

Deductive verification of SYCL in VerCors*

Ellen Wittingen, Marieke Huisman^[0000–0003–4467–072X], and Ömer
Şakar^[0000–0003–3457–5446]

University of Twente, Enschede, The Netherlands

Abstract. SYCL is a C++ programming model for the development of heterogeneous programs. It uses the concept of kernels, where multiple instances of a computation are executed concurrently on a computing unit. This concurrency entails that the set of possible program behaviours can be of considerable size, which makes these programs error-prone. Formal verification could be used to ensure the correctness of all these possible program behaviours. However, there exist no formal verification tools for SYCL.

In this paper, SYCL support is added to VerCors, a formal verification tool for concurrent software, by encoding SYCL constructs into VerCors’ internal language COL. To the extend of our knowledge, this is the first deductive verification tool for SYCL. We show how SYCL’s basic- and ND-range kernels are encoded, along with the encoding and challenges related to scheduling kernels and the execution order of those kernels. In addition, we discuss how SYCL’s buffers and data accessors are encoded, focusing on the challenges related to it, in particular enabling memory transfer between host and device. The usability of the added SYCL support and how it was evaluated are discussed as well.

Keywords: VerCors · SYCL · Formal verification · Heterogeneous computing.

1 Introduction

SYCL [18] is a C++ programming model for developing heterogeneous programs, where code is executed on multiple, possibly different, computing units. It allows such programs to be declared on a high level and code for all computing units to be declared inside the main body of a program, instead of separately. This has a smaller learning curve for programmers new to heterogeneous programming than, for example, OpenCL [17]. Moreover, it supports computing units from multiple vendors, unlike, for instance, CUDA [24]. It uses the concept of kernels, where multiple instances of a computation are executed in parallel on a computing unit. The number of possible program behaviours increases exponentially with the size of the parallel parts of a program and the number of threads that execute it [6]. This means that the concurrency of SYCL’s kernels results in many possible program behaviours. This has the problem that bugs

*This work is supported by NWO TTW grant 17249 for the ChEOPS project.

which only occur in a few possible program behaviours are easily overlooked. To solve this, formal verification can be used to reason about the correctness of all possible behaviours of a program. There exist various formal verification tools for heterogeneous programs, e.g. GPUVerify [5], PUG [22], CIVL [32], the tool by Xing et al. [30], and VerCors [6]. However, none of these verify SYCL.

In this paper, we present the deductive verification of SYCL programs in VerCors. To the extend of our knowledge, this is the first deductive verification tool for SYCL. VerCors is a formal verification tool targeted at concurrent programs in various languages and frameworks, including subsets of the heterogeneous computing frameworks OpenCL, CUDA, and OpenMP. It aims to be language-independent, such that support for new languages can be added without redesigning the entire toolset. For every supported language, it has an encoding into a program in its internal language COL, which is then verified against specifications provided by the user.

To enable developers to formally verify their programs, support was added for SYCL to VerCors, by encoding several SYCL constructs into COL.

Concretely, the contributions of this paper are as follows:

- The encodings and the encountered challenges of the following SYCL constructs and behaviours are explained and illustrated with examples:
 - Basic- and ND-range kernel executions and the ability to explicitly wait on their termination (Section 3).
 - The construction and destruction of buffers (Section 4).
 - The construction of data accessors and how they interact with elements of their buffers (Section 5).
 - The execution ordering of kernels (Section 6).
- An evaluation of the implementation and usability of the supported subset of SYCL (Section 7).

The support for SYCL partially builds on top of the current support for OpenCL and CUDA. The common encoding of kernels between SYCL, OpenCL and CUDA was used as a basis on top of which SYCL-specific behavior is encoded. What is newly supported is: 1) the scheduling of (SYCL) kernels, where SYCL kernels can also be executed out of order and 2) memory transfer between the host and device through SYCL’s buffers and data accessors.

These contributions only cover the most relevant SYCL constructs and behaviours. For more details on the added support, and the encodings of all the supported SYCL constructs, we refer to the related Master’s thesis [29].

2 Background

2.1 VerCors

Verification process VerCors [6] uses separation logic to prove partial correctness of concurrent programs. Figure 1 shows an overview of how VerCors achieves

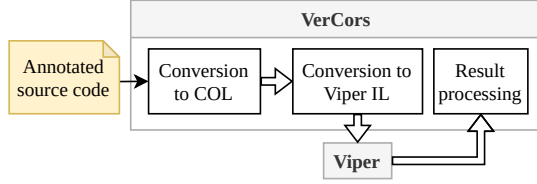


Figure 1: Overview of VerCors' verification process.

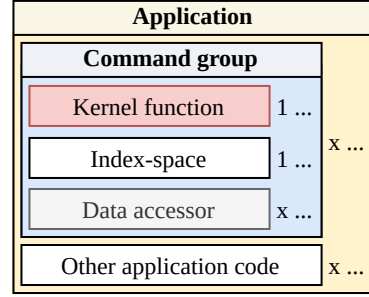


Figure 2: Overview of SYCL's application structure. *The number of components is denoted by $y...$, where x is arbitrary.*

this. It takes source code annotated with specifications and encodes this into its internal language COL, a Java-like language with specific constructs to express concurrency. A running example of such an encoding is the method `someMethod` in Listing 3 (covered in detail in Section 3). It is encoded into a similarly named COL method on lines 18-25 in Listing 4. This COL encoding is then converted in several steps to Viper's [23] intermediate verification language. VerCors then passes this off to Viper, which verifies it using Z3 [14]. Then, the results from Viper's verification are processed in VerCors and shown to the user.

Specifications In VerCors, specifications are written in an annotation language similar to JML [21]. In this annotation language, requires `expr` represents a pre-condition and ensures `expr` represents a post-condition. Inside these specifications, $(\backslash\text{forall } \text{decls}; \text{cond}; \text{expr})$ denotes that `expr` must hold for all possible values for the variables declared in `decls` that meet the conditions described `cond`. VerCors extends these specifications with permission-based separation logic (PBSL) [2,9], which allows users to reason about what data on the heap is accessible when and by which threads. By default, it is assumed that a thread does not access any heap data. However, access to an element `x` can be modelled using the notation $\text{Perm}(\mathbf{x}, \mathbf{p})$, where `p` represents the kind of permission. The permissions relevant to this paper are read and write, which represent, respectively, read-only and read-write permission. Internally, permissions are fractions, where write represents 1 and read a fraction $[0, 1)$. An example of a specification in VerCors can be seen on line 1 in Listing 3, where a thread requires read-write permission to some global variable `x` to execute `someMethod`.

2.2 SYCL

Programming model SYCL [18] is a C++ programming model that enables different types of computing units to be used in tandem in a single application, where a computing unit is a physical component in a system that can perform calculations. Examples of computing units are CPUs, GPUs, and FGPAs

Listing 3: SYCL program where a basic kernel is submitted to a queue, and then, after some arbitrary host code, the submitted kernel’s termination is awaited.

```

1  //@ requires Perm(x, write);
2  void someMethod(sycl::queue queueObject) {
3      sycl::event ev = queueObject.submit(
4          [&](sycl::handler& commandGroupHandler){
5              commandGroupHandler.parallel_for(sycl::range<2>(6,4),
6                  //@ requires kernel pre-conditions;
7                  //@ ensures kernel post-conditions;
8                  [=] (sycl::item<2> it){ kernel body });
9          });
10     arbitrary host code
11     ev.wait();
12 }

```

[25,27,31]. To enable this, a SYCL application consists of *host code* and *kernels*. Host code consists of code that is executed on the main computing unit (the *host*) of the system. Code inside kernels can be executed many times in parallel on a computing unit of choice, concurrently with host code.

Kernels There are two main types of SYCL kernels: *basic kernels* and *ND-range* (*N-dimensional range*) *kernels*. These kernels are invoked with a specific index-space, where for every point an instance of the kernel is executed by a *work-item*. Work-items are organized based on the type of kernel and the index-space.

Both basic and ND-range kernels have a one-, two-, or three-dimensional index-space. In addition to this, the work-items of an *ND-range kernel* are organised into *work-groups*. A work-group is a collection of work-items that can synchronize using barriers.

In SYCL, a kernel and its properties, such as its index-space and data-accessors (discussed in Section 5), are declared inside a *command group*, as illustrated in the overview of SYCL’s application structure in Figure 2. An example of a command group can be seen on lines 4-9 in Listing 3, where an invocation of the `parallel_for` method inside it is used to declare the kernel’s index-space and body. The index-space is represented by a `range` object for a basic kernel or an `nd_range` for an ND-range kernel. The index-space described on line 5 in Listing 3 states that the work-items are organized in two dimensions, 6 by 4 work-items and those will execute the body of the basic kernel declared inside the lambda method on line 8.

3 Encoding of SYCL’s kernels

Listing 3 shows a basic SYCL kernel submission to a queue, the waiting for the termination of that kernel, and arbitrary host code in between. Listing 4 shows the COL encoding of this program. The encoding of ND-range kernels is similar to basic kernels, unless indicated otherwise.

Listing 4: The encoding of the SYCL program in Listing 3. *Constructs with the same highlighting as in Listing 3 are the encoding of that specific construct.* α represents the event-class, β the kernel-runner, γ the kernel-parblock, and ϵ the host code. context *expr*; is shorthand for requires *expr*; ensures *expr*;

```

1   $\alpha$       class SYCL_EVENT_CLASS {
2   $\alpha$       int dim0; int dim1;
3   $\alpha$ 
4   $\alpha$   $\beta$       context Perm(this.dim0, read) ** this.dim0 >= 0;
5   $\alpha$   $\beta$       context Perm(this.dim1, read) ** this.dim1 >= 0;
6   $\alpha$   $\beta$       requires (\forall int ID_0, int ID_1; 0 <= ID_0 && 0 <= ID_1 &&
7   $\alpha$   $\beta$        $\hookrightarrow$  ID_0 < this.dim0 && ID_1 < this.dim1; kernel pre-conditions);
8   $\alpha$   $\beta$       ensures (\forall int ID_0, int ID_1; 0 <= ID_0 && 0 <= ID_1 &&
9   $\alpha$   $\beta$        $\hookrightarrow$  ID_0 < this.dim0 && ID_1 < this.dim1; kernel post-conditions);
10  $\alpha$   $\beta$       run {
11  $\alpha$   $\beta$   $\gamma$       par SYCL_BASIC_KERNEL(
12  $\alpha$   $\beta$   $\gamma$           int ID_0 = 0 .. this.dim0, int ID_1 = 0 .. this.dim1
13  $\alpha$   $\beta$   $\gamma$       ) context Perm(this.dim0, read) ** this.dim0 >= 0;
14  $\alpha$   $\beta$   $\gamma$           context Perm(this.dim1, read) ** this.dim1 >= 0;
15  $\alpha$   $\beta$   $\gamma$           requires kernel pre-conditions;
16  $\alpha$   $\beta$   $\gamma$           ensures kernel post-conditions; { kernel body } } }
17
18  $\epsilon$       requires Perm(x, write);
19  $\epsilon$       void someMethod(ref queueObject) {
20  $\epsilon$           SYCL_EVENT_CLASS sycl_event_ref;
21  $\epsilon$           sycl_event_ref = new SYCL_EVENT_CLASS(6, 4);
22  $\epsilon$           fork sycl_event_ref;
23  $\epsilon$           arbitrary host code
24  $\epsilon$           join sycl_event_ref;
25  $\epsilon$       }

```

3.1 Kernel submissions

SYCL's behaviour A SYCL kernel is *submitted to a queue* in the host code. The queue then schedules the execution of the kernel, whilst the host continues with executing the statements that come after the kernel submission. Currently, only one of SYCL's kernel submission patterns is supported. This pattern is used in various SYCL teaching materials [10,12,11,15], and appears to be the simplest way to declare a kernel. However, adding support for the other submission patterns would not be difficult, as the COL encoding could be the same since they contain the same key elements, albeit using different syntax. In the supported pattern, the `submit` method of the `queue` class is invoked with a command group as argument. An example of a kernel submission can be seen on lines 3-9 in Listing 3. To enable users to reason about SYCL kernels in VerCors, work-item-level specifications, referred to as *user-kernel-specifications*, can be added as a con-

tract to the lambda expression in which a kernel’s body is declared, as can be seen on lines 6-7 in Listing 3.

Encoding For each submission of a kernel, a new COL class, referred to as the *event-class*, is created in the encoding, as can be seen on lines 1-16 in Listing 4. On lines 20-21 an instance of that class is created in the encoding of the host code by calling its constructor with the dimension sizes of the kernel’s index-space as arguments. The constructor¹ then sets the fields of the *event-class*, shown on line 2, to those dimension sizes. This allows the sizes of the dimensions, which do not have to be a constant, to be used inside the *event-class*. A run-method, referred to as the *kernel-runner*, is added to the *event-class*, which can be seen on lines 4-16. This *kernel-runner* is executed in parallel with the *host code* when the instance of the *event-class* is forked on line 22. This models the behaviour that the host continues executing when a kernel is submitted to a queue.

The kernel’s body is encoded similarly to how CUDA kernel declarations are encoded into COL [26]: they are encoded as a single nested parblock for basic kernels, or a double nested parblock for ND-range kernels. Parblocks are denoted by `par NAME(work_items) specs { body }`, where `body` is executed parallelly by the work-items declared in `work_items`. The work-item-level specifications denoted by `specs` are a pre-/poststyle contract which should hold right before and after the execution of `body` respectively.

The encoding of the parallel execution of the kernel body from Listing 3 can be seen on lines 11-16 in Listing 4, where the kernel body is put inside a parblock in the *kernel-runner* on line 16, referred to as the *kernel-parblock*, which has `dim0` by `dim1` work-items.

To enable the kernel body in the *kernel-parblock* to use the *event-class*’s dimension fields, specifications are added to the contracts of the *kernel-runner* and *kernel-parblock*, which check for read access to those fields and that they are not negative. The *user-kernel-specifications* are inserted into those contracts as well. The contract of the *kernel-parblock* is on the same work-item-level as the *user-kernel-specifications*, so the *user-kernel-specifications* are simply inserted, as can be seen on lines 15-16 in Listing 4. In the *kernel-runner*’s contract, the *user-kernel-specifications* are quantified over a range with the same number of work-items as the kernel’s index-space, as can be seen on lines 6-9. Furthermore, all the permissions a *kernel-runner* is supposed to have are already included in its contract by the encoding (such as permissions to data accessors, described in Section 5), so the *user-kernel-specifications* cannot be blindly inserted into its contract. If they were, permissions to an element `x` could end up being declared twice, which makes the encoding unsound. Therefore, all statements regarding permissions in the *user-kernel-specifications* are filtered out, such that they are not added to the *kernel-runner*’s contract.

¹Definition has been omitted from this paper for brevity.

Listing 5: Construction and destruction of a two-dimensional buffer in SYCL.

```

1 { sycl::buffer<T,2> buff =
2   ↪ sycl::buffer(hostData, sycl::range<2>(dim0, dim1));
3   ...
4 } // the buffer is destroyed here

```

3.2 Explicitly waiting on a kernel’s termination

Calling `wait()` on an `event` returned by the submission of a kernel forces the host to wait for that kernel to terminate. For instance, on line 11 in Listing 3 the host waits on the kernel submitted on lines 3-9. In the encoding, the instance of the *event-class* belonging to the kernel is joined, as shown on line 24 in Listing 4, which assumes the *kernel-runner*’s post-conditions to hold at that point.

4 Encoding of SYCL’s buffers

4.1 Buffer constructions

A SYCL buffer can be seen as a copy of a piece of data on the host. It allows this data to be accessed in kernels through data accessors (see Section 5). Buffers are typically declared in the host code, and can be constructed with a pointer to data, called the `hostData`, together with a one-, two-, or three-dimensional `range` object, which describes how the buffer’s copy will appear to be structured to the user. An example of such a buffer construction can be seen on lines 1-2 in Listing 5, where a buffer is constructed with `hostData` as its data and a `dim0` by `dim1` two-dimensional structure. The encoding of creating a copy of the `hostData` when a buffer is constructed is shown on lines 1-2 in Listing 6. As can be seen, a new COL array is created, referred to as a *buffer-array*, and the part of `hostData` within the provided range is copied to that array using the generated method `copy_hostdata_to_buffer`². It was decided to always generate a one-dimensional *buffer-array* as specifications about one-dimensional arrays are easier to verify than specifications about multi-dimensional arrays.

The SYCL specification [18] (page 128) states that when a buffer is constructed with `hostData`, the buffer acquires exclusive access to `hostData`, such that `hostData`’s contents are unspecified during the buffer’s lifetime. However, the extent of this exclusive access is not stated, so the assumption was made that only the part within the bounds of the buffer’s range parameter will be unspecified. The buffer’s exclusive access to that part of `hostData` is encoded by the generated predicate `exclusive_hostdata_access`² on line 3. This predicate is then *folded*, i.e. the contents of the predicate is exchanged for an instance of the predicate. Folding the predicate has the effect that it hides the read-write permission to the part of `hostData` within the buffer’s range from the host. This means that from that point onwards, it will be verified that the host does not access it. This encoding does have as a result that read-write permission to

²Definition has been omitted from this paper for brevity.

Listing 6: The encoding of the buffer construction and destruction in Listing 5. *Constructs with the same highlighting as a construct in Listing 5 are the encoding of that specific construct.*

```

1 T[] buff;
2 buff = copy_hostdata_to_buffer(hostData, dim0 * dim1);
3 fold exclusive_hostdata_access(hostData, dim0 * dim1);
4 ...
5 waiting on kernel terminations
6 copy_buffer_to_hostdata(hostData, buff);
7 unfold exclusive_hostdata_access(hostData, dim0 * dim1);

```

`hostData` is required to construct a buffer, even if the buffer remains unused. If the buffer does later turn out to claim the entirety of its `hostData`, the predicate should cover the entire size of the `hostData` pointer instead.

4.2 Buffer destructions

When a buffer variable goes out of scope, such as on line 4 in Listing 5, it is destroyed. When it is destroyed, the host first waits for the termination of all submitted kernels for which it is required that the buffer's contents are copied back to the host. The assumption was made that this covers all kernels with *read-write* access to the buffer, as those can alter the state of the buffer, and thus the state of the `hostData` on the host. Then, if necessary, the contents of the buffer are copied to the `hostData` to make sure its values match the buffer's latest contents. After the buffer's destruction, the entire contents of `hostData` can be used again without undefined behaviour. On lines 5-7 in Listing 6 the encoding of the buffer destruction in Listing 5 is shown. First, all running kernels with *read-write* access to the buffer are explicitly waited upon, using the same encoding as for invocations of the `event::wait()` method (described in Section 3.2). After that, the buffer's contents are copied to `hostData` using the generated method `copy_buffer_to_hostdata`³. This copy is always executed, to avoid having to evaluate whether it is necessary. Last of all, the predicate `exclusive_hostdata_access` is unfolded to make read-write permission to access `hostData` available to the host again.

5 Encoding of SYCL's data accessors

In this section, SYCL's data accessors and its encoding are explained, which together with SYCL's buffers define SYCL's memory transfer between host and device. The encoding of SYCL's buffers and data accessors is new in VerCors.

³Definition has been omitted from this paper for brevity.

Listing 7: Construction of a two-dimensional data accessor in SYCL.

```

1 void someMethod(sycl::queue q, sycl::buffer<T,2> buff) {
2   q.submit([&](sycl::handler& cgh) {
3     sycl::accessor<T,2> acc = sycl::accessor(buff, cgh, mode);
4     cgh.parallel_for(...); });
5 }

```

5.1 Data accessor constructions

SYCL’s behaviour Data accessors allow memory objects on the host, such as buffers, to be accessed inside kernels. A data accessor can be constructed inside a command group with a buffer object, the command group’s handler object and an access mode. The access mode determines how the accessor’s buffer can be accessed. The access modes chosen to be supported in VerCors are *read-only* and *read-write*, as they could directly be translated to COL permissions. An example of a data accessor construction can be found on line 3 in Listing 7, where a data accessor is constructed with `mode` access to the two-dimensional buffer `buff`.

Some of the unsupported access modes are used in SYCL teaching materials [10,12,15]. However, in those materials they could be replaced by supported ones whilst maintaining a similar, but slightly less efficient behaviour. The unsupported access mode *no-init-read-write* is similar to the *read-write* access mode, except that the previous contents of the buffer are discarded, so it is possible to add support for it. Adding support for SYCL *write-only* and *no-init-write-only* access modes would be more complicated, as there is no direct way to express write permission without read permission in PBSL.

Encoding The encoding of the data accessor construction in Listing 7 is shown in Listing 8. Because a data accessor allows a buffer to be used inside the kernel’s body, the encoding of that buffer needs to be accessible inside the encoding of the kernel’s body, i.e. the *kernel-parblock*. Furthermore, changes made to the buffer in the encoding of the kernel’s body need to be accessible to the encoding of the host code, where the encoding of the buffer resides. These challenging requirements could only be satisfied by adding a field to the *event-class* with the same contents as the buffer’s *buffer-array*, referred to as the *buffer-field*. Furthermore, to share the size of each dimension of the data accessor’s buffer, a field is added for each dimension, referred to as *buffer-range-fields*. These two kinds of fields can be seen on line 2 in Listing 8. To pass the contents of the *buffer-array* and its dimension sizes from the *host code* to these fields, the constructor of the *event-class* is extended to take them as parameters and to use them to set those fields.

To access these fields in the body of the *kernel-parblock*, specifications are added to the contracts of the *kernel-runner* and the *kernel-parblock*, as can be seen on lines 4-8 and 11 in Listing 8. They state that all the fields are read-only. Furthermore, to verify that subscripts to the data accessor are within the *buffer-field*’s bounds, the length of the *buffer-field* is specified to be equal to the length of the *buffer-array*, by stating it is the product of all the *buffer-range-fields*. On line 8, the body of the *kernel-runner* is additionally given read-write permission

Listing 8: Additions to the *event-class* for the encoding of the data accessor construction in Listing 7. The notation `\array(name, size)` denotes that array *name* is not null and has size *size*. For the ternary operator inside the bordered frame only the result is inserted. Constructs with the same highlighting as in Listing 7 are the encoding of that specific construct. The kernel dimension fields and related specifications have been omitted.

```

1  class SYCL_EVENT_CLASS {
2      T[] acc; int acc_dim0; int acc_dim1;
3
4      context Perm(this.acc_dim0, read);
5      context Perm(this.acc_dim1, read);
6      context Perm(this.acc, read);
7      context \array(this.acc, this.acc_dim0 * this.acc_dim1);
8      context Perm(this.acc[*], mode == read_write ? write : read);
9      run {
10         par SYCL_BASIC_KERNEL(dimensions in index-space)
11             same as lines 4-7
12         { ... }
13     } }

```

to all elements of the *buffer-field* if the accessor’s access mode is *read-write*, or read-only permission if the accessor’s access mode is *read-only*. To let the user decide what work-items have access to what parts of the buffer, this last specification is not added to the *kernel-parblock*. These described specifications are positioned before the encoding of the *user-kernel-specifications*, such that users have enough permission to use the data accessors in their kernel specifications.

Data accessors with same buffer The possibility in SYCL to declare multiple data accessors for the same buffer object for the same kernel added a challenge to the encoding. It affects the kernel’s access to a buffer: it will equal the most permissive access mode of the data accessors for that buffer [18] (page 26-27). To give an example, a kernel with a *read-write* and a *read-only* data accessor to the same buffer will be given *read-write* access to that buffer. Another challenge was that changes written to one data accessor must be present in all other data accessors for that buffer as well. In the encoding this is solved by only creating a single *buffer-field* and a single set of *buffer-range-fields* for each buffer the kernel has one or more data accessors for. All data accessor usages for a buffer are encoded to use those fields, and the permission given in the contract of the *kernel-runner* to the elements of the *buffer-field* is the permission that corresponds with the combined access mode. One downside to this approach is that if a kernel contains a *read-write* and a *read-only* data accessor for the same buffer, the user is allowed to write to the *read-only* data accessor in the kernel body, because the kernel body has read-write access to the *buffer-field* in the encoding due to the access mode combination. However, the state of the buffer is not affected by which data accessor is used, so the encoding is sound.

Listing 9: Subscripting a two-dimensional data accessor in a kernel in SYCL.

```

1 //@ context idx0 < acc.get_range().get(0);
2 //@ context idx1 < acc.get_range().get(1);
3 //@ context Perm(acc[idx0][idx1], write);
4 [=] (sycl::item<2> it) { acc[idx0][idx1] = value; }

```

5.2 Interacting with elements of a data accessors' buffer

The elements of an accessor's buffer can be accessed in kernels and their contracts in the same way as C++ arrays are accessed: with subscripts on the data accessor object, where the number of required subscripts equals the number of dimensions of the data accessor's buffer. To give an example, the data accessor `acc`, which provides access to a two-dimensional buffer, declared on line 3 in Listing 7 would be accessed with two subscripts, as can be seen on line 4 in Listing 9. These data accessor subscripts are encoded as a one-dimensional subscript on the data accessor's *buffer-field*, by means of linearization with the buffer's dimension sizes.

Required user specifications To interact with a data accessor, the user needs to specify in the contract of the kernel what elements of the buffer each work-item is allowed to access, as that contract is merged with the contract of the *kernel-parblock*. An example of such a specification can be seen on line 3 in Listing 9. VerCors checks whether subscripts on an array are within the array's bounds. However, VerCors can often not infer this by itself for data accessors due to the complexity of the encoding. In such cases, the user needs to specify in the kernel's contract that the subscripts are within the bounds of the dimensions they index. To give an example, in Listing 9 the user needs to specify on lines 1-2 that the subscripts are smaller than the bounds of the data accessor.

Updating the buffer In SYCL, when a user updates an element of a buffer through a data accessor, the data accessor writes this update to the buffer. In the encoding, all updates are written to the data accessor's *buffer-field* instead. This means that changes made to the *buffer-field* need to be manually copied to the *buffer-array* in the *host code*. This needs to happen before another kernel attempts to read the buffer, otherwise it might read old values. This problem was solved by, for all data accessors with *read-write* access, assigning a data accessor's *buffer-field* to the *buffer-array* in the *host code* right after the kernel terminates, i.e. right after it is joined. At that point in time no further updates will be made to the *buffer-field*, and other kernels cannot access the buffer yet, as they need to wait till the kernel has finished terminating.

6 Encoding of SYCL's kernel execution order

As mentioned in Section 3.1, the queue schedules the execution of submitted kernels. In the encoding, the execution order is determined by the rules below, where data accessors for the same buffer are considered equivalent. The support for scheduling kernels is new in VerCors.

When a new kernel is submitted and it is...

- ... **independent** from the currently running kernels: it has either an independent set of data accessors or an intersection, where all data accessors in the intersection provide *read-only* access, it is immediately started.
- ... **dependent** on one or more of the currently running kernels: it has an intersection of data accessors, where one or more data accessors in the intersection provide *read-write* access, then the host halts to wait for all the running kernels with those data accessors to terminate (by joining their *event-classes*, as described in Section 3.2) before starting the new kernel.

Out-of-order scheduling The SYCL specification states that kernels are executed *in an out-of-order fashion based on dependency information*, but that kernels dependent of each other are executed in order of submission to the queue [18] (page 22-23). This implies that submitted kernels that are independent of each other could be re-ordered at run-time.

To give an example of a possible out-of-order execution, suppose we have three kernels CG_{ind} , CG_a and CG_b where CG_{ind} is independent of CG_a and CG_b and suppose CG_a and CG_b are already submitted to the queue in that order. If CG_{ind} is submitted, it might be that CG_b is waiting for CG_a if CG_b depends on CG_a . The scheduling of CG_{ind} can then still be before CG_b due to the independences of the kernels.

In the encoding the *host code* halts when a submitted kernel needs to wait, so any kernel submissions declared after it will not be submitted till it starts executing. This means that kernels are always executed *in-order* in the encoding. The encoding is sound because SYCL can only reorder kernels independent of each other, and their independence means that their results are the same regardless of their execution order.

Halting host code In SYCL, host code does not halt when a submitted kernel is waiting on other kernels to terminate before executing, which allows host code declared directly after a submission of a kernel CG_k to be executed before CG_k is started. In the example above, *arbitrary host code* can be executed before CG_a has finished executing and CG_b starts executing, because CG_b waits on CG_a to terminate. In the encoding this is not possible, as it halts till CG_b starts executing. The encoding is sound, because this does not influence the state of the heap, and thus the program's results. The only heap-data that kernels and host code share are buffers, but host code can neither read nor write to them or their **hostData**. A kernel could be submitted in the host code, which uses a buffer that a currently running kernel CG_k uses as well. However, that kernel is either independent of CG_k , in which case the ordering does not influence the results, or dependent on CG_k , in which case it has to wait on CG_k to finish, in both SYCL and the encoding. A buffer's **hostData** becomes accessible to the host code after the buffer has been destroyed. However, at that point all kernels writing to **hostData** have terminated, and any changes written to **hostData** by the host code do not affect the still running kernels with *read-only* access, because they read from a copy made at the point they were submitted.

Listing 10: Test program that reads from a part of `a` that is currently claimed by `aBuffer`. *\pointer(a, 10, write)* denotes that pointer `a` is not null, and that `a` to `a+9` are valid locations with read-write permission to them.

```

1 #include <sycl/sycl.hpp>
2 //@ requires \pointer(a, 10, write);
3 void test(bool* a) {
4     sycl::buffer<bool, 1> aBuffer =
5         ↪ sycl::buffer(a, sycl::range<1>(10));
6     bool x = a[5]; // Not allowed
7 }

```

Table 11: Number of tests per construct.

Construct	Tests
Kernels	15
Item methods ⁴	12
Buffers	16
Data accessors	23
Local accessors ⁴	12

Table 12: Profiling of the usability programs.

Phase	Time (s) for Program 1	Time (s) for Program 2
Parsing	80,1	129,7
Name Resolution	1,1	1,7
Transformation	13,5	21,3
Verification	39,8	154,9
Total	144,4	314,7

7 Evaluation

7.1 Correctness of the implementation

The implementation of the encoding of SYCL into COL was purely additive: code was added to VerCors’ source code [28] that performs the encoding of SYCL, but the already existing implementations of encodings of other languages and frameworks were not altered, nor how the resulting encodings are verified.

The encoding of SYCL into COL was developed in such a way that if the COL program is proven correct, then the SYCL program is proven correct. This was not formally proven. The implementation of the encoding was tested on a large number of examples: examples that are expected to verify and examples that are expected to fail. VerCors already had a framework for this, which takes programs and their expected results, which can be a successful verification or a certain error being thrown. An example of an integration test is the test program in Listing 10, of which the verification is expected to fail because the program reads a part of data claimed by a buffer. For every supported SYCL concept, it was tested whether the implementation matches the described encoding. Furthermore, integration tests were added to test the occurrence of specific error in specific cases. The number of tests per SYCL construct can be seen in Table 11.

All these tests pass and were run using the ScalaTest [4] test runner in IntelliJ [16]. Averaged over 5 runs⁵, executing all tests takes 9 minutes and 30 seconds in

⁴Not covered in this paper.

⁵Run in Fedora 38 on a laptop with an Intel Core i7-1165G7 CPU (4 cores, 2 threads per core, avg. 2.8GHz, max. 4.7 GHz) and 16 GB RAM (DDR4, max. 3200MT/s).

total. This paper is accompanied with an artifact to run and evaluate the SYCL examples, which can be found in [33].

7.2 Usability of the supported subset of SYCL

To show that the supported subset of SYCL is extensive enough to verify statements about SYCL programs performing full tasks, two annotated, real-life SYCL programs were created, which can be successfully verified by VerCors⁶.

Program 1 (Listing 13) has a method `vector_add` which performs vector addition: adding the elements of arrays `a` and `b` of size `n` and storing the results in array `c`. Vector addition was chosen because most SYCL teaching materials [10,12,15] use vector addition as an example program. This means that SYCL developers, most likely, know how vector addition is performed, and can now see how they could verify that SYCL program.

Program 2 is a more complex, real-life SYCL program. It has been developed as a typical program that SYCL developers would write. In addition, it showcases all the different supported SYCL features. The program contains a method which applies a function to each element of a two-dimensional matrix and then transposes the results. For the interested reader, the program is listed in Appendix A.2 in the related Master’s thesis [29].

VerCors occasionally fails to establish the pre-conditions of one of the kernels in the method, even though they are sound. Normally, non-deterministic results should be impossible for formal verifiers. However, for quantified expressions without triggers, which indicate when a quantification should be instantiated, Z3 uses heuristics and randomisation to instantiate them. Triggers are not included in the encoding of SYCL programs, because they need to be inserted in such a way that quantifications are instantiated in all necessary cases, which can be complicated to determine. Therefore, it is most likely that the incidental verification failures are caused by a failure to instantiate one of the triggerless quantified expressions in the encoding.

Profiling The profiling of the two programs for the different phases in VerCors can be seen in Table 12. Both programs were verified five times and an average was taken for each of VerCors’ phases^{5,7}. As can be seen in the table, verification of *Program 1* took 2 minutes and 24 seconds, and verification of *Program 2* took 5 minutes and 14 seconds. From the table, we read that a significant portion of the total time is spent in the parsing phase, so a large portion of the recorded time is spent on parsing rather than verifying the encoding of these programs. This is most likely caused by the fact that VerCors uses the C++ ANTLR grammar provided by ANTLR, into which VerCors-related grammar-rules were inserted. Some initial investigation has shown that the grammar was written inefficiently, but it requires further investigation to pinpoint the precise issue.

⁶The VerCors version used for running the examples is commit `acdf83` [28].

⁷The flag `-no-infer-heap-context-into-frame`, which disables an optimization step, needs to be enabled to verify these SYCL programs.

Listing 13: A SYCL program that performs vector addition of two arrays.

```

1  #include <sycl/sycl.hpp>
2
3  //@ context n >= 0;
4  //@ context \pointer(a, n, write);
5  //@ context \pointer(b, n, write);
6  //@ context \pointer(c, n, write);
7  //@ ensures (\forallall int i; 0 <= i && i < n; c[i] == a[i] + b[i]);
8  void vector_add(int n, int* a, int* b, int* c) {
9      sycl::queue q;
10     sycl::buffer<int,1> a_buf = sycl::buffer(a,sycl::range(n));
11     sycl::buffer<int,1> b_buf = sycl::buffer(b,sycl::range(n));
12     sycl::buffer<int,1> c_buf = sycl::buffer(c,sycl::range(n));
13
14     q.submit([&](sycl::handler& cgh) {
15         sycl::accessor<int, 1, sycl::access_mode::read> a_acc =
16             ↪ sycl::accessor(a_buf, cgh, sycl::read_only);
17         sycl::accessor<int, 1, sycl::access_mode::read> b_acc =
18             ↪ sycl::accessor(b_buf, cgh, sycl::read_only);
19         sycl::accessor<int, 1> c_acc =
20             ↪ sycl::accessor(c_buf, cgh, sycl::read_write);
21
22         cgh.parallel_for(sycl::range(n),
23             //@ context it.get_id(0) < a_acc.get_range().get(0);
24             //@ context Perm(a_acc[it.get_id(0)], read);
25             //@ context it.get_id(0) < b_acc.get_range().get(0);
26             //@ context Perm(b_acc[it.get_id(0)], read);
27             //@ context it.get_id(0) < c_acc.get_range().get(0);
28             //@ context Perm(c_acc[it.get_id(0)], write);
29             //@ ensures c_acc[it.get_id(0)] ==
30                 ↪ a_acc[it.get_id(0)] + b_acc[it.get_id(0)];
31             [=](sycl::item<1> it) {
32                 c_acc[it.get_id(0)] =
33                 ↪ a_acc[it.get_id(0)] + b_acc[it.get_id(0)];
34             } ); }); }

```

8 Related works

Our work is the first to offer formal verification for SYCL programs. In terms of informal verification, there do not seem to exist any SYCL-specific testing frameworks, but there does exist a performance benchmark tool called SYCL-Bench [19,20], and a runtime debugger based on GDB [1]. The support added to VerCors for SYCL is built upon on the encodings of GPGPU languages similar to SYCL into languages similar to COL, described by Blom et al. [7] and Darabi [13] for a subset of OpenMP, and described by Safari & Huisman [26] for a subset of CUDA. Inspiration was also taken from the approaches to verifying GPGPU kernels and their incorporation into VerCors described by Amighi et

al. [3], Blom et al. [8], and Darabi [13]. There also exist other formal verification tools for GPGPU programs, which might be able to extend their support to SYCL as well, such as GPUVerify by Betts et al. [5] for OpenCL and CUDA, PUG by Li & Gopalakrishnan [22] for CUDA, the tool by Xing et al. [30] for CUDA, and CIVL by Zheng et al. [32] for kernels from different languages.

9 Conclusion

This paper described how a core subset of SYCL constructs are encoded into VerCors’ internal language COL to support verification of SYCL programs. More precisely, the encodings of the following SYCL constructs were discussed and illustrated with examples: submissions of basic- and ND-range kernels, explicitly waiting on their termination, constructions and destructions of buffers, constructions of data accessors, and interactions with elements of their buffers. Furthermore, the encoding of SYCL’s kernel ordering and its soundness were discussed. To evaluate the added support, integration testing was used. The usability of the supported subset of SYCL was also shown. Vector addition, used as an example in most SYCL teaching materials, can be verified with VerCors. Function applications to a matrix and transposing the results using a combination of all of the supported SYCL features can be verified as well, may it be with the occasional false negatives due to the lack of triggers in the encoding. All in all, our work makes VerCors the first tool to formally verify (a subset of) SYCL programs.

10 Future work

To solve the issue of VerCors occasionally giving false negatives, we plan to investigate how to insert triggers in the right places in the encoding of SYCL programs. We would also like to add support to VerCors for more SYCL constructs that could be useful to developers. SYCL’s group barriers and memory fences can be used to control the reordering of memory loads and stores in a kernel and allow work-items to synchronise with each other. To add support for these constructs we could take inspiration from the encoding of their OpenCL and CUDA counterparts. SYCL has a virtual generic address space that overlaps its global, local, and private address spaces (which are already supported in VerCors). It can be used in cases where it is not known in advance, or irrelevant, in what address space data is stored. There exists no VerCors support for any similar constructs at the time of writing, making it also interesting to investigate. As always, we plan to do more complex case-studies to both test our SYCL support and find the limits of the current support. Last of all, we would like to include the often required user specifications about the bounds of the subscripts on data accessors in the encoding, such that the user will no longer need to specify them.

References

1. Aktemur, B., Metzger, M., Saiapova, N., Strasuns, M.: Debugging SYCL programs on heterogeneous Intel® architectures. In: Proceedings of the International Workshop on OpenCL. pp. 1–10 (2020)
2. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multi-threaded Java programs. *Logical methods in computer science* **11**(1), 2 (Feb 2015), doi=10.2168/LMCS-11(1:2)2015
3. Amighi, A., Darabi, S., Blom, S., Huisman, M.: Specification and verification of atomic operations in GPGPU programs. In: *Software Engineering and Formal Methods*. pp. 69–83. Springer International Publishing (2015)
4. Artima: ScalaTest support in the IntelliJ Scala plugin, https://www.scalatest.org/user_guide/using_scalatest_with_intellij, accessed on 15-04-2024
5. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems languages and Applications*. p. 113–132. OOPSLA ’12, Association for Computing Machinery (2012), doi=10.1145/2384616.2384625
6. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: *Integrated Formal Methods*. pp. 102–110. Springer International Publishing (2017), doi=10.1007/978-3-319-66845-1_7
7. Blom, S., Darabi, S., Huisman, M., Safari, M.: Correct program parallelisations. *International Journal on Software Tools for Technology Transfer* **23**(5), 741–763 (Oct 2021), doi=10.1007/s10009-020-00601-z
8. Blom, S., Huisman, M., Mihelčič, M.: Specification and verification of GPGPU programs. *Science of Computer Programming* **95**, 376–388 (2014), Special Section: ACM SAC-SVT 2013 + Bytecode 2013
9. Bornat, R., Calcagno, C., O’Hearn, P.W., Parkinson, M.: Permission accounting in separation logic. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. pp. 259–270 (2005), doi=10.1145/1040305.1040327
10. Codeplay Software Ltd: Introduction to SYCL, <https://tech.io/playgrounds/48226/introduction-to-sycl/hello-world>, accessed on 15-04-2024
11. Codeplay Software Ltd: SYCL Guide, <https://developer.codeplay.com/products/computecpp/ce/2.11.0/guides/sycl-guide>, accessed on 15-04-2024
12. Codeplay Software Ltd: SYCL Academy on GitHub (Apr 2024), <https://github.com/codeplaysoftware/syclacademy>
13. Darabi, S.: Verification of program parallelization. Ph.D. thesis, University of Twente (Mar 2018), iPA Dissertation Series No. 2018-02 IDS PhD. Thesis Series No. 18-458
14. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS, March 29–April 6, 2008. Proceedings 14*. pp. 337–340. Springer (2008)
15. Intel: Explore SYCL with Samples from Intel (Mar 2023), <https://www.intel.com/content/www/us/en/docs/oneapi/code-samples-dpcpp/2023-1/overview.html>
16. JetBrains s.r.o.: IntelliJ IDEA – the Leading Java and Kotlin IDE, <https://www.jetbrains.com/idea/>, accessed on 15-04-2024
17. Khronos® OpenCL Working Group: OpenCL™ Specification (Apr 2024), https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf

18. Khronos® SYCL™ Working Group: SYCL™ 2020 Specification (revision 7) (Apr 2023), <https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf>, compiled from <https://github.com/KhronosGroup/SYCL-Docs/tree/SYCL-2020/final-rev7>
19. Lal, S., Alpay, A., Salzmann, P., Cosenza, B., Hirsch, A., Stawinoga, N., Thoman, P., Fahringer, T., Heuveline, V.: SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing. In: Euro-Par 2020: 26th International European Conference on Parallel and Distributed Computing. Springer International Publishing (2020)
20. Lal, S., Alpay, A., Salzmann, P., Cosenza, B., Stawinoga, N., Thoman, P., Fahringer, T., Heuveline, V.: SYCL-Bench: A Versatile Single-Source Benchmark Suite for Heterogeneous Computing. In: Proceedings of the International Workshop on OpenCL. IWOCCL '20, Association for Computing Machinery (2020), doi=10.1145/3388333.3388669
21. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML reference manual, <http://www.jmlspecs.org/refman/jmlrefman.pdf>, DRAFT, Rev. 2344
22. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. pp. 187–196 (2010)
23. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Verification, Model Checking, and Abstract Interpretation. pp. 41–62. Springer Berlin Heidelberg (2016)
24. NVIDIA: CUDA Toolkit Documentation 12.1 (Apr 2024), <https://docs.nvidia.com/cuda/index.html>
25. Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., Tian, X.: Data Parallel C++. Apress (2021), doi=10.1007/978-1-4842-5574-2
26. Safari, M., Huisman, M.: Formal verification of parallel prefix sum and stream compaction algorithms in CUDA. Theoretical Computer Science **912**, 81–98 (2022), doi=10.1016/j.tcs.2022.02.027
27. Siegel, H.J., Antonio, J.K., Metzger, R.C., Tan, M., Li, Y.A.: Heterogeneous computing. ECE Technical Reports p. 206 (1994)
28. UTwente FMT Group: VerCors' source code on GitHub (Apr 2024), <https://github.com/utwente-fmt/vercors>
29. Wittingen, E.: Deductive verification for SYCL. Master's thesis (January 2024), <http://essay.utwente.nl/97976/>
30. Xing, Y., Huang, B., Gupta, A., Malik, S.: A formal instruction-level GPU model for scalable verification. In: 2018 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2018 - Digest of Technical Papers. Institute of Electrical and Electronics Engineers Inc. (Nov 2018), doi=10.1145/3240765.3240771
31. Zahran, M.: Heterogeneous computing: Here to stay: Hardware and software perspectives. Queue **14**(6), 31–42 (Dec 2016), doi=10.1145/3028687.3038873
32. Zheng, M., Rogers, M.S., Luo, Z., Dwyer, M.B., Siegel, S.F.: CIVL: Formal verification of parallel programs. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 830–835 (2015), doi=10.1109/ASE.2015.99
33. Şakar, Ö., Wittingen, E., Huisman, M.: Artifact for paper (Deductive verification of SYCL in VerCors) (2024), doi=10.4121/45d37292-cce5-4fb7-8d4e-a1b32cfa3028