API [Khronos Group 2022j], which is deployed to billions of Android devices worldwide and is also available as standard on Windows and Linux systems. It is also one of the kernel programming languages supported by the industry-standard OpenCL programming model [Khronos Group 2022e]. From a distance, SPIR-V resembles LLVM IR [LLVM Compiler Infrastructure 2022]: SPIR-V programs are written as a list of labelled basic blocks of instructions, with gotos to jump between them. But SPIR-V also has several distinctive features related to control flow. For example, (1) each diverging branch must be annotated with the label of a basic block at which control flow merges again, and (2) each loop must be annotated with the labels of two basic blocks, one heading a special "continue construct", executed at the end of each iteration, and another at which control flow merges after execution leaves the loop. Associated with these annotations are a set of rules governing their correct usage, and thus the allowed form of control flow structures (we discuss several of these rules later in the paper). Conformance to these rules can be mechanically checked by `spirv-val`, a validator for SPIR-V provided the Khronos Group [Khronos Group 2022i].

The SPIR-V specification refers to these annotations as *structured control flow*. The full SPIR-V specification also supports arbitrary unstructured control flow that does not require the use of annotations. However, Vulkan—the main context in which SPIR-V is used in practice—requires structured control flow annotations to be used throughout a SPIR-V program [Khronos Group 2022j, Appendix A]. In the rest of the article we focus on this subset of SPIR-V where all control flow is structured, and henceforth drop "structured" and simply talk about "control flow".

The aims of these annotations are threefold: (1) to afford SPIR-V most of the expressivity benefits associated with unstructured control flow, yet (2) to allow communication of high-level program structure when compiling from higher-level languages into SPIR-V, and (3) to equip downstream compilers that translate SPIR-V into GPU-specific machine code with information on where the control flow of threads diverges and merges, relieving them of significant control-flow analysis burden. Compilers that process SPIR-V are free to assume that input programs obey the rules that govern correct use of these annotations, and can thus leverage the annotations to generate architecture-specific control-flow instructions. For example, Intel Ice Lake GPUs feature a `brc`

[1] More precisely, this rule applies only if the threads are in the same *workgroup*.
[2] SPIR-V stands for Standard Portable Intermediate Representation; the V distinguishes it from an earlier representation, SPIR [Khronos Group 2014], and does not stand for anything.

(branch converging) instruction, with an operand that "should reference the instruction where all [threads] are expected to come together" [Intel 2020, p. 237], and a `break` instruction with an operand that "should be the offset to the end of the inner most conditional or loop block" [Intel 2020, p. 241]. Other GPU architectures differ in details, but architectural support for control flow is common.

Unfortunately, prior to our work, the semantics of these control flow annotations, as set out in the SPIR-V specification, suffered from problems of complexity, ambiguity, inconsistency with associated tooling, and in some cases failure to meet the intent of the language designers. Owing to these problems (of which we give examples in Section 3), developers have had a hard time working with the SPIR-V format. For example, Dzmitry Malyshau, while working as a developer at Mozilla, devoted a section of his "Horrors of SPIR-V" blog post to criticising the language's "head scratching" approach to control flow [Malyshau 2021]; Jason Ekstrand, developer at Collabora, who has contributed significantly to the Mesa 3D graphics stack and Intel's open-source driver, comments in an article about Mesa's intermediate representation that "the only thing that has been a challenge has been dealing with SPIR-V's less than obvious structure rules" [Ekstrand 2022]; and Sean Baxter, author of the Circle C++ compiler (which can target GPU architectures) expressed confusion about various control flow rules on a Khronos Group forum [Baxter 2020], and subsequently tweeted sarcastically: "Targeting SPIR-V is super easy and the structurization requirements totally won't make you want to throw yourself off a cliff" [Baxter 2021].[3]

Our work. Over several months, we studied the SPIR-V specification and its associated control-flow-related problems in detail. It became clear that the overall intent of the language designers was simple and reasonable—to provide a framework for allowing features of high-level language constructs to be encoded in an otherwise unstructured IR, and to enforce various restrictions on how these constructs should interact (e.g. prohibiting arbitrary branching between constructs). However, putting these ideas into practice via a set of definitions and rules had proved to be challenging due to the relative difficulty of working at the level of control-flow graphs (CFGs) compared with high-level program syntax, and the numerous possibilities for unanticipated and undesirable interactions between rules. With the aim of rectifying this situation, we set out to create a mechanised formal model of SPIR-V control flow capable of: (1) determining whether or not a given SPIR-V CFG is valid, and (2) generating examples of valid and invalid SPIR-V CFGs.

In this article, we report on our experience designing and implementing such a formal model using the Alloy modelling language and tool [Jackson 2019], iteratively refining the model in collaboration with SPIR-V language experts to fix problems of ambiguity, complexity, inconsistency, and mismatched intent. Cross-checking our model against other SPIR-V references along the way led to us finding and fixing problems in: (1) various SPIR-V conformance test suites, (2) `spirv-val`, a validation tool for SPIR-V programs [Khronos Group 2022i], and (3) (early versions of) our model itself, as depicted in Fig. 1. Once we had converged upon a mature model that agreed with the (improved) conformance tests and validator, and that we also felt was elegant, we proposed a revision of the SPIR-V specification in which we substantially rewrote the control flow definitions and rules to match our formal model. The Khronos Group reviewed and accepted our changes, publishing them in the recent SPIR-V 1.6 revision 2 specification [Khronos Group 2022a].

Our work contributes the following key ideas:

Key idea 1: Structural dominance. Fixing the fundamental problems associated with SPIR-V control flow required introducing a new concept into the language that we call *structural dominance*,

[3]In these quotations, "structure rules" and "structurization requirements" refer to rules governing correct usage of SPIR-V's control flow annotations, which we discuss in detail later in the article.

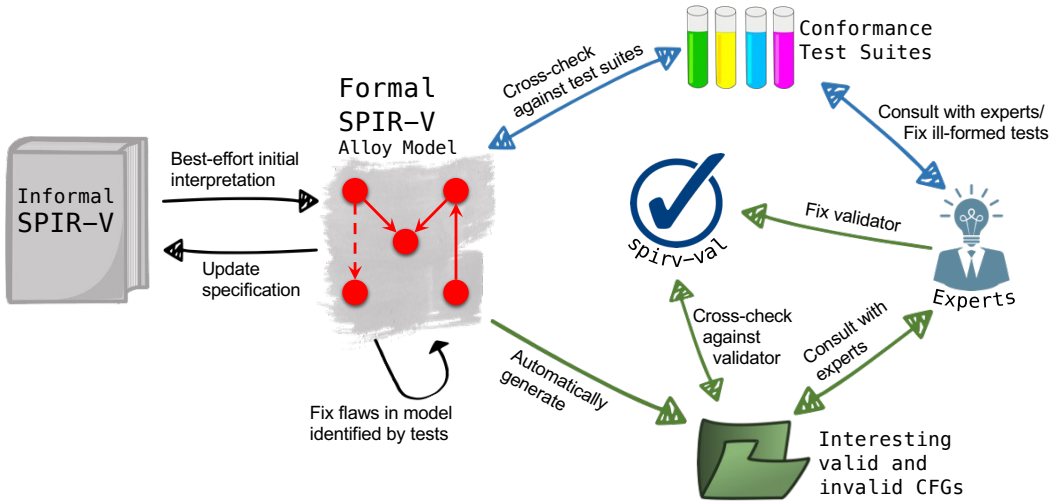


Fig. 1. The virtuous cycle of validating the SPIR-V formal model

described in detail in Section 4. Structural dominance is a very simple idea and seems pleasingly obvious with hindsight, but it was our formal modelling work that led us to the idea and aided us in working it out in full. The expert reader wishing to get a quick sense of what structural dominance is can look ahead to page 6. Prior to our work, SPIR-V control flow constructs were designed in terms of dominators and post dominators in CFGs, considering only 'branch' edges that correspond to the transfer of control between blocks (Fig. 2c). The key idea of structural dominance is to consider CFGs to be annotated with additional 'merge' and 'continue' edges (Fig. 2d) and then to include those edges when calculating dominance and post dominance relations. Armed with structural dominance, it is possible to re-formulate SPIR-V's control flow definitions and rules in a simpler, more elegant form that better achieves the intent of the language designers.

Key idea 2: Test generation via CFG fleshing. Because our formal model can be used to generate interesting SPIR-V CFGs, we have also leveraged it to obtain a novel method for compiler fuzzing: we take a generated SPIR-V CFG and *flesh* it into an executable program that is equipped, by construction, with a strong test oracle. We have applied this fuzzing technique to six industrial-strength SPIR-V compilers and translators, leading to the discovery and reporting of twenty distinct bugs, fourteen of which have already been fixed. The bugs include eight miscompilation bugs and twelve compiler or translator crashes.

Contributions. In summary, our main *conceptual* contributions are:

- the notion of *structural dominance*, the key concept that has allowed SPIR-V control-flow to be defined clearly and concisely for the first time and that is at the heart of the fundamental changes to SPIR-V that arose from our modelling efforts; and
- control-flow graph *fleshing*, a novel technique for compiler fuzzing that synthesises oracle-equipped test cases from CFG skeletons generated by our formal model;

while our main *impact arising* contributions are:

- a large case study showing the value of applying formal techniques from the programming languages domain to an industry-standard intermediate representation, including how we set

up a virtuous cycle of continuous improvement between our model, the SPIR-V specification, and validation tooling and conformance tests for the language;
- a substantial rewrite of the sections of the SPIR-V specification related to control flow, which has been incorporated into the latest public release of the specification;
- a reworking of the `spirv-val` validator (which previously had to attempt to validate programs in the context of an ambiguous language), bringing it in line with our reformulation of SPIR-V control flow, as well as corresponding changes to SPIR-V conformance test suites
- the use of control-flow graph fleshing to discover twenty distinct, previously unknown bugs in SPIR-V shader compilers from Intel, Google, the Khronos Group, and Mozilla.

We hope that the impact of this work will inspire other programming languages researchers to undertake analogous case studies in other application domains.

Outline. We provide necessary background on the SPIR-V intermediate representation and its control flow features (Section 2). Via a number of examples, we explain the problems with the way in which control flow in SPIR-V was defined in the SPIR-V specification prior to our work (Section 3). We then explain how we have reformulated the SPIR-V specification to fix these problems, via a new concept called *structural dominance* (Section 4). This concept arose thanks to our formal modelling effort; we describe the model and the process of applying it in practice (Section 5). We then show how we were able to further leverage our formal model by using it as the basis of a novel compiler fuzzing technique which has led to the discovery of numerous new bugs in tools that process SPIR-V (Section 6). After a discussion of related work (Section 7) we present our conclusions and ideas for future work (Section 8).

Artifact and source code. We have made available an artifact that provides the version of the formal model and tooling associated with this paper [Klimis et al. 2022a]. A repository containing the latest version of our model, the code for our tooling, and a set of issues giving insight into the iterative approach we followed in designing our model, is also available [Klimis et al. 2022b].

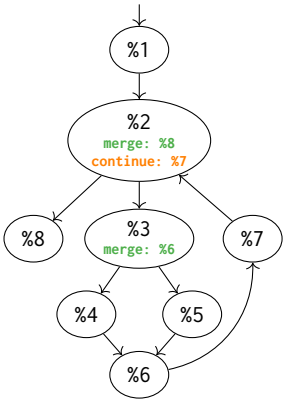
2 OVERVIEW OF SPIR-V AND ITS CONTROL FLOW

The SPIR-V language is an intermediate representation whose purpose, like LLVM IR, is to avoid the need for every GPU vendor to write their own compiler for each high-level language. This is a pressing problem due to the proliferation of GPU architectures from a diverse range of vendors and the high-level languages that aim to target them, such as the OpenGL Shading Language (GLSL) [Khronos Group 2019], Microsoft's High-Level Shading Language (HLSL) [Microsoft 2019], the OpenCL C kernel programming language [Khronos Group 2022e] and the WebGPU shading language [W3C 2022]. Instead, each vendor implements a SPIR-V compiler, and a small number of GPU-agnostic, industry-standard tools, such as `glslang` from the Khronos Group [Khronos Group 2022b], `clspv` from Google [Google 2022c], `tint` from Google [Google 2022f], and `naga` from Mozilla [Rust Graphics Mages 2022] are invoked to translate from high-level languages into SPIR-V. At present, SPIR-V is principally tied to the Vulkan programming model [Khronos Group 2022j], but can also be used with OpenCL [Khronos Group 2022e] and OpenGL [Segal and Akeley 2022]. Furthermore, several projects implement other graphics APIs, such as OpenGL, DirectX and WebGPU, on top of Vulkan [Google 2022b,d; Rebohle 2022].

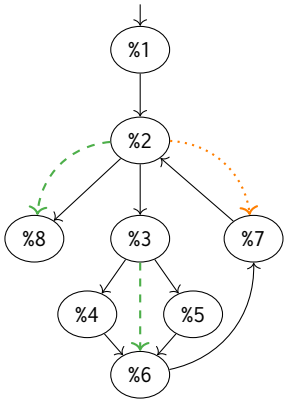
The SPIR-V specification is maintained and published by the Khronos Group, who also provide a number of tools for working with the format, including `spirv-opt`, a SPIR-V-to-SPIR-V optimiser that implements several target-agnostic compiler optimisations, and `spirv-val`, a static validator that aims to determine whether a given SPIR-V module obeys the rules of the language (including its control flow rules) [Khronos Group 2022i].

```
void main() {
  int x = 0, i = 0; // %1
  while(i < 100) { // %2
    if (i < 50) // %3
      x += 2; // %4
    else
      x += 4; // %5
    // %6
    i++; // %7
  } // %8
}
```

(a) A simple OpenGL shader



(c) Its CFG, with loop/selection headers annotated with merge/continue targets



(d) The same CFG with explicit edges for merge (-→) and continue (⋯→) targets

```
%44 = OpFunction %42 None %43
%1 = OpLabel
%48 = OpVariable %47 Function
%10 = OpVariable %47 Function
     OpStore %48 %9
     OpStore %10 %9
     OpBranch %2
%2 = OpLabel
%16 = OpLoad %46 %10
%19 = OpSLessThan %18 %16 %17
     OpLoopMerge %8 %7 None
     OpBranchConditional %19 %3 %8
%3 = OpLabel
%20 = OpLoad %46 %10
%22 = OpSLessThan %18 %20 %21
     OpSelectionMerge %6 None
     OpBranchConditional %22 %4 %5
%4 = OpLabel
%26 = OpLoad %46 %48
%27 = OpIAdd %46 %26 %25
     OpStore %48 %27
     OpBranch %6
%5 = OpLabel
%30 = OpLoad %46 %48
%31 = OpIAdd %46 %30 %29
     OpStore %48 %31
     OpBranch %6
%6 = OpLabel
     OpBranch %7
%7 = OpLabel
%32 = OpLoad %46 %10
%34 = OpIAdd %46 %32 %33
     OpStore %10 %34
     OpBranch %2
%8 = OpLabel
     OpReturn
     OpFunctionEnd
```

(b) A possible SPIR-V representation; instructions unrelated to control flow are greyed out

Fig. 2. Compiling a simple OpenGL shader into SPIR-V

2.1 SPIR-V by Example

We introduce SPIR-V by showing how it can represent the high-level program of Fig. 2a. This program is expressed in the OpenGL shading language (GLSL) [Khronos Group 2019], which we use for examples throughout the article as it can illustrate all the necessary control flow constructs that we discuss, whilst retaining the C-like syntax that will be familiar to most readers.

A SPIR-V program comprises:

- a sequence of administrative instructions related to issues such as the memory model the program uses, and whether it represents a graphics shader or a compute kernel;
- a sequence of instructions declaring types, constants, and global variables; and
- a series of function definitions, where each function is described via a list of parameters and a series of *basic blocks*.

The program of Fig. 2a features a loop with a conditional statement inside its body. Fig. 2b illustrates the SPIR-V code that might be emitted by a translator that compiles the program of Fig. 2a into SPIR-V. We are only concerned with SPIR-V's control flow constructs in this article,

and so restrict attention to the control flow between basic blocks of a single function. We thus omit from Fig. 2b all instructions outside the function definition and grey out instructions that are not directly related to control flow. We have annotated Fig. 2a to indicate in which basic block of Fig. 2b the SPIR-V code associated with each high-level language statement resides.

SPIR-V is a static single assignment (SSA) representation, so that each value in a SPIR-V program has a unique identifier expressed in the form `%v` where `v` is a positive integer. Each block starts with a *label* instruction of the form `%b = OpLabel`, where integer *b* gives the block a unique identifier. This is followed by zero or more instructions that perform memory accesses and computation. A block ends with a *terminator* instruction. The terminators relevant to this article are:

- `OpReturn`, causing control to return to the function's caller (optionally returning a value);
- `OpBranch %c`, causing control to transfer unconditionally to block *c*;
- `OpBranchConditional %v %c %d`, causing control to transfer conditionally to block *c* if the Boolean value associated with id *v* is *true*, and to *d* otherwise.

SPIR-V supports five different kinds of *structured control-flow constructs*: selection, loop, continue, switch and case constructs. For ease of presentation, we do not discuss switch and case constructs in this paper. They are handled in full by our formal model, which did lead to us finding and fixing several issues related to the way these constructs were defined, but the problems and associated fixes were not conceptually interesting, and the details of these constructs are rather involved.

As discussed in Section 1, when targeting SPIR-V from a high-level language, a code generator can embed information about source-level structures in the (otherwise unstructured) SPIR-V code. Intuitively, selection constructs can be used to model if-then-else constructs from high-level languages, as well as short-circuit evaluation of operators such as `&&`. Loop and continue constructs support translation of loops. As we discuss below, a continue construct captures that part of the loop that is executed at the end of each iteration.

Selection and loop constructs each start with a basic block, the *header block*, that includes a special *merge instruction*. There are two types of merge instructions, `OpSelectionMerge %m`, and `OpLoopMerge %m %c`, which signal the start of a *selection construct* or *loop construct* respectively.[4] A merge instruction allows control flow to diverge after the header block but declares the intent to reconverge at block `%m`, called the *merge block*.[5] For a loop header it also specifies that at the end of each loop iteration a single-entry-single-exit region of blocks—the loop's *continue construct*—will be executed, and block `%c`, the *continue target* of the loop, indicates the entry block of the continue construct. The SPIR-V specification requires that the entry block of a function is not a loop header.

In Fig. 2b, block `%3` is a *selection header* because it contains an `OpSelectionMerge` instruction; its merge block is block `%6`. Block `%2` is a *loop header* because it contains an `OpLoopMerge` instruction; the merge block and continue construct for this loop header are blocks `%8` and `%7`, respectively.

Fig. 2c shows the blocks of Fig. 2b arranged as a CFG, where, e.g., the `OpBranch %6` instruction at the end of block `%4` in Fig. 2b gives rise to the edge $4 \longrightarrow 6$ and the `OpBranchConditional ... %4 %5` at the end of block `%3` gives rise to the edges $3 \longrightarrow 4$ and $3 \longrightarrow 5$. In Fig. 2c we have annotated the loop and selection header blocks, `%2` and `%3`, with their associated merge blocks and (in the case of the loop header) continue target.

The CFG of Fig. 2d shows the same information in a different way: instead of annotating header blocks with merge/continue information, we use special merge and continue edges. A merge edge $h \dashrightarrow m$ indicates that m is the merge block for header block h. A continue edge $h \cdots\!\!\rightarrow c$ indicates

[4] Merge instructions also feature an additional "selection control" argument that we ignore here since it is unrelated to control flow semantics.

[5] As we shall see, control is allowed to bypass the merge block in a limited set of scenarios: by returning from the function, breaking from an enclosing loop, or continuing to the next iteration of an enclosing loop.

that c is the continue target for loop header block h. Henceforth we shall use CFGs augmented with merge and continue edges when discussing SPIR-V examples. We emphasise that these edges do *not* indicate how flow of control can transfer between blocks—e.g. the merge edge 3 -⇢ 6 in Fig. 2d does not mean that control can flow directly from block 3 to block 6.

2.2 Definitions of SPIR-V Control Flow Constructs

Having described the conceptual notions of SPIR-V control flow constructs by example, we now turn to the definitions of these constructs that are provided in the SPIR-V specification.

We present definitions that have remained essentially unchanged since the original SPIR-V 1.0 specification [Khronos Group 2017]. These definitions are simple and intuitive, and work well for straightforward examples such as the illustrative example of Section 2.1. However, as we show in Section 3, the simple intuitive definitions are problematic for more complex examples. In more recent versions of the SPIR-V specification they have been extended with additional conditions and caveats in an effort to remedy this. In what follows, we quote definitions from SPIR-V 1.6 revision 1 (1.6r1), the specification version prior to our changes, but omit the caveats and side conditions alluded to above, which we discuss in Section 3. This allows us to present the original spirit of SPIR-V control flow, but using the style and terminology of the more recent specification version makes it easier to show how we have reformulated the specification when we discuss our changes in Section 4.

Control flow constructs in SPIR-V are defined in terms of *dominance* and *post dominance* relations, which are standard in the compilers literature (see e.g. [Aho et al. 2007, p. 656 and p. 728]):

> *Dominate*: A block A dominates a block B, where A and B are in the same function, if every path from the function's entry point to block B includes block A. *A strictly dominates B* only if *A dominates B* and A and B are different blocks.
>
> *Post Dominate*: A block B post dominates a block A, where A and B are in the same function, if every path from A to a function-return instruction goes through block B.
>
> [SPIR-V 1.6r1, §2.2.5]

A *selection construct* is then defined in terms of the dominance relation:

> a *selection construct*: includes the blocks dominated by a selection header, while excluding blocks dominated by the selection construct's merge block [SPIR-V 1.6r1, §2.11]

Applying this definition to Fig. 2d, we see that the blocks $\{3, 4, 5\}$ form a selection construct headed by block 3: block 3 is a *selection header* with block 6 as its merge block (due to the `OpSelectionMerge` instruction in the block; see Fig. 2b); block 3 dominates blocks $\{3, 4, 5, 6, 7\}$; block 6 dominates blocks $\{6, 7\}$; and the difference between these sets of dominators is $\{3, 4, 5\}$. With respect to Fig. 2a, this is intuitive: the selection construct corresponds to the **`if`** statement.

Before defining loops, SPIR-V 1.6r1 defines the notions of *back edge* and *back-edge block*:

> *Back Edge*: A branch is a *back edge* if there is a depth-first search starting at the entry block of the CFG where the branch branches to one of its ancestors.[6] A *back-edge block* is a block containing such a branch instruction. [SPIR-V 1.6r1, §2.2.5]

For example, edge 7 → 2 in Fig. 2d is a back edge, and thus block 7 is a back-edge block.

SPIR-V partitions the blocks that make up a loop into a *continue construct*—a single-entry single-exit region that must be executed each time control returns to the head of the loop at the end of an

[6] This is what is usually called a *retreating edge* in the compilers literature [Aho et al. 2007, p. 661], with *back edge* usually involving the extra condition that the head of the edge dominates its tail [Aho et al. 2007, p. 662]. However, SPIR-V CFGs are reducible, meaning that these definitions coincide [Aho et al. 2007, pp. 662–664].

iteration, and a *loop construct*, which comprises the main body of the loop. One motivation for this partitioning is to ease syntax-directed translation of the "increment" expression in a C-style **`for`** loop (which feature in most high-level graphics shading languages), which must be executed each time the loop iterates, whether due to the end of the loop body being reached or an early **`continue`** statement being executed. This expression is often simple, e.g. `i++` in a typical counting loop, but in general it may invoke functions that perform non-trivial computation which the compiler may choose to inline. To cater for this, a SPIR-V continue construct is thus a *region* of blocks.

> a *continue construct*: includes the blocks dominated by an `OpLoopMerge` Continue Target and post dominated by the corresponding loop's back-edge block
>
> a *loop construct*: includes the blocks dominated by a loop header, while excluding both that header's *continue construct* and the blocks dominated by the loop's merge block
>
> [SPIR-V 1.6r1, §2.11][7]

In our running example, block 7 of Fig. 2d forms a single-block continue construct. This is because: 7 is both the *continue target* operand of the `OpLoopMerge` instruction in Fig. 2b, and the back-edge block for the loop header associated with this `OpLoopMerge` instruction; 7 is the only block dominated by 7; and 7 is the only block post dominated by 7.

The loop construct headed by block 2 is then defined as the set of blocks $\{2, 3, 4, 5, 6\}$, because: the header block 2 dominates the set $\{2, 3, 4, 5, 6, 7, 8\}$; block 7 is excluded from this set as it forms the continue construct associated with the loop; and block 8 is excluded, being the loop header's merge block (and in fact the only block dominated by the loop header's merge block).

3 PROBLEMS WITH SPIR-V CONTROL FLOW

We now illustrate a number of problems related to control flow definitions and rules that we identified when studying the SPIR-V specification and comparing its requirements with the rules checked by `spirv-val`. Most of the examples are presented via a GLSL source program and a corresponding SPIR-V CFG, with statements in the GLSL source annotated to indicate the associated SPIR-V basic block. The `spirv-val` results reported in this section are with respect to version v2020.1 of the tool—a stable release that precedes the improvements that we made during this work.

The problems discussed in Section 3.1 and Section 3.2 were known to members of the community for some time, and we discuss language that was added to the SPIR-V specification in an attempt to remedy them. We discovered the remaining problems via our formal modelling work, and do not know to what extent practitioners interested in SPIR-V were aware of these particular problematic examples.

Several of the problems that we discuss relate to definitions in the SPIR-V specification being *unintuitive* due to mismatches between (a) the basic blocks that a compiler from a high level langauge to SPIR-V would emit when performing syntax-directed translation of a high level language construct, and (b) the set of blocks that comprise a SPIR-V structured control flow construct according to the definitions in the specification. This matters in practice for two main reasons. First, it has led to cases where the SPIR-V validator deviates from the specification in the direction of what would be achieved by more intuitive definitions (we give an example in Section 3.3). Second, it has the potential to prevent compiler-writers taking advantage of structured control flow information to implement optimisations, since the non-intuitive sets of blocks that comprise structured control flow constructs do not correspond to coherent CFG subgraphs that can be meaningfully optimised.

[7] The full SPIR-V 1.6r1 definition adds a non-intuitive extra constraint compared with the original SPIR-V 1.0 definition, which we discuss in Section 3.

```
void main() {
  int i = 0; // %1
  do { // %2
    if (i > 50) { // %3
      break;
    } else {
      // no-op // %4
    }
    i++; // %5
  }
  while(i < 100); // %6
  // %7
}
```

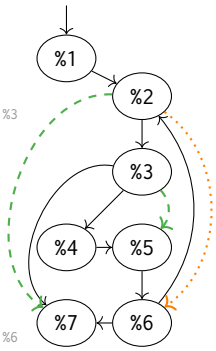


Fig. 3. A loop with an early break

```
void main() {
  for (int i = 0; // %1
       i < 100; // %2
       i++) { // %5
    if (i > 50) // %3
      continue;
    i++; // %4
  } // %6
}
```

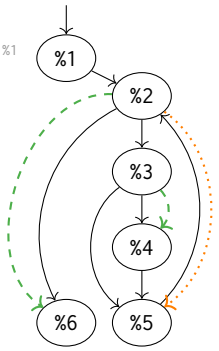


Fig. 4. A loop with an early continue

3.1 First Problem: 'Selection Construct' Ill-Defined in Presence of Early Breaks

The GLSL program of Fig. 3 features a loop with an early break statement; the CFG associated with a reasonable syntax-directed translation to SPIR-V is also shown. The CFG features a loop construct headed by block 2 and a selection construct headed by block 3, corresponding to the **do**...**while** loop and **if** statement of Fig. 3, respectively. Recall that the non-solid edges indicate the merge blocks and (in the case of loops) continue targets associated with header blocks and do *not* represent control flow paths, e.g. control cannot transfer directly from block 2 to block 7.

Intuitively, the selection construct headed by block 3 corresponds to the **if** statement of Fig. 3, and thus we would expect this selection construct to comprise the set of blocks $\{3, 4\}$. However, if we apply the definition of 'selection construct' from Section 2.2 we find that the blocks dominated by the header 3 is the set $\{3, 4, 5, 6, 7\}$, and the blocks dominated by merge block 5 is the set $\{5, 6\}$. Thus according to the definition, the selection construct headed at 3 comprises blocks $\{3, 4, 7\}$. This is intuitively wrong: block 7, which is associated with the end of the program, should *not* be considered part of the set of blocks that model the **if** statement of Fig. 3. The problem is that the edge $3 \longrightarrow 7$, associated with the early break from the loop, leads to the selection header 3 dominating the loop merge block 7.

In an attempt to fix the definition to cater for examples such as this, the following caveat was added to the specification, which applies to all control flow constructs including selection constructs:

> Furthermore, these structured control-flow constructs are additionally defined to exclude all outer constructs' continue constructs and exclude all blocks dominated by all outer constructs' merge blocks. [SPIR-V 1.6r1, §2.11]

With respect to the CFG of Fig. 3, this could be interpreted as eliminating block 7 from the selection construct, because 7 is the merge block of a loop that contains the selection construct.

However, this caveat is far from satisfactory. First, it makes the definition of a construct rather complex and unwieldy. Secondly, in its current form, the caveat makes the definition of a control flow construct circular: the caveat forms part of the definition of the blocks that comprise a construct, and yet is defined in terms of the blocks that comprise constructs. Thirdly, the specification does not define what is meant by 'outer construct'. Finally, while it would likely be possible to precisely define 'outer construct' and avoid circularity via an inductive definition, this would make the definitions of control flow constructs even more complex.

```
void main() {
  // %1
  do {
    ... // %2
  } while(false); // %3
  // %4
}
```

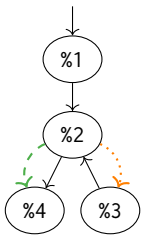


Fig. 5. The "do while false" idiom

```
void main() {
  if (...) // %1
    return; // %2
  else
    return; // %3
  // unreachable
  if (...) // %4
    ... // %5
  else
    ... // %6
  // %7
}
```

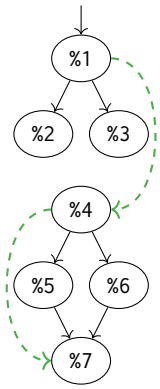


Fig. 6. A program that naturally compiles to a SPIR-V program with an unreachable control-flow subgraph

3.2 Second Problem: 'Selection Construct' Ill-Defined in Presence of Early Continues

The GLSL program of Fig. 4 features a loop with an early **`continue`** statement. The figure includes a potential associated CFG. Intuitively, we would like the selection construct headed by block 3 to comprise just block 3. However, according to the initial definition of a selection construct (see the first quotation in Section 3.1), the selection construct comprises the set of blocks $\{3, 4, 5\}$ dominated by the selection header 3, minus the singleton set $\{4\}$ of blocks dominated by the associated merge block. The result is the set $\{3, 5\}$. In particular, the loop's continue target, block 5, is considered part of the selection construct according to the initial definition, which is undesirable.

As in Section 3.1, the caveat that was subsequently added to SPIR-V, about constructs being defined to exclude certain blocks associated with outer constructs, could be interpreted as rectifying this problem, but again the caveat is complex and not clearly defined.

3.3 Third Problem: Back-Edge Rules Fail for Loops with Unreachable Continue Targets

The program of Fig. 5 illustrates the "do while false" idiom, which is commonly used in C-like languages to allow preprocessor macros that span multiple lines to be invoked as seemingly stand-alone statements [Gennaro 2015, p. 45]. The CFG of Fig. 5 is associated with a reasonable syntax-directed translation of this idiom. The body of the loop is represented by block 2, which unconditionally branches to block 4, the loop's merge block. The loop's continue target, block 3, is unreachable (due to the **`false`** condition), so the edge $3 \longrightarrow 2$, representing continuing to the next loop iteration, can never be traversed.

The SPIR-V specification states the following rule about loop headers and back edges:

> all CFG back edges must branch to a loop header, with each loop header having exactly one back edge branching to it [SPIR-V 1.6r1, §2.11]

The CFG of Fig. 5 is invalid according to this rule: edge $3 \longrightarrow 2$ is *conceptually* the back edge for the loop, but is *not* a back edge according to the definition in the specification (see Section 2.2) since the edge is not reachable from the entry block of the CFG. Hence the loop header block 2 has no back edges branching to it. Nevertheless, the SPIR-V code for this example *is* deemed valid by `spirv-val`. We discussed this example with the SPIR-V language designers and they advised that

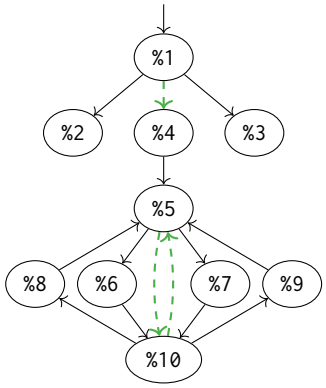


Fig. 7. Unusual CFG featuring cyclic merge relationships

```
void main() {
  // %1
  while(true) { // %2
    do { // %3
      ...
    } while(...); // %4
    // %5
    ... // %6
  } // %7
}
```

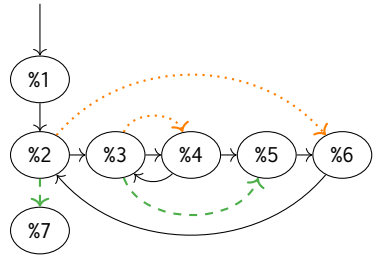


Fig. 8. A program that leads to a CFG with no paths to exit blocks

it is indeed desirable for examples such as this to be considered valid, to allow for straightforward syntax-directed translations from high-level languages to SPIR-V.

3.4 Fourth Problem: Unreachable Constructs

Before our changes, the SPIR-V specification was somewhat complicated by caveats related to unreachable blocks, e.g.:

> each header block must strictly dominate its merge block, unless the merge block is unreachable in the CFG [SPIR-V 1.6r1, §2.11]

This caveat is surprising since every block in a CFG *vacuously* dominates every unreachable block: a given block appears on all paths from the CFG's entry block to an unreachable block because the set of such paths is empty. It is not clear what the 'unless the merge block is unreachable' clause is meant to achieve.

Vacuous dominance relationships also highlight a number of problems with definitions of constructs in SPIR-V 1.6r1.

In the program of Fig. 6, both branches of the first `if` statement end with `return` so that the following `if` statement is unreachable. This is made clear in the corresponding CFG. The SPIR-V specification allows this example, `spirv-val` agrees that it is valid, and we confirmed with SPIR-V experts that it is desirable to allow such examples (to support simple syntax-directed translators). Intuitively, we would like the selection construct headed at 4 to comprise the set of blocks $\{4, 5, 6\}$ corresponding to the second `if` statement in Fig. 6. But according to the definition of selection construct given in Section 2.2 this selection construct contains *no* blocks. Every unreachable block vacuously dominates every unreachable block, so the selection construct headed at 4 comprises all blocks dominated by 4—all of the unreachable blocks—minus all blocks dominated by 7—also all of the unreachable blocks, yielding an empty set.

Things get murkier still when we consider less intuitive CFGs featuring unreachable blocks. Fig. 7 features a strange subgraph, unreachable via control flow edges, with two selection header blocks, 5 and 10, each of which has the other as its merge block. Intuitively such an example should be invalid, because SPIR-V structured control flow constructs are supposed to model control flow constructs in high-level languages that may be nested or sequenced but that cannot have a cyclic relationship. However, SPIR-V 1.6r1 appears to allow this CFG. The requirement "each header block must strictly dominate its merge block, unless the merge block is unreachable in the CFG"

is doubly-satisfied for blocks 5 and 10 because (a) the blocks strictly dominate one another in a vacuous manner, and (b) the blocks are both unreachable in the CFG.

Despite the fact that the specification appears to allow it, the example is *rejected* by `spirv-val`, but with the following rather misleading error (misleading because $8 \longrightarrow 5$ is not a back edge):

```
error: line 22: Back-edges (8 -> 5) can only be formed between a block and a loop header.
```

We discussed this example with SPIR-V experts, who agreed that if the specification is supposed to be unambiguous about the validity of CFGs such as that of Fig. 6, it needs to be unambiguous about the validity rules for *arbitrary* CFGs featuring unreachable blocks.

3.5 Fifth Problem: Syntactically Infinite Loops

The GLSL program and associated CFG of Fig. 8 features an infinite outer loop that contains another loop. Due to the infinite outer loop, all paths starting at the entry block of the CFG are infinite; the outer loop's merge block, block 7, is not reachable. In this sense the loop is *syntactically* infinite.

According to the definition of post dominance (see Section 2.2), if there are *no* paths from a block B to a block that exits the CFG then *every* block in the CFG post dominates B.

Looking at the loop headed at block 3 in Fig. 8, intuitively we would like the continue construct for this loop to comprise the block 4, as this is both the continue target and back-edge block for the loop header. However, according to the definition of 'continue construct' in the original SPIR-V specification (see Section 2.2), the construct comprises those blocks that are both dominated by the continue target and post dominated by the back-edge block—i.e., both dominated and post dominated by block 4. Now, block 4 dominates the set of blocks $\{4, 5, 6\}$, and 4 post dominates each of these blocks because none of the blocks lie on a path that reaches a CFG exit block. This means the continue construct is defined to be $\{4, 5, 6\}$, which is not intuitive.

The caveat that was added to SPIR-V mandating that a construct should exclude blocks dominated by the merge blocks of outer loops, as well as the continue constructs of outer loops, might be interpreted to eliminate block 6 from this set, as it is the continue construct of the outer loop headed at block 2. However, block 5 is the merge block of the loop construct headed at block 3, and because loop constructs and continue constructs are defined to be disjoint, this loop construct is not defined to enclose the continue construct headed at 4, so the caveat does not get rid of block 5.

Perhaps in response to examples such as this, the definition of a *continue* construct was extended with an extra condition, which we highlight in bold:

> a *continue construct*: includes the blocks dominated by an `OpLoopMerge` Continue Target and post dominated by the corresponding loop's back-edge block, **while excluding blocks dominated by that loop's merge block** [SPIR-V 1.6r1, §2.11]

For the CFG of Fig. 8 this extra condition does get rid of block 5, yielding $\{4\}$ as the continue construct, as desired. However, while not too onerous, we argue that this additional condition is unintuitive and inelegant.

3.6 Sixth Problem: Ill-Formed Exit Rules

The SPIR-V language designers wished to limit the scenarios under which a control flow edge could leave a structured control flow construct, for example limiting "loop break" edges so that they can only break from the innermost loop construct.

The specification had attempted to encode these rules by defining concepts such as "break block" and "continue block" to refer to blocks that branch to merge blocks and continue targets, respectively, and then mandating rules such as:

> - a break block is valid only for the innermost loop it is nested inside of
> - a continue block is valid only for the innermost loop it is nested inside of
>
> [SPIR-V 1.6r1, §2.11]

However, the formulation of such exit rules was fundamentally difficult because of the ambiguities surrounding the definitions of structured control flow constructs that we have showcased in this section. We also identified a number of technical problems with the definitions and rules. For example, it is permissible for a loop header to serve as its own continue target, but this means that a predecessor of the loop header from which the loop is *entered* is regarded as a continue block.

4 FIXING THESE PROBLEMS WITH STRUCTURAL DOMINANCE

We now describe *structural dominance*, our proposal for modifying the control flow definitions and rules of the SPIR-V language, which has been adopted in the latest revision of the specification. We motivate the idea and provide an intuitive overview (Section 4.1) before describing our changes to the specification in detail (Section 4.2). We then explain how these changes rectify each of the problems discussed in Section 3 (Section 4.3), at the expense of a small amount of backwards-incompatibility that was deemed acceptable by the SPIR-V language designers (Section 4.4). We also briefly discuss design choices related to how the specification should handle "wholly unreachable" blocks (Section 4.5).

4.1 Structural Dominance: Motivation and Intuition

A common theme associated with the problematic examples studied in Section 3 is *unwanted dominance relationships* between blocks. In the CFG of Fig. 3 (Section 3.1) it is the fact that block 3 dominates block 7 that leads to 7 being erroneously incorporated into the selection construct headed by 3 unless additional conditions are imposed—this dominance relationship arises due to the early break edge $3 \longrightarrow 7$. Likewise, in Fig. 4 (Section 3.2), the continue target 5 becomes an unwanted element of the selection construct headed at 3 (without additional conditions) due to the early continue edge $3 \longrightarrow 5$.

Another root cause of the problems discussed in Section 3 is *lack of reachability*. The loop construct headed at block 2 in Fig. 5 (Section 3.3) has no back edge due to block 3—the loop's continue target—being unreachable. In Fig. 7 (Section 3.4), the cyclic merge relationship between blocks 5 and 10 is allowed to hold due to these blocks being unreachable.

Furthermore, there is *interplay* between dominance-related issues and reachability-related issues. In Fig. 6 (Section 3.4), the selection construct headed by block 4 is empty due to mutual domination between unreachable blocks; and in Fig. 8 (Section 3.5), the mutual post dominance relationship between blocks that complicates the definition of 'continue construct' is due to the CFG's exit block being unreachable.

Our key observation is that in all these examples, the unwanted dominance and/or lack of reachability would go away if 'merge' and 'continue' relationships between blocks were incorporated into the definitions of reachability, dominance and post dominance.

As a concrete illustration, consider the "early break" example of Fig. 3 (Section 3.1), which features a selection construct inside a loop. Using the regular notion of dominance, the header of the selection construct (block 3) dominates the loop's merge block (block 7), however, the selection construct's merge block (block 5) does not. Defining the blocks of a selection construct as the blocks dominated by the header block minus the blocks dominated by the merge block thus leads to 7 being erroneously included. However, if we included the merge edges $2 \dashrightarrow 7$ and $3 \dashrightarrow 5$ and the continue edge $2 \cdots\!\!> 6$ in the dominance calculation then 3 would no longer dominate 7, due to the path $1 \longrightarrow 2 \dashrightarrow 7$ that reaches 7 from the entry block of the CFG without going through 3.

4.2 Structural Dominance: Definitions and Rules

We now describe structural dominance in detail by presenting the engineer-focused prose definitions and rules that have been accepted into the SPIR-V 1.6 specification as a result of our work [Khronos Group 2022g]. These definitions and rules arose as a result of formally modelling rules from earlier SPIR-V specification versions, discussing them with experts and validating them against conformance test suites and external tools, and then translating them back into prose matching the general style of the SPIR-V specification. We discuss our formal modelling efforts in Section 5.

Before our changes, SPIR-V did not explicitly define the notion of a branch edge between basic blocks—being the only kind of edge of interest it was obvious from context what "branch" meant. As our changes introduce merge and continue edges, we define all three:

> *Branch Edge*: There is a *branch edge* from block A to block B if the terminator of A is a branch instruction and B is one of the target blocks for the branch instruction.
>
> *Merge Edge*: There is a *merge edge* from block A to block B if A contains a merge instruction and B is the merge block of this merge instruction.
>
> *Continue Edge*: There is a *continue edge* from block A to block B if A is a loop header and B is the Continue Target of the loop header's OpLoopMerge instruction. [SPIR-V 1.6r2, §2.2.5]

We then introduce the term "structured control-flow edge" as a catch-all to refer to any of the above kinds of edge, and the notion of a sequence of contiguous structured control-flow edges as a "structured control-flow path":

> *Structured Control-Flow Edge*: There is a structured control-flow edge from block A to block B if there is a branch edge, merge edge, or continue edge from A to B.
>
> *Structured Control-Flow Path*: A sequence of blocks $B_0, B_1, \ldots, B_n$ where for each $0 \leq i < n$ there is a structured control-flow edge from B_i to B_{i+1}. This forms a *structured control-flow path* from B_0 to B_n.
>
> *Structurally Reachable*: A block B is *structurally reachable* if there exists a structured control-flow path from the entry block of the function containing B to B. [SPIR-V 1.6r2, §2.2.5]

For example, $1 \longrightarrow 2 \longrightarrow 6 \longrightarrow 7$ is not a regular control flow path in the CFG of Fig. 3, but $1 \longrightarrow 2 \cdots\!\!\rightarrow 6 \longrightarrow 7$ is a structured control-flow path, involving a combination of branch edges ($\longrightarrow$) and continue edges ($\cdots\!\!\rightarrow$).

We can then define notions of *structural* dominance and *structural* post dominance. These exactly mirror the definitions of regular dominance and post dominance (see Section 2.2), with "path" (which in the old definitions referred to a regular control flow path) replaced with "structured control-flow path":

> *Structurally Dominate*: A block A structurally dominates a block B, where A and B are in the same function, if every structured control-flow path from the function's entry block to block B includes block A. A strictly structurally dominates B if A structurally dominates B and A and B are different blocks.
>
> *Structurally Post Dominate*: A block B structurally post dominates a block A, where A and B are in the same function, if every structured control-flow path from A to a function termination instruction includes block B. [SPIR-V 1.6r2, §2.2.5]

To illustrate these concepts, consider the CFG of Fig. 3 again. Block 3 dominates block 7 because every regular control-flow path from the CFG's entry block to 7 starts with the prefix $1 \longrightarrow 2 \longrightarrow 3$ of branch edges. But 3 does not *structurally* dominate 7 due to the structured control flow path $1 \longrightarrow 2 \dashrightarrow 7$ comprised of branch and merge edges.

Relationship between regular dominance and structural dominance. For blocks A and B in a SPIR-V CFG, if A structurally dominates B, i.e. every structured control-flow path from the function's entry block to block B includes block A, then in particular every *regular* control-flow path from the function's entry block to block B includes block A, since a regular control-flow path is a special case of a structured control flow path, hence A dominates B. Thus 'A structurally dominates B' implies 'A dominates B'.

By a similar argument, 'B structurally post dominates A' implies 'B post dominates A'.

Our changes to the SPIR-V specification mean that the definition of regular post dominance is no longer required, and it has been removed. The definition of regular dominance is still required, because the SSA form used by SPIR-V instructions requires every use of an id to be dominated by the definition of the id (with the exception of uses that occur in SSA phi instructions).

We also adapt the notion of "back edge" so that it is defined over structured control-flow edges, rather than merely branch edges:

> *Back Edge*: A branch edge that branches to one of its ancestors in a depth-first search over structured control-flow edges starting at the function's entry block. [SPIR-V 1.6r2, §2.2.5]

Note that a back edge must still be a *branch* edge, because the notion of back edge relates to execution of a loop, and only branch edges describe how control can transfer between blocks at runtime. However, the depth-first search that identifies a back edge can involve all three edge types.

The definition of "back-edge block" is unchanged (see Section 2.2), but its meaning is different as it refers to this new notion of back edge.

Armed with these definitions, we simplified rules and definitions associated with structured control flow as follows. First, we were able to remove caveats about unreachability from rules that govern dominance relationships between header blocks, merge blocks and continue targets (text that we added is shown in **bold**; text that we removed is ~~struck through~~):[8]

> each header block must strictly **structurally** dominate its merge block~~, unless the merge block is unreachable in the CFG~~ [SPIR-V 1.6r2, §2.11.1]

and later in the same section:

> for a given loop header, its `OpLoopMerge` Continue Target, and corresponding back-edge block:
>
> - the loop header must **structurally** dominate the Continue Target~~, unless the Continue Target is unreachable in the CFG~~
> - the Continue Target must **structurally** dominate the back-edge block
> - the back-edge block must **structurally** post dominate the Continue Target
>
> [SPIR-V 1.6r2, §2.11.1]

Secondly, we were able to give simpler, clearer definitions of selection, continue and loop constructs (again, we use **bold** for added text and removed text is ~~struck through~~):

> - a *selection construct*: the blocks **structurally** dominated by a selection header, while excluding blocks **structurally** dominated by the selection header's merge block
> - a *continue construct*: the blocks that are both **structurally** dominated by an `OpLoopMerge` Continue Target and **structurally** post dominated by the corresponding loop's back-edge block ~~while excluding blocks dominated by that loop's merge block~~

[8] Our changes also incorporate some minor wording changes to improve clarity; we do not highlight these changes in order to focus attention on the more fundamental changes.

> - a *loop construct*: the blocks **structurally** dominated by a loop header, excluding both the **loop** header's *continue construct* and the blocks **structurally** dominated by the loop header's merge block
>
> ~~Furthermore, these structured control-flow constructs are additionally defined to exclude all outer constructs' continue constructs and exclude all blocks dominated by all outer constructs' merge blocks.~~ [SPIR-V 1.6r2, §2.11.2]

The major advantage of our revised definitions is that they are unambiguous and non-circular. Removing the final caveat—the source of circularity—also makes the rules more concise and elegant, and thus easier to work with and reason about.

In Section 5 we discuss how we have worked to bring the SPIR-V validator and conformance tests in line with these new definitions and rules, by cross-checking them against our formal model.

4.3 Structural Dominance Applied to Problematic Examples

In Section 4.1 we already discussed how structural dominance solves the problem discussed in Section 3.1 associated with a loop break appearing in a selection construct. We briefly discuss how our changes fix the other problems of Section 3 in an elegant manner.

Well defined selection constructs featuring early loop continues. Recall the problem of Section 3.2, where unwieldy caveats were required to avoid the continue construct of a loop becoming part of a selection construct inside the loop body. Structural dominance avoids the need for such caveats: there is always a continue edge from a loop header to its continue target, so the loop header is the only block in a loop construct that can structurally dominate the loop's continue target (assuming that the loop header is structurally reachable).

Back edges in loops with unreachable continue constructs. The problem identified by the CFG of Fig. 5 (Section 3.3) is that with the old SPIR-V definitions, the loop headed at block 2 had no associated back edge. With our new definitions, branch edge $3 \longrightarrow 2$—which we would conceptually like to be the back edge—is indeed a back edge, because it is a retreating edge in the structured control flow path $1 \longrightarrow 2 \dashrightarrow 3 \longrightarrow 2$ that starts from the CFG's entry block.

Unreachable constructs are not empty, and cyclic merge relationships are not allowed. In Section 3.4 we discussed the problem of mutual dominance between unreachable blocks. This led to the CFG of Fig. 6 having a selection construct that is technically empty, and to the strange CFG of Fig. 7 being deemed valid. Structural dominance solves both of these issues. In Fig. 6, the selection construct headed by block 4 is now defined to be the block *structurally* dominated by the header block 4, i.e. the set $\{4, 5, 6, 7\}$, minus the blocks *structurally* dominated by the merge block 7, i.e. the set $\{7\}$, yielding the set $\{4, 5, 6\}$, which matches our intuition based on the block labels presented in the source code of Fig. 6. Turning to Fig. 7, this example is now deemed invalid, because header block 10 does not structurally dominate its merge block 5.

There is always a structured path to an exit block. The CFG of Fig. 8 (Section 3.5) was problematic due to there being no regular control flow path from certain blocks to an exit block. Such blocks are vacuously post dominated by all blocks in the CFG. With *structural* post dominance, there is always a structured path from a structurally reachable block to an exit block. This is because cycles can only be achieved via loop constructs, a loop construct must have an associated header block, that header block must feature a merge block that is not part of the cycle, and thus the merge edge from loop header to merge block provides a way out of the cycle. With respect to Fig. 8, the continue construct for the loop headed by block 3 comprises the set of blocks $\{4\}$ as desired, because 4 is

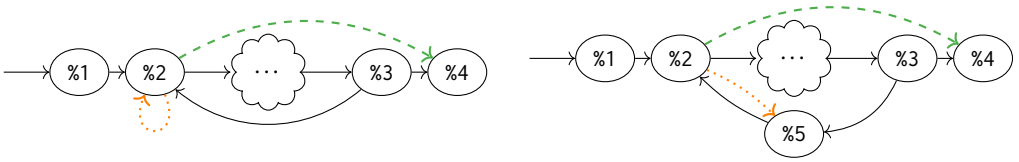


Fig. 9. A now-invalid control-flow idiom that `spirv-opt` would generate (left), and how to rectify it (right)

the only block that is both structurally dominated by the loop's continue target (block 4) and post dominated by the loop's back-edge block (also block 4).

Unambiguous exit rules. Thanks to our new, unambiguous definitions of control flow constructs, we were able to articulate a precise set of rules dictating when it is acceptable for a branch edge to leave a construct, replacing the problematic existing exit rules that we briefly discussed in Section 3.6. We present three of these rules for illustration:

> if a branch edge from block A to block B exits the structured control-flow construct S, then the exit must correspond to one of the following:
>
> - Breaking from a selection construct: S is a selection construct, S is the innermost structured control-flow construct containing A, and B is the merge block for S
> - Breaking from the innermost loop: S is the innermost loop construct containing A, and B is the merge block for S
> - Entering the innermost loop's continue construct: S is the innermost loop construct containing A, and B is the continue target for S [SPIR-V 1.6r2, §2.11.3]

The choice of rules was based on discussion with the SPIR-V language designers. The key point is that our improved definitions of constructs now make it possible to articulate such rules.

4.4 Backwards Incompatibility

We encountered two CFG idioms where our changes to SPIR-V control flow are not backwards-compatible. Both are related to the requirement that (since our changes) a loop's back-edge block should (structurally) post dominate its associated continue target. Intensive automated analysis using our Alloy model of structural dominance (see Section 5) gives us a high degree of confidence that backwards incompatibility is limited to these cases.

Fig. 9 illustrates two CFGs where in each case the blob labelled "…" is meant to denote an arbitrary sub-CFG that does not itself contain an exit block. Prior to our work, the inlining pass of `spirv-opt`, the SPIR-V optimiser, would sometimes yield CFGs with the form of the left CFG in Fig. 9. Consider loop header 2, which is its own continue target and has associated back-edge block 3. The old SPIR-V rules required 3 to post dominate 2, which it does. Our new rules require 3 to *structurally* post dominate 2, which it does not: the merge edge 2 -⇢ 4 means that there is a structural path from 2 to an exit block that does not pass through 3.

The language designers agreed that it was acceptable to regard programs of this form as invalid. We contributed a change to the optimiser's inlining pass that avoids the problem by adding an empty intermediate block as the loop's continue target, as illustrated by the right CFG in Fig. 9, which features the additional block 5. We also fixed a number of test cases in SPIR-V-related test suites that suffered from this issue because their associated code had been passed through `spirv-opt`.

The other idiom we found is rather complex and contrived: it involves a loop whose continue construct contains a loop that in turn contains a selection construct with an unreachable merge block. The unreachable merge block leads to the selection construct of the outer loop failing to

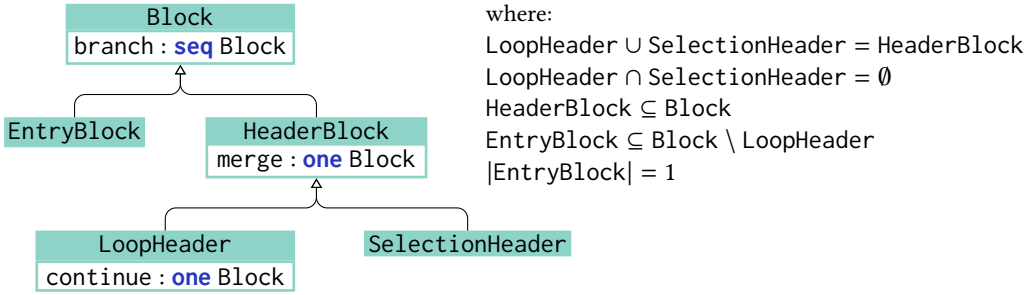


Fig. 10. Hierarchy of Alloy signatures that represent types of block (with ◁— denoting inheritance), together with some constraints on their relationships

meet the property that its back-edge block must post dominate its continue target. We have not encountered a real-world SPIR-V program where this backwards incompatibility matters, and the language designers agreed it was acceptable for this contrived example to become invalid.

4.5 Structurally Unreachable Blocks

Unlike the use cases in Section 3 involving blocks that are unreachable by regular control flow yet still structurally reachable, there are no compelling uses cases for structurally unreachable blocks. We discussed with the language designers whether to simply ban such blocks. In the end it was agreed to allow them but have the specification place no restrictions on structurally unreachable control flow constructs so they can in effect be ignored.

5 THE DESIGN AND APPLICATION OF OUR FORMAL MODEL

We now describe how we have formalised SPIR-V control flow using the Alloy modelling language [Jackson 2019], and used the Alloy Analyzer to put that model to use in two ways: generating CFGs that probe the corner cases of the model, and testing whether a given CFG is valid.

Alloy has three features that make it a good fit for our context. First, it allows the definition of directed graphs with customisable vertex types (called 'signatures') and edge labels. This is an ideal match for CFGs, which are, of course, graphs. Second, it allows constraints to be defined on those graphs, via a language that combines first-order logic and Tarski's relation algebra [Tarski 1941]. We use this feature to concisely capture validity constraints on CFGs. Third, it allows a subtyping hierarchy to be defined among the vertex signatures. We use this feature to capture hierarchical relationships in the CFG, such as: some blocks in the CFG are header blocks, and some header blocks are loop header blocks.

Figure 10 shows how we have captured the hierarchy of different types of CFG blocks using Alloy signatures. All vertices of type `Block` have outgoing `branch` edges to an ordered **seq**uence of `Blocks` (the branch targets of that block). `HeaderBlock` is a subtype of `Block`; blocks of this type have an outgoing `merge` edge to exactly **one** `Block` (the merge target). `LoopHeader` and `SelectionHeader` are two subtypes of `HeaderBlock`; blocks of the former type have a `continue` edge to exactly **one** `Block` (the continue target). As can be seen in the additional constraints on the right-hand side of the figure, every `HeaderBlock` is either a `LoopHeader` or a `SelectionHeader` but not both. Finally, `EntryBlock` is a further subtype of `Block`, and as can be seen from the last two constraints on the right-hand side, there is exactly one `EntryBlock` per CFG and it cannot be a `LoopHeader` (as discussed in Section 2.1, this is forbidden in SPIR-V).

```
pred LoopHeaderStructurallyDominatesContinueTarget
{
  let StructurallyReachableBlock = EntryBlock . *(branch.elems + merge + continue) |
  StructurallyReachableBlock <: continue in structurallyDominates
}
```

Fig. 11. Capturing one of the validity constraints on CFGs as an Alloy predicate

Fig. 11 illustrates one of our model's 26 predicates that capture CFG validity constraints, all of which must hold for a CFG to be deemed valid. This predicate encodes the requirement that every (structurally reachable) loop header in the CFG must structurally dominate its continue target (see Section 4.2). It works in two steps. On the first line, it defines the set of structurally reachable blocks as those that can be reached by starting at the `EntryBlock` and then taking zero or more steps, each of which is either a `branch`,[9] `merge`, or `continue`. On the second line, the predicate constructs the set of pairs of blocks (A, B) where there is a continue edge from A to B and A is in the set of structurally reachable blocks (the `<:` syntax restricts the domain of a binary relation). The predicate requires that this set is a subset of ('`in`') the structural dominance relation; that is, that A structurally dominates B.

Having set up this system of constraints, we can synthesise CFGs that witness them by fixing upper bounds on the number of vertices per signature and then invoking the Alloy Analyzer. The Analyzer works by reducing the constraint-solving problem to a Boolean satisfiability problem and then invoking an off-the-shelf SAT solver. We can, for instance, ask the Analyzer to generate CFGs (up to a given size) that satisfy all 26 of the validity predicates. We can also negate one of the validity predicates and re-run the Analyzer to generate CFGs that are invalid, but are nonetheless close to the valid/invalid boundary.

5.1 From SPIR-V to Alloy and Back

To enable validation of the CFGs of SPIR-V programs using our Alloy model we have written a translation tool, `spirv-to-alloy`, that takes a binary SPIR-V module and turns it into a format ready for analysis with Alloy. Fig. 12a shows the SPIR-V function from Fig. 2a in a form Alloy can process. Conversely, when the Alloy Analyzer yields an example CFG from our model, it is useful to be able to turn this into a SPIR-V program that exhibits the CFG, to assess whether `spirv-val` agrees with our model on the validity of the CFG. For this purpose we have implemented `alloy-to-spirv`, which takes as input an Alloy-generated example (in the form of an XML file), and yields a corresponding SPIR-V program. The SPIR-V program is rather meaningless because it does not compute anything: it merely contains the label, branch and merge instructions needed to syntactically express the CFG of interest (with *true* used when a Boolean value is required in a conditional branch instruction). Nevertheless, it is meaningful to ask `spirv-val` whether such a program obeys the rules of the language. In Section 6 we discuss strategies for fleshing out such skeletal programs into meaningful test cases.

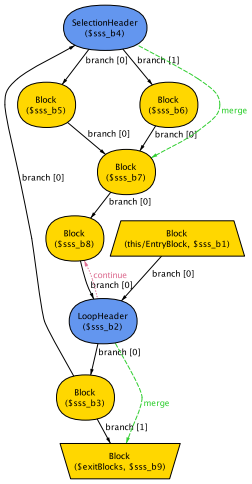
5.2 Putting the Model into Practice

In Section 4 we gave an overview of our "finished product": a revised set of control flow definitions and rules based on our new concept of structural dominance. However, this solution was far from obvious to us initially. We arrived at it via an iterative process summarised by Fig. 1 (Section 1).

[9]Technical detail: `branch.elems` converts the ordered sequence of branch targets into an unordered set.

```
module simple_shader
open AlloyModel/StructuralDominanceCFG as validCFG
pred sampleCFG {
  some disj b1,b2,b3,b4,b5,b6,b7,b8,b9: Block {
    EntryBlock  = b1
    HeaderBlock = b2 + b4
    LoopHeader  = b2
    SwitchBlock = none
    branch = (b1 ->  (0 -> b2))
           + (b2 ->  (0 -> b3))
           + (b3 -> ((0 -> b4) + (1 -> b9)))
           + (b4 -> ((0 -> b5) + (1 -> b6)))
           + (b5 ->  (0 -> b7))
           + (b6 ->  (0 -> b7))
           + (b7 ->  (0 -> b8))
           + (b8 ->  (0 -> b2))
    merge  = (b2 -> b9)
           + (b4 -> b7)
    continue = (b2 -> b8)
  }
}
run { sampleCFG ∧ validCFG/Valid } for 9 Block
```

(a) The input module to Alloy



(b) Predicate is consistent

Fig. 12. A SPIR-V Function in Alloy and the generated instance

We first designed a best-effort Alloy model based on a careful reading of the specification and discussion with SPIR-V experts. We then proceeded to cross-validate our model against two sources of truth about SPIR-V: the `spirv-val` validator, and a number of conformance test suites.

We extracted 340 distinct SPIR-V CFGs from the Vulkan Conformance Test Suite (CTS) [Khronos Group 2022c], 243 distinct CFGs from an open source repository of SPIR-V control flow examples [David Neto 2022], and 134 distinct CFGs from the `spirv-val` test suite [Khronos Group 2022i], using `spirv-to-alloy` to allow feeding these inputs to our model. The `spirv-val` test suite yielded a mixture of CFGs that purported to be valid and invalid, while the CFGs from the other sources all purported to be valid. For each extracted CFG, we used the Alloy Analyser to check whether our model agreed with the CFG's claimed validity. We discussed mismatches with SPIR-V experts and proposed fixes to the relevant test suite or our model, accordingly.

To cross-check against `spirv-val`, we used the Alloy Analyzer to generate a large number of CFGs of interesting shapes and varying sizes, imposing all of our model's constraints to generate valid CFGs, and negating selected constraints to generate invalid (but *almost*-valid) CFGs. We used `alloy-to-spirv` to convert these into skeletal SPIR-V programs and cross-checked the validity of each example as claimed by our model against `spirv-val`'s verdict. In the case of mismatches, we either fixed `spirv-val` or refined our model, guided by discussions with experts.

We iterated these two forms of cross-checking until we reached a point where (a) our model, the test suites and `spirv-val` were all in agreement, and (b) we and the SPIR-V language designers were satisfied with the simplicity, elegance and intuitive nature of the definitions and rules we had formulated. We then rewrote the relevant sections of the SPIR-V specification based on our formal model (but using precise English rather than mathematics), and worked with the Khronos Group to get these changes incorporated into the 1.6r2 release of the language specification.

During this journey of model construction and refinement we tracked numerous issues that arose along the way via our GitHub repository [Klimis et al. 2022b]. The issues related to the modelling process have the `modelling` label. While they are not organised systematically, they may still be

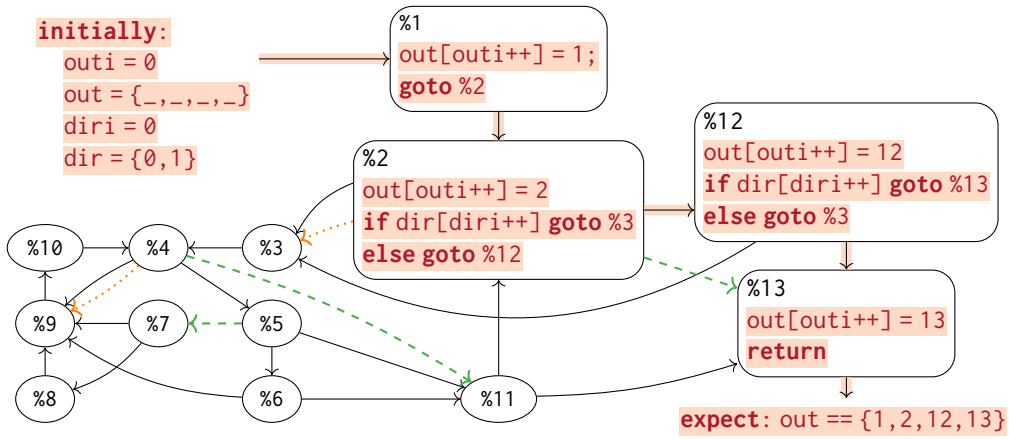


Fig. 13. Example of a fleshed CFG. The original CFG comprises blocks `%1` through `%13`. The path chosen for fleshing is marked (⟹). The code added to the blocks to force execution to follow that path is `highlighted`.

interesting for a reader wishing to gain insight into how we worked through the details of various intricate examples.

6 FUZZING SPIR-V COMPILERS VIA CONTROL-FLOW GRAPH FLESHING

We leverage our formal model of SPIR-V control flow to obtain a novel method for compiler fuzzing, which we call *fleshing*. We report on results of a pilot study that shows that straightforward fleshing is effective, finding twenty distinct new bugs in six SPIR-V commercial compilers and translators. The bugs we found include eight miscompilations and twelve compiler or translator crashes, plus two crashes in a related downstream compiler.

Fleshing is a three-stage process that involves:

(1) producing a valid SPIR-V CFG *skeleton*,
(2) choosing a random path through the CFG, and fleshing out the CFG with instructions that force this path to be taken and record the path actually taken at runtime, and
(3) augmenting the program with a test oracle asserting that the path recorded at runtime exactly matches the expected path.

We detail our fleshing approach (Section 6.1) and present the results of our pilot study (Section 6.2).

6.1 Control Flow Graph Fleshing

The aim of fleshing is to produce an executable test case for which we have a strong oracle that can detect miscompilation errors, and such that the test case is small enough to be easy for a human developer to understand, so that they can use it for debugging. Fig. 13 shows an example of a fleshed test case that was able to find a miscompilation bug in the Intel Mesa driver [Intel 2022].

The first stage of fleshing is to produce a valid SPIR-V CFG skeleton. We use two methods to obtain skeleton CFGs: (1) we use the Alloy Analyzer to produce an example CFG via our formal model, and then use `alloy-to-spirv` to turn this into a skeletal SPIR-V program exhibiting this CFG, and (2) we mine CFGs from the Vulkan Conformance Test Suite (CTS) [Khronos Group 2022c] test cases, validating them against our formal model to ensure that they are valid. It would also be possible to write a random generator of SPIR-V CFGs, however, since our Alloy model already gives us a mechanism to produce interesting, valid SPIR-V CFGs, we did not implement this.

The second stage of fleshing takes as input a valid SPIR-V CFG and chooses a particular path through the CFG that the fleshed program must follow. By default this produces a single path that will be executed by a single thread. We also provide an option whereby multiple threads each follow a distinct path. The path is chosen by performing a random traversal of the CFG's branch edges until an exit block is reached, or a given maximum path length is exceeded. In the latter case, the path is extended to reach an exit block via the shortest route possible. Our algorithm pre-computes those blocks that cannot reach a CFG exit node via branch edges and excludes them from consideration. In the example in Fig. 13, the chosen path is marked ($\Rightarrow$).

In the third and final stage, we add flesh to the bones of our skeletal SPIR-V CFG to turn it into an executable test case. This involves adding instrumentation to force the program to follow the path chosen in the previous step, and to record the actual path taken via a program output. The actual path can then be compared with the expected path once the program has been executed. To achieve this, a directions array, `dir`, is created to store the control-flow decisions that will be made when executing each block that terminates with a conditional instruction. Since blocks may be visited multiple times, `dir` may contain multiple entries, so we use an index variable, `diri`, to select a potentially different direction on each visit, incrementing it after each visit. In the example in Fig. 13, `dir` is configured with `{0,1}` so that the path takes the 'false' branch from block `%2` and then the 'true' branch from block `%12`. An output array `out` and associated index `outi` are used to track the path taken at runtime. Whenever a block is visited, the block's integer ID is written to the output array at index `outi` after which the index is incremented. The compiler must preserve stores to the output array, because the array is in memory that will become visible to the host CPU. Thus if the program is compiled correctly, the full path taken will always be recorded. This forms the basis for our test oracle, as we know both the expected path and the actual path taken at runtime. In Fig. 13, which exposed a bug in the Intel Mesa driver, we found that the compiled code led to the `out` array not containing the expected values `{1,2,12,13}`; in fact it was not written to at all.

Our fleshed programs can then be used to (a) find bugs in SPIR-V compilers inside vendor-specific drivers, and (b) find bugs in cross compilation tools that turn SPIR-V into other formats. Because the size of a fleshed test case is proportional to the size of the fleshed CFG, and due to the simple nature of the instrumentation that we propose, fleshed test cases that expose bugs are straightforward for developers to comprehend and thus provide a good starting point for debugging.

For direct execution on a device, we wrap the fleshed program in AmberScript [Google 2022a] code which handles the scaffolding of executing SPIR-V code and comparing the output array to the expected path. With this approach, we can detect both crashes and miscompilations: miscompilations are found when there is a mismatch between the IDs in the output array and the expected path IDs, or if a timeout is exceeded, indicating an erroneous infinite loop or GPU hang. For cross-compilation, our fleshed programs are output as pure SPIR-V assembly which is fed to both SPIRV-Cross [Khronos Group 2022h] and `naga` [Rust Graphics Mages 2022] to produce code in a range of target languages such as HLSL [Microsoft 2019], GLSL [Khronos Group 2019] and the Metal Shading Language (MSL) [Apple 2022]. For each target language, we run the resulting code through its native downstream toolchain: `glslangValidator` for GLSL, the DirectX Shader Compiler for HLSL and Apple's shader validator for MSL. If a program is rejected by one of the downstream tools it indicates a likely translation error which can be investigated. We can also detect crashes in the downstream tools, but currently not miscompilations as this would require engineering test scaffolding for each target language.

Using phi instructions. Basic fleshing uses memory loads and stores to maintain the index variables associated with the direction and output arrays. Since the SPIR-V specification requires programs

Table 1. Number of bugs found by each test set for every compiler and translator tested.

		Compilers				Translators	
Test set	**Number of test cases**	Mesa	SwiftShader	MoltenVK	NVIDIA	SPIRV-Cross	Naga
Basic CTS	5,668	2	0	0	0	0	2
Basic Alloy	536,601	7	3	0	0	3	0
Phi CTS	34,751	1	0	0	0	0	0
Phi Alloy	310,540	0	0	0	0	2	0

to be in SSA form, it is relatively easy to replace these load and store instructions with SSA phi instructions, allowing fleshing to exercise a different aspect of SPIR-V compilation.

Multiple independent paths. We also provide a mode where instead of fleshing code that forces one thread to follow a particular path, a number of distinct threads are forced to explore an independent path each, in parallel. We support an arbitrary number of threads spread across an arbitrary number of workgroups, although in practice the maximum number is limited by the limits of the execution platform. Whilst compatible with phi instructions, we do not include them in this mode.

6.2 Evaluation

To evaluate the effectiveness of our fuzzer, we conducted a basic fuzzing campaign. We focused our effort on six SPIR-V compilers and translators. Using the output of the SPIR-V translators, we were also able to test for crash bugs in three high-level language compilers for HLSL, GLSL, and MSL.

Method. We started our fuzzing campaign by generating two sets of CFG skeletons that we used throughout. The first set of 400 skeletons was generated by scraping the Vulkan CTS for valid CFGs and turning them into skeletons ready for fleshing. The second set of skeleton CFGs were generated from our Alloy model and consisted of 2499 skeletons that ranged widely in size. We generated this set by running the Alloy analyzer on several variants of our model, each augmented with a different set of additional constraints on the allowed form of CFGs. These additional constraints include: preventing Alloy from lazily building long branch-less patterns by adding a constraint ensuring that there does not exist blocks A, B, C such that B is A's only structural successor and C is B's only structural successor; enforcing that if there is a branch edge from block A to block B then at least one of A or B must be a header block, merge block or continue target (focusing attention on structures that make maximal use of structured control flow annotations); imposing the constraint that there should not exist two selection constructs with the same number of blocks, with similar constraints for loop and switch constructs (to increase diversity in the kinds of structures that are generated); and imposing constraints that require some nesting between constructs to be present.

Once we had generated the skeletons, we began fleshing. For each of the fleshing modes (basic, phi and independent paths), we generated two sets of test cases by invoking our fleshing tool on the Vulkan CTS skeletons and the Alloy skeletons, respectively. This led to a total of six test sets of varying size. To avoid needlessly executing duplicate test cases, we removed test cases within the same test set that both followed exactly the path and exhibited the same set of features.

Test Setup. The CFGs were generated by Alloy Analyzer 5.1.0, using MiniSat as its SAT solver, running on a MacBook Pro i7-1068NG7 with 32GB RAM and Intel(R) Iris(TM) Plus Graphics. MoltenVK 1.1.10 was tested on the same machine, as were the naga and SPIRV-Cross to MSL translators. Mesa, SwiftShader and the remaining SPIRV-Cross and naga translator configurations were tested on a laptop running Ubuntu 20.04 with an i7-1165G7 CPU, 16GB RAM and Intel(R) Xe Graphics (TGL GT2). To test the NVIDIA driver, we used a machine with an Intel Xeon E5-2640

Table 2. Number of bugs found by category in compilers and translators.

	Compilers				Translators	
Bug Type	Mesa	SwiftShader	MoltenVK	NVIDIA	SPIRV-Cross	Naga
Crash	6	1	0	0	3	2
Miscompilation	4	2	0	0	2	0

Table 3. CFGs found by Alloy in 4-hour periods.

	Number of blocks			
	8	10	12	14
CFGs	178,588	97,522	3,052	21

v3 CPU, 32GB RAM and an NVIDIA GTX 980 GPU. The compiler/translator versions tested were SwiftShader [Google 2022e] (commit d15c4248); Mesa version 22.0.1 and merge request 17922 (Basic test sets), version 22.1.0 (phi test sets) and version 22.1.2 (independent paths test sets); NVIDIA Linux driver version 510.47.03; MoltenVK [Khronos Group 2022d] version 1.1.10; naga (commit b3d5e6d8); and finally SPIRV-Cross (commit 6ae7ddb9).

Results. Overall, our campaign was effective, finding twenty new bugs in the six SPIR-V compilers and translators tested, as well as two crash bugs in Microsoft's DirectX Shader Compiler [Microsoft 2022]. We manually examined all bugs before reporting them to ensure they were all distinct. Table 1 shows the distinct bugs found by each test set, listing test sets in increasing order of sophistication, with later test sets having more features and/or a larger number of test cases than earlier test sets. Since many test sets were capable of finding the same bug, we assign each bug to the simplest test set that could find it. The independent paths test sets found many bugs that earlier test sets had already discovered, but did not find any distinct bugs, so we exclude them from Table 1. The test sets generated from our Alloy model found three times as many distinct bugs (15) as the test sets from the Vulkan CTS (5), demonstrating the value of the Alloy model for producing new test cases. Table 2 provides a breakdown of the type of bugs found for each SPIR-V compiler and translator.

For both the SPIR-V compilers and translators, we classify bugs into two groups: (1) miscompilations, which occur when the code produced by the compiler or translator produced the wrong result, either due to a mismatch in the expected path taken or by entering an infinite loop, and (2) crashes, which encompass all other bugs, which include but are not limited to crashes in the compilers themselves, and translators producing invalid code that is rejected by the target language compiler.

We found miscompilation and crash bugs in both the Mesa and SwiftShader compilers, as well as in the SPIRV-Cross translator. We found only crash bugs in naga, however, these crash bugs were triggered by over 95% of test cases, limiting our ability to find deeper bugs. We did not find any bugs in the MoltenVK and NVIDIA compilers. However, MoltenVK uses SPIRV-Cross as a SPIR-V front-end, so we were able to find the miscompilation bugs in SPIRV-Cross by virtue of testing MoltenVK, and these bugs would impact MoltenVK users. As well as testing the release versions of the various compilers and translators, we also worked closely with a member of the Intel Mesa team to test a new version of their SPIR-V control flow handling implementation. As a result, we found five bugs in the new implementation (including both crash and miscompilation bugs) before release. We believe this demonstrates the value of integrating our fuzzer as part of an ongoing development process. We reported all twenty of the bugs found and fourteen of them have been fixed, with a further three bugs confirmed.

Throughput. The throughput of our fuzzing is limited by the execution speed of the underlying platform. We use the Independent Paths test set (658, 388 test cases) to characterise the throughput of the execution platforms, as it was at least as expensive to execute as the other test sets. The execution times ranged from just over ten hours on the fastest platform (Mesa) to over thirty-nine hours on the slowest (NVIDIA). We believe this is due to the overhead of copying the direction and

output buffers to and from dedicated GPU memory. It took our fleshing tool just over three hours (on the Ubuntu laptop described above) to generate the Independent Paths test set, demonstrating that the fleshing stage is not the bottleneck in the process.

Table 3 shows how the number of blocks in a CFG affects the throughput of the CFG generation process. We measured the number of CFGs of a given size that could be generated in 4-hour periods, using block counts of eight, ten, twelve, and fourteen respectively. It is clear that the number of blocks in a CFG greatly affects the CFG generation throughput, which is not surprising given Alloy uses a SAT solver under the hood.

This highlights a trade-off in the fuzzing pipeline, where it is possible to generate a large number of small CFGs, which even with a small number of fleshed test cases will saturate the execution stage, or to use a small number of large CFGs and flesh them more thoroughly generating many more test cases per CFG. Our experience is that it is the static structural features of a CFG that are more likely to trigger a bug – for example, having nested loops with break and continue statements. We found that the vast majority of bugs could be triggered by many of the test cases for a particular CFG, suggesting that it is more important to explore a wider range of different CFGs than fleshing an individual CFG more thoroughly.

7 RELATED WORK

Model-based test generation using Alloy. Ours is the latest in a long line of research that has used the Alloy Analyzer to generate test cases from models written in the Alloy modelling language. The subjects of prior work have ranged from inter-app communication protocols for Android [Jing et al. 2012] to functional requirements of embedded systems [Wang et al. 2022], and several authors have used Alloy to generate conformance tests from axiomatisations of memory models [Iorga et al. 2021; Lustig et al. 2017; Raad et al. 2020, 2019; Wickerson et al. 2017]. We have found Alloy generally well suited to our needs, but one inherent limitation is that its verification results are bounded, in our context meaning that only CFGs up to a given size can be considered. To move to unbounded verification, it may be possible to attach a theorem-proving backend to Alloy [Macedo and Cunha 2012] or to port our models to the Ivy verification system [Padon et al. 2016].

Extended control-flow graphs. Our modelling efforts have been based around an extended form of CFG – CFGs augmented with 'merge' and 'continue' edges. Control-flow graphs have been extended before, in different ways and with different aims; for instance, Ball and Horwitz [1993] extended CFGs by modifying how jumps are handled so that techniques for program slicing can cope with arbitrary control flow, and CFGs extended with edges that represent data-flow are widely used in high-level synthesis [Amellal and Kaminska 1993].

Related compiler-fuzzing techniques. Our fleshing approach generates "self-checking" test programs: programs that, although randomly generated, come equipped with oracles related to what should be computed. Other compiler fuzzers that produce self-checking test cases include Orange 3 [Nagai et al. 2014] and YARPGen [Livinskii et al. 2020], which generate C/C++ programs. Fleshing is more restricted than these approaches because it is only concerned with control flow. However, both Orange 3 and YARPGen have limited or no support for loops, while fleshing is specifically designed to produce test cases with interesting control flow, including complex use of loops.

Differential and metamorphic compiler-testing techniques have been successful in a range of domains over the last decade or so (see e.g. [Le et al. 2014; Yang et al. 2011] and a recent survey [Chen et al. 2020]). This includes work on testing GPU compilers for OpenCL [Lidbury et al. 2015], OpenGL [Donaldson et al. 2017] and SPIR-V [Donaldson et al. 2021]. We have not undertaken a detailed comparison with our fleshing approach and the `spirv-fuzz` tool of Donaldson et al. [2021], but we note that fleshing is much simpler than the involved metamorphic approach employed

by `spirv-fuzz`, so that even if there is overlap in the bugs the tools can find, there is merit to being able to find bugs using a simpler technique.

Our proposal for CFG fleshing bears some conceptual resemblance to the 'skeletal program enumeration' (SPE) [Zhang et al. 2017] program generation technique. In SPE, an outer loop generates 'syntactic structures' (programs with all identifiers replaced by 'holes') and then an inner loop fills in those structures with various collections of identifiers. By comparison, we have an outer loop that generates CFGs, and then an inner loop that fills in those graphs with various concrete instructions. A key difference between the approaches is that fleshing can find miscompilation bugs because it produces well defined test cases with strong oracles. In contrast, SPE makes no guarantees about the semantic validity of the programs that are generated. In particular, they may trigger dynamic undefined behaviour, which means that although they are effective at finding compiler crashes, they are not suitable for assessing whether a compiler produced correct code.

Formalisation of GPU programming features. We have focused on formalising structured control flow in a GPU programming language. Other GPU language features have been formalised in prior work, e.g. memory operations [Alglave et al. 2015; Batty et al. 2016; Gaster et al. 2015; Lustig et al. 2019], forward-progress guarantees [Sorensen et al. 2021], and lock-step execution [Betts et al. 2012; Collingbourne et al. 2013; Habermaier and Knapp 2012], and many of these formalisation efforts have, like our work, led to clarifications and corrections being made to the original specifications.

8 CONCLUSIONS AND FUTURE WORK

We have presented a success story in applying lightweight formal methods to improve the design of an industrial language for GPU computing. Thanks to our formal modelling work, we developed the notion of *structural dominance*, which has proved key to rectifying a number of problems in the SPIR-V intermediate representation related to control flow. Our model was instrumental in settling on a precise set of replacements for the problematic definitions and rules in earlier versions of the specification, and our changes have been incorporated into the latest public release.

Other aspects of SPIR-V could benefit from formalisation efforts, e.g. its memory model. More generally, we believe there is broad scope for applying similar modelling and cross-checking techniques to bring clarity to difficult features of other programming languages and intermediate representations. With respect to CFG fleshing, we plan to explore more advanced fleshing strategies for SPIR-V that integrate features such as execution barriers that are known to have complex interactions with control flow. We are also interested in investigating the effectiveness of our fleshing idea as a means for finding bugs in compilers for other CFG-based representations.

DATA AVAILABILITY STATEMENT

All tooling and experimental data created/used for this research are openly available for reproduction, reuse, and scrutiny at [Klimis et al. 2022a].

ACKNOWLEDGEMENTS

We are grateful to Pinghi Yu, Chengsong Tan and the anonymous POPL 2023 reviewers for valuable feedback on an earlier draft of this work. This work was supported by the IRIS EPSRC Programme Grant (EP/R006865/1).

REFERENCES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers, Principles, Techniques & Tools, Second Edition.* Pearson.

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the*

Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 577–591. https://doi.org/10.1145/2694344.2694391

Said Amellal and Bozena Kaminska. 1993. Scheduling of a Control and Data Flow Graph. In *1993 IEEE International Symposium on Circuits and Systems, ISCAS 1993, Chicago, Illinois, USA, May 3-6, 1993*. IEEE, 1666–1669.

Apple. 2022. Metal Shading Language. https://developer.apple.com/metal/, last accessed 2022-07-04.

Thomas Ball and Susan Horwitz. 1993. Slicing Programs with Arbitrary Control-flow. In *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93, Linköping, Sweden, May 3-5, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 749)*, Peter Fritszon (Ed.). Springer, 206–222. https://doi.org/10.1007/BFb0019410

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 634–648. https://doi.org/10.1145/2837614.2837637

Sean Baxter. 2020. Khronos Group forum post: Clarify meaning of merge block. https://community.khronos.org/t/clarify-meaning-of-merge-block/106006

Sean Baxter. 2021. Tweet about SPIR-V control flow. https://twitter.com/seanbax/status/1348780718797807622

Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 113–132. https://doi.org/10.1145/2384616.2384625

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2020), 4:1–4:36. https://doi.org/10.1145/3363562

Peter Collingbourne, Alastair F. Donaldson, Jeroen Ketema, and Shaz Qadeer. 2013. Interleaving and Lock-Step Semantics for Analysis and Verification of GPU Kernels. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 270–289. https://doi.org/10.1007/978-3-642-37036-6_16

David Neto. 2022. SPIR-V samples for WebGPU. https://github.com/dneto0/spirv-samples, last accessed 2022-07-04.

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *PACMPL* 1, OOPSLA (2017), 93:1–93:29. https://doi.org/10.1145/3133917

Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1017–1032. https://doi.org/10.1145/3453483.3454092

Jason Ekstrand. 2022. In defense of NIR. https://www.jlekstrand.net/jason/blog/2022/01/in-defense-of-nir/

Benedict R. Gaster, Derek Hower, and Lee W. Howes. 2015. HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models. *ACM Trans. Archit. Code Optim.* 12, 1 (2015), 7:1–7:26. https://doi.org/10.1145/2701618

Davide Di Gennaro. 2015. *Advanced Metaprogramming in Classic C++*. Apress.

Google. 2022a. Amber Repository. https://github.com/google/amber, last accessed 2022-07-04.

Google. 2022b. ANGLE - Almost Native Graphics Layer Engine. https://chromium.googlesource.com/angle/angle, last accessed 2022-07-07.

Google. 2022c. The clspv project. https://github.com/google/clspv, last accessed 2022-06-30.

Google. 2022d. Dawn, a WebGPU implementation. https://dawn.googlesource.com/dawn, last accessed 2022-07-07.

Google. 2022e. SwiftShader, CPU-based Vulkan Implementation. https://swiftshader.googlesource.com/SwiftShader, last accessed 2022-07-07.

Google. 2022f. The Tint project. https://dawn.googlesource.com/tint, last accessed 2022-06-30.

Axel Habermaier and Alexander Knapp. 2012. On the Correctness of the SIMT Execution Model of GPUs. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 316–335. https://doi.org/10.1007/978-3-642-28869-2_16

Intel. 2020. Intel Iris Plus Graphics and UHD Graphics Open Source, Programmer's Reference Manual For the 2019 10th Generation Intel Core Processors based on the "Ice Lake" Platform, Volume 2a. https://01.org/sites/default/files/documentation/intel-gfx-prm-osrc-icllp-vol02a-commandreference-instructions_2.pdf, last accessed 2022-07-06.

Intel. 2022. Mesa 3D Graphics Stack Repository. https://gitlab.freedesktop.org/mesa/mesa, last accessed 2022-07-06.

Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. 2021. The semantics of shared memory in Intel CPU/FPGA systems. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485497

Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76. https://doi.org/10.1145/3338843

Yiming Jing, Gail-Joon Ahn, and Hongxin Hu. 2012. Model-Based Conformance Testing for Android. In *Advances in Information and Computer Security - 7th International Workshop on Security, IWSEC 2012, Fukuoka, Japan, November 7-9, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7631)*, Goichiro Hanaoka and Toshihiro Yamauchi (Eds.). Springer, 1–18. https://doi.org/10.1007/978-3-642-34117-5_1

Khronos Group. 2014. SPIR Specification, Version 1.2. https://www.khronos.org/registry/SPIR/specs/spir_spec-1.2.pdf, last accessed 2022-06-30.

Khronos Group. 2017. SPIR-V Specification, Version 1.0, Revision 12. https://www.khronos.org/registry/SPIR-V/specs/1.0/SPIRV.html

Khronos Group. 2019. The OpenGL Shading Language Version 4.60.7. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf, last accessed 2022-06-30.

Khronos Group. 2022a. A complete registry of all official SPIR-V specifications. https://www.khronos.org/registry/SPIR-V/

Khronos Group. 2022b. glslang GitHub repository. https://github.com/KhronosGroup/glslang, last accessed 2022-06-30.

Khronos Group. 2022c. Khronos Vulkan, OpenGL, and OpenGL ES conformance tests. https://github.com/KhronosGroup/VK-GL-CTS, last accessed 2022-07-02.

Khronos Group. 2022d. MoltenVK, a Vulkan Portability Implementation. https://github.com/KhronosGroup/MoltenVK, last accessed 2022-07-07.

Khronos Group. 2022e. OpenCL-Docs. https://github.com/KhronosGroup/OpenCL-Docs, last accessed 2022-06-30.

Khronos Group. 2022f. SPIR-V Specification, Version 1.6, Revision 1, Unified. https://web.archive.org/web/20220613184046/https://www.khronos.org/registry/SPIR-V/specs/unified1/SPIRV.pdf Also cited as SPIR-V 1.6r1.

Khronos Group. 2022g. SPIR-V Specification, Version 1.6, Revision 2, Unified. https://www.khronos.org/registry/SPIR-V/specs/unified1/SPIRV.html Also cited as SPIR-V 1.6r2.

Khronos Group. 2022h. SPIRV-Cross Repository. https://github.com/KhronosGroup/SPIRV-Crosss, last accessed 2022-07-04.

Khronos Group. 2022i. SPIRV-Tools Repository, including spirv-opt and spirv-val. https://github.com/KhronosGroup/SPIRV-Tools, last accessed 2022-06-30.

Khronos Group. 2022j. Vulkan 1.3 - A Specification (with all registered Vulkan extensions). https://www.khronos.org/registry/vulkan/specs/1.3-extensions/html/vkspec.html, last accessed 2022-07-05.

Vasileos Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. 2022a. Artifact for "Taking Back Control in an Intermediate Representation for GPU Computing", POPL 2023. https://doi.org/10.5281/zenodo.7152484

Vasileios Klimis, Jack Clark, John Wickerson, and Alastair F. Donaldson. 2022b. Repository containing SPIR-V control flow Alloy model and fuzzer. https://github.com/mc-imperial/spirv-control-flow

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. 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*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 65–76. https://doi.org/10.1145/2737924.2737986

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. https://doi.org/10.1145/3428264

LLVM Compiler Infrastructure. 2022. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html, last accessed 2022-07-06.

Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 257–270. https://doi.org/10.1145/3297858.3304043

Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 661–675. https://doi.org/10.1145/3037697.3037723

Nuno Macedo and Alcino Cunha. 2012. Automatic Unbounded Verification of Alloy Specifications with Prover9. *CoRR* abs/1209.5773 (2012), 17 pages. arXiv:1209.5773 http://arxiv.org/abs/1209.5773

Dzmitry Malyshau. 2021. Horrors of SPIR-V. http://kvark.github.io/spirv/2021/05/01/spirv-horrors.html

Microsoft. 2019. Reference for HLSL. https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-reference, last accessed 2022-06-30.

Microsoft. 2022. DirectX Shader Compiler Repository. https://github.com/microsoft/DirectXShaderCompiler, last accessed 2022-07-06.

Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. *IPSJ Trans. Syst. LSI Des. Methodol.* 7 (2014), 91–100. https://doi.org/10.2197/ipsjtsldm.7.91

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 614–630. https://doi.org/10.1145/2908080.2908118

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.* 4, POPL (2020), 11:1–11:31. https://doi.org/10.1145/3371079

Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 135:1–135:27. https://doi.org/10.1145/3360561

Philip Rebohle. 2022. DXVK. https://github.com/doitsujin/dxvk/, last accessed 2022-07-07.

Rust Graphics Mages. 2022. The Naga project. https://github.com/gfx-rs/naga, last accessed 2022-06-30.

Mark Segal and Kurt Akeley. 2022. OpenGL 4.6 Core Profile. https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf, last accessed 2022-07-07.

Tyler Sorensen, Lucas F. Salvador, Harmit Raval, Hugues Evrard, John Wickerson, Margaret Martonosi, and Alastair F. Donaldson. 2021. Specifying and testing GPU workgroup progress models. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485508

Alfred Tarski. 1941. On the Calculus of Relations. *J. Symb. Log.* 6, 3 (1941), 73–89. https://doi.org/10.2307/2268577

W3C. 2022. WebGPU Shading Language W3C Working Draft. https://www.w3.org/TR/WGSL/, last accessed 2022-06-30.

Chunhui Wang, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2022. Automatic Generation of Acceptance Test Cases From Use Case Specifications: An NLP-Based Approach. *IEEE Trans. Software Eng.* 48, 2 (2022), 585–616. https://doi.org/10.1109/TSE.2020.2998503

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 190–204. https://doi.org/10.1145/3009837.3009838

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 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 347–361. https://doi.org/10.1145/3062341.3062379

Received 2022-07-07; accepted 2022-11-07