

MASTER'S THESIS

# Deductive verification for SYCL

*Author:*  
Ellen Wittingen

*Supervisors:*  
Prof. Dr. M. Huisman  
Ö.F.O. Şakar, MSc.

*Exam committee:*  
Prof. Dr. M. Huisman  
Dr. ir. N. Alachiotis

*Date:*  
January, 2024

FORMAL METHODS AND TOOLS GROUP

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE

UNIVERSITY OF TWENTE.

## Abstract

A heterogeneous computing system is a system composed of different types of computing units. SYCL is a software development framework with which programs can be developed for such systems. It uses the concept of kernels, where a kernel executes code inside it in parallel, and different kernels can be executed concurrently on multiple computing units. These concurrent executions of kernels and their contents means that the set of possible program behaviours can be of considerable size, which makes it easy for developers to overlook problems which only occur in a few possible program behaviours. To solve this, formal verification can be performed to ensure a program's correctness. However, at the time of writing, there do not seem to exist any formal verification tools for SYCL programs.

The VerCors toolset, which enables users perform deductive verification on programs, supports reasoning about parallel and concurrent constructs. This makes VerCors suitable for verification of programs written for heterogeneous systems. Moreover, it aims to be language-independent, such that support for new languages can be added without redesigning the entire toolset.

In this thesis, it is investigated how the VerCors toolset could be extended to enable users to formally verify a subset of programs written in SYCL. This subset consists of programs that can contain two kinds of kernel definitions, data sharing between devices and the host, and declarations of data in the various address spaces of computing units. For each feature in this subset, this thesis describes its semantics and discusses in detail how these semantics are encoded into VerCors, and what considerations had to be made to facilitate the construction and soundness of these encodings. Finally, the efforts taken to ensure the soundness of the encoding and the usefulness of the supported SYCL subset are discussed.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Overview . . . . .	5
<b>2</b>	<b>Goal</b>	<b>6</b>
2.1	Research questions . . . . .	6
<b>3</b>	<b>Background</b>	<b>8</b>
3.1	Heterogeneous computing systems . . . . .	8
3.1.1	Definition . . . . .	8
3.1.2	Relevance . . . . .	8
3.1.3	Software development frameworks . . . . .	9
3.2	SYCL . . . . .	18
3.2.1	Application structure . . . . .	18
3.2.2	Execution model . . . . .	18
3.2.3	Memory model . . . . .	22
3.2.4	Synchronization . . . . .	24
3.2.5	Error handling . . . . .	25
3.3	Formal verification of heterogeneous programs . . . . .	26
3.3.1	Motivation . . . . .	26
3.3.2	Deductive verification . . . . .	26
3.3.3	Contracts . . . . .	27
3.3.4	Auxiliary annotations . . . . .	29
3.3.5	Separation logic . . . . .	31
3.4	VerCors . . . . .	37
3.4.1	Overview of the verification process . . . . .	37
3.4.2	Supported deductive verification concepts . . . . .	38
3.4.3	Reasoning about other constructs . . . . .	41
3.4.4	Supplementary clauses, operators, expressions, and types . . . . .	43
3.4.5	Ghost code . . . . .	45
3.4.6	Predicates . . . . .	48
<b>4</b>	<b>Related works</b>	<b>51</b>
4.1	SYCL verification tools . . . . .	51
4.2	Publications about VerCors . . . . .	51
4.2.1	Verification of heterogeneous programming constructs in VerCors . . . . .	51
4.2.2	Additions to VerCors' support for non-heterogeneous constructs. . . . .	52
4.3	Front-ends for Viper . . . . .	52
<b>5</b>	<b>Added support for basic C++ constructs</b>	<b>53</b>
5.1	Added C++ support to VerCors' initial phases . . . . .	53
5.2	Constructs with implemented C-equivalent constructs . . . . .	53
5.2.1	Logical expressions . . . . .	53
5.2.2	Arithmetic expressions . . . . .	53
5.2.3	Primitive types . . . . .	53
5.2.4	Arrays . . . . .	54
5.2.5	Pointers . . . . .	54
5.2.6	Variables . . . . .	54

5.2.7	Control structures . . . . .	54
5.2.8	Methods . . . . .	55
5.3	Constructs without implemented C-equivalent constructs . . . . .	55
5.3.1	Namespaces . . . . .	55
5.3.2	Class methods . . . . .	56
5.3.3	Class constructors . . . . .	57
5.3.4	Lambda methods . . . . .	57
<b>6</b>	<b>Added support for SYCL's basic and ND-range kernels</b>	<b>58</b>
6.1	Declaration and execution of kernels . . . . .	58
6.1.1	Definitions of supported methods . . . . .	58
6.1.2	Semantics . . . . .	59
6.1.3	Encoding into VerCors . . . . .	60
6.2	Synchronizing with the host . . . . .	70
6.2.1	Definitions of supported methods . . . . .	70
6.2.2	Semantics . . . . .	70
6.2.3	Encoding into VerCors . . . . .	70
6.3	Querying a work-item's id and index-space . . . . .	71
6.3.1	Definitions of supported methods . . . . .	71
6.3.2	Semantics . . . . .	72
6.3.3	Encoding into VerCors . . . . .	72
<b>7</b>	<b>Added support for SYCL's data sharing between devices and the host</b>	<b>76</b>
7.1	Buffers . . . . .	76
7.1.1	Definition of the supported constructor . . . . .	76
7.1.2	Semantics . . . . .	76
7.1.3	Encoding into VerCors . . . . .	78
7.2	Data accessor declarations . . . . .	82
7.2.1	Definitions of supported constructor and access modes . . . . .	82
7.2.2	Semantics . . . . .	82
7.2.3	Encoding into VerCors . . . . .	83
7.3	Accessing elements of a buffer through a data accessor . . . . .	88
7.3.1	Definitions of supported methods . . . . .	88
7.3.2	Semantics . . . . .	88
7.3.3	Encoding into VerCors . . . . .	89
7.4	Updating a buffer's contents . . . . .	90
7.4.1	Semantics . . . . .	90
7.4.2	Encoding into VerCors . . . . .	90
<b>8</b>	<b>Added support for SYCL's address spaces</b>	<b>91</b>
8.1	Global and private address spaces . . . . .	91
8.2	Local address space . . . . .	91
8.2.1	Definition of supported constructor . . . . .	91
8.2.2	Semantics . . . . .	92
8.2.3	Encoding into VerCors . . . . .	93
<b>9</b>	<b>Validation of added support</b>	<b>95</b>
9.1	Formal proof . . . . .	95
9.2	Integration testing . . . . .	95
9.2.1	Overview of tests . . . . .	95
9.2.2	Examples . . . . .	96
9.2.3	Testing speed . . . . .	98
9.3	Adhering to SYCL's behaviour . . . . .	98
9.4	Full SYCL programs . . . . .	99

<b>10 Conclusion</b>	<b>100</b>
10.1 Future work . . . . .	101
10.1.1 Extending SYCL support . . . . .	101
10.1.2 Extending C++ support . . . . .	101
<b>Bibliography</b>	<b>102</b>
<b>Glossary</b>	<b>107</b>
<b>Appendix A Examples of VerCors-verifiable SYCL programs</b>	<b>109</b>
A.1 Vector addition . . . . .	109
A.2 Matrix mapping and transposition . . . . .	110
<b>Appendix B Overview of all assumptions and limitations</b>	<b>112</b>
<b>Appendix C Overview of errors that can be thrown</b>	<b>114</b>
<b>Appendix D Proofs of linearization methods</b>	<b>115</b>
D.1 Linearization of two ids . . . . .	115
D.1.1 Assumption 1: Minimum value . . . . .	115
D.1.2 Assumption 2: Maximum value . . . . .	115
D.1.3 Assumption 3: Injectivity . . . . .	116
D.2 Linearization of three ids . . . . .	117
D.2.1 Assumption 1: Minimum value . . . . .	117
D.2.2 Assumption 2: Maximum value . . . . .	117
D.2.3 Assumption 3: Injectivity . . . . .	118

# 1 Introduction

A heterogeneous computing system is a system composed of different types of computing units [58, 66, 79]. Most machines, such as desktop computers, laptops, tablets, and smartphones are heterogeneous systems [79], and so are some of the most powerful commercially available computer systems around the world [73]. There exist several software development frameworks, such as OpenCL, OpenMP, CUDA, HIP, and SYCL, with which software can be developed that takes advantage of the heterogeneity of a system. The thesis will focus on SYCL, which is a C++ programming model that enables different types of computing units to be used in a single application [71, 72]. SYCL uses the concept of kernels, where any code declared in a kernel can be executed in parallel on a chosen computing unit. For these kernels it distinguishes multiple address spaces, allows synchronization through memory fences and group barriers, and allows declaring what data should be available when a kernel is executed.

Programs written for heterogeneous systems are often concurrent. For example, many instances of a kernel can be executed in parallel. One problem with concurrent software is that it is notoriously error-prone. The set of possible program behaviours increases exponentially with the size of the parallel parts of a program and the number of processes that execute it. With many possible program behaviours, it is easy for developers to overlook problems which only occur in a few possible program behaviours. To ensure a concurrent program’s correctness, formal verification can be performed. [12]

Deductive verification is a subset of formal verification. In deductive verification a system’s desired properties are specified using logical formulas. These specifications are then proven using deduction in a logic calculus. [31] There are various tools which can automatically perform deductive verification, such as Dafny [46], OpenJML [24], KeY [1], VCC [65], and VeriFast [41], and VerCors. VerCors uses static analysis to prove partial correctness of programs. It allows for reasoning about parallel and concurrent constructs, which makes VerCors suitable for verification of programs written for heterogeneous systems. VerCors uses permission-based separation logic (PBSL), which allows for reasoning about what data on a heap can be accessed when and by which processes, by declaring for each block of code what access permissions they have for the heap [77].

At the time of writing, the aforementioned deductive verification tools cannot verify SYCL programs. Moreover, there do not seem to exist any formal verification tools for SYCL programs. VerCors, however, supports reasoning about parallel and concurrent constructs, and supports partial verification of subsets of OpenCL, CUDA, and OpenMP. This makes VerCors suitable for verification of programs written for heterogeneous systems. Moreover, it aims to be language-independent, such that support for new languages can be added without redesigning the entire toolset. This means that the VerCors toolset could be extended to enable users to formally verify programs written in SYCL.

The main goal of this thesis is to investigate how the VerCors toolset could be extended to enable users to be able to formally verify a subset of SYCL programs using the VerCors toolset. This *subset of programs* consists of programs containing basic kernels and ND-range kernels, data sharing between devices and the host, and data declarations on devices in their various address spaces.

This goal is achieved by answering the following research questions in this thesis:

- RQ 1** How can reasoning about basic C++ constructs be supported in VerCors?
- RQ 2** How can reasoning about SYCL’s basic and ND-range kernels be supported in VerCors?
- RQ 3** How can reasoning about SYCL’s data sharing between devices and the host be supported in VerCors?
- RQ 4** How can reasoning about SYCL’s address spaces for devices be supported in VerCors?

## 1.1 Overview

This thesis is built up as follows:

First, in chapter 2, the main goal of this thesis explained in further detail and the choices and scopes of the research questions are discussed. Then, in chapter 3 the background information upon which this thesis is built shared. This background information covers heterogeneous computing systems, SYCL, formal verification of heterogeneous programs and VerCors. In Chapter 4, works related to this thesis are discussed.

After this groundwork, the research questions are investigated. In chapter 5, RQ1 is investigated by first examining to what extent VerCors' C implementation can be copied to the C++ implementation to add support for the required C++ constructs. Then, for C++ constructs without C-equivalent constructs it is explored how they can be implemented from scratch. In chapters 6-8, RQ2 - RQ4 are investigated. For each research question, it is first shown for what SYCL constructs support should be added to VerCors to answer the research question. The semantics of those constructs, such as their syntax and behaviour, are explored after that. Then, the research question is answered by sharing how those SYCL constructs and their semantics are encoded into VerCors and what considerations had to be made to facilitate the construction and soundness of these encodings.

Finally, the efforts taken to ensure validity of the added support to reason about SYCL programs in VerCors are described and the usefulness of the supported subset of SYCL constructs is discussed.

## 2 Goal

The main goal of this thesis is:

*To be able to deductively verify a subset of SYCL programs using the VerCors toolset.*

This *subset of SYCL programs* consists of programs containing basic- and ND-range kernels definitions, data sharing between devices and the host, and data declarations on devices in their various address spaces. This goal is refined in the next section by explaining what research questions are examined in this thesis.

### 2.1 Research questions

In this section, the research questions that are examined in this thesis, in order to achieve the aforementioned goal, are discussed. For each research question, its scope and the reasons why it was chosen are described. Because SYCL has many features and multiple implementations, support was not added to VerCors for all SYCL features. However, for these research questions, it can be assumed that when support for reasoning about a SYCL feature is described, all of VerCors' verification concepts described in section 3.4 should be able to be used when reasoning about it.

#### RQ 1 How can reasoning about basic C++ constructs be supported in VerCors?

As mentioned in subsection 3.1.3.7, SYCL is a library built on top of C++, so all its features are expressed using C++ syntax. Therefore, basic C++ constructs should be supported in VerCors to achieve the main goal of the thesis. This research question covers the ability to reason about the C++ constructs mentioned below. In this list, the constructs highlighted in blue are essential, as the SYCL features mentioned in the goal cannot be expressed without them. The remaining C++ constructs were added to this research question to allow users to perform basic tasks (which can also be put inside SYCL kernels), such as simple mathematical calculations and executing code in a loop.

- The following primitive types: `bool`, `int`, `long`, `double`, `float`, `char`, and `void`.
- Pointers (which can denote arrays) containing values of the aforementioned primitive types.
- Statements: variable declarations and assignments, method calls.
- Declarations of (lambda) methods, namespaces, and classes.
- Logical expressions `!a`, `a && b`, `a || b`, `a == b`, `a != b`, `a < b`, `a <= b`, `a > b`, `a >= b`.
- Arithmetic expressions `a + b`, `a - b`, `a * b`, `a / b`, `a % b`.
- Control structures: if statements, while loops and for loops.

#### RQ 2 How can reasoning about SYCL's basic and ND-range kernels be supported in VerCors?

As mentioned in subsection 3.2.1, only code defined in kernels can be executed on a device. This makes kernels a critical feature for heterogeneous computing, as without kernels, code cannot be executed on the devices in a heterogeneous system. This research question looks at these two kinds of kernels: basic kernels and ND-range kernels (see subsection 3.2.2.2). These two kinds were chosen as they are the only ones that the specification describes in detail and that should be supported in every SYCL implementation. These two kinds have as an added bonus that they are device-independent: they have the same semantics on any device that supports them.



### **RQ 3 How can reasoning about SYCL’s data sharing between devices and the host be supported in VerCors?**

Being able to share data between devices and the host in a heterogeneous system is another important feature for heterogeneous computing. It is important because it allows results of tasks executed on devices to be communicated to the host, which in turn enables those results to be processed further on the host or another device.

As discussed in subsection 3.2.3.1, the SYCL specification describes two ways to share data: using a combination of buffers and data accessors, or using unified shared memory (USM). In the SYCL teaching materials by Codeplay Software Ltd [20, 21, 22], who created the ComputeCPP SYCL implementation [19], USM is not covered or described after buffers and data accessors. In the SYCL example programs from Intel [36], who created the DPC++ implementation of SYCL as part of their oneAPI product [40], buffers and data accessors are used more than USM. This seems to indicate that buffer and data accessors are considered more accessible to new SYCL developers than USM. To make deductive verification accessible to as many SYCL developers as possible, it was chosen to examine how to add support and reason about buffers and data accessors in this research question.

### **RQ 4 How can reasoning about SYCL’s address spaces for devices be supported in VerCors?**

As described in subsection 3.2.3.2, there are four types of address spaces for data on devices: the global address space, the local address space, the private address space, and the generic address space. These address spaces are useful as they allow data to be stored on a device with control over what work-items have access to it. For ND-range kernels, it is particularly useful, because the local address space allows data to only be visible to work-items in the same work-group. The generic address space is quite different from the three others, as it is a virtual space that overlaps all of them.

The generic address space is not covered in the SYCL teaching materials by Codeplay Software Ltd and Intel [20, 21, 22, 36], whereas the other three address spaces are. This seems to indicate that the other three address spaces are considered more accessible to new SYCL developers than the generic address space. To make deductive verification accessible to as many SYCL developers as possible, it was chosen to examine how to add support and reason about the global, local, and private address spaces in this research question.

## 3 Background

In this section, the background information required to understand and investigate the research questions are discussed. First, heterogeneous computing systems and software development frameworks, with which software can be developed that takes advantage of the heterogeneity of a system, are explained to show SYCL's place in the field of heterogeneous computing. After this, an overview of SYCL will be given. Then, deductive verification, on which VerCors' proving methods are based, are discussed. Last of all, an overview of VerCors' verification process is given and the VerCors features that are relevant to this thesis are explained.

### 3.1 Heterogeneous computing systems

In this section, the definition and relevance of heterogeneous computing are described. After that several frameworks for the development of software for heterogeneous systems are discussed.

#### 3.1.1 Definition

A heterogeneous computing system is a computing system composed of different types of computing units. In this definition, two computing units are considered different when their capabilities, and/or execution models are different. [58, 66, 79]

An example of a heterogeneous computing system is a system containing both a CPU and a GPU (Graphical Processing Unit), because they have different capabilities. CPU cores run at high frequencies and have large caches to minimize latency of single threads, whereas GPUs contain more cores than CPUs, but they run on a lower frequency and have smaller caches. [47] Other examples of computing units that can be included in heterogeneous systems are: FPGAs (Field-Programmable Gate Arrays), DSPs (Digital Signal Processors), ASICs (Application-Specific Integrated Circuits), and AI chips [58].

Examples of different execution models are vector, single instruction multiple data (SMID), multiple instruction multiple data (MIMD), dataflow, and special-purpose models [66]. For instance, a GPU uses the SMID model [79].

In some cases, a system containing only a single CPU can be considered a heterogeneous system as well [79]. An example of such a system is a system containing a 12th Generation Intel Core Processor. It contains two types of processor cores in its CPU with different capabilities on the same die: performance and efficiency cores. Performance cores are physically larger high-performance processor cores designed for speed, and optimized for low-latency single-threaded performance and AI workloads. Efficiency cores are physically smaller than performance cores and are optimized for scalable multi-threaded performance and efficient offload of background tasks.[38, 39]

#### 3.1.2 Relevance

Heterogeneous systems are widely in use. Most machines, such as desktop computers, laptops, tablets, and smartphones are heterogeneous systems, usually containing both a CPU and a GPU [79]. Heterogeneous systems are also used in the world of supercomputers. In November 2022, at least 36% of the 500 most powerful commercially available computer systems around the world were heterogeneous, containing co-processors or accelerators [73].

### 3.1.3 Software development frameworks

There are several development frameworks with which software can be developed that takes advantage of the heterogeneity of a system. These frameworks include: OpenMP, OpenCL, CUDA, HIP, and SYCL. There exist more frameworks than these, but the ones just mentioned are the most well-known and/or support different kinds of devices or at least devices from multiple manufacturers, making them applicable to a wide variety of heterogeneous systems.

First, an overview will be given of heterogeneous programming concepts that most of the aforementioned frameworks use. After this, the frameworks themselves are discussed and for every framework it is shown how an example task could be performed in that framework's language.

#### 3.1.3.1 Concepts

Most of the aforementioned software development frameworks distinguish similar resources and have similar grouping methods for the work-items of a kernel.

**Resources** The following resources are often distinguished in the aforementioned frameworks:

- *Platform*: can be seen as a device vendor's runtime and the devices accessible through it.
- *Context*: contains a collection of devices that can be used by the host, and manages memory objects that can be shared between devices.
- *Host*: can be seen as the main computing unit. On this computing unit the source code which manages what source code is run on what computing unit is executed.
- *Device*: represents a computing unit on which kernels can be executed. Devices are provided by a platform.
- *Kernel*: a method or some collection of statements that will be executed in parallel on a device.

In Figure 3.1 an overview is given of how the resources relate to each other.

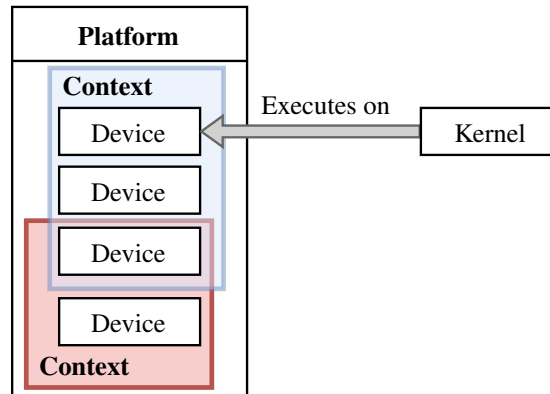


Figure 3.1: Overview of the relations between the resources usually present in the aforementioned frameworks.

**Grouping of work-items** In most of the aforementioned frameworks, the code inside a kernel can be executed many times in parallel. Every instance of the parallelly executed code inside the kernel is called a work-item. Every work-item has its own *work-item id* to distinguish it from the others. Often, work-items can be organised into *work-groups*. An example of this can be seen in Figure 3.2. Each work-group gets an unique *work-group id*, and each work-item is assigned a *local id*, which is unique inside its work-group. Each work-item still has a global id as well. This means that each work-item can be uniquely identified by either its global id or by a combination of its local id and work-group id. Work-groups can be subdivided into *sub-groups* as well.

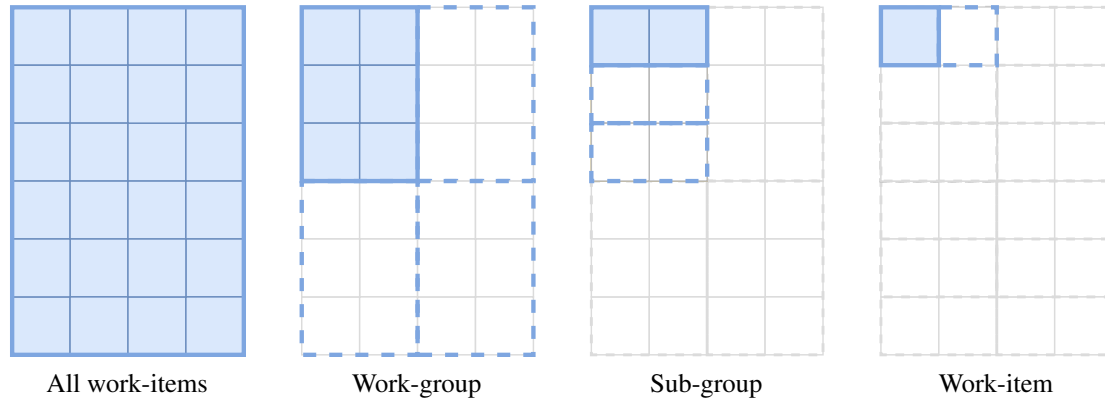


Figure 3.2: A two-dimensional range divided into work-groups, sub-groups, and work-items. *The squares with a blue border around them represent a single instance of the range, group, or item listed in the text below each block.*

### 3.1.3.2 Example task

To show how each framework can be used, the task of zipping two arrays using addition is used as the example task. A visualization of this task can be seen in Figure 3.3. This zipping is done on arrays that contain 10 elements, by 10 work-items in the same work-group, on the first device that can be found. Work-item *i* adds `a_array[i]` and `b_array[i]` and stores the result in `result_array[i]` directly, or stores the result in the device's memory, which is then copied to `result_array` when all work-items are done.

```

a_array      [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
+
b_array      [ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]
=
result_array [ 10, 10, 10, 10, 10, 10, 10, 10, 10, 10 ]

```

Figure 3.3: Zipping of arrays `a_array` and `b_array` to the array `result_array` using addition.

This example task is used to show in the next subsections how tasks are defined in each framework. For comparability between the frameworks, the code implementing this task is written in the C++ version of each framework. Code that handles exceptional states has been omitted in the example implementations.

### 3.1.3.3 OpenMP

OpenMP is a specification set of compiler directives, library routines, and environment variables. It allows developers to create, manage, debug and analyse parallel programs by extending the C, C++, and Fortran languages. OpenMP is maintained by the OpenMP ARB, which are a group of computer hardware and software vendors. Originally, only CPU-based systems were supported, but now OpenMP can also be used to create software for various accelerators and DSPs. Numerous vendors, such as AMD, NVIDIA, and Intel have created compilers for OpenMP such that OpenMP source code can be compiled to be executed on their devices. [53, 54]

In Listing 3.4 the kernel and host code written in OpenMP for the zip addition task from subsection 3.1.3.2 can be seen.

```
1  #include <iostream>
2  #include <omp.h>
3
4  int main() {
5
6      // Create the arrays
7      int a_array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
8      int b_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
9      int result_array[10];
10
11     // Execute a kernel with 10 work-items on a device
12     // Copies a_array and b_array to the device's memory,
13     // and copies the results from device memory to result_array
14     #pragma omp target device(0) map(from: result_array[0:10])
15         map(to: a_array[0:10], b_array[0:10])
16     #pragma omp parallel num_threads(10)
17     {
18         // The kernel
19         int tid = omp_get_thread_num();
20         result_array[tid] = a_array[tid] + b_array[tid];
21     }
22
23     // Print the results
24     for (int i = 0; i < 10; i++) {
25         std::cout << result_array[i];
26     }
27
28     return 0;
29 }
```

Listing 3.4: OpenMP source code for the zip addition task from subsection 3.1.3.2. *Inspired by [30].*

### 3.1.3.4 OpenCL

OpenCL is a framework for cross-platform and parallel programming of diverse computing units. It is maintained by the Kronos Group. Many vendors (such as Intel, NVIDIA, and AMD [69]) of devices such as CPUs, GPUs, DSPs, and FPGAs have developed OpenCL drivers to allow OpenCL to interface with their devices. OpenCL consists of a programming framework and a runtime. Its programming framework is low-level. Developers have direct and explicit control over when and on what device which parts of their software are executed, how the memory is allocated and how the host and devices synchronize their operations to ensure correctness. The most commonly used version of OpenCL is called *OpenCL C*, which is based on C99. However, there also exist bindings for C++. [70] In OpenCL the kernel is specified separately from the host code, often in a separate file or as a string in the host code.

In Listing 3.5 the kernel code, and in Listing 3.6 the host code written in OpenCL for the zip addition task from subsection 3.1.3.2 can be seen. On line 36 of the host code, "kernel.cl" refers to file in which the kernel code shown in Listing 3.5 is stored.

```
1 kernel void add(global int* a, global int* b, global int* result) {
2     int tid = get_global_id(0);
3     result[tid] = a[tid] + b[tid];
4 }
```

Listing 3.5: OpenCL source code for the kernel for the zip addition task from subsection 3.1.3.2. Inspired by [68, 74].

```
1 #include <fstream>
2 #include <iostream>
3 #include <CL/opencl.hpp>
4
5 #define CL_HPP_ENABLE_PROGRAM_CONSTRUCTION_FROM_ARRAY_COMPATIBILITY;
6
7 int main(void) {
8     // Get the first available platform
9     cl::Platform platform = cl::Platform::get();
10
11     // Get the first available device
12     std::vector<cl::Device> all_devices;
13     platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices);
14     cl::Device device = all_devices[0];
15
16     // Get the context for the device
17     cl::Context context({device});
18
19     // Create buffers for the device context
20     cl::Buffer a_buffer(context, CL_MEM_READ_ONLY, sizeof(int) * 10);
21     cl::Buffer b_buffer(context, CL_MEM_READ_ONLY, sizeof(int) * 10);
22     cl::Buffer result_buffer(context, CL_MEM_WRITE_ONLY, sizeof(int) * 10);
23
24     // Create the arrays
25     int a_array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
26     int b_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
27     int result_array[10];
28
29     // Create a queue to schedule kernels on the device
30     cl::CommandQueue queue(context, device);
31
32     // Copy values of a_array and b_array to the device's memory
33     queue.enqueueWriteBuffer(a_buffer, CL_TRUE, 0, sizeof(int) * 10, a_array);
34     queue.enqueueWriteBuffer(b_buffer, CL_TRUE, 0, sizeof(int) * 10, b_array);
35
36 }
```

```

37 // Read the kernel source code from a file
38 std::ifstream sourceFile("kernel.cl");
39 std::string sourceCode((std::istreambuf_iterator<char> (sourceFile)),
    ↪ std::istreambuf_iterator<char>());
40 cl::Program::Sources sources;
41 sources.push_back({ sourceCode.c_str(), sourceCode.length() });
42
43 // Create a program object from the source code of the kernel
44 cl::Program program = cl::Program(context, sources);
45
46 // Build the program for the current device
47 program.build({ device });
48
49 // Create a kernel object
50 cl::compatibility::make_kernel<cl::Buffer, cl::Buffer, cl::Buffer> add(
    ↪ cl::Kernel(program, "add"));
51
52 // Submit the kernel to the queue
53 cl::NDRange global(10);
54 add(cl::EnqueueArgs(queue, global), a_buffer, b_buffer, result_buffer).wait();
55
56 // Put the results in result_array
57 queue.enqueueReadBuffer(result_buffer, CL_TRUE, 0, sizeof(int) * 10,
    ↪ result_array);
58
59 // Print the results
60 for (int i = 0; i < 10; i++) {
61     std::cout << result_array[i];
62 }
63
64 return 0;
65 }

```

Listing 3.6: OpenCL source code for the host for the zip addition task from subsection 3.1.3.2. Inspired by [68, 74].

### 3.1.3.5 CUDA

The CUDA Toolkit provides a development environment to create GPU-accelerated applications [50]. The toolkit includes GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler, and a runtime library to deploy applications. CUDA is maintained by NVIDIA. In contrast to OpenCL, which is supported on many different devices, CUDA is only supported on GPUs created by NVIDIA [51].

In Listing 3.7 the kernel and host code written in CUDA for the zip addition task from subsection 3.1.3.2 can be seen.

```
1  #include <iostream>
2  #include <cuda.h>
3
4  // The kernel
5  __global__ void add(int* a, int* b, int* result) {
6      int tid = threadIdx.x;
7      result[tid] = a[tid] + b[tid];
8  }
9
10 int main() {
11     // Create the arrays
12     int a_array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
13     int b_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
14     int result_array[10];
15
16     // Create buffers for the device to use
17     int* a_buffer;
18     int* b_buffer;
19     int* result_buffer;
20     cudaMalloc(&a_buffer, sizeof(int) * 10);
21     cudaMalloc(&b_buffer, sizeof(int) * 10);
22     cudaMalloc(&result_buffer, sizeof(int) * 10);
23
24     // Copy values of arrays a and b to the device
25     cudaMemcpy(a_buffer, a_array, sizeof(int) * 10, cudaMemcpyHostToDevice);
26     cudaMemcpy(b_buffer, b_array, sizeof(int) * 10, cudaMemcpyHostToDevice);
27
28     // Execute the kernel with 10 work-items
29     add<<<1, 10>>>(a_buffer, b_buffer, result_buffer);
30
31     // Put the results in result_array
32     cudaMemcpy(result_array, result_buffer, sizeof(int) * 10,
33         ↪ cudaMemcpyDeviceToHost);
34
35     // Print the results
36     for (int i = 0; i < 10; i++) {
37         std::cout << result_array[i];
38     }
39     return 0;
40 }
```

Listing 3.7: CUDA source code for the zip addition task from subsection 3.1.3.2. *Inspired by [49].*



### 3.1.3.6 HIP

The Heterogeneous Interface for Portability (HIP) is a C++ runtime API and kernel language to create portable software for AMD and NVIDIA GPUs. It is maintained by AMD. Developers can either write code directly in HIP's C++ language extension or automatically convert their CUDA source code to HIP. The developers of HIP claim that the HIP API is less verbose than OpenCL and more familiar to CUDA developers. They also claim that porting applications from CUDA to HIP is easier than porting from CUDA to OpenCL. However, it is made clear that HIP is not a drop-in replacement for CUDA, and platform-specific tuning and optimization needs to be done when porting from CUDA to HIP. [3, 5]

In Listing 3.8 the kernel and host code written in HIP for the zip addition task from subsection 3.1.3.2 can be seen. It can be observed here that the source code for HIP is similar to the code for CUDA.

```
1  #include <iostream>
2  #include <hip/hip_runtime.h>
3
4
5  // The kernel
6  __global__ void add(int* a, int* b, int* result) {
7      int tid = hipThreadIdx_x;
8      result[tid] = a[tid] + b[tid];
9  }
10
11 int main() {
12     // Create the arrays
13     int a_array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
14     int b_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
15     int result_array[];
16
17     // Create buffers for the device to use
18     int* a_buffer;
19     int* b_buffer;
20     int* result_buffer;
21     hipMalloc(&a_buffer, sizeof(int) * 10);
22     hipMalloc(&b_buffer, sizeof(int) * 10);
23     hipMalloc(&result_buffer, sizeof(int) * 10);
24
25     // Copy values of arrays a and b to the device
26     hipMemcpy(a_buffer, a_array, sizeof(int) * 10, hipMemcpyHostToDevice);
27     hipMemcpy(b_buffer, b_array, sizeof(int) * 10, hipMemcpyHostToDevice);
28
29     // Execute the kernel with 10 work-items
30     hipLaunchKernelGGL(add, 1, 10, 0, 0, a_buffer, b_buffer, result_buffer)
31
32     // Put the results in result_array
33     hipMemcpy(result_array, result_buffer, sizeof(int) * 10,
34               ↪ hipMemcpyDeviceToHost);
35
36     // Print the results
37     for (int i = 0; i < 10; i++) {
38         std::cout << result_array[i];
39     }
40
41     return 0;
42 }
```

Listing 3.8: HIP source code for the zip addition task from subsection 3.1.3.2. *Inspired by [4, 6].*

### 3.1.3.7 SYCL

SYCL (pronounced *sickle*) is a C++ programming model for heterogeneous computing [72]. SYCL is maintained by the Kronos Group. The main goal of SYCL is to enable different heterogeneous devices to be used in a single application. It does this by providing an extension to C++, APIs, and an ecosystem [71]. SYCL builds on top of the underlying concepts, portability and efficiency of other standards such as OpenCL and CUDA by using them as backends. SYCL's compilation process makes use of one or more of such backends to provide optimized instructions for the devices the program will execute on. An overview of this process can be seen in Figure 3.9, where a developer's source code is split up into host and device code, and the device code is compiled by the SYCL compiler using one of the backends listed at the bottom of the picture. Because of the use of other standards as backends, SYCL can support many different computing units, depending on the backends chosen to be used. The Khronos Group does not provide an implementation of SYCL, only its programming model. However, there exist several implementations made by other parties. The most notable implementations are DPC++, which is part of Intel's oneAPI [40] project, and ComputeCPP [19] by Codeplay Software Ltd<sup>1</sup>.

SYCL allows host and device code to be written in one program using ordinary C++ syntax, and includes libraries that can perform many simple tasks. This reduces the learning curve for developers new to the concept of heterogeneous programming, as they can focus on the heterogeneous concepts without having to learn a new syntax or implement simple tasks. This makes SYCL stand out from OpenCL and CUDA, which separate device code from host code and require the user to implement more tasks themselves to accomplish the same goals. [72] (pages 15-16)

In Listing 3.10 on the next page, the kernel and host code written in SYCL for the zip addition task from subsection 3.1.3.2 can be seen.

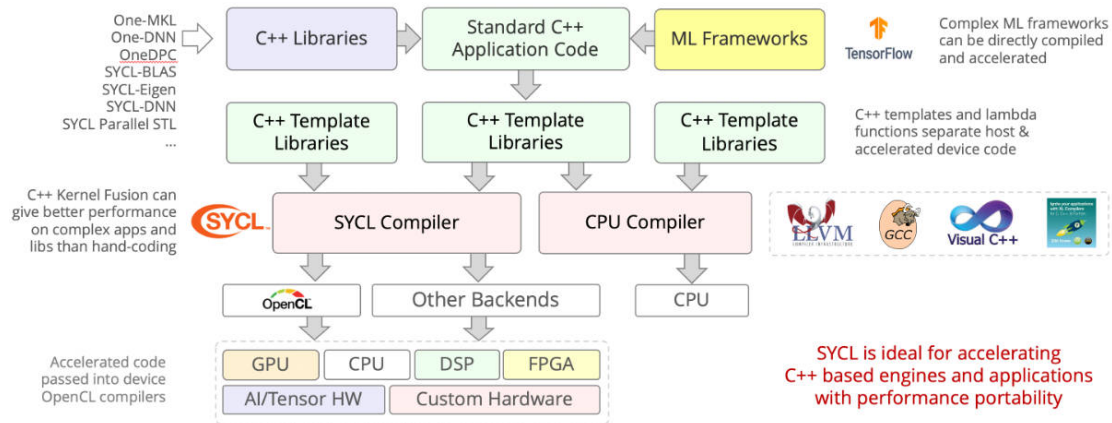


Figure 3.9: Overview of SYCL's compilation process. *Image taken from [71].*

<sup>1</sup>ComputeCPP deprecated during the writing of this thesis, on September 1st 2023. The developers are working on integrating its key features into Intel's DPC++ compiler. [23]

```

1  #include <iostream>
2  #include <sycl/sycl.hpp>
3
4  int main(void) {
5      // Create the arrays
6      int a_array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
7      int b_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
8      int result_array[10];
9
10     // Create a queue to schedule kernels on the default device
11     sycl::queue queue;
12
13     {
14         // Create buffers for the device to use
15         sycl::buffer<int, 1> a_buffer(a_array, sycl::range<1>(10));
16         sycl::buffer<int, 1> b_buffer(b_array, sycl::range<1>(10));
17         sycl::buffer<int, 1> result_buffer(result_array, sycl::range<1>(10));
18
19         // Submit the kernel to the queue
20         queue.submit([&] (sycl::handler& cgh) {
21             // Accessors
22             sycl::accessor a_accessor = { a_buffer, cgh, sycl::read_only };
23             sycl::accessor b_accessor = { b_buffer, cgh, sycl::read_only };
24             sycl::accessor result_accessor = { result_buffer, cgh, sycl::write_only };
25
26             cgh.parallel_for(10, [=] (sycl::item<1> tid) {
27                 // The kernel
28                 result_accessor[tid] = a_accessor[tid] + b_accessor[tid];
29             });
30         });
31     }
32
33     // Print the results
34     for (int i = 0; i < 10; i++) {
35         std::cout << result_array[i];
36     }
37
38     return 0;
39 }

```

Listing 3.10: SYCL source code for the zip addition task from subsection 3.1.3.2. *Inspired by [72].*

## 3.2 SYCL

In this section, an overview is given of SYCL, discussing its application structure, execution model, memory model, and its synchronization and error handling. This section is based on information presented in SYCL's specification [72] and the book *Data Parallel C++* [58].

SYCL's specification makes use of OpenCL's terminology. Therefore, OpenCL's terminology is used in this section and the rest of this thesis. In the glossary of this report a translation from the OpenCL terms used in this thesis to CUDA terminology can be found.

### 3.2.1 Application structure

A SYCL application consists of three scopes: the *application scope*, *command group scope*, and *kernel scope*. An overview of a SYCL's application structure, with each scope listed, can be seen in Figure 3.11. The application scope contains code that is meant to be executed on the host. It also contains command group declarations, which have their own command group scope. Code in the command group scope is also executed on the host. Kernel functions can be declared in the command group scope. The command group scope can contain various SYCL and C++ statements as well. A kernel function is a scoped block of statements (referred to by SYCL's specification as the *kernel scope*) that is compiled using a device compiler and is executed on a device. Because the code in the kernel scope will run on a device, it has to adhere to restrictions on what operations can be performed on the device.

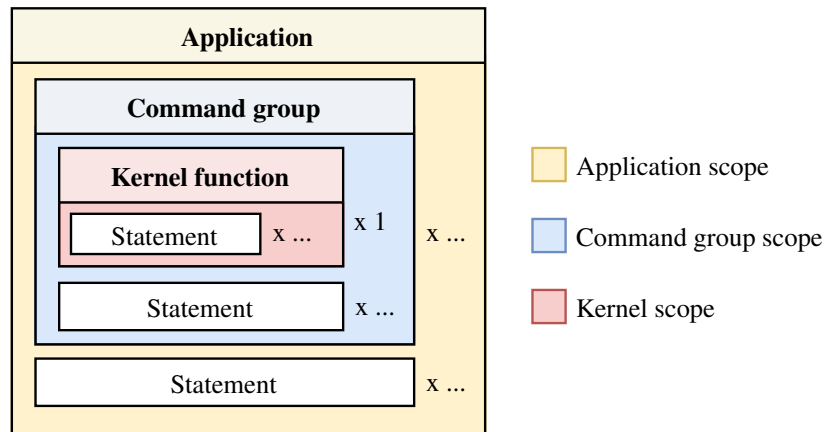


Figure 3.11: Overview of SYCL's application structure.

### 3.2.2 Execution model

SYCL has two execution models: the application execution model and the kernel execution model.

#### 3.2.2.1 Application execution model

The application execution model governs code in the application- and command group scopes. Command groups are submitted for execution through a *queue* object, which specifies the device the kernel will be executed on. This action of submitting a command group for execution is also called "submitting a kernel to a device". Command groups can be submitted to multiple queues.

**Resources** The SYCL runtime, which is integrated in the SYCL application, manages the resources required by the SYCL backend API to manage the devices. These managed resources and their intricacies are:

- *Platform*: can be seen as a device vendor's runtime and the devices accessible through it. They implement all features of SYCL backend APIs.
- *Context*: contains a collection of devices that can be used by the host, and manages memory objects that can be shared between devices. Devices in the same context must be able to access each other's global memory. A context can only contain devices owned by a single platform.
- *Device*: represents a computing unit on which kernels can be executed. Devices are provided by a platform.
- *Kernel*: a method or some collection of statements that will be executed on a device. In SYCL, a kernel is a C++ function object.
- *Kernel bundle*: Kernels are stored internally by SYCL as device images, which can be grouped into kernel bundles. These bundles provide a way for the application to control the online compilation of kernels.
- *Queue*: Kernels are often submitted to queues, which are associated with a context, a platform and a device. The queues execute the submitted kernels on its associated device.

In Figure 3.12 an overview is given of how the resources managed by SYCL relate to each other.

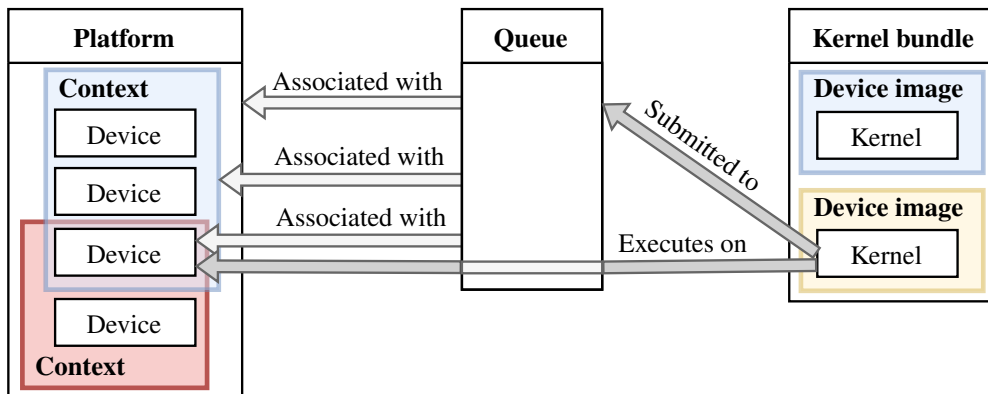


Figure 3.12: Overview of the relations between the resources managed by the SYCL runtime.

**Execution order** By default, kernel functions in a queue are executed in an *out-of-order* fashion, based on their execution requirements. These execution requirements are defined in the command group using *data accessors*, which specify what data must be accessible in order to execute a kernel. The SYCL runtime guarantees that kernels are executed in an order that guarantees correctness with regards to the required data.

An execution requirement that must be satisfied to execute the kernel function in a command group  $CG$  on a particular device is called a **requisite** ( $r$ ). The set of requisites of a command group may be evaluated at any time after it has been submitted to the queue. This evaluation is called *the processing of a command group*.  $CG$ 's kernel is not executed until all its  $r_i$  are satisfied. The processing of a command group does not block execution and can be performed multiple times.

The collection of implementation-defined operations that must be performed to satisfy a requisite  $r_i$  is called an **action** ( $a_i$ ). This collection can be empty if  $r_i$  is already satisfied. Examples of actions are: memory copy operations, mapping operations, host-side synchronisation, and implementation-specific behaviour.

Command groups with a non-empty intersection of requisites are said to depend on each other, and are executed in order of submission to the queue. When these command groups are submitted to different queues or by multiple threads, the order is determined at runtime. Command groups that are independent of each other can be executed in parallel.

To give an example, take a look at Figure 3.13. Three command groups are submitted for execution to a queue. The first two have different requisites, so they are independent and can be executed in parallel. Command group *c*, however, depends on command groups *a* and *b*, as it shares a requisite with both. Therefore command group *c* is not executed till command groups *a* and *b* are done.

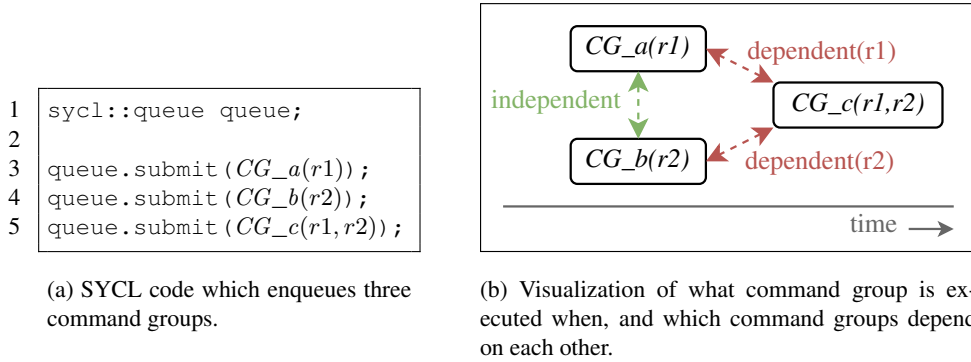


Figure 3.13: Example of how the queued command groups are scheduled by SYCL. *Inspired by [72].*

The execution order can also be controlled explicitly through *event* objects, which are returned when submitting a command group to a queue. Developers can tell the host to wait on an event to finish before continuing. Event objects can also be used to define requisites between command groups, which tell the runtime that one or more command groups must have finished executing before some other command group is allowed to be executed.

An example of controlling the execution order using events can be seen in Figure 3.14. The code is nearly the same, except that command group *b* additionally has the event *e* as a requisite. This makes command group *b* depend on *a*, so now *b* needs to wait for *a* to be finished before it can execute.

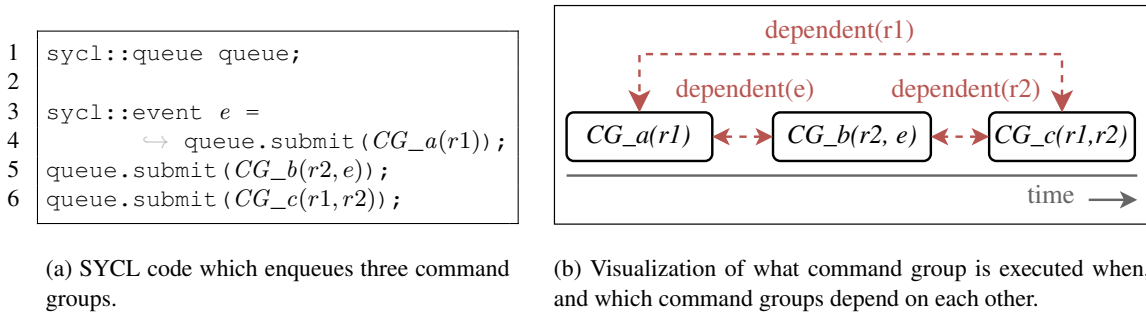


Figure 3.14: Example of how event can be used to influence the scheduling of the queued command groups. *Inspired by [72].*

### 3.2.2.2 Kernel execution model

The kernel execution model governs code in the kernel scope. For each submitted kernel function, an index-space is defined. For each point in this index-space, there is a work-item which executes the kernel function. Each work-item has a unique *global id* based its position in the index-space. There are three types of kernels: basic, ND-range, and backend-specific kernels.

**Basic kernels** A basic kernel has a one-, two-, or three-dimensional index-space, where for every point in the index-space a work-item is executed.

**ND-range kernels** Just as the basic kernel type, an ND-range kernel has a one-, two-, or three-dimensional index-space. However, in ND-range kernels work-items are organised into work-groups.

Each work-group gets a unique *work-group id*, and each work-item is assigned a *local id*, which is unique inside its work-group. Each work-item also still has a global id. This means that each work-item can be uniquely identified by either its global id or by a combination of its local id and work-group id. All work-items in the same work-group execute concurrently on the processing elements of a single compute unit.

Work-groups can optionally be subdivided into *sub-groups*, which have a one-dimensional index-space. It cannot be assumed that work-items within sub-groups execute in an particular order, or that work-groups are divided into sub-groups in a specific way.

To give an example of how work-items can be identified, take a look at Figure 3.15. In the figure, the blue work-item has the following ids:

- global id: (2,4)
- work-group id: (1,1)
- local id: (0,1)

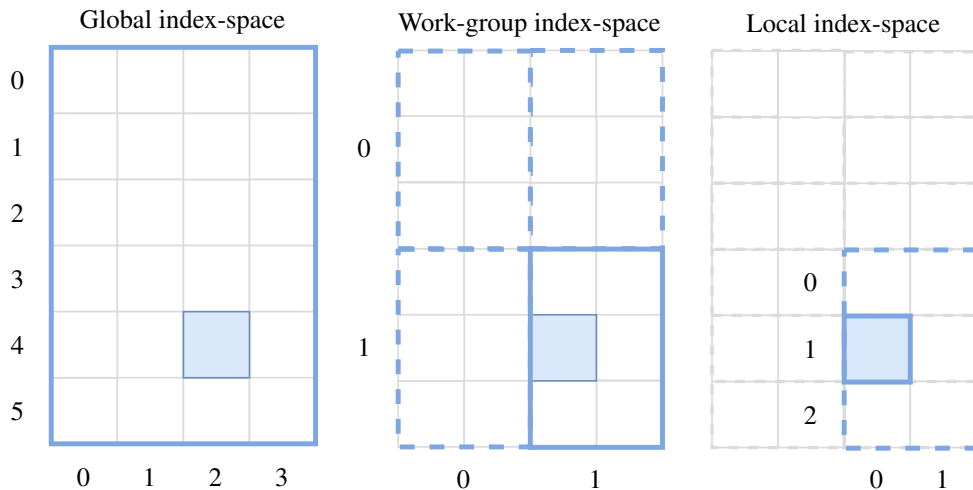


Figure 3.15: Overview of the three index-spaces in an ND-range kernel. One work-item has been highlighted in blue.

**Backend-specific kernels** SYCL backends are allowed to expose fixed functionality as non-programmable built-in kernels. The availability and behaviour of these built-in kernels is backend-specific, and does not have to follow the SYCL execution and memory models.

### 3.2.3 Memory model

SYCL's memory model is split up into an application and kernel memory model. SYCL also has a memory consistency model.

#### 3.2.3.1 Application memory model

To allocate memory in the global address space, buffer objects or USM allocation functions are used. To make buffers accessible to kernels on a device, data accessors can be used.

**Buffers** SYCL's `buffer` class defines a shared array of one, two, or three dimensions. Buffers allow an object to be shared between the host and devices with different contexts, platforms or backends.

**Unified Shared Memory** USM stands for unified shared memory, which uses a single unified address space across the host and the devices. This means that pointers to USM allocations are consistent across the host and devices and can be directly passed to kernels as arguments. [37] There are three types of USM allocations:

- *Device allocations* are allocations to the device's memory and can only be read directly by devices. To read the data on the host, explicit data copies must be made.
- *Host allocations* are allocations to the host memory and can be read by the host and the device. However, kernels that want to interact with this data need to do it remotely, over a potentially slower bus.
- *Shared allocations* are accessible on both the host and the device. Shared allocations allow the data to automatically be migrated between the host and device memories to give one or the other better performance.

USM memory objects are bound to a specific context, thus can only be accessed by devices with the same context. Accessing USM allocations does not alter the order of execution, however, users can influence it by specifying the requirements necessary for valid execution.

**Data accessors** To give a kernel access to memory objects on the host, such as buffers, a *data accessor* must be defined in the kernel's command group. When a data accessor object is constructed, the requirements are processed as requisites, regardless of whether the memory is actually used. A data accessor definition contains the required data-access, an access target, and a memory object that needs to be accessed. The required data-access can be *read-only*, *write-only*, or *read-write*. There are also two other types of access: *no-init-write-only*, and *no-init-read-write*, which tell the system to discard any previous values in the memory. The access target tells SYCL in which memory the data will be contained, for example, in the global or local memory.

When processing all accessors in the same command group that request access to the same memory object, accessors with the same access targets are combined into single requisites. This means that there will be no requisites for the same memory object with the same access target. Only one of the remaining requisites for the same memory object is allowed to have write or read-write access, otherwise a runtime exception will be thrown. A requisite with write or read-write access determines the side effects of the command group on the memory object.

Data accessors can also be created for the host, which are called *host accessors*. They define when what memory is required to be available on the host. *Local accessors* also exist, and provide access to the local memory of the device the kernel is executed on.



### 3.2.3.2 Device memory model

The memory model for devices is based on OpenCL 1.2's memory model. It distinguishes four types of address spaces:

**Global memory** Global memory is accessible to all work-items in all work-groups. All work-items can read from and write to any location within the global memory. This address space is persistent across kernel executions. If multiple kernels access a location in USM concurrently, where at least one of the kernels modifies it, there will be a data-race unless memory fences or atomic operations are used.

**Local memory** Local memory is accessible to all work-items in the same work-group. Accessing this memory from another work-group causes undefined behaviour. A variable will automatically be allocated in local memory when it is defined in the `sycl::parallel_for_work_group` scope, or declared with a local accessor.

**Private memory** Private memory is accessible to a single work-item. Accessing this memory from another work-item causes undefined behaviour. A variable will automatically be allocated in private memory when it is defined in the `sycl::parallel_for` or `sycl::parallel_for_work_item` scope.

**Generic memory** Generic memory is a virtual address space which overlaps the global, local, and private address spaces. A pointer which points to an address in the generic address space represents an address in either of the three address spaces. Thus, generic memory can be seen as an abstraction of global, local, and private memory. This abstraction allows developers to write a single function that can take arguments from any of the three address spaces by defining the arguments as generic memory address pointers [44].

SYCL has `multi_ptr` class which represents a pointer to a location in a particular address space, where the address space can be any of the aforementioned. This class allows for conversion from a pointer to an address in generic memory to a pointer to either an address in global, local or private memory. However, the result of this conversion is undefined when the generic memory address does not represent an address in the provided address space.

### 3.2.3.3 Memory consistency model

SYCL's memory consistency model is based on C++'s memory consistency model. Operations within a work-item are ordered according to C++'s *sequenced before* relation (described in [25]). To ensure memory consistency across work-items, group barriers, memory fences, atomic operations can be used. The operations across different work-items are ordered according to C++'s *happens before* relation (described in [26]).

On a device, group barriers can be used to ensure consistency of the local and/or global memory across work-items in the same group (see subsection 3.2.4.2). If the device supports *acquire-release* memory ordering (described in [26]) or *sequentially consistent* memory ordering (described in [16]), memory fences and atomic operations can also be used.

Memory consistency between the host and devices, and between devices in the same context, can be controlled using synchronization on the application level, as described in subsection 3.2.4.1. If the device supports concurrent atomic accesses to USM and *acquire-release* memory ordering or *sequentially consistent* memory ordering, cross-device memory can also be made consistent using memory fences and atomic operations.

### 3.2.4 Synchronization

Different kinds of synchronization are possible on the application and kernel levels.

#### 3.2.4.1 Application synchronization

There are multiple synchronization points between the host and devices.

- *Buffer destruction*: When buffers are requested to be destroyed, the host waits till all submitted work on these buffers is completed and copied back to the host memory. However, the destructors only wait if the object was constructed with attached host memory and if data needs to be copied back to the host.
- *Host accessors*: Host accessor constructors wait for all kernels that modify its buffer to complete and then copy data back to the host memory. Command groups that require to same memory objects as in the host accessor cannot be executed till the host accessor has been destroyed.
- *Queue operations*: Queue operations can be used by the developer to block execution of a thread until all command groups that are submitted to the queue have finished execution.
- *Event objects*: As mentioned in subsection 3.2.2.1, event objects can be used for synchronization.

#### 3.2.4.2 Kernel synchronization

Work-items in a kernel can synchronize in multiple ways. Memory fences give control over what memory loads and stores are allowed to be reordered when. Synchronisation between work-items in the same work- or sub-group can be done through group barriers.

**Memory fences** To control the reordering of memory loads and stores, memory fences can be used. The memory fence's behaviour is dependent on its *memory order*. What work-items are affected by the fence is determined by the fence's *memory scope*.

The supported memory orders (which have the same behaviour as in the C++ core language) are:

- *relaxed*: Load and store operations before or after the fence can be reordered without restrictions.
- *acquire*: Load and store operations after the fence cannot be reordered before it.
- *release*: Load and store operations before the fence cannot be reordered after it. Moreover, store operations before the fence are guaranteed to be visible to other work-items after acquire memory fence on the same.
- *acq\_rel*: Has the combined behaviour of *acquire* and *release*.
- *seq\_cst*: Has *acquire* behaviour for loads, *release* behaviour for stores, and *acq\_rel* behaviour for load-modify-store operations. All operations are observed in a sequentially consistent order.

The supported memory scopes are:

- *work-item*: The fence only applies to the work-item that executes the fence. However, all operations, except from image operations, already execute in program order.
- *sub-group*: The fence only applies to work-items in the same sub-group as the work-item that executes the fence.
- *work-group*: The fence only applies to work-items in the same work-group as the work-item that executes the fence.
- *device*: The fence applies to work-items on the same device as the work-item that executes the fence.
- *system*: The fence applies to all work-items on the entire system.

**Group barriers** Group barriers allow for synchronisation between work-items in the same work- or sub-group. In SYCL, group barriers serve two functions: First of all, a group barrier synchronizes executing work-items in a work- or sub-group. This allows, for example, a work-item to wait until another work-item is finished before using its results. Secondly, it synchronizes the state of memory for each work-item, to allow for memory consistency.

When a barrier is declared, it must be specified whether the barrier is for sub-groups or work-groups. When a work-item reaches a group barrier, it has to wait at the barrier until all work-items in the same group have reached the barrier. The barrier also performs two memory fences: before synchronising at the barrier all work-items execute a release fence, and after synchronising all work-items execute an acquire fence. By default, the memory scope of the fences is the narrowest scope which includes all work-items in the group. However, the memory scope can be altered by defining the desired scope in the barrier declaration.

### **3.2.5 Error handling**

There are two types of errors in SYCL: synchronous errors and asynchronous errors. Synchronous errors, such as a failure to construct an object, can be detected and thrown immediately when an API call is made. Asynchronous errors, such as an error occurring during the execution of a kernel on a device, can only be detected after an API call has returned and are reported via an asynchronous error-handler mechanism.

### 3.3 Formal verification of heterogeneous programs

In this section, the motivation for carrying out formal verification is discussed. After this, deductive verification, method contracts, and auxiliary notations used in deductive verification are explained. Last of all, separation logic and its extensions: concurrent separation logic and permission-based separation logic are described.

#### 3.3.1 Motivation

Software for heterogeneous systems can be concurrent, containing sections of code that are executed in parallel, such as kernels. An example of concurrently executed code on heterogeneous systems are kernels, which are often executed by multiple work-items in parallel.

One problem with concurrent software is that it is notoriously error-prone. The set of possible program behaviours increases exponentially with the size of the parallel parts of the program and the number of processes that execute it, as it contains all possible interleavings of the atomic instructions of each process. With many possible program behaviours, it is easy for developers to overlook problems which only occur in a few possible program behaviours, such as data-races.

To ensure a software's correctness, tests could be written that cover every possible program behaviour. However, this is often impossible, or highly impractical. A more practical way to ensure a concurrent program's correctness, is to perform formal verification, of which deductive verification is a subset. [12]

#### 3.3.2 Deductive verification

A *deductive verification* method is a verification method where a system's desired properties are specified using logical formulas. These specifications are then proven using deduction in a logic calculus. [31]

To give an example of proving a specification deductively, take a look at Listing 3.16, where it needs to be proven that the assertion that `result` must be smaller or equal to both `a` and `b` right before the method returns holds. In this case, one can deduce that the value of `result` depends on the condition `a > b` in the if-statement. Therefore, to proof that the assertion holds, we can distinguish two cases, `a > b` and `a <= b`, which together cover all program behaviours.

In the case of `a > b`, the if-statement will be entered, so `result` will equal `b`. If we replace `result` with `b` in the assertion, we get `b <= a && b <= b`. This can be rewritten to `b <= a`, which is true because we are looking at the case `a > b`. So the assertion holds for the case `a > b`.

In the case of `a <= b`, the if-statement not will be entered, so `result` will equal `a`. If we replace `result` with `a` in the assertion, we get `a <= a && a <= b`. This can be rewritten to `a <= b`, which is true because we are looking at the case `a <= b`. So the assertion holds for the case `a <= b`.

This means that we have now proven deductively that the assertion holds for all possible program behaviours.

```
1 public int min(int a, int b) {  
2     int result = a;  
3     if (a > b) {  
4         result = b;  
5     }  
6     //@ assert result <= a && result <= b;  
7     return result;  
8 }
```

Listing 3.16: Example of a small Java program and a specification. *The specification is written in blue, using JML syntax (see [45]). Statements prepended with `assert` are statements that must always be true when the program reaches the statement.*

In this example, proving that the assertion holds was rather trivial. However, in cases where programs, for example, contain user input, concurrently executed statements, method calls, or loops, proving their specifications becomes more complicated, and extra annotations are required to prove a program's correctness. These annotations will be described in the next subsections.

### 3.3.3 Contracts

To prove the correctness of programs that contain method calls, every method in a program can be annotated with a *contract*. Alshnakat et al [2] describes a contract as the following:

Suppose  $f$  is a method with formal parameters  $a_1, \dots, a_n$  and a formal result variable  $r$ . A contract for  $f$  is a pair  $(Pre_f, Post_f)$  consisting of a pre-condition  $Pre_f$  over the arguments  $a_1, \dots, a_n$ , and a post-condition  $Post_f$  over the arguments  $a_1, \dots, a_n$  and the result  $r$ .

In this definition, a pre-condition states the legal inputs of a method, and a post-condition states properties about the method's result in relation to the method's inputs. Note that expressions used in the pre- and post-conditions must be *pure*, meaning that they cannot have any side-effects (see subsection 3.3.4.3). This prevents specifications from altering the program's state.

Contracts are usually denoted using a Hoare triple [33], where  $S_f$  is the method body of  $f$ :

$$\{Pre_f\} S_f \{Post_f\}$$

An example of a method with a contract can be seen in Listing 3.17. The method `div` can be described by the following Hoare triple:

$$\{b \neq 0\} \text{ return } a / b; \{\backslash\text{result} == a / b\}$$

In method `div`, parameter `a` is divided by parameter `b`. In division, it is not allowed to divide a number by zero, which is why its pre-condition states that parameter `b` is not allowed to be zero. Its post-condition states that `div` should always return the value of `a` divided by `b` if the pre-condition holds and the method terminates.

```
1  //@ requires b != 0;
2  //@ ensures \result == a / b;
3  int div(int a, int b) {
4      return a / b;
5  }
```

Listing 3.17: Example of the Java method `div` with a contract. *The contract is written in blue, using JML syntax. Statements prepended by `requires` represent pre-conditions and statements prepended by `ensures` represent post-conditions. `\result` represents the possible values that can be returned by the method.*

Note that if multiple pre-conditions or multiple post-conditions are declared in a contract, such as in the contract for `min` in Listing 3.18, the actual pre- or post-condition is a conjunction of all the declared pre- or post-conditions. Thus, the actual pre-condition for `min` is:

$$(a > 0 \ \&\& \ a < 1000) \ \&\& \ (b > 0 \ \&\& \ b < 1000)$$

```
1  //@ requires a > 0 && a < 1000;
2  //@ requires b > 0 && b < 1000;
3  //@ ensures a > b ==> \result == b;
4  //@ ensures a <= b ==> \result == a;
5  int min(int a, int b) {
6      return a > b ? b : a;
7  }
```

Listing 3.18: Example of the Java method `min` with a contract. *The contract is written in blue, using JML syntax. The arrow `==>` represents logical implication.*

### 3.3.3.1 Verification process

When every called method  $f$  in a program has a contract, the program can be verified using the following steps:

**1. Verify that all methods satisfy their contract.** First it is verified for all methods whether they satisfy their contract. This verification is done in a modular fashion, where each method is verified in isolation. Method  $f$  satisfies its contract if for every combination of values for parameters  $a_1, \dots, a_n$  that satisfy  $Pre_f$  when  $f$  is called and allow  $f$  to terminate,  $Post_f$  is satisfied when  $f$  returns.

To give an example, the contract for method `div` in Listing 3.17 is satisfied if for every possible combination of values for  $a$  and  $b$ , except for combinations where  $b$  is zero, `div` returns  $a/b$ . As can be seen in the method body of `div`, it satisfies its contract because it always returns  $a/b$ , except when  $b$  is zero, in which case Java throws an exception [55].

**2. Verify the rest of the program.** The rest of program can be verified by surrounding every call to  $f$  with assumptions and assertions based on its contract. Since it is already known (from step 1) that each method  $f$  adheres to its contract, the method's body does not need to be interacted with for each call to  $f$ . Instead, it is checked that the input of the call to  $f$  does not violate  $f$ 's pre-conditions. If this is the case, the post-conditions of  $f$ 's contract can be assumed to be true. These post-condition-based assumptions can help a prover to prove other specifications further down the line.

An example of this second step can be seen in Listing 3.19. The main method first calls the method `min` (see Listing 3.18), so it needs to be verified whether the inputs `x` and `y` satisfy `min`'s pre-conditions, using the inserted `assert` statements. Since `x` and `y` are both within the bounds, the pre-conditions are satisfied. That means that `min`'s post-conditions can be assumed after `min` returns. Next, the method `div` (see Listing 3.17) is called, so `div`'s pre-conditions must be met, so `smallest` must not equal zero. From `min`'s post-conditions, the prover knows that `smallest` equals `x` or `y`. And since neither `x` nor `y` equals 0, `div`'s pre-condition is met. The process of asserting pre-conditions and assuming post-conditions is then continued for the rest of the program's code.

```
1 public static void main(String[] args) {  
2     int x = 5;  
3     int y = 10;  
4     //@ assert x > 0 && x < 1000;  
5     //@ assert y > 0 && y < 1000;  
6     int smallest = min(x, y);  
7     //@ assume x > y ==> smallest == y;  
8     //@ assume x <= y ==> smallest == x;  
9  
10    //@ assert smallest != 0;  
11    int z = div(49, smallest);  
12    //@ assume z == 49 / smallest;  
13 }
```

Listing 3.19: Example of the second step of verification. *The verification statements (using JML syntax) are written in pink, and do not actually appear in the user-defined code, but are added for demonstrative purposes. Statements prepended with `assume` are assumed to be true by the prover and are not checked for correctness.*

**Proving specifications formally** In the examples of the two aforementioned verification steps, specifications were proven using informal deductive reasoning methods for demonstrative purposes. Normally, specifications are proven in a more formal fashion. This can be done by, for example, using a combination of Hoare logic (see [33]) and weakest pre-condition reasoning. [2, 10].

### 3.3.4 Auxiliary annotations

In the example used in the verification second step in subsection 3.3.3.1, proving that the precondition of `div` is satisfied was rather trivial. However, oftentimes it is harder to prove that a certain specification holds because the number of possible program behaviours to evaluate is very large or because the gap between the verifier's assumptions and what needs to be proven is large, requiring the verifier to take many intermediate steps. Therefore, users can often add extra annotations, such as assumptions, assertions and loop invariants to give the prover extra information to help it prove specifications. These auxiliary notations are described in this subsection.

#### 3.3.4.1 Assumptions and assertions

Assumptions and assertions could already be seen in Listing 3.16 and Listing 3.19. Assumptions are statements that are assumed to be true by the prover and are not checked for correctness. Assertions are statements that must always be true when the program reaches said statement.

#### 3.3.4.2 Loop invariants

Loops are often hard to verify because, in general, it is impossible to statically evaluate the loop body repeatedly until the loop condition is false. Usually the number of loop iterations that are executed depends on user input and the state of the program at runtime. To help a prover verify specifications for programs containing loops, users can annotate loops with so-called *loop invariants*. A loop invariant is a statement which is true when the loop is entered, at the start and end of each iteration of the loop, and right after the loop is finished. [35]

Listing 3.20 gives an example of a loop invariants. The function `fib_array` fills an array with length `length` with the Fibonacci sequence. The loop fills an array from the second position till the `length` parameter with the Fibonacci sequence. The three loops invariants state the following:

When the loop is entered, at the start and end of every iteration of the loop, and right after the loop is finished,

- variable `i` will be greater or equal to 2 and smaller or equal to the value of the function's `length` parameter.
- `array` has value 0 at index 0, and value 1 and index 1.
- for all elements in `array` with an index greater or equal to 2 and smaller than the current value of `i`, the value of the element is the sum of the two elements before it.

First we will check whether the loop invariants are true when the loop is entered: When the loop is entered, `array` looks like `[0,1] + ([0] * (length - 2))` and `i == 2`. For the first two loop invariants it is easy to deduce that they are true. The third loop invariant is also true because no items in the array are checked because `j` cannot attain any value as the lowest value of `j` already equals `i`.

Now take a look at the loop invariants for all the following iterations of the loop: The first loop invariant remains true for all iteration steps and right after the loop is done, because the range specified in the loop invariant contains every value `i` will ever attain in the loop. `array[0]` and `array[1]` are never assigned in the loop, so in every iteration step and right after the last iteration the second loop invariant stays true. The third invariant also remains true for all iteration steps and right after the loop is finished, because it looks at all the past assignments the loop made, which were all assigned to `array[j-2] + array[j-1]`, which this loop invariant also states.

```

1  //@ requires length >= 2;
2  //@ ensures \result != null && \result.length == length;
3  //@ ensures \result[0] == 0 && \result[1] == 1;
4  //@ ensures (\forallall int i; i >= 2 && i < length; \result[i] ==
5                  ⇔ \result[i-2] + \result[i-1]);
6  public int[] fib_array(int length) {
7      int[] array = new int[length];
8      array[0] = 0;
9      array[1] = 1;
10
11     //@ loop_invariant i >= 2 && i <= length;
12     //@ loop_invariant array[0] == 0 && array[1] == 1;
13     //@ loop_invariant (\forallall int j; j >= 2 && j < i; array[j] ==
14                             ⇔ array[j-2] + array[j-1]);
15     for (int i = 2; i < length; i++) {
16         array[i] = array[i-2] + array[i-1];
17     }
18     return array;
19 }

```

Listing 3.20: Example of a Java method containing a loop with loop invariants. *The verification statements are written in blue, using JML syntax. Loop invariants are prepended with loop\_invariant. Expressions in the form of (\forallall declaration; condition; boolean expression) are true if the boolean expression is always true when the condition is true, where variables declared in the declaration can be used in the condition and boolean expression.*

### 3.3.4.3 Pure methods

As mentioned before in subsection 3.3.3, expressions inside specifications must be pure. An expression is pure if all of the following is true [45]:

- The expression has no side-effects. So no class- or global variables are assigned and no non-pure methods are called.
- The expression must be deterministic: for the same program state and inputs it must always return the same value.
- If the expression is a method call, the expression must terminate normally or throw an exception, even for calls that do not satisfy the method's pre-conditions.

For complicated methods, it can be hard for the prover to verify whether a method is pure. To help the prover, programmers can mark methods that they know are pure<sup>2</sup> as such, as can be seen in Listing 3.21.

```

1  /*@ pure */
2  public int add(int a, int b) {
3      return a + b;
4  }

```

Listing 3.21: Example of a Java method annotated as pure. *The verification statements are written in blue, using JML syntax. Methods prepended with pure are pure methods.*

<sup>2</sup>Note that for the remainder of this report, methods that are pure will be referred to as *functions*.



### 3.3.5 Separation logic

Separation logic can be used to reason about what data on a heap can be accessed when and by which processes. Separation logic can be seen as an extension to the deductive verification concepts shown in the previous subsections. In this section, first the basic concepts of separation logic are explained. After this, concepts from concurrent separation logic and permission-based separation logic, which both extend separation logic, are discussed.

#### 3.3.5.1 Relations

Separation logic adds three different relations and a notation for an empty heap [52, 59, 60]:

**Points-to relation** The points-to relation  $x \mapsto v$  states that the heap must contain a cell at address  $x$ , with value  $v$ . When talking about pointers,  $x \mapsto v$  can also mean that pointer  $x$  must point to value  $v$ .

**Separating conjunction** The separating conjunction  $P * Q$  states that the heap can be split into two disjoint parts, one part where  $P$  must hold and one part where  $Q$  must hold.

**Separating implication** The separating implication  $P \multimap Q$  (also called a *magic wand*) states that if the current heap is extended with a disjoint partition where  $P$  holds, then  $Q$  must hold in this extended heap.

**Empty heap** The expression  $\text{emp}$  states that the heap is empty. It is used as the unit of  $*$ , meaning that:  $P = \text{emp} * P = P * \text{emp}$ .

#### 3.3.5.2 Rules and behaviour

There are several rules and observations involving separation logic [15, 52, 59, 60], which will be explained below.

**Frame rule** The most important rule in separation logic is the frame rule, which states the following:

$$\frac{\{P\} S \{Q\}}{\{P * R\} S \{Q * R\}} \quad (\text{modifies } S \cap \text{vars } R = \emptyset)$$

This rule states that local specifications can be extended with a predicate  $R$ , and that reasoning about  $R$  and its effects can be done separately from  $S$ , because  $R$  is disjoint from  $P$  and  $Q$ . Note that in this rule,  $R$  cannot contain any variables that are modified by  $S$ .

**Monotonicity** The separation conjunction is monotone with respect to implication:

$$\frac{P_1 \Rightarrow Q_1 \quad P_2 \Rightarrow Q_2}{P_1 * P_2 \Rightarrow Q_1 * Q_2}$$

**Modus ponens** The modus ponens rule of inference  $((P \rightarrow Q) \wedge P \models Q)$  found in propositional logic can be replicated in separation logic as follows:

$$(P \multimap Q) * P \models Q$$

Here  $(P \multimap Q)$  states that if a disjoint partition is added where  $P$  holds, then  $Q$  holds. The separation conjunction with  $P$  states that a disjoint partition where  $P$  holds exists, which entails that  $Q$  holds.

**Interesting  $\mapsto$  and  $*$  observations** There are some interesting observations regarding the  $\mapsto$  and  $*$  relations:

- $x \mapsto v * x \mapsto v$  is always false, as there is no way to divide any heap in a way that  $x$  is in both partitions.
- $x \mapsto v_1 * y \mapsto v_2$ , where  $v_1$  and  $v_2$  can be anything, even the same value, implicitly states that  $x$  and  $y$  are not aliases (distinct names for the same location). This is true because  $*$  split the heap into a disjoint partition where  $x$  exists, and a different disjoint partition where  $y$  exists, and a single location cannot exist in multiple disjoint partitions at the same time.

### 3.3.5.3 Concurrent separation logic

Concurrent separation logic (CSL) focuses on how and when concurrent processes, such as work-items in kernels, can access data. Two rules introduced by CSL are discussed in this subsection, namely the *parallel composition rule* and the *conditional critical region rule*. [15, 18]

**Parallel composition rule** The parallel composition rule states the following:

$$\frac{\{P_1\} \text{ process}_1 \{Q_1\} \dots \{P_n\} \text{ process}_n \{Q_n\}}{\{P_1 * \dots * P_n\} \text{ process}_1 \parallel \dots \parallel \text{ process}_n \{Q_1 * \dots * Q_n\}}$$

In this rule, a parallel composition holds if the heap can be separated into disjoint parts such that the pre-condition of each process holds, and that after the processes terminate, the heap can be separated into disjoint parts again where the post-condition of each process holds. If the parallel composition is valid, it means that each process can be assessed completely independently.

To give an example: suppose the methods `setA` and `setB` from Listing 3.22 are executed in parallel, with the following pre- and post-conditions:

$$\begin{array}{c} \{\text{this.a} \mapsto - * \text{this.b} \mapsto -\} \\ \text{setA(x); } \parallel \text{ setB(y); } \\ \{\text{this.a} \mapsto x * \text{this.b} \mapsto y\} \end{array}$$

As can be seen, there are disjoint pre- and post-conditions for the variables `this.a` and `this.b`. Since `this.a` is only used in `setA` and `this.b` is only used in `setB`, the process that executes `setA` can be assessed separately from the process that executes `setB`. This gives:

$$\begin{array}{cc} \{\text{this.a} \mapsto -\} & \{\text{this.b} \mapsto -\} \\ \text{setA(x); } & \text{setB(y); } \\ \{\text{this.a} \mapsto x\} & \{\text{this.b} \mapsto y\} \end{array}$$

```

1 class Data {
2     Object a;
3     Object b;
4
5     public setA(Object newA) {
6         this.a = newA;
7     }
8
9     public setB(Object newB) {
10        this.b = newB;
11    }
12 }
```

Listing 3.22: Java code containing two methods that can be assumed to be executed in parallel.

**Conditional critical region rule** The parallel composition rule prevents multiple processes from having access to the same addresses on the heap. However, parallel programs often have critical sections where multiple processes do need access to the same addresses on the heap. Many critical regions can be described as a conditional critical region (CCR):

with  $r$  when  $B$  do  $C$

In a CCR,  $r$  is a resource or a group of resources on the heap,  $B$  is a boolean guard, and  $C$  is critical region's code.

A CCR is executed in the following fashion:

1. Acquire resource  $r$ .
2. Evaluate the boolean guard  $B$ .
  - If  $B$  is true, execute  $C$  and release  $r$ .
  - If  $B$  is false, release  $r$  and try again.

In separation logic, the conditional critical region rule can be used to verify CCRs:

$$\frac{\{(P * RI_r) \wedge B\} C \{Q * RI_r\}}{\{P\} \text{ with } r \text{ when } B \text{ do } C \{Q\}}$$

This rule adds the idea that processes can "own" a certain partition of the heap, and that they can transfer this ownership to another process during runtime. This allows ownership of resources to be transferred to processes that enter CCRs and transferred back when CCRs are finished executing. In this rule,  $r$  is associated with a *resource invariant*  $RI_r$ , which must be true before the CCR can execute, and must be true when the CCR is finished as well. The rule makes the execution of all CCRs that want to access  $r$  mutually exclusive. When a process enters a CCR, only the  $B$  and  $C$  in the CCR are allowed to access  $r$ .

An example of an application of the conditional critical region rule can be found in Listing 3.23. It contains a class `Cell` with two methods, `produce` and `consume`, which will be executed in parallel. Both methods alter the variables `value` and `hasValue`. The methods should not write to those variables at the same time, to avoid `value` from being read when its empty or being overwritten when it contains an unconsumed value. Therefore, the variables are set in a conditional critical region that can only be entered if `hasValue` has a certain value. Method `produce` contains the following CCR:

```
with {value, hasValue} when ¬hasValue
do {value = i; hasValue = true;}
```

For this example, the following pre- and post-conditions and resource invariant were chosen:

{i ↦ -}      *pre-condition*

{i ↦ -}      *post-condition*

(hasValue \* value ↦ -) ∨ (¬hasValue \* emp)      *resource invariant*

The result of the application of conditional critical region rule to the CCR in the `produce` method can be seen in Listing 3.23 on lines 7 and 10.

```

1 class Cell {
2   private Integer value = null;
3   private int hasValue = true;
4
5   public produce(Integer i) {
6     while (hasValue) {}
7     {(i ↦ - * ((hasValue * value ↦ -) ∨ (¬hasValue * emp))) ∧ ¬hasValue}
8     value = i;
9     hasValue = true;
10    {i ↦ - * ((hasValue * value ↦ -) ∨ (¬hasValue * emp))}
11  }
12
13  public int consume() {
14    Integer result;
15    while (!hasValue) {}
16    {(emp * ((hasValue * value ↦ -) ∨ (¬hasValue * emp))) ∧ hasValue}
17    result = value;
18    value = null;
19    hasValue = false;
20    {result ↦ - * ((hasValue * value ↦ -) ∨ (¬hasValue * emp))}
21    return result;
22  }
23 }

```

Listing 3.23: Example application of the conditional critical region rule on Java code. *The coloured expressions correspond with expressions with the same colour in the explanation text above. This example is based on Figure 2 in [15].*

### 3.3.5.4 Permission-based separation logic

Sometimes multiple processes can access the same partition of the heap without causing concurrency issues. For example, if processes only read the value at the same address, and do not alter it. The separation logic explained till now, however, does not allow reasoning about concurrent access to the same location by multiple processes. To solve this, permission-based separation logic (PBSL) extends separation logic to allow data to be accessed concurrently when it is safe to do so [15, 59]

**Ownership** In order to reason about when multiple processes are allowed to use the same heap locations, the definition of  $x \mapsto v$  relation is extended. When  $x \mapsto v$  is true for a certain process, it means that the process has *ownership* of  $x$ . This concept was already mentioned in the explanation of the conditional critical region rule, in the previous paragraph, where it was also mentioned that this ownership can be transferred between processes.

**Fractional permissions** The concept of fractional permissions allows ownership to be split into multiple pieces, to allow multiple processes to access the same heap locations. This is done by associating a *permission*  $z$  with the  $\mapsto$  relation:  $x \overset{z}{\mapsto} v$ . Here  $z$  is a fractional real number, such that  $0 < z \leq 1$ . When a process has a permission  $z = 1$  for location  $x$ , it has the same meaning as  $x \mapsto v$ : the process is the single owner of  $x$ , and is allowed to read and write to  $x$ . When  $z < 1$ , the process is only allowed to read the value stored at location  $x$ . Permissions for the same location  $x$  adhere to the following conservation law:

$$x \overset{z}{\mapsto} v * x \overset{z'}{\mapsto} v \iff x \overset{z+z'}{\mapsto} v$$

The sum of permissions for a location  $x$  is not allowed to exceed 1. This has as a result that either one process has read and write access to  $x$ , or multiple processes have read access to  $x$ . This makes sure that two or more process are never allowed to concurrently write to  $x$ , which can cause data-race behaviour in programs.

An example of fractional permissions can be seen in Listing 3.24. Here the method `get` has the pre-condition that it needs read access to `this.amount`, and the post-condition that it releases its read-access to `this.amount` when the method terminates. Because the permission in the pre-condition states it needs 0.5 of the total ownership, it means that two processes can execute `get` in parallel, because  $0.5 + 0.5 = 1$ . If, for example, a maximum  $n$  processes are allowed to execute `get` in parallel, the permission for `this.amount` should be changed to `this.amount  $\xrightarrow{1/n}$  -`.

```

1 class Account {
2   int amount = 0;
3
4   //@ requires Perm(this.amount, 0.5);
5   //@ ensures Perm(this.amount, 0.5);
6   int get() {
7     return this.amount;
8   }
9 }

```

Listing 3.24: Example of fractional permissions on a Java function. *VerCors*' permission syntax is used here, where **Perm**( $x$ ,  $z$ ) means  $x \xrightarrow{z} -$ .

**Implicit dynamic frames** PBSL can be used in combination with implicit dynamic frames to solve the frame problem. The frame problem states that a specification contract should specify an upper bound on the set of cells on the heap that can be modified by a method (or some other piece of code, like a kernel) that the contract belongs to. If this is not done, it becomes hard to prove anything about the state of the heap. To give an example, in Figure 3.25 the assertion in `createAccount` cannot be proven because it is unknown whether the method `set` alters `y`, so after line 5 in `createAccount`, it is unknown whether `y` still equals 5. [67]

```

1 class Account {
2   public int amount = 0;
3   public int y = 5;
4
5   //@ ensures this.amount == x;
6   void set(int x) {
7     this.amount = x;
8   }
9 }

```

(a) Account class in Java.

```

1 class Main {
2
3   public createAccount() {
4     Account acc = new Account();
5     acc.set(40);
6     //@ assert acc.y == 5;
7   }
8
9 }

```

(b) Main class in Java.

Figure 3.25: Example of the frame problem in Java. *Inspired by [67].*

To solve this issue, the authors of [67] introduce implicit dynamic frames, which add the following verification rules for a cell  $x$  on the heap and a method  $m$ :

- $m$  is only allowed to access  $x$  if it has permission to do so.
- Inside  $m$ ,  $x$  is only accessible if it is specified as such in the  $m$ 's pre-conditions. This means that a method's pre-conditions specifies the upper bound on the set of cells on the heap that can be modified by the method.
- If  $x$  was accessible before calling  $m$  and access to  $m$  is **not** specified in  $m$ 's pre-conditions, then  $x$  is still accessible after the call to  $m$ .
- If  $x$  was accessible before calling  $m$  and access to  $m$  is specified in  $m$ 's pre-conditions, then  $x$  is only still accessible after the call to  $m$  if access to  $x$  is ensured in  $m$ 's post-conditions.

This solves the frame problem because if  $x$  is not mentioned in  $m$ 's pre-condition, it is not accessible in  $m$ , which means it cannot be altered by  $m$ . PBSL can be used to declare in contracts what cells on the heap are accessible by declaring write permissions for those cells. An example of this can be seen in Listing 3.26, where `set` has been altered in such a way that the assertion in `createAccount` in Figure 3.25 can be proven, because `set`'s contract states that only  $x$  is accessible in `set`, which implicitly declares that  $y$  will not be altered by `set`, which means its value stays equal to 5.

```

1  class Account {
2      public int amount = 0;
3      public int y = 5;
4
5      //@ requires Perm(this.amount, write);
6      //@ ensures Perm(this.amount, write) ** this.amount == x;
7      void set(int x) {
8          this.amount = x;
9      }
10 }

```

Listing 3.26: Example of using PBSL in combination with implicit dynamic frames in Java. *VerCors*' permission syntax is used here, where  $\mathbf{Perm}(x, z) ** x == v$  means  $x \xrightarrow{z} v$ . Inspired by [67].

### 3.4 VerCors

VerCors is a deductive verification tool which uses static analysis to prove partial correctness of programs. It uses permission-based separation logic (PBSL) as its logical foundation, which allows for reasoning about parallel and concurrent constructs. This makes VerCors suitable for verification of programs written for heterogeneous systems. VerCors currently supports reasoning about Java, C, OpenCL, CUDA, OpenMP, and PVL. PVL stands for Prototypal Verification Language and is developed alongside VerCors. Users annotate a program with its specifications and VerCors then verifies whether the program adheres to these specifications. For most supported programming languages, these specifications are written in a comment language similar to JML. In fact, the JML specification examples in section 3.3 are valid syntax in VerCors. Note should be taken that VerCors, by default, only proves *partial* correctness: a program is verified under the assumption that the program terminates. However, as of recent, VerCors optionally allows for verifying, to a certain extent, whether a program terminates.

**Alternatives** There exist alternatives to VerCors which are also capable of carrying out static verification. Such alternatives are: Dafny [46], OpenJML [24], KeY [1], VCC [65], and VeriFast [41]. However, VerCors distinguishes itself from these alternatives in two ways. First of all, VerCors supports verification of different parallel and concurrent language constructs from multiple programming languages. Secondly, it aims to be language-independent, such that support for other languages can be added without redesigning the entire toolset.

**Overview of this section** In this section, first an overview is given of how VerCors verifies annotated programs. Then, it is discussed what deductive verification concepts from section 3.3 are supported by VerCors and what additional concepts that are relevant to this thesis it supports. All the knowledge shared in this section and inspiration for some the examples shown in this section came from VerCors' wiki pages [77]. All the source code in the examples in this section is written in Java, unless stated otherwise.

#### 3.4.1 Overview of the verification process

VerCors' verification process of an annotated program can be divided into different phases, of which an overview can be seen in Figure 3.27. The phases are explained in more detail on the next page.

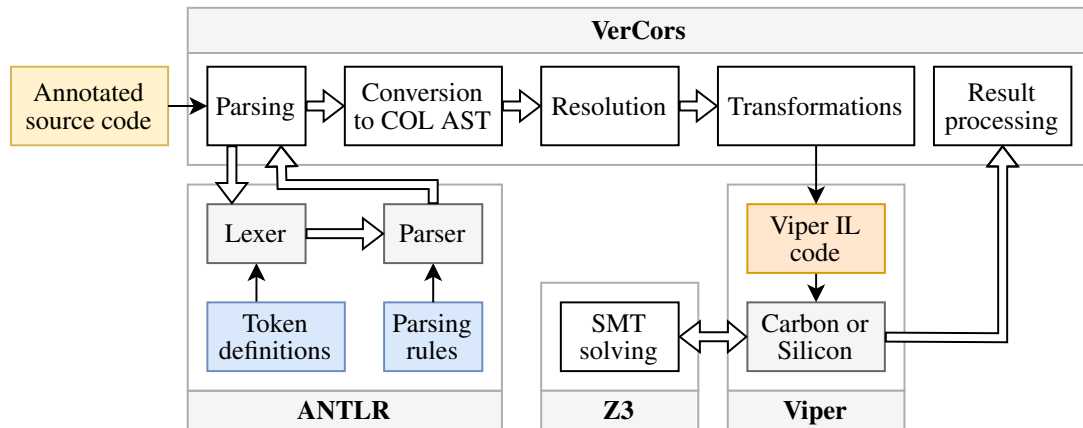


Figure 3.27: Overview of VerCors' verification process. The phases are represented by white boxes and the software component usages by gray boxes. The thick arrows indicate the ordering of these phases and component usages. The boxes encapsulating phases and components show what tool they are part of. Inputs and outputs for some of the phases and components are represented by the coloured boxes. The thin arrows indicate what phases and components the inputs and outputs belong to.

**Parsing** First, the annotated program’s source code is parsed into an abstract syntax tree (AST) using a lexer and a parser. This lexer and parser were generated by ANTLR [57] when the tool was built, based on token definitions and parsing rules for the source code’s language.

**Conversion to COL AST** The AST that was created in the parsing phase is then converted to a COL AST. COL stands for common object language and is VerCors’ internal intermediate representation in which language features from all the languages supported by VerCors can be expressed. The COL language is very similar to PVL, so most PVL syntax is also valid COL syntax and vice versa. Because of this, PVL and COL are often used interchangeably in literature.

**Resolution** After this, all the names, such as variable and method names, in the COL AST are resolved by finding their declarations. For each name, a reference to the found declaration is then stored in the name, to make this information available to later phases.

**Transformations** Then the COL AST goes through many transformations, where each transformation is called a *pass*. The VerCors user can decide what passes are applied. There are several categories of passes which perform different kinds of tasks, namely: reducing features, checking for consistency, standardization, importing built-ins, and optimizations. Finally, the COL AST is transformed into Viper’s intermediate language [48] (referred to as Viper IL for the remainder of this report).

**Verification** The generated Viper IL code is then verified by either *Carbon* or *Silicon*, which are both provers for Viper. Internally, these provers make use of Z3 [28], which is an SMT (satisfiability modulo theories) solver.

**Result processing** When the Viper verifier has finished verifying the Viper IL code, the verification results are read by the VerCors toolset. These results are then translated into useful information about the annotated source code and displayed to the user.

### 3.4.2 Supported deductive verification concepts

In this subsection, VerCors’ support for deductive verification concepts such as contracts, loop invariants, and permissions, as described in section 3.3, will be discussed.

#### 3.4.2.1 Contracts

VerCors supports method contracts containing pre- and post-conditions in a similar fashion as described in subsection 3.3.3.

In post-conditions the following additional two keywords can be used:

- `\result`: The value returned by a method.
- `\old(expression)`: The value of `expression` in the method’s pre-state.

**Context** In a method contract, if an expression must be true in both the pre- and post-states of a method call, the short-hand notation `context expression`; can be used. An example of this can be seen in Listing 3.28, where the meaning of the context clause is annotated in pink. This short-hand notation is useful in cases where there are many pre- and post-conditions that are the same, for example, when a method requires and returns the same permissions for multiple objects on the heap, as then it shortens the contract, making it more readable. Moreover, if `context` is used for all duplicate pre- and post-conditions, then when the contract contains a `requires` or `ensures` clause, a user can instantly see it is only true in the pre- or post-condition, respectively, without having to inspect the rest of the contract.



```

1  //@ context b != 0;
2  //@ requires b != 0;
3  //@ ensures b != 0;
4  public int div(int a, int b) {
5      return a / b;
6  }

```

Listing 3.28: Example of the context clause in VerCors. *The verification statements written in pink do not actually appear in the user-defined code, but are added for demonstrative purposes.*

**Context-everywhere** If an expression must be true in the pre- and post-states of a method, but also right before and right after every iteration of all loops in a method, the expression `context_everywhere expression;` can be used. An example of this can be seen in Listing 3.29, where the meaning of the context-everywhere clause is annotated in pink. This short-hand notation is useful in cases where there are many pre- and post-conditions and loop invariants that are the same, for example, when a method and its loops require and return the same permissions for multiple objects on the heap, as then it shortens the contract, making it more readable.

```

1  //@ context_everywhere number > 0;
2  //@ requires number > 0;
3  //@ ensures number > 0;
4  public int factorial(int number) {
5      int result = 1;
6      //@ loop_invariant number > 0;
7      for(int i = 2; i <= number; i++) {
8          result *= i;
9      }
10     return result;
11 }

```

Listing 3.29: Example of the context-everywhere clause in VerCors. *The verification statements written in pink do not actually appear in the user-defined code, but are added for demonstrative purposes.*

### 3.4.2.2 Assumptions and assertions

VerCors supports assumptions and assertions in a similar fashion as described in subsection 3.3.4.1. It also supports a `refute expression;` statement, which is the inverse of an assertion, because it is true when `expression` is false. Using `refute expression;` instead of `assert !expression;` helps with readability as it makes it easier to see that the expression should be false.

### 3.4.2.3 Loop invariants

VerCors supports loop invariants above loops in a similar fashion as described in subsection 3.3.4.2. Since VerCors checks for partial correctness, it does, by default, not check whether a loop actually terminates. Loop termination can be checked by using a `decreases` clause (see subsection 3.4.4.1).

### 3.4.2.4 Pure methods

VerCors supports marking methods as pure in a similar fashion as described in subsection 3.3.4.3. VerCors then internally reduces the method to a single expression, which is then transformed to a function. It also checks whether the method is actually pure. If the method cannot be rewritten into a single expression or it is not actually pure, an error will be shown to the user. Marking pure methods as such is useful, as any method that has been marked and verified as pure can be used in annotations that do not allow side-effects, such as contracts.

### 3.4.2.5 Permissions

VerCors supports permission-based separation logic as described in subsection 3.3.5. It verifies that there is enough permission to interact with elements on the heap, and shows an error to the user when it concludes this is not the case. To guide the tool, methods, loops and other constructs need to be annotated with what permissions are required or can be assumed. Permissions can be declared anywhere in a specification, such as in pre- and post-conditions, loop invariants, etc.

**Relations** In VerCors, the relations mentioned in subsubsection 3.3.5.1 are represented by the following notations:

- **\*\*** represents the separating conjunction  $*$ .
- **-\*** represents the separating implication  $-*$ .
- **PointsTo**( $x, z, v$ );  
**Perm**( $x, z$ ) &&  $x == v$ ;  
**Perm**( $x, z$ ) \*\*  $x == v$ ;  
all represent the permission  $x \xrightarrow{z} v$  (in this case, **\*\*** should be read as &&, and not as the separating conjunction).

The notations using **Perm**( $x, z$ ) allow users to define permissions separately from specifications about the value  $v$ . However, the permissions for an element should always be declared before the specifications about its value.

Permission value  $z$  can have the following values:

- For write permission: `1` or `write`.
- For read permission: `read`, fraction `x\y`, or a floating point value.
- For no permission: `none`

Note should be taken that when `read` is used, the actual fraction of permission is unspecified. However, the permission can be split infinitely. When combining it with other permissions for  $x$  using the conservation law, the new permission is also an unspecified fraction. Thus, `read` permission cannot be combined with other permissions into a write permission.

- **Value**( $x$ ) represents read permission to  $x$ , where the fractional value does not matter. However, **Value** permissions can be split infinitely. A **Value** permission cannot be combined with other permissions into a write permission.

All of the notations just mentioned have the type **resource** in VerCors, instead of boolean. Usually both boolean and resource types can be used in annotations, but not always.

**Example** To give an example of VerCors' permission notation, take a look at the method `set` in Listing 3.30. If the permission declarations in the method's contract are removed, VerCors will state that the method cannot be verified, because there is insufficient permission to assign the field `amount` on line 7. This is caused by the fact that `amount` is visible to and alterable by other processes (i.e. stored on the heap), which might cause undesired behaviour such as data races.

```

1 class Account {
2   public int amount = 0;
3
4   //@ requires Perm(this.amount, 1);
5   //@ ensures Perm(this.amount, 1) ** this.amount == x;
6   void set(int x) {
7     this.amount = x;
8   }
9 }

```

Listing 3.30: Example of permission declarations in a method contract in VerCors.

To be allowed to execute the method, a process must have write permission to `amount`. In Listing 3.30 this is done by declaring in the pre- and post-condition `Perm(this.amount, 1)`, which means that the method can only be executed if a process has write permission for `amount` and that this write permission is returned when the method terminates.

### 3.4.3 Reasoning about other constructs

In this section, VerCors' support for constructs such as kernels, parallel blocks, fork/join constructs, pointers, and arrays are discussed.

#### 3.4.3.1 OpenCL and CUDA kernels

VerCors has the following support for reasoning about kernels in OpenCL and CUDA programs:

**Kernel contracts** Contracts can be specified for kernels in the same way as for methods. If the pre-conditions hold before the kernel is executed, then the post-conditions can be assumed to hold right after the kernel finished executing.

**Work-items** The shorthand `\gtid` can be used to get a work-item's global id in kernel contracts.

**Work-groups** The shorthand `\ltid` can be used to get a work-item's local id in specifications about kernel functions. The declaration `opencl_gcount == x` in a kernel contract specifies that the work-items that will execute the kernel are grouped into  $x$  work-groups.

**Barriers** Contracts can be specified for barriers inside kernels as well: if the pre-conditions hold right before the barrier is reached, the post-conditions are assumed to hold right after the barrier. Also, the contracts of a barrier are used to redistribute permissions among the work-items that execute the barrier.

#### 3.4.3.2 Parallel blocks

In PVL, there is a so-called parblock construct, which allows the code in its body to be executed by any number of threads, or, using OpenCL's notation, work-items, in parallel. An example of using this construct can be seen in Listing 3.31. Here, a parblock is created with a variable `index` which ranges from 0 (inclusive) to `result.length` (exclusive). This means that `result.length` work-items will execute the body of the parblock on line 8. Each work-item can retrieve its unique id using the just-declared `index` variable. As can be seen on lines 4-6, contracts can also be defined for parblocks. These contracts are seen as work-item-level specifications: they are verified for every work-item individually. Parblocks can also be optionally named: in the example the parblock has been named `adder`.

```

1 void zipAdd(int[] arr1, int[] arr2) {
2   int[] result = new int[arr1.length];
3   par adder (int index = 0 .. result.length)
4     context Perm(arr1[index], 1/2) ** Perm(arr2[index], 1/2);
5     context Perm(result[index], 1);
6     ensures result[index] == arr1[index] + arr2[index];
7   {
8     result[id] = arr1[index] + arr2[index];
9   }
10 }
```

Listing 3.31: Example of a parblock in PVL which zips two arrays by adding their items and storing the results in a third array. *The program is written in PVL, where specifications are written as statements instead of comments. The method contract has been omitted here.*

**Nested parblocks** Parblock can also be nested inside each other, using either of the two (equivalent) notations: declaring a parblock inside another parblock, as is done in Listing 3.32, or declaring multiple iteration variables in a single parblock, as is done in Listing 3.33.

```
1 par outerNestedPar (int y = 0 .. 6) {
2   par innerNestedPar (int x = 0 .. 8) {}
3 }
```

Listing 3.32: Example of declaring a nested parblock using two parblocks with one iteration variable.

```
1 par nestedPar (int y = 0 .. 6, int x = 0 .. 8) {}
```

Listing 3.33: Example of declaring a nested parblock by using one parblock with two iteration variables.

### 3.4.3.3 Fork/Join constructs

PVL also contains fork/join constructs, which are similar to Java threads. When a PVL class contains a run-method, such as the one on lines 3-5 of Listing 3.34, that method can be executed on a different thread from the "main" thread. By writing `fork x;` on an instance `x` of a class containing a run-method, a new thread is spawned which executes the run-method, whilst the code on the current thread continues executing the statements that come after `fork x;`. An example of this can be seen on line 9 in Listing 3.34, which causes the run-method on line 3-5 to execute on a different thread, whilst the current thread continues to execute lines 10 and 11. The user can also wait on a forked instance `x` to finish executing by writing `join x;`. This `join` statement blocks until the run-method terminates, and only then continues executing the statements that come after it.

One thing to note is that run-methods do not take any arguments nor return any values. However, a run-method can indirectly take arguments and return values by reading and updating fields of the class it resides in.

```
1 class Test {
2
3   run {
4     // code to be run in another thread
5   }
6
7   void main() {
8     Test t = new Test();
9     fork t;
10    // code to be run in the current thread
11    join t;
12  }
13
14 }
```

Listing 3.34: Example of the class `Test` being forked and joined again.

### 3.4.3.4 Pointers

VerCors contains the following two expressions to reason about pointers in C<sup>3</sup>:

- `\pointer(name, size, permission)`: Denotes that pointer `name` is not `null`, `name + 0` to `name + size - 1` are valid locations, and the program has `permission` permission for the locations `name` points to.
- `\pointer_index(name, index, permission)`: Denotes that pointer `name` is not `null`, `name + index` is a valid location, and the program has `permission` permission for the location `name + index`.

<sup>3</sup>Support for these expressions was also added for C++ in this thesis

### 3.4.3.5 Arrays

VerCors contains the following shorthand notations to reason about arrays:

**Array length** The expression `\array(name, N)` can be used to denote that array `name` is not `null` and has length `N`.

**Matrices** The expression `\matrix(name, M, N)` can be used to specify there is an `M` by `N` matrix `name` which is not `null` and where every row is a different array instance.

## 3.4.4 Supplementary clauses, operators, expressions, and types

In this subsection, VerCors' supplementary clauses, operators, expressions, and types, which can be used in any language, are discussed.

### 3.4.4.1 Decreases clause

The clause `decreases expression;`, where `expression` must return an integer, can be placed in contracts of loops and recursive methods. It states that `expression` must decrease by at least one and should never become negative in every iteration of a loop or every recursive call of a method. If the right `expression` is chosen, the decreases clause can be used to proof that a loop or recursive method terminates.

```
1  //@ decreases number;
2  public int factorial(int number) {
3      if (number <= 1) {
4          return 1;
5      }
6      return number * factorial(number - 1);
7  }
```

Listing 3.35: Example usage of a decreases clause in VerCors.

An example of a decreases clause can be seen in Listing 3.35 on the previous page. The decreases clause states that `number` must decrease by at least one and should never become negative in every recursive call of `factorial`. In the example, this is indeed the case, as it is called with `number - 1`, which means that `factorial` keeps being called with a smaller `number` parameter. And because of the if-statement, it stops recursively calling `factorial` when `number` equals 1, so it also never becomes negative. Because the decreases clause can be verified, it is now known that `factorial` will always terminate.

### 3.4.4.2 Operators

The following operators are also supported by VerCors:

- `==>` is a logical implication operator, where `a ==> b` means  $a \implies b$ .
- `?.` is a null-coalescing operator, where `a?.b` means  $a \neq \text{null} \implies a.b$ .

### 3.4.4.3 Let expressions

The expression `(\let T name = expression1; expression2)` means that `expression1` is assigned to a variable `name` of type `T`. This variable can then be referenced in `expression2` instead of writing `expression1`. This feature allows for better readability when `expression1` is long or repeated multiple times in `expression2`.

#### 3.4.4.4 Quantified expressions

In logic, the quantifiers  $\forall$  and  $\exists$  are used to specify something that must, respectively, always hold, or hold in at least one case. VerCors supports such reasoning as well, with the following expressions:

- To denote  $\forall$ : (**\forall** declaration, ...; condition; expression) or ( $\forall$  declaration, ...; condition; expression)
- To denote  $\exists$ : (**\exists** declaration, ...; condition; expression) or ( $\exists$  declaration, ...; condition; expression)

In these notations, `declaration` denotes the declaration of one or more variables, examples of which are: `int i` and `int j=a..b`, where `a..b` denotes an integral type (in this case `int`) in the interval  $[a, b]$ . The argument `condition` denotes the constraints on what cases are considered, in the form of a boolean expression. Usually, these are constraints on the range of the just declared variables, such as `0 >= i && i < n`. If there are no constraints, `condition` can be omitted. The argument `expression` is a boolean expression which must hold for all cases or at least one case that abides by the constraints in `condition`.

**Quantified permissions** To define permissions for multiple elements on the heap at the same time, such as elements of an array, a special kind of quantified expression can be used:

```
(\forallall* declaration, ...; condition; resource)
```

In this expression, `resource` is a resource expression, such as `Perm(a[i], z) ** a[i] == v`. This expression means that every case for which `condition` is true, `resource` should hold.

One can also quantify permissions for all elements of an array using the following short-hand notation: `Perm(array[*], z)`.

**Triggers** To apply a quantified expression to a concrete case, it need to be instantiated. For example, suppose the quantifier

```
(\forallall int i; i >= 0 && i < array.length; array[i] == 0)
```

holds when the verifier comes across a statement containing `array[5]` (and `array.length > 5`). Then the quantifier needs to be instantiated, which replaces `i` with the concrete value 5 in the quantified expression, to be able to know that `array[5] == 0` holds. This instantiation is usually done automatically by VerCors, using some heuristics and randomisation. However, for complex expressions it is hard for VerCors to automatically recognise where a quantified expression can be instantiated, in which case verification might fail even though the program is sound, take a long time or get stuck. To combat this, a trigger can be defined inside a quantified expression, which tells VerCors when to instantiate it. To mark part of an expression inside a quantified expression as a trigger, the notation `{:part_of_expression :}` is used. The trigger must contain all variables declared in the quantifier. Note that when a trigger is present, automatic instantiation is disabled, so the quantified expression will only be instantiated when an expression matches the trigger. VerCors also supports some more advanced trigger syntax and behaviour such as multiple triggers in one quantified expression and triggers in nested quantified expressions, but those will not be discussed.

To give an example of the behaviour of a quantified expression with a trigger, consider the method `sumWithLast` in Listing 3.36 on the next page. In the pre-condition the quantified expression has the trigger `a[i]`. On line 4 `a[a.length-1]` is used, which matches the trigger, so the quantified expression is instantiated there. This means the verifier knows that on line 4 that `a[a.length-1] == 0`, so it can deduce that `sum == 0 + b`. With that information it can prove the post-condition of the method.

```

1  //@ requires a != null && a.length > 0;
2  //@ context Perm(a[*], read);
3  //@ requires (\forallall int i = 0 .. a.length; { : a[i] : } == 0);
4  //@ ensures \result == b;
5  public int sumWithLast(int[] a, int b) {
6      int sum = a[a.length-1] + b;
7      //@ assert a[a.length-1] == 0;
8      return sum;
9  }

```

Listing 3.36: Example of triggers in a quantified expression in VerCors. *The statement written in pink does not actually appear in the user-defined code, but is added for demonstrative purposes.*

### 3.4.4.5 Primitive types

VerCors adds two primitive data types on top of the primitive data types from the supported languages:

- **bool**: A boolean type which is useful for target languages that do not support a boolean type. However, they can be used in any target language.
- **frac**: A permission in PBSL is always a real value in the interval  $(0, 1]$ . To make denoting fractional permissions more convenient the, **frac** type can be used. A **frac** value is denoted in the following way:  $x \backslash y$ , which means an  $\frac{x}{y}$  fraction of permission.

### 3.4.4.6 Axiomatic data types

Axiomatic data types (ADTs) are data types that are defined by a set of uninterpreted functions whose behaviour is described by axioms (assumptions). VerCors contains the following axiomatic data types:

- **Sequence**: an ordered collection of values. A sequence is finite and immutable. It has the type `seq<T>`, where `T` is the type of the values.
- **Bag**: an unordered collection of values. A bag is finite and immutable. It has the type `bag<T>`, where `T` is the type of the values.
- **Set**: an unordered collection of unique values. A set is finite and immutable. It has the type `set<T>`, where `T` is the type of the values.
- **Tuple**: an ordered collection containing exactly two elements. A tuple is immutable. It has the type `tuple<F, S>`, where `F` is the type of the first element, and `S` the type of the second element.
- **Map**: an unordered collection of key-value pairs, with unique keys. A map is finite and immutable. It has the type `map<K, V>`, where `K` is the type of the keys, and `V` the type of the values.
- **Option**: a container which is empty or holds one element. It has the type `option<T>`, where `T` is the type of the item it might hold. The value of an option is either `None` or `Some(e)`, where element `e` is of type `T`.

To support domain-specific assumptions, users can also define their own ADTs in PVL.

### 3.4.5 Ghost code

VerCors supports so-called *ghost code*. Ghost code is code that can only be seen by the prover. It can be used to create helper methods or variables to, for example, keep track of the intermediate state inside a method. The variables declared in ghost code make up the *ghost state*. Ghost code can be placed anywhere where statements are allowed in a program, for example inside a method, or in a global context. Also, any code that is legal in the target language is allowed to be ghost code, so entire ghost classes can be created as well. Code is seen as ghost code if it is placed inside an annotation comment which starts with the keyword **ghost**. However, when the ghost code contains no constructs from the target language, the **ghost** keyword can be omitted.

### 3.4.5.1 Ghost methods and variables

Ghost methods and variables can be declared. They have the same syntax as actual methods and variables, except that they live inside an annotation comment that starts with **ghost**. An example of a ghost method can be seen in Listing 3.37, where the ghost method `isPositive` has been declared. As can be seen, ghost methods can have contracts too. An example of a ghost variable is also present in Listing 3.37: the ghost variable `positive` is declared on line 12, assigned a value on line 18 and read on lines 15 and 21.

```
1 public class Main {
2     /*@ ghost
3         ensures \result == (current && a >= 0);
4         boolean arrayIsPositive(boolean current, int a) {
5             return current && a >= 0;
6         }
7     */
8
9     //@ context_everywhere a != null;
10    //@ context_everywhere Perm(a[*], read);
11    public int sum(int[] a) {
12        //@ ghost boolean positive = true;
13        int result = 0;
14        //@ loop_invariant i >= 0 && i <= a.length;
15        //@ loop_invariant positive ==> (\forallall int j = 0 .. i; a[j] >= 0);
16        //@ loop_invariant (\forallall int j = 0 .. i; a[j] >= 0) ==> result >= 0;
17        for(int i = 0; i < a.length; i++) {
18            //@ ghost positive = arrayIsPositive(positive, a[i]);
19            result += a[i];
20        }
21        //@ assert positive ==> result >= 0;
22        return result;
23    }
24 }
```

Listing 3.37: Example of a ghost method and variable in VerCors.

**Pure ghost methods** Pure ghost methods are ghost methods that have been declared pure. In pure ghost methods the body must be a single expression with no side-effects. Pure ghost methods are useful, as they can be used in annotations that do not allow side-effects, such as pre- and post-conditions. The function `isPositive` in Listing 3.38 is an example of a pure ghost method.

```
1 /*@
2     ensures \result == a >= 0;
3     pure boolean isPositive(int a) = a >= 0;
4 */
```

Listing 3.38: Example of a pure ghost method in VerCors.

### 3.4.5.2 Ghost parameters and results

The header of a method can be extended with ghost parameters and result variables. In a contract, **given** `type param_name;` declares a ghost parameter `param_name` of type `type`, which can be used in pre- and post-conditions and is seen as an extra parameter for annotations inside the method body. In Listing 3.39, function `add` has a ghost parameter `min`, which must be greater or equal to zero right before `add` is executed. When a method with ghost parameters is called, all ghost parameters must be assigned by writing **given** `{param_name = value, ...}`; behind the invocation. In the function caller the ghost parameter `min` of function `add` is assigned the value 5.



In a contract, **yields** type result\_name; can be used in a contract to declare a ghost result result\_name of type type. A ghost result can be used in post-conditions and is assigned in annotations inside the method body. In Listing 3.39, function add has a ghost result all\_ge\_min. The post-condition states that if this ghost result is true, the function's return value must be greater or equal to min \* 2. When a method with ghost parameters is called, its ghost results can be assigned to ghost variables with **yields** {name = result\_name, ...}; behind the invocation. In the function caller the ghost result all\_ge\_min of the function add is assigned to the local ghost variable items\_ge\_5.

```

1 public class Main {
2
3     /*@
4     given int min;
5     yields boolean all_ge_min;
6     requires min >= 0;
7     ensures all_ge_min ==> \result >= min * 2;
8     */
9     public int add(int a, int b) {
10         /*@ ghost all_ge_min = a >= min && b >= min;
11         return a + b;
12     }
13
14     public int caller(int[] a) {
15         /*@ ghost boolean items_ge_5;
16         int result = sum(a) /*@ given { min = 5 } */ /*@ yields {
17                                     ↪ items_ge_5 = all_ge_min } */ ;
18         return result;
19     }
20 }

```

Listing 3.39: Example of declaration and usage of ghost parameters and results in VerCors.

### 3.4.5.3 Abstract methods

The body of a (ghost) method can be omitted if one is only interested in the pre- and post-conditions of the method, but not the actual implementation. Because the method has no body, VerCors cannot check if the post-conditions hold given the pre-conditions, so instead it assumes the post-condition is true. This is useful, as it speeds up the verification process. Moreover, it allows verification of programs where not all methods have been implemented yet, but their specifications are known already.

The function div and ghost function isPositive in Listing 3.40 are examples of abstract (ghost) methods.

```

1 public class Main {
2     /*@ ghost
3     ensures \result == a >= 0;
4     boolean isPositive(int a);
5     */
6
7     /*@ requires b != 0;
8     /*@ ensures \result == a / b;
9     public int div(int a, int b);
10 }

```

Listing 3.40: Example of declaration and usage of abstract (ghost) methods in VerCors.

#### 3.4.5.4 Inline functions

Inline functions can be seen as macros. Before analysing a program, VerCors replaces every invocation of an inline function with its body. This is mostly useful for long or complicated expressions that are used in many annotations. In such cases, the expression can be put in an inline function and the annotations then refer to the function instead, which makes the annotations easier to comprehend. Because VerCors replaces the function invocation with the actual body, it is as if the actual expression is still present in the annotations, rather than some specification about it. Only simple inline functions are allowed to be defined, so concepts such as recursion are not allowed. Inline functions are declared using the `inline` keyword, as can be seen in Listing 3.41.

```
/*@ inline pure boolean isPositive(int a) = a >= 0;
```

Listing 3.41: Example of a declaration of an inline function in VerCors.

#### 3.4.6 Predicates

Contracts can expose restricted information, such as what private variables another class contains. An example of this can be seen in Figure 3.42, where for the method `manageAccount` write permission needs to be declared for `Account`'s private variable `amount`, in order to execute the method call `acc.set(50)`. This means a private variable is exposed to another class, which might be undesirable. Moreover, if the implementation of the `Account` class changes, the contracts in other unrelated classes need to be changed as well. To give an example: if another private field is added for which the method `set` needs write permissions, the contract of `set` will change, which forces the contract for `manageAccount` to be changed as well. This might be undesired, because it means that even a small change in a contract requires changes in contracts in unrelated classes as well.

```
1 class Account {
2   private int amount = 0;
3
4   //@ context Perm(amount, 1);
5   //@ ensures amount == x;
6   public void set(int x) {
7     this.amount = x;
8   }
9 }
```

(a) Account class.

```
1 class Main {
2
3   //@ requires acc != null;
4   //@ context Perm(acc.amount, 1);
5   //@ ensures acc.amount == 50;
6   void manageAccount(Account acc) {
7     acc.set(50);
8   }
9 }
```

(b) Main class.

Figure 3.42: Demonstration of a private variable being exposed to another class through a contract.

To mitigate these undesired effects, users can define predicates in a class body using the following syntax: `resource name(S param1, T param2, ...) = body;`. A predicate can be seen as a function called `name` with parameters `param1` of type `S` and `param2` of type `T`, ... and function body `body`, which returns a *resource* (which is the same type that `Perm` has). The only difference is that predicates are not called, but folded and unfolded.

In Listing 3.43 on the next page, the predicates `pre_state` and `state` have been added to `Account`, and are used in the contract for `set` and `manageAccount`. This way, `manageAccount` cannot see `Account`'s private field `amount`, and if the implementation of `Account` changes, only the bodies of `pre_state` and `state` have to be altered.

```

1 class Account {
2     private int amount = 0;
3
4     //@ resource pre_state() = Perm(amount, write);
5     //@ resource state(int value) = Perm(amount, write) ** amount == value;
6
7     //@ requires pre_state();
8     //@ ensures state(x);
9     public void set(int x) {
10         //@ assume pre_state();
11         //@ unfold pre_state();
12         //@ assume Perm(amount, write);
13         this.amount = x;
14         //@ assume Perm(amount, write) ** amount == x;
15         //@ fold state(x);
16         //@ assume state(x);
17     }
18 }
19
20 class Main {
21     //@ requires acc != null;
22     //@ requires acc.pre_state();
23     //@ ensures acc.state(50);
24     void manageAccount(Account acc) {
25         acc.set(50);
26     }
27 }

```

Listing 3.43: Example of declaration and usage of a predicate in VerCors. *The verification statements written in pink do not actually appear in the user-defined code, but are added for demonstrative purposes.*

**Predicate folding** Predicates can be folded and unfolded to translate between an invocation of the predicate and its body. A predicate is folded using **fold** name(arg1, arg2, ...); and unfolded with **unfold** name(arg1, arg2, ...);, where name is the name of the predicate and arg1, arg2, ... are the arguments for the predicate's parameters. Folding and unfolding of a predicate can be seen as folding and unfolding a piece of paper with an invocation of the predicate written on the outside and the predicate's body written on the inside. When the paper is folded up, only the invocation of the predicate can be seen, and when the paper is unfolded, only its body can be seen. Figure 3.44 shows a visualization of this.

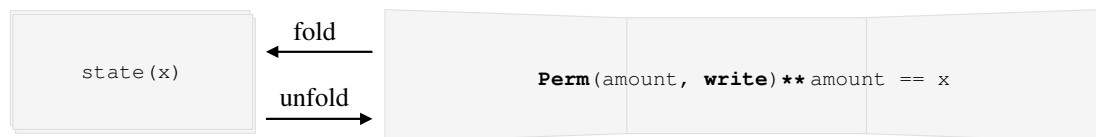


Figure 3.44: Visualisation of folding and unfolding a of predicate in Vercors.

An example of folding and unfolding can be seen inside the method `set` in Listing 3.43. For the statement `amount = x` on line 13, write permission for `amount` is required. Because of `set`'s pre-condition, we know that the start of the method's body that `pre_state` holds. However, we do not know if `pre_state` gives this write permission. Therefore, we unfold it, which reveals that there is write permission, so `amount = x`; is allowed to be executed. Now, in the post-condition, we see that `state(x)` must hold when the method terminates. However, we only know that `Perm(amount, write) ** amount == x` at the moment. So, to satisfy the post-condition, it needs to be folded up to `state(x)` before the method returns.

**Unfolding predicates in expressions** Sometimes it is desirable to use the body of a predicate in an expression to, for example, temporarily gain permission to access a certain field. This can be done by writing `\unfolding name(arg1, arg2, ...) \in expression;`, where `name` is the name of the predicate and `arg1, arg2, ...` are the arguments for the predicate's parameters. This statement means the same as the following:

```
/*@ unfold name(arg1, ...); */ expression /*@ fold name(arg1, ...); */
```

Listing 3.45: Meaning of unfolding a predicate for an expression in VerCors.

**Scaling and splitting** Sometimes a permission is split up into multiple permissions. This is also possible for permissions inside predicates, where one can take a fraction of a predicate. When one has a fraction  $f$  of a predicate, it means that for each permission declaration in the predicate, you get a fraction  $f$  of it. A fraction  $f$  of a predicate is denoted the following way: `[f]name(arg1, arg2, ...)`, where `name` is the name of the predicate and `arg1, arg2, ...` are the arguments for the predicate's parameters.

**Inline predicates** An inline predicate is declared in a similar fashion as a predicate, but prepended with the keyword `inline`. When an inline predicate is used, it is always unfolded. One downside to inline predicates is that they cannot be recursive.

**Recursion** Predicates are allowed to be recursive, as long as they are not inline.

## 4 Related works

Three categories of works related to the goal of the thesis can be identified: SYCL verification tools, publications about VerCors, and front-ends for Viper. These are discussed in this chapter.

### 4.1 SYCL verification tools

There do not seem to exist any deductive verification tools for SYCL programs, nor any tools which offer some other form of formal verification for SYCL programs.

### 4.2 Publications about VerCors

This category consists of publications about VerCors by authors that have worked on VerCors' implementation or used VerCors to model certain constructs. The publications that have been examined can be divided into two subcategories: The first subcategory consists of publications that describe how certain heterogeneous programming constructs are verified in VerCors. The second subcategory consists of additions to VerCors' support for non-heterogeneous constructs. VerCors' project structure has changed a few times over the years, most recently in September 2022. Therefore, to get an up-to-date view of the state of the VerCors project, the second subcategory only consists of more recent publications, from 2020 onwards.

#### 4.2.1 Verification of heterogeneous programming constructs in VerCors

At the time of writing, VerCors supports three heterogeneous programming languages, namely OpenCL, CUDA, and OpenMP. Several publications have been written about how these heterogeneous programming languages are supported by VerCors. For OpenCL and CUDA, VerCors supports reasoning about kernels. For OpenMP it does not support reasoning about code that runs on different devices, but it does support OpenMP constructs which allow parts of code to be executed with the SMID execution model instead of the CPU's normal execution model. Therefore, publications about OpenMP are still considered related works as this does meet the definition of heterogeneous computing (see subsection 3.1.1 for the definition).

In the papers [13] and [27] it is explained how VerCors' supported subset of OpenMP can be translated into an intermediate representation language called PPL (Parallel Programming Language). It also briefly touches upon how PPL is encoded into a Viper IL in VerCors. The authors of [63] explain how a CUDA program can be transformed into a PVL program. In [7], [14], and [27] it is explained how to reason about OpenCL and CUDA kernels. [14] focusses on kernels that contain barriers, whereas [7] focusses on kernels that contain atomics. [27] address on both. First, an extension of KPL (Kernel Programming Language) is described in these papers, which is used to formalize kernels. Then a verification method is explained which can be used to verify specifications about kernels. The author of [27] further describes that OpenCL programs are encoded into PVL, which is then encoded into Viper IL. The authors [14] described VerCors' previous, now outdated, translation process where OpenCL is encoded into Chalice code, instead of Viper IL. The paper [12] gives a short summary of how kernels are verified in VerCors.

One thing to note about these related works is that they describe that heterogeneous programming constructs are translated into PPL, KPL, or PVL, which is then translated to Viper IL. However, in the actual implementation (as described in subsection 3.4.1) a program is translated directly to COL, which is then translated to Viper IL. Nevertheless, these works were included in this subsection as it does give insights in the thought processes behind the implementation of VerCors’ support for heterogeneous programs.

#### **4.2.2 Additions to VerCors’ support for non-heterogeneous constructs.**

Only four works about additions to VerCors’ support for non-heterogeneous constructs have been written between 2020 and the time when work on this thesis was started. In [64] it is described how an ADT for maps was added to VerCors’ axiomatic data types, and how functionality of existing ADTs was extended. An implementation-level overview of VerCors’ verification process is also given, and so are descriptions of five different general approaches to implementing a feature in VerCors. The author of [61] and [62] discusses how support for Java exceptions was added to VerCors, and how Java’s inheritance could be implemented in the future. Two design guidelines for the implementation of new features were also described. In [11] it is briefly described how support was added to VerCors to verify JavaBIP annotations.

### **4.3 Front-ends for Viper**

As mentioned before, VerCors uses Viper as its back-end. This means that VerCors can be seen as a front-end for Viper. This begs the question on whether there exist any other front-ends to Viper. If this were the case, then some inspiration could be taken from how certain heterogeneous programming constructs are translated to Viper IL in those other front-ends. There exist Viper front-ends for many programming languages, such as: Nagini for Python [29], Gobra for Go [78], Prusti for Rust [9], Soothsharp for C# [34], and Scala2Silver (formerly known as Scala2SIL) for Scala [17]. However, there do not seem to exist any Viper front-ends for programming languages targeted at heterogeneous computing.

## 5 Added support for basic C++ constructs

The first research question (RQ1, see subsection 2.1 (page 6)) states that basic C++ constructs should be supported, as all of SYCL’s features are expressed using C++ constructs. To be able to support C++ at all, support for basic C++ constructs first had to be added to VerCors’ initial phases. This added support is discussed first. The C++ constructs that should be supported by VerCors can be split into two categories: constructs for which VerCors already supports their C-equivalent constructs, and constructs for which this is not the case. The added support to VerCors for both categories is described in this chapter as well.

### 5.1 Added C++ support to VerCors’ initial phases

To be able to support C++ at all, support for basic C++ constructs first had to be added to VerCors’ parsing, conversion to COL, and resolution phases. In order to take C++ files as input, VerCors requires an ANTLR lexer and parser for C++ in its parsing phase. A lexer and parser for C++ 14 were found in a collection of ANTLR v4 grammars on GitHub [76]. This lexer and parser were copied, with some minor adjustments to generate more suitable ASTs, to VerCors’ parsing phase. Since the lexer and parser for C++ creates a similar AST as VerCors’ lexer and parser for C, the C implementation of the conversion to COL AST could be copied and slightly altered to implement the conversion of C++ ASTs to COL. The C implementation of the resolution phase could be copied and slightly altered as well to form the base of C++’s resolution phase.

### 5.2 Constructs with implemented C-equivalent constructs

At the time of implementing support for C++ constructs, VerCors already had support for basic C constructs. Because C++ was originally created as an extension of C, many of the basic constructs in C++ are inherited from C’s basic constructs [56]. This means that for many required C++ constructs, the implementation for their equivalent C constructs could simply be copied and slightly extended to implement support for them. In this section, the C++ constructs for which the implementation of a C-equivalent construct could be copied are presented, and the extensions that were made to the copied implementations are discussed.

#### 5.2.1 Logical expressions

Support was added for the logical expressions `!a`, `a && b`, `a || b`, `a == b`, `a != b`, `a < b`, `a <= b`, `a > b`, and `a >= b`.

#### 5.2.2 Arithmetic expressions

Support for the arithmetic expressions `a + b`, `a - b`, `a * b`, `a / b`, and `a % b` was implemented. Additionally, support for the following equivalent arithmetic expressions, but with side-effects, was implemented: `a++`, `a--`, `a += b`, `a -= b`, `a *= b`, `a /= b`, and `a %= b`.

#### 5.2.3 Primitive types

Support was added for the types `bool`, `int`, `long`, `double`, `float`, `char`, and `void`. C does not have the boolean values `true` and `false` (although VerCors provides a header file that assigns the values 1 and 0 to them respectively). However, a subset of Java, which does have boolean values, is supported by VerCors, so the support of the `bool` type was copied from VerCors’ Java implementation.

### 5.2.4 Arrays

Arrays were implemented for C++ to the same extent as they were implemented for C at the time of writing. This means that local arrays can be declared without an initializer, with the following notation: `int a[7]`. If the array is one-dimensional, it can also be declared with a literal array as initializer, with the following notation: `int a[] = {4, 5, 6}`. Furthermore, the elements of arrays can be read and re-assigned, and arrays can be passed as parameters to other methods.

### 5.2.5 Pointers

Support for pointers was also implemented to the same extent as the C implementation. This means that pointers can be declared inside methods and be parameters, but the `&` operator cannot be used. However, SYCL command group lambda methods, which need to be supported in VerCors, require a `&` parameter. To support this parameter, a workaround was added: when a (lambda) method parameter contains a single addressing operator `&`, the parameter is processed as if `&` was not there. This workaround might not be entirely accurate behaviour-wise, as using `&` in a method parameter could change it from pass-by-value to pass-by-reference. However, it avoided having to fully implement addressing in C++, which was considered out of scope for this thesis. To minimize inaccuracy, only `&` parameters of type `sycl::handler` are supported, as only support for such parameters is required to support the required SYCL features.

### 5.2.6 Variables

Declaring, assigning, and referencing of variables is supported in local and global scopes. C++'s visibility and reassigning rules, with regards to what scope the variable is (reassigned) in, are also supported. To give an example, in Listing 5.1, variable `a` is reassigned in an inner scope on line 5, which permanently changes the value of `a`. In that same inner scope, `b` is re-declared on line 6, which, just as in C++, only changes the value of `b` inside that scope.

```
1 void test() {  
2     int a = 10;  
3     int b = 10;  
4     {  
5         a = 20;  
6         int b = 20;  
7         //@ assert a == 20 && b == 20;  
8     }  
9     //@ assert a == 20 && b == 10;  
10 }
```

Listing 5.1: Example of scoping inside a method.

### 5.2.7 Control structures

In terms of support for control structures, support was implemented for all variations of if-statements and while-loops. Support was also added for for-loops, but only for for-loops declared with an init-statement (may be empty), a condition (may be empty), and an iteration-expression (may be empty), as can be seen below. Support for range-based for-loops was not added because they are also not supported in the C implementation and are mostly syntactic sugar, so not considered a necessity.

```
for (int i = 0; i < max; i++) { /* loop body */ }
```



## 5.2.8 Methods

Support was added for the declaration of C++ methods in the form of:

```
1 type methodname(type param1, type param2, ...) {  
2     statement1;  
3     statement2;  
4     ...  
5 }
```

and abstract methods in the form of:

```
type methodname(type param1, type param2, ...);
```

These method declarations are also allowed to be recursive. They are also allowed to be declared as VerCors ghost methods, and all other VerCors annotations described in subsection 3.4.2 (page 38), subsection 3.4.4.1 (page 43), subsection 3.4.5 (page 45), and subsection 3.4.6 (page 48) are also supported.

In C, overloading of methods is not supported. However, it is supported in C++ and used for some SYCL methods. Therefore, support for C++ method overloading was added, by implementing the following: When resolving a method invocation in the resolution phase, the implementation searches for a method declaration with not only the same name as the invocation, but also with the same parameter types as the invocation's arguments. This approach is similar to the already existing implementation for resolving Java methods, which, excluding the Java-specific conditions, only matches methods with the same name and parameter types as the invocation's name and arguments as well.

VerCors' C implementation, and thus also its C++ implementation, allow two methods to be created in the same scope with the same name and parameters, i.e. with an identical signature. Then when a user invokes one of those methods, the one that was declared first is invoked. This is not allowed in C++. However, as long as the user only verifies C++ programs in VerCors that can be compiled without errors by a C++ compiler, this disparity in behaviour should not be an issue.

## 5.3 Constructs without implemented C-equivalent constructs

There are also some C++ constructs for which there were no C-equivalent constructs, or for which the C-equivalent constructs were not supported in VerCors. This meant that these constructs had to be implemented from scratch. How these constructs were implemented is described in this section.

### 5.3.1 Namespaces

Support was added for declarations and usage of basic namespaces, as many required SYCL methods and classes are declared inside namespaces, and thus also accessed with the namespace-reference notation, using the `::` operator. Namespaces are also used in the implementation of support for some SYCL class methods (see subsection 5.3.2). However, in both use-cases, only support for method declarations inside namespaces was necessary. Therefore, it was chosen to only support method declarations inside namespaces and to not support other constructs, such as namespace fields. The implementation of namespace support is described in the next two paragraphs.

**Transformation of namespace declarations** Namespaces are transformed in the parsing phase. The parser stores a list of names that should be prepended to method names. When coming across a method declaration, all the names in this list are prepended to the method name. When coming across a namespace declaration in the parser, the parser stores the name of that namespace in that list, causing this name to be appended to every method declaration it comes across inside the namespace. This also works for nested namespaces, as the prepended names are a list, so the nested namespace can simply be appended to it. Then, instead of nesting the method declarations inside namespaces, they are declared in the same scope as methods outside of namespaces, but with their prepended name.

To give an example, the method declarations in Figure 5.2a are transformed to the method declarations in Figure 5.2b in the parsing phase.

```

1 void method1();
2 namespace a {
3     void method2();
4     namespace b { void method3(); }
5 }

```

(a) Method declarations outside and inside (nested) namespaces.

```

1 void method1();
2
3 void a::method2();
4
5 void a::b::method3();

```

(b) Method declarations from Figure 5.2a after the parsing phase.

Figure 5.2: Examples of how method declarations are transformed in the parsing phase.

**Resolving references to namespace methods** The transformed namespace methods can be resolved by their name, using the same method as resolving methods outside of namespaces. When calling a method inside a namespace, for example, when calling `method2` from Figure 5.2a, one would write `a::method2()`. Instead of looking for a method inside a namespace, a search is done for a method with the name `a::method2`, which exists, as can be seen in Figure 5.2b.

### 5.3.2 Class methods

Support for class instance methods is required, as most SYCL methods that are required for working with kernels are class instance methods. However, adding full support for classes was considered out of scope for two reasons: First of all, there does not exist support for classes in C, so support for classes would have to be implemented from scratch. Secondly, a simpler solution could be found, and since the main goal of this thesis is to add support for SYCL features, and not for C++ features, a simpler solution for C++ features was preferred. This solution consists of two parts: the addition of a header file and a method to resolve class instance methods in the resolution phase.

**Addition of a header file** When a user includes a header file in their C++ code, VerCors automatically searches for a header file with the same name in its own folders. If it finds such a file, it adds the code in that header file at the top of the file the user wants to verify. Therefore, most SYCL (class instance) methods supported by VerCors were manually added as abstract methods to a header file called `sycl.hpp`. This filename is the same as the name of the actual SYCL header file, which is often already imported by the user to be able to use SYCL in C++ outside of VerCors.

Because classes in C++ are not supported, they cannot be added to the SYCL header file in VerCors in that form. Therefore, classes are instead added as namespaces with the same name. An example of this can be seen in Listing 5.3 for the SYCL class `queue`, which has an instance method called `submit`. As can be seen, the `queue` class has been added as namespace to the header file.

```

1 namespace sycl {
2     namespace queue {
3         sycl::event submit(...);
4     }
5
6     ...
7 }

```

Listing 5.3: Excerpt from the `sycl.hpp` header file in VerCors. *The method parameters have been left out for brevity.*

**Resolving references to class instance methods** When calling instance methods, they need to be resolved. Since all classes are namespaces in the SYCL header file, they are resolved differently than static methods. When coming across an method call to a class instance method, for example `myQueue.submit` on line 5 in Listing 5.4, two resolving steps are done. First, the classname of the class instance on which the method is called is resolved. In the example, `myQueue` is an instance of the `sycl::queue` class. The resolved classname is then used in the second resolving step, which searches for a method with the same name as the instance method being called, but prepended with the previously resolved classname. In the example, this means the method being searched is `sycl::queue::submit`. And this method exists in the transformed version of the header file seen in Listing 5.3.

```

1 #include <sycl/sycl.hpp>
2
3 void main() {
4     sycl::queue myQueue;
5     myQueue.submit(...);
6 }

```

Listing 5.4: Example of calling an instance method. *The method parameters have been left out for brevity.*

### 5.3.3 Class constructors

Support for invocation of class constructors was also implemented, to let the user construct SYCL objects. Only the syntax that can be seen below is supported:

```

type(type param1, type param2, ...);
type<generic_arg1, ...>(type param1, type param2, ...);

```

When a method invocation is found, and it does not match any method in the SYCL header nor any user-defined method, it is checked if the name of the constructor matches any supported SYCL type. If this is the case, it is checked whether the (generic) parameter types match any of the supported SYCL constructors. When this is true, a constructor invocation is generated and encoded in the transformation phase into the appropriate COL code.

Constructors were not added to the SYCL header file as some constructors take a generic arguments. To support such constructors, support for C++ templates would have to be added, which would have been more complex than the solution stated above.

### 5.3.4 Lambda methods

Some SYCL methods take a lambda method as one of their arguments. Therefore, it was decided to partially support lambda methods. All lambda methods in C++ code are accepted in the parsing phase and reach the transformation phase. Then, if the lambda method is a parameter for a SYCL method that was chosen to be supported in VerCors, the lambda method is encoded to what is needed for that SYCL method. For example, the lambda method parameter of SYCL's `queue.submit` method is encoded to a COL class, as described in subsection 6.1.3. Then, for any lambda methods that were not encoded, i.e. all lambda methods that are not an argument of a supported SYCL method call, an error is thrown that such lambda methods are not supported.

Note should be taken captures, such as `[=]` and `[&]`, which define what variables declared outside the lambda expression are accessible in what way inside the lambda body, are ignored. This avoids the complexity of distinguishing between pass-by-value and pass-by-reference variables. Instead, the capture value is assumed to have a certain value, based on what SYCL method the lambda is an argument for. These assumptions are documented in the sections that cover those SYCL methods.

## 6 Added support for SYCL's basic and ND-range kernels

The second research question (RQ2, see subsection 2.1 (page 6)) states that SYCL's basic and ND-range kernels should be supported, as kernels are a critical feature for heterogeneous computing. In this chapter, the added support to VerCors for SYCL's kernel declarations, kernel executions and related methods is discussed.

### 6.1 Declaration and execution of kernels

This section first shares the definitions of the SYCL methods related to kernel declarations for which support was added to VerCors. Then the semantics and the encoding into VerCors for the declaration and execution of kernels is discussed.

#### 6.1.1 Definitions of supported methods

The definitions of SYCL's `submit` and `parallel_for` methods that are supported by VerCors are described below. There are two versions for the `parallel_for` method, one that declares a basic kernel, and one that declares an ND-range kernel. After that, the constructors for the `range` and `nd_range` classes are shown. These definitions are simplified versions of the method descriptions in the SYCL specification [72] (on pages 93, 247, 250, 286, and 288).

```
submit(T lambda) → sycl::event
```

**In class:** `sycl::queue`

**Arguments:**

- `lambda`: a lambda method with a `sycl::handler` object as parameter.

**Returns:** An event which is linked to the command group definition in `lambda`.

**Description:** Submits the command group definition in `lambda` to the queue, which the queue then executes as soon as possible.

```
parallel_for(sycl::range<S> range, T lambda) → void
```

**In class:** `sycl::handler`

**Arguments:**

- `range`: a `sycl::range` object. This determines the dimensions of the kernel.
- `lambda`: a lambda method with a `sycl::item` object as parameter. This will be the body of the kernel.

**Description:** Declares a basic kernel inside the command group, with the `lambda` method as body and an index-space as described by the `range` parameter.

```
parallel_for(sycl::nd_range<S> range, T lambda) → void
```

**In class:** `sycl::handler`

**Arguments:**

- `range`: a `sycl::nd_range` object. This determines the dimensions of the kernel's index-space.
- `lambda`: a lambda method with a `sycl::nd_item` object as parameter. This will be the body of the kernel.

**Description:** Declares an ND-range kernel inside the command group, with the `lambda` method as body and an index-space as described by the `range` parameter..

```
range(int dim0) → sycl::range<1>
```

```
range(int dim0, int dim1) → sycl::range<2>
```

```
range(int dim0, int dim1, int dim2) → sycl::range<3>
```

**In class:** `sycl::range`

**Arguments:**

- `dim0, ...`: One, two, or three integers that indicate the number of work-items in each dimension.

**Returns:** A `range<1>` object with one dimension if one parameter, a `range<2>` object with two dimensions if two parameters, etc.

**Description:** Declares a `range` instance with the number of parameters determining the number of dimensions and the parameters themselves indicating the number of work-items in each dimension.

```
nd_range(sycl::range<S> globalRange, sycl::range<S> localRange)  
→ nd_range<S>
```

**In class:** `sycl::nd_range`

**Arguments:**

- `globalRange`: A range object indication the total number of work-items in each dimension.
- `localRange`: A range object indication the number of work-items per work-group in each dimension.

**Returns:** An `nd_range<S>` object with the same dimensions as its `sycl::range<S>` parameters.

**Description:** Declares an `nd_range` instance with the parameters determining the total number of work-items and the number of work-items per work-group.

## 6.1.2 Semantics

As mentioned in subsection 3.2.1 (page 18), kernels are declared inside command groups. And, as mentioned in subsection 3.2.2.1, to execute the kernel, its command group must be submitted to a queue. The most straight-forward way to declare and execute a kernel in SYCL, is to do both of these actions at the same time, using the pattern shown in Listing 6.1. In the host code (indicated with yellow), a queue's `submit` method is called, which immediately submits its argument to the queue. This argument is a lambda method, such as the one on lines 4-11, which describes the command group (indicated with blue). Inside the command group declaration, on line 5, the `parallel_for` method is called on the `handler` object it was given. It takes as first argument a `range` object (in case of a basic kernel, or an `nd_range` object, in case of an ND-range kernel) that describes the kernel's index-space: how many dimensions the index-space has, and how many work-items exist in each dimension. Its second parameter is a lambda method describing the kernel body, as indicated with red on lines 7-9. As can be seen in Listing 6.1, the kernel body lambda method has as argument an `item` object (in case of a basic kernel, or an `nd_item` object, in case of an ND-range kernel), which can be used to query a work-item's id and the size of each dimension in the index-space.

```

1  sycl::queue myQueue;
2
3  sycl::event myEvent = myQueue.submit(
4      [&](sycl::handler& cgh) {
5          cgh.parallel_for(
6              sycl::range<2>(6,3),
7              [=] (sycl::item<2> it) {
8                  // kernel body
9              }
10         );
11     }
12 );

```

Listing 6.1: Example creation and execution of a basic kernel in SYCL. *The colours indicate the scope of the code, using the same colours as in Figure 3.11 (page 18).*

#### 6.1.2.1 Other kernel declaration methods

There exist a few other ways to declare and execute kernels as well. However, these methods were considered out of scope for this thesis because each declaration method has a different structure. These differences in declaration structure mean that there would be differences in VerCors' implementation for them as well. These differences would create a snowball effect in implementation-effort when taking into account that support is added for data- and local accessors in this thesis as well, which are declared inside these different kernel declarations. Therefore, it was decided to only support one declaration structure, i.e. the declaration structure above, which is used in various SYCL examples, and appears to be the simplest way for the user to declare a kernel.

#### 6.1.2.2 Failing to enqueue or to execute a command group

In the SYCL specification (page 281), it is stated that enqueueing a command group, or executing it, can fail. However, it is unspecified in what situations it would fail and what happens in such cases. Therefore, in this thesis the assumption is made that enqueueing a command group to a queue or executing it cannot fail if the user has declared a valid command group. Here 'valid' means that the command group declaration passes the checks mentioned throughout this chapter, such as not having negative range dimensions, and global range dimensions being divisible by local range dimensions in `nd_range` objects.

### 6.1.3 Encoding into VerCors

In this subsection, it is explained how the declaration and execution of kernels are encoded into VerCors. First, the general idea of the implementation is described, and then each step of the encoding is explained in further detail in their own subsubsections.

#### 6.1.3.1 The general idea

Most of the encoding of declarations and executions of SYCL kernels was implemented by extending the transformation pass for C++ in VerCors' transformation phase. In this pass, invocations of SYCL's `submit` method are treated differently from normal method invocations. The transformation pass searches inside the command group argument of a `submit` invocation for a `parallel_for` method invocation, which contains the index-space and kernel declaration.

These detected components are then encoded into COL code. In Listing 6.4, a general template is shown which indicates what components of the command group submission, as shown in Figure 6.2, are encoded into what components in COL. These newly generated COL components are grouped into five sections, for which the legend is shown in Figure 6.3.

```

1  someType someMethod() {
2      queueObject.submit(
3          [&](sycl::handler& handlerObject){
4              handlerObject.parallel_for(
5                  range object,
6                  /*@ kernel contract */
7                  [=] (item parameter){
8                      // kernel body
9                  });
10     });
11 }

```

Figure 6.2: General pattern of submitting a basic kernel to a queue in SYCL.

$\alpha$	<i>event-class</i>
$\beta$	<i>kernel-runner</i>
$\gamma$	<i>kernel-parblock</i>
$\delta$	<i>event-constructor</i>
$\epsilon$	<i>host code</i>

Figure 6.3: Legend of generated COL sections for kernels.

```

1   $\alpha$       class SYCL_EVENT_CLASS {
2   $\alpha$       int dimSize; // for every dimension
3   $\alpha$ 
4   $\alpha$   $\beta$       context Perm(this.dimSize, read); // for every dimension
5   $\alpha$   $\beta$       context this.dimSize >= 0; // for every dimension
6   $\alpha$   $\beta$       quantified kernel contract
7   $\alpha$   $\beta$       run {
8   $\alpha$   $\beta$   $\gamma$           par SYCL_BASIC_KERNEL(
9   $\alpha$   $\beta$   $\gamma$               int iterVar = 0 .. this.dimSize // for every dimension
10  $\alpha$   $\beta$   $\gamma$           ) context Perm(this.dimSize, read); // for every dimension
11  $\alpha$   $\beta$   $\gamma$           context this.dimSize >= 0; // for every dimension
12  $\alpha$   $\beta$   $\gamma$           kernel contract {
13  $\alpha$   $\beta$   $\gamma$               kernel body
14  $\alpha$   $\beta$   $\gamma$           }
15  $\alpha$   $\beta$       }
16  $\alpha$       }
17
18  $\delta$       requires dimension checks;
19  $\delta$       ensures \result != null;
20  $\delta$       ensures \typeof(\result) == \type(SYCL_EVENT_CLASS);
21  $\delta$       ensures Perm(\result.dimSize, read); // for every dimension
22  $\delta$       ensures \result.dimSize == dimSize; // for every dimension
23  $\delta$       ensures idle(\result);
24  $\delta$       SYCL_EVENT_CLASS event_constructor(dimension sizes);
25
26  $\epsilon$       someType someMethod() {
27  $\epsilon$           SYCL_EVENT_CLASS sycl_event_ref;
28  $\epsilon$           sycl_event_ref = new SYCL_EVENT_CLASS(dimension sizes);
29  $\epsilon$           fork sycl_event_ref;
30  $\epsilon$       }

```

Listing 6.4: A general template that shows what components of a basic kernel submission are encoded to what components in COL.

The kernel declarations in `parallel_for` invocations are encoded similarly to how CUDA kernel declarations are encoded [63]: they are encoded as a single nested parblock (as described in subsection 3.4.3.2) for basic kernels, and a double nested parblock for ND-range kernels. Parblocks can run a block of code in parallel over a range of work-items, which is similar to the behaviour of SYCL and CUDA kernels. This means that SYCL kernel declarations can be encoded into a COL AST with similar behaviour with components that already existed in VerCors. A basic kernel declaration together with its index-range is encoded into the *kernel-parblock* shown in Listing 6.4. An ND-range kernel declaration together with its index-range is encoded into the *kernel-parblock* shown in Listing 6.5.

```

1  α      quantified over all work-groups kernel contract
2  α β    run {
3  α β γ   par SYCL_ND_RANGE_KERNEL_WORKGROUPS (
4  α β γ   work-group ranges in index-space
5  α β γ   ) context Perm(this.dimsizes, read); // for every dimension
6  α β γ   context this.dimsizes >= 0; // for every dimension
7  α β γ   quantified over work-items in work-group kernel contract {
8  α β γ   par SYCL_ND_RANGE_KERNEL_WORKITEMS (
9  α β γ   work-item ranges in index-space
10 α β γ   ) context Perm(this.dimsizes, read); // for every dimension
11 α β γ   context this.dimsizes >= 0; // for every dimension
12 α β γ   kernel contract {
13 α β γ   kernel body
14 α β γ   }
15 α β γ   }
16 α β    }

```

Listing 6.5: A general template that shows the encoding of an ND-range kernel submission into a *kernel-runner* and *kernel-parblock* in COL.

One particularity in SYCL is that when a kernel has been submitted to a queue and is running, the host code continues executing the commands that come after that kernel's submission. This behaviour is encoded with VerCors' fork/join mechanism (as described in subsection 3.4.3.3), which allows code in a class' run-method to be executed at the same time as the code it is forking from. This is why the generated *kernel-parblock* for the kernel declaration is placed inside a run-method in a class, which is forked from the encoded host code. This encoding can be seen in Listing 6.4, where the generated *kernel-parblock* resides inside a *kernel-runner*, which resides in the *event-class*. Furthermore, in the *host code*, an instance of the *event-class* is made by invoking the *event-constructor* with the dimension sizes of the kernel, which is then forked on line 21. This encoding allows a clear separation between code executed on the host and code executed on a device as well, as all code executed on the device is encapsulated by the *event-class*.

To enable users to actually reason about a SYCL kernel in VerCors, contracts can be added to kernel body declarations, as can be seen in orange on line 6 in Figure 6.2. These contracts are then added above the *kernel-parblock*, as can be seen in orange on line 6 in Listing 6.4. This contract, but quantified over all work-items, is also added to the *kernel-runner* in order to verify whether it can be forked from the *host code*. It also allows the post-conditions given by the user to be assumed in the *host code* when the *kernel-runner* is joined (which will be discussed in section 6.2).

### 6.1.3.2 Detecting the submission of a kernel

Detecting when a kernel is submitted to a queue is quite trivial. As mentioned in subsection 5.3.2 (page 56), supported SYCL methods are declared in a header file, and their names are prepended with the names of the namespaces in which they reside. SYCL's `submit` method is declared in this header file under the namespaces `sycl` and `queue`, since in reality the `submit` method is an instance method of the `queue` class, which resides in the `sycl` namespace. This means that internally in VerCors, the name of the `submit` method is `sycl::queue::submit`. This means that when encoding a method invocation, if the called method's name is equal to `sycl::queue::submit`, it is known that SYCL's `submit` method is being called and thus that a kernel is being submitted to a queue.

### 6.1.3.3 Finding the kernel declaration

When the `submit` method has been detected, the body of its lambda method argument is searched for invocations of a method with the name `sycl::handler::parallel_for`. When such an invocation has been found, the implementation moves onto the next steps, which are described in the next subsections.



**Allowed number of kernel declarations** SYCL only allows one kernel to be declared per command group at runtime. In cases where more than one is declared, a runtime exception should be thrown. It further states that a command group can statically contain more than one kernel declaration, as long as at runtime only one of them is executed. ([72], page 280) However, the choice was made not to support the latter in VerCors, as statically verifying that only one kernel is declared at runtime can be quite complex. Therefore, regardless of how many declared `parallel_for` invocations would actually be invoked at runtime, an error is displayed to the user if more than one `parallel_for` invocation is found in the command group scope.

SYCL also allows submission of command groups without a kernel declaration inside them, but the specification is unclear about what happens in such situations. Therefore, it was decided to throw an error when no `parallel_for` invocation is found, to avoid having to handle undefined behaviour.

**Allowed code inside command groups** In SYCL, it is allowed for the command group scope to contain code other than invocations of the `parallel_for` method. Allowing this in VerCors would be hard, as the code would have to be added into one of the various parts a kernel is encoded into. The code could be added in the *host code* before the *event-constructor* is called, inside the *event-constructor*, or at the start of the *kernel-runner*. Each position has its pros and cons, however none of the positions can cover all usages of a variable declared in the command group. For example, if a variable is used in the range parameter of the `parallel_for` method and used in the kernel body, it must be present in both the *event-constructor* and the *kernel-runner* to be able to use it.

However, most non-SYCL code put inside a command group scope could be placed in the host code right before the invocation of the `submit` method as well, so allowing non-SYCL code other than `parallel_for` invocations did not seem a necessity. For that reason, the choice was made to avoid the complexity of where to place the other code in the encoding and to simply not support it. Instead, when non-SYCL code is detected in the command group scope, an error message is shown to the user that it is unsupported.

For the same complexity reason not all SYCL code is supported by VerCors in command groups either: the only code that is allowed are invocation of the `parallel_for` method, and declarations of data- and local accessors (which are covered in section 7.2 (page 82) and section 8.2 (page 91)). These are allowed as the position of their placement in the encoding is clear. For all other SYCL-code inside command groups an error message is shown to the user that it is unsupported.

**Local variable usage** In SYCL, local variables that have been declared outside the command group scope of the `submit` method invocation are allowed to be used inside the command group scope. However, this is not possible in VerCors for the following reason: The entire kernel scope is encoded to an *event-class*. The *event-class* has its own scope, from which variables in the host code cannot be seen, as they are encoded into *host code*. This could be solved by detecting what local variables are used and copying them to fields in the *event-class*. However, this also requires permissions to read those fields and statements about the values of the fields to be added as well. Moreover, the values of the fields are influenced by whether the lambda method has a [=] or [&] capture. To avoid this complexity, it was decided to let VerCors throw an error that it cannot find the local variables. If a user does want to use those local variables inside the command group lambda, they can create a data accessor (see section 7.2 (page 82)) for it and access the local variable through it.

The only exception to this rule is that local variables declared outside the command group scope are allowed in the range parameter of invocations to the `parallel_for` method. These are allowed because it allows user to, for example, state that kernel range is the same as the range of the array the kernel will process. The range parameter is put as several arguments in the invocation of the *event-constructor*, which is invoked in the host code where the local variables are in scope. Also, because a given range object and its dimension parameters cannot be altered, the value of the lambda capture does not have any effects on this usage of local variables.

### 6.1.3.4 Processing the kernel declaration

When the `parallel_for` method invocation has been found inside the command group declaration, the type of its first parameter, which contains the index-space, is checked. This argument also determines which `parallel_for` constructor from subsection 6.1.1 is called, which determines whether a basic or an ND-range kernel is being declared. If the type of the first argument is a `range`, the kernel in the second argument is expected to be a basic kernel lambda with an `item` parameter, and if it is an `nd_range`, the kernel in the second argument is expected to be an ND-range kernel lambda with an `nd_item` parameter. If the expected kernel type and the argument of the lambda method in the second argument do not match, an error is shown to the user.

To illustrate how kernel declarations are encoded, the basic- and ND-range kernels in Listing 6.6 and Listing 6.7 respectively will be used as examples in this subsection.

```

1  cgh.parallel_for(
2      sycl::range<2>(6,4),
3      /*@
4          requires 1 + 2 == 3;
5          ensures 3 * 5 == 15;
6      */
7      [=] (sycl::item<2> it) {
8          // kernel body
9      }
10 );

```

Listing 6.6: Example of a `parallel_for` invocation with a basic kernel.

```

1  cgh.parallel_for(sycl::nd_range<2>(
2      sycl::range<2>(6,4), sycl::range<2>(2,2)
3  ),
4      /*@
5          requires 1 + 2 == 3;
6          ensures 3 * 5 == 15;
7      */
8      [=] (sycl::nd_item<2> it) {
9          // kernel body
10     }
11 );

```

Listing 6.7: Example of a `parallel_for` invocation with an ND-range kernel.

**Encoding of range dimensions** As mentioned previously, the dimensions of the index-space parameter of `parallel_for` methods are allowed to be expressions using local variables from the host code. This means that the dimension expressions cannot be directly inserted into the *event-class*, as it has a disjoint scope from the *host code*. To solve this, a field is added to the *event-class* for every dimension in the index-space. These fields are initialized to the dimension expressions by the *event-constructor*, which is called from the *host code* with the dimension expressions as arguments. To be able to read the range dimension fields in the *event-class*, read permission to the fields are ensured by the *event-constructor* and required and ensured by the *kernel-runner* and *kernel-parblock*.

The constructor generated for the kernel declaration in Listing 6.6 where the index-space has dimensions 6 and 4, can be seen in Listing 6.8. The constructor ensures on lines 4-5 that the values of the fields `dim0` and `dim1` in the *event-class* equals its parameters, so they will be 6 and 4 respectively. The constructor generated for the kernel declaration in Listing 6.7, where the index-space has global dimensions 6 and 4 and local dimensions 2 and 2, can be seen in Listing 6.8. The constructor ensures on lines 5-7 that the values of the field `group_dim0`, which represents of number of work-groups in the first dimension is the division of its parameters `global_dim0` and `local_dim0`, which will be 6 and 2, so `group_dim0` will be  $6/2 = 3$ , and that the field `local_dim0`, which represents the number of work-items in the first dimension equals its `local_dim0` parameter, which will be 2. Using the same reasoning for the second dimension, `group_dim1` will be  $4/2 = 2$  and `local_dim1` will be 2.

```

1  requires dim0 >= 0 && dim1 >= 0;
2  ensures \result != null && \typeof(\result) == \type(SYCL_EVENT_CLASS);
3  ensures Perm(\result.dim0, read) ** \result.dim0 == dim0;
4  ensures Perm(\result.dim1, read) ** \result.dim1 == dim1;
5  ensures idle(\result);
6  SYCL_EVENT_CLASS event_constructor(int dim0, int dim1);

```

Listing 6.8: The *event-constructor* generated for the basic kernel declaration in Listing 6.6.

```

1  requires global_dim0 >= 0 && local_dim0 > 0 && global_dim0 % local_dim0 == 0;
2  requires global_dim1 >= 0 && local_dim1 > 0 && global_dim1 % local_dim1 == 0;
3  ensures \result != null && \typeof(\result) == \type(SYCL_EVENT_CLASS);
4  ensures Perm(\result.local_dim0, read) ** \result.local_dim0 == local_dim0;
5  ensures Perm(\result.group_dim0, read);
6  ensures \result.group_dim0 == global_dim0 / local_dim0;
7  ensures \result.group_dim0 * \result.local_dim0 == global_dim0;
8  ensures Perm(\result.local_dim1, read) ** \result.local_dim1 == local_dim1;
9  ensures Perm(\result.group_dim1, read);
10 ensures \result.group_dim1 == global_dim1 / local_dim1;
11 ensures \result.group_dim1 * \result.local_dim1 == global_dim1;
12 ensures idle(\result);
13 SYCL_EVENT_CLASS event_constructor(int global_dim0, int local_dim0,
14                                  ↪ int global_dim1, int local_dim1);

```

Listing 6.9: The *event-constructor* generated for the ND-range kernel declaration in Listing 6.7.

**Range dimension checking** To make sure that the kernel dimensions given by the user are valid, requirements are added to the *event-constructor*, as can be seen on line 18 in Listing 6.4. For basic kernels, it states that the range of each dimension should be greater or equal to zero. A dimension range of zero is allowed in SYCL and means the kernel is not actually executed. This is also supported for VerCors’ parblocks, so it was decided to allow this. The statement for the given *range* on line 2 in Listing 6.6 can be seen on line 1 in Listing 6.8.

For ND-range kernels, the pre-conditions state that the global dimensions should be divisible by the local dimensions and that the local dimensions ranges should be greater than zero, to avoid division errors when calculating the number of work-groups. Also, similar to the basic kernel, the global dimension ranges should be greater or equal to zero. The statement for the given *nd\_range* on lines 1-3 in Listing 6.7 can be seen on lines 1-2 in Listing 6.9.

**Encoding of basic kernels** For a *parallel\_for* invocation with a basic kernel, such as the one in Listing 6.6, a nested *kernel-parblock* is generated with an iteration variable for every dimension in the *range*, where its range is same as that dimension’s size. This means that the resulting *kernel-parblock* maintains the concept of an index-space with multiple dimensions. This technique is similar to how CUDA kernels are encoded into VerCors [63].

The generated *event-class* for Listing 6.6 can be seen in Listing 6.10. In Listing 6.6 on line 2, the *range* object has two dimensions with sizes 6 and 4, so  $6 * 4 = 24$  work-items will be executed in total. This is encoded as a nested *kernel-parblock* with two iter-variables: one with *dim0* work-items and one with *dim1* work-items, so also  $6 * 4 = 24$  work-items in total.

```

1  class SYCL_EVENT_CLASS {
2      int dim0; int dim1;
3
4      context Perm(dim0, read) ** dim0 >= 0 ** Perm(dim1, read) ** dim1 >= 0;
5      requires [dim1][dim0] (1 + 2 == 3);
6      ensures [dim1][dim0] (3 * 5 == 15);
7      run {
8          par SYCL_BASIC_KERNEL(
9              int GLOBAL_ID0 = 0 .. dim0, int GLOBAL_ID1 = 0 .. dim1
10             ) context Perm(dim0, read) ** dim0 >= 0 ** Perm(dim1, read) ** dim1 >= 0;
11                 requires 1 + 2 == 3;
12                 ensures 3 * 5 == 15; {
13                     // kernel body
14                 }
15             }
16 }

```

Listing 6.10: The *event-class* generated for the basic kernel declaration in Listing 6.6. The notation  $[x][y] \text{expr}$  resembles a quantification of *expr* over  $x*y$  items.

**Encoding of ND-range kernels** For a `parallel_for` invocation with an ND-range kernel, such as the one in Listing 6.7, a nested outer *kernel-parblock* and inner *kernel-parblock* are generated. The nested inner *kernel-parblock* has an iteration variable for every dimension in the local `range` parameter, where its range is the same as that dimension's size. The nested outer *kernel-parblock* has an iteration variable for every dimension in the local `range` parameter, where its range is same as that dimension's work-group size. This work-group size is calculated by dividing the dimension's size in global range parameter by the dimension's size in the local range parameter. This means that the resulting *kernel-parblocks* also maintain the concept of an index-space with multiple dimensions. This technique is also similar to how CUDA kernels are encoded into VerCors [63]. The generated COL encoding for Listing 6.6 can be seen in Listing 6.10. In Listing 6.6 on lines 2-3, the `nd_range` object has global dimension sizes 6 and 4, and local dimension sizes 2 and 2. This means that  $6 * 4 = 24$  work-items will be executed in total, using  $6/2 = 3$  by  $4/2 = 2$  work-groups. This is encoded into an inner *kernel-parblock* with two iter-variables: one with `local_dim0` and one with `local_dim1` work-items, so  $2 * 2 = 4$  work-items in per work-group, and an outer *kernel-parblock* with two iter-variables: one with `group_dim0` work-groups and one with `group_dim1` work-groups. This means that  $(3 * 2) * (2 * 2) = 24$  work-items are executed in total.

```

1  class SYCL_EVENT_CLASS {
2      int group_dim0; int local_dim0; int group_dim1; int local_dim1;
3
4      context Perm(group_dim0, read) ** group_dim0 >= 0;
5      context Perm(local_dim0, read) ** local_dim0 >= 0;
6      context Perm(group_dim1, read) ** group_dim1 >= 0;
7      context Perm(local_dim1, read) ** local_dim1 >= 0;
8      requires [group_dim1][group_dim0][local_dim1][local_dim0] (1 + 2 == 3);
9      ensures [group_dim1][group_dim0][local_dim1][local_dim0] (3 * 5 == 15);
10     run {
11         par SYCL_ND_RANGE_KERNEL_WORKGROUPS (
12             int GROUP_ID0 = 0 .. group_dim0, int GROUP_ID1 = 0 .. group_dim1
13         ) context Perm(group_dim0, read) ** group_dim0 >= 0;
14             context Perm(local_dim0, read) ** local_dim0 >= 0;
15             context Perm(group_dim1, read) ** group_dim1 >= 0;
16             context Perm(local_dim1, read) ** local_dim1 >= 0;
17             requires [local_dim1][local_dim0] (1 + 2 == 3);
18             ensures [local_dim1][local_dim0] (3 * 5 == 15); {
19                 par SYCL_ND_RANGE_KERNEL_WORKITEMS (
20                     int LOCAL_ID0 = 0 .. local_dim0, int LOCAL_ID1 = 0 .. local_dim1
21                 ) context Perm(group_dim0, read) ** group_dim0 >= 0;
22                     context Perm(local_dim0, read) ** local_dim0 >= 0;
23                     context Perm(group_dim1, read) ** group_dim1 >= 0;
24                     context Perm(local_dim1, read) ** local_dim1 >= 0;
25                     requires 1 + 2 == 3;
26                     ensures 3 * 5 == 15; {
27                         // kernel body
28                     }
29                 }
30             }
31     }

```

Listing 6.11: The *event-class* generated for the ND-range kernel declaration in Listing 6.7. The notation  $[x] [y] \text{expr}$  resembles a quantification of *expr* over  $x*y$  items.

**Similarities to the encoding of CUDA kernels** In [63] it is described how CUDA kernels are transformed to PVL (COL) code. It only shows a transformation for the CUDA-equivalent of a SYCL ND-range kernel with a one-dimensional index-space. However, when verifying a CUDA kernel with VerCors and inspecting the generated COL AST right after a CUDA kernel is transformed, it can be seen that a nested parblock is created for every dimension in the range <sup>1</sup>. This is also done in the SYCL implementation.

### 6.1.3.5 Handling asynchronicity between running kernels and host code

In VerCors, all work-items in a parblock must finish executing before any statements after the parblock statement are executed. However, submitted kernels in SYCL run asynchronously from host code: the host code continues executing the commands that come after a kernel's submission to the queue. This means that modelling a SYCL command group only using the *kernel-parblock*, as described in the previous subsections, does not fully match reality. To match reality, the *kernel-parblock* should be executed in parallel with the *host code* that comes after the kernel submission.

This asynchronicity between running kernels and the host is similar to the behaviour of VerCors' fork/join construct (described in subsection 3.4.3.3 (page 42)), where the 'host code' forks a class, which executes that class' run-method, whilst at the same time also continuing executing the 'host code' that comes after the fork. To illustrate the similarity in behaviour more clearly, look at Figure 6.12, where the comments and matching colours indicate the similarities.

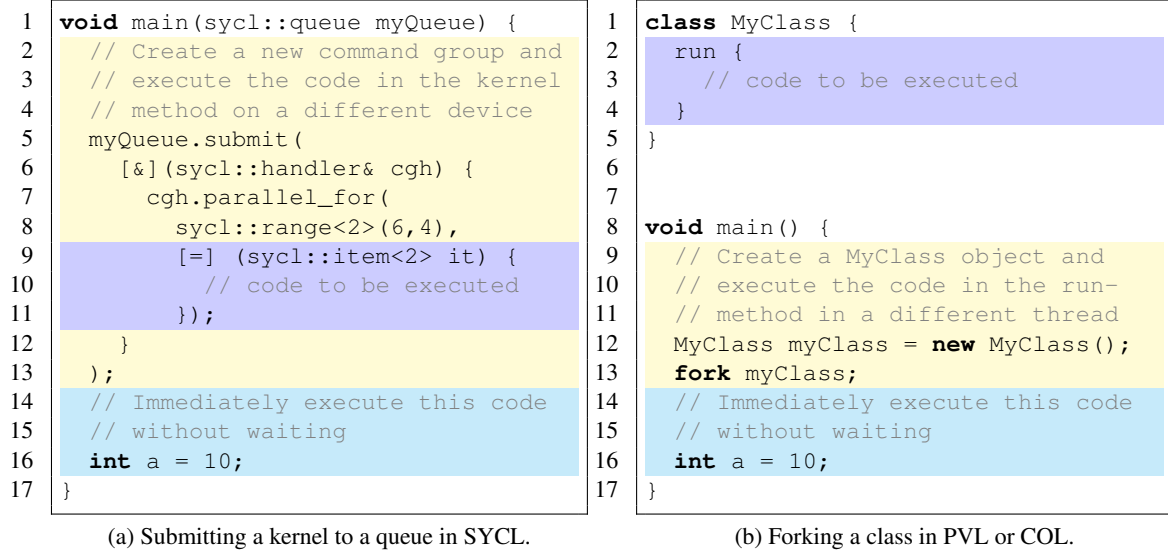


Figure 6.12: Similarities in behaviour between submitting a kernel to a queue in SYCL and forking a class in PVL or COL.

Because of this similarity in behaviour, the fork/join construct was used to encode the asynchronous behaviour of SYCL host code continuing whilst a kernel is being executed: the generated *kernel-parblock* is placed inside a run-method (*kernel-runner*) inside a class (*event-class*), which is forked from the encoded *host code*. This encoding can be seen in Listing 6.4, where the generated *kernel-parblock* resides inside a *kernel-runner*, which resides in the *event-class*. Then, in the *host code* an instance of the *event-class* is made using the *event-constructor*, and is forked.

Besides the similarity in behaviour, this encoding has another advantage: it shows a clear separation between code executed on the host and code executed on a device, as all code executed on the device is encapsulated by the *event-class*.

<sup>1</sup>It should be noted here that VerCors does not seem to support the C syntax for invoking a CUDA kernel with more than one dimension for its global and local index-space, but in the transformation phase there does exist code to handle multiple dimensions.

**Alternative** The asynchronicity between running kernels and the host code could also have been encoded using VerCors’ simultaneous parblocks, which allows two parblocks to be executed simultaneously. The first parblock would be the *kernel-parblock*, and the host code would be put inside the second parblock, with an iter-variable of range 1, so the host code is executed simultaneously, but only once. A visualization of this approach can be seen in Listing 6.13. However, the fork/join construct approach was chosen because the encoding into simultaneous parblocks has multiple challenges: First of all, when submitting multiple kernels, parts of the host code could end up nested in multiple parblocks. Secondly, the user can wait on a kernel to finish executing in the host code by calling `wait()`. This means that the host code parblock must only contain the host code up to the call to `wait()`. This means, that the implementation needs to read ahead to find `wait()` statements.

```

1 parallel {
2   kernel-parblock
3   par (int i = 0 .. 1) { host code }
4 }

```

Listing 6.13: Encoding of asynchronicity between a running kernel and host code using simultaneous parblocks.

### 6.1.3.6 Handling user-defined contracts

In order for users to actually reason about SYCL kernels in VerCors, specifications can be added as a contract to kernel body declarations. An example of such a contract can be seen on lines 4-5 in Listing 6.6 and on lines 5-6 Listing 6.7. The contract written above the kernel declaration is seen as a work-item-level specification, so it will be verified for every work-item in the kernel individually. In order to successfully verify these specifications, they are inserted in two places in the generated code.

**Contract above the *kernel-parblock*** The first place the specifications are added, is in the contract of the (inner) *kernel-parblock*. VerCors then automatically verifies the contract for every work-item in the parblock. To give an example, the added specifications for the basic kernel in Listing 6.6 can be seen on lines 11-12 in Listing 6.10. For ND-range kernels, placing the specification in the contract of the inner *kernel-parblock* is not enough, as it only proves the specifications for work-items inside a single work-group. Therefore, the specifications are added to the contract of the outer *kernel-parblock* as well, but then quantified over all work-items of the work-group, which allows VerCors to verify the contract for every work-group. To give an example, the added contract for the ND-range kernel in Listing 6.7 can be seen on lines 17-18 and on lines 25-26 in Listing 6.11.

The quantification of the specifications that is done for the outer *kernel-parblock* in the encoding ND-range kernels is also done in the implementation of CUDA, which has its own version of ND-range kernels. [63]

**Contract above the *kernel-runner*** The second place where the specifications are added, is in the contract of the *kernel-runner*. This is done because the *kernel-parblock* resides in the *kernel-runner*. As mentioned in subsection 3.3.3.1, methods are verified in isolation, where the requirements state what must be true in order to ensure its post-conditions. This means that the pre-conditions of the *kernel-parblock* only have the information provided by the pre-conditions of the *kernel-runner*. If the specifications are added to the contract of the *kernel-runner*, then if the pre-conditions of the *kernel-runner* can be established, the pre-conditions of the *kernel-parblock* can always be established as well. Furthermore, to pass the assumptions from the post-conditions of the *kernel-parblock* to the host after the kernel has finished executing (and the host waited on it, as described in the next section), the post-conditions of the *kernel-parblock* should also be present in the post-conditions of the *kernel-runner*. However, the specifications above the kernel body declaration are written on the work-item-level. So in the contract for the *kernel-runner*, the specifications are quantified over all work-items. One thing to note is that user specifications involving permissions are not put in the contract of the *kernel-runner*, as the contract for the *kernel-runner* already contains all permissions the user is allowed to have. An example of the contract of a *kernel-runner* can be seen on lines 5-6 in Listing 6.10, where the contract on lines 4-5 of Listing 6.6 is quantified over all dimensions of all work-groups.

The quantification of the specifications in the contract of the *kernel-runner* to get the specification for the whole kernel is also done in the implementation of CUDA, where the generated parblocks reside inside a method [63].

**Challenge of quantifications** These quantifications, especially in case of ND-range kernels, where they are nested, might have the same challenge as the CUDA implementation, which is that (nested) quantified expressions can be hard to prove for Z3 [63]. However, as stated in [63], simplification rules were added which syntactically replace complicated quantified expressions by simpler ones to help Z3 with proving CUDA kernels. These simplification rules might also simplify the quantified expressions generated for SYCL kernels.

## 6.2 Synchronizing with the host

This section first shares the definitions of SYCL’s `wait` method for which support was added to VerCors. Then the semantics and the encoding into VerCors for explicit synchronisation of running kernels with the host code are discussed.

### 6.2.1 Definitions of supported methods

The definition of the `wait` method in SYCL’s `event` class is described below. This definition is a simplified version of the method description in the SYCL specification [72] (on page 102).

**`wait()` → `void`**

**In class:** `sycl::event`

**Description:** Wait for the command group linked to this event to finish its execution.

### 6.2.2 Semantics

As mentioned previously, when a command group is submitted to a queue, the host continues executing code that comes after the queue submission. However, sometimes it is desirable to let the host wait until a submitted command group has finished executing. This can be done in SYCL by calling the `wait` method on the event object that is returned when submitting a command group to a queue. An example of this can be seen in Figure 6.14a. On line 2 in yellow, a kernel is submitted to a queue, and the resulting event is stored in the variable `ev`. After that, variable `a` is assigned, which is executed asynchronously with the kernel. Then, on line 4 in green, the `wait` method is called on the event variable linked to the kernel submission on line 2. This means that the host waits until that kernel has finished executing. Finally, once the kernel has terminated, variable `b` is declared.

```
1 void main(sycl::queue q) {
2   sycl::event ev = q.submit(...);
3   int a = 5;
4   ev.wait();
5   int b = 10;
6 }
```

(a) Example code of the host waiting on a submitted kernel to finish executing. *The command group declaration has been omitted here.*

```
1 void main(ref q) {
2   SYCL_EVENT_CLASS sycl_event_ref;
3   sycl_event_ref =
4     ↪ new SYCL_EVENT_CLASS();
5   fork sycl_event_ref;
6   int a; a = 5;
7   join sycl_event_ref;
8   int b; b = 10;
9 }
```

(b) The resulting *host code* in the encoding of Figure 6.14a.

Figure 6.14: Example of the *host code* of the encoding of the host waiting on a submitted kernel.

### 6.2.3 Encoding into VerCors

In subsection 6.1.3.5 (page 67), it was explained that the generated *event-class* is forked from the *host code* to allow them to execute asynchronously. Internally, the generated class instance of *event-class* is linked to the variable to which the return value of a queue’s `submit` was assigned. This makes the implementation of the `wait` method straightforward: when the `wait` method is called on an event, the generated class instance linked to event is joined using the `join` statement of VerCors’ `fork/join` construct. An example of this can be seen on line 7 in green in Figure 6.14b, where `ev.wait()` in Figure 6.14a is encoded to `join sycl_event_ref;`



## 6.3 Querying a work-item's id and index-space

This section first shares the definitions of the SYCL methods related to querying a work-item's id and index-space for which support was added to VerCors. Then, the semantics and the encoding into VerCors for querying a work-item's id and index-space are discussed.

### 6.3.1 Definitions of supported methods

The methods in the `item` and `nd_item` classes that are supported in VerCors are described below. These definitions are simplified versions of the method descriptions in the SYCL specification [72] (on pages 255-256, 258-259).

**`get_range(int dimension) → int`**

**In class:** `sycl::item`

**Arguments:**

- `dimension`: What dimension of the index-space to query.

**Returns:** The number of work-items in the dimension `dimension`.

**Description:** Returns the number of work-items in a basic kernel, in the requested dimension of the index-space.

**`get_id(int dimension) → int`**

**In class:** `sycl::item`

**Arguments:**

- `dimension`: What dimension of the index-space to query.

**Returns:** The id of the work-item in the dimension `dimension`.

**Description:** Returns the id of the current work-item in a basic kernel, in the requested dimension of the index-space.

**`get_linear_id() → int`**

**In class:** `sycl::item`

**Returns:** The linear id of the work-item.

**Description:** Returns the linear id of the current work-item in a basic kernel, calculated using the formula from section 3.11.1 of the SYCL specification [72].

**`get_local_range(int dimension) → int`  
**`get_group_range(int dimension) → int`  
**`get_global_range(int dimension) → int`******

**In class:** `sycl::nd_item`

**Arguments:**

- `dimension`: What dimension of the index-space to query.

**Returns:** The local, group, or global number of work-items in the dimension `dimension`.

**Description:** Returns the local, group, or global number of work-items in an ND-range kernel, in the requested dimension of the index-space.

```
get_local_id(int dimension) → int
get_group(int dimension)2 → int
get_global_id(int dimension) → int
```

**In class:** `sycl::nd_item`

**Arguments:**

- `dimension`: What dimension of the index-space to query.

**Returns:** The local, group, or global id of the work-item in the dimension `dimension`.

**Description:** Returns the local, group, or global id of the current work-item in an ND-range kernel, in the requested dimension of the index-space.

```
get_local_linear_id() → int
get_group_linear_id() → int
get_global_linear_id() → int
```

**In class:** `sycl::nd_item`

**Returns:** The local, group, or global linear id of the work-item.

**Description:** Returns the local, group, or global linear id of the current work-item in an ND-range kernel, calculated using the formula from section 3.11.1 of the SYCL specification [72].

## 6.3.2 Semantics

Oftentimes, it is desirable in a work-item's body to know what the id of the work-item is or many work-items there are in the index-space. For example, when working with arrays, a work-item's id can be used to figure out what index of the array it should read from or write to. As mentioned previously, the kernel declaration lambda argument in SYCL's `parallel_for` method invocations take an instance of an `item` or `nd_item` class as parameter. Both these classes contain methods to query a work-item's id or the range of a dimension in the index-space. The methods for which support was added to VerCors are described in detail in the previous subsection.

## 6.3.3 Encoding into VerCors

To implement support for the aforementioned methods, additions were made to the SYCL header file and to VerCors' transformation phase. These additions are described in this subsection.

### 6.3.3.1 Additions to the SYCL header file

To aid the encoding of the `get_linear_..._id`, `get_global_id`, and `get_global_range` methods, some functions were added to the SYCL header file. These functions are described below.

**Functions for linearized ids** To calculate a linear index out of two or three subscripts, four functions were added to the SYCL header file. Since these functions do not exist in SYCL, they were added as comments, to keep separation between SYCL methods and verification helper functions. These functions can be seen on the next page in Listing 6.15.

---

<sup>2</sup>One would expect this method to be named `get_group_id` to be conformant with `get_local_id` and `get_global_id`, but this is not the case.

```

1  requires id0 >= 0 && id0 < dim0 && id1 >= 0 && id1 < dim1;
2  requires dim0 >= 0 && dim1 >= 0;
3  ensures \result == sycl::linearize2formula(id0, id1, dim1);
4  ensures \result >= 0 && \result < dim0 * dim1;
5  ensures (\forallall int ida0, int ida1;
6    ida0 >= 0 && ida0 < dim0 && ida1 >= 0 && ida1 < dim1 &&
7    (ida0 != id0 || ida1 != id1) ==>
8    \result != { : sycl::linearize2formula(ida0, ida1, dim1) :}
9  );
10 pure int linearize2(int id0, int id1, int dim0, int dim1);
11
12 pure int linearize2formula(int id0, int id1, int dim1) = id1 + (id0 * dim1);
13
14 requires id0 >= 0 && id0 < dim0 && id1 >= 0;
15 requires id1 < dim1 && id2 >= 0 && id2 < dim2;
16 requires dim0 >= 0 && dim1 >= 0 && dim2 >= 0;
17 ensures \result == sycl::linearize3formula(id0, id1, id2, dim1, dim2);
18 ensures \result >= 0 && \result < dim0 * dim1 * dim2;
19 ensures (\forallall int ida0, int ida1, int ida2;
20   ida0 >= 0 && ida0 < dim0 && ida1 >= 0 &&
21   ida1 < dim1 && ida2 >= 0 && ida2 < dim2 &&
22   (ida0 != id0 || ida1 != id1 || ida2 != id2) ==>
23   \result != { : sycl::linearize3formula(ida0, ida1, ida2, dim1, dim2) :}
24 );
25 pure int linearize3(int id0, int id1, int id2, int dim0, int dim1, int dim2);
26
27 pure int linearize3formula(int id0, int id1, int id2, int dim1, int dim2) =
28   ⇐ id2 + (id1 * dim2) + (id0 * dim1 * dim2);

```

Listing 6.15: Excerpt from the SYCL header file showing the functions `linearize2`, `linearize2formula`, `linearize3`, and `linearize3formula`.

The abstract function `linearize2` models the calculation of a linear index out of two indices, without actually implementing it. This allows the results to be assumed to be true, instead of having to be proven, which would take extra time to verify. To make sure that the assumed results are sound, they were proven to be true by hand. These proofs can be found in Appendix D. The parameters `id0` and `id1` represent the indices from which a linear index should be calculated. The parameters `dim0` and `dim1` represent the size of each dimension. The requirements of the method check that the requested indices are within the bounds of the dimension sizes and that each given dimension size is non-negative. If those requirements are met, the post-conditions ensure three things:

- The result is equal to `linearize2formula` which calculates the linearization of the two indices, using the formula described in section 3.11.1 of the SYCL specification [72]. This post-condition is highlighted in yellow on line 3 in Listing 6.15.
- The resulting linear index is non-negative and within the bounds of the linear range size. This post-condition was added as it can be necessary for certain proofs to know the bounds of the result. This post-condition is highlighted in blue on line 4 in Listing 6.15.
- The result is injective, i.e. the result of `linearize2` is unique from every result for any other set of ids `id0`, `id1` within bounds of the dimensions `dim0`, `dim1`. This post-condition is highlighted in purple on lines 5-9 in Listing 6.15.

The function `linearize3` requires and ensures the same for the linearization of three-dimensional indices.

**Function for `get_global_range`** The function `get_global_range` calculates the sizes of the global dimensions by multiplying a local dimension size with a group dimension size. As can be seen, this calculation is rather trivial. However, it was required to encapsulate it in a function, as without this encapsulation the verifier was not able to verify some statements.

```
pure int get_global_range(int local_dim, int group_dim) =
    local_dim * group_dim;
```

### 6.3.3.2 Additions to the transformation phase

Invocations of the aforementioned methods are encoded to COL in the transformation phase. As mentioned in subsection 6.1.3, a kernel declaration is encoded into several components, such as an *event-class* and a *kernel-parblock*. The *event-class* contains fields holding the sizes of the dimensions of the index-space. The *kernel-parblock* has iteration variables ranging from 0 to the value of a dimension field. An example of this encoding can be seen in Figure 6.16, where for a 2D basic kernel the *event-class* has dimension size fields `dim0` and `dim1`, and the *kernel-runner* has iteration variables `id0` and `id1`.

```
1 cgh.parallel_for(
2   sycl::range<2>(x,y),
3   [=] (sycl::item<2> it) {...}
4 );
```

(a) An example of a 2D basic kernel declaration.

```
1 par SYCL_BASIC_KERNEL(
2   int id0 = 0 .. this.dim0,
3   int id1 = 0 .. this.dim1
4 ) { ... }
```

(b) The *kernel-parblock* generated for the 2D basic kernel in Figure 6.16a.

Figure 6.16: Example of the *kernel-parblock* generated for a 2D basic kernel.

These dimension size fields and iteration variables are used to encode `(nd_)item` methods as follows:

- Invocations to methods `get_range(x)`, `get_local_range(x)`, and `get_group_range(x)` return a subscript `[x]` on a sequence containing all basic, local, or group dimension size fields, respectively. An example can be seen in Figure 6.17 where on line 1 in both listings `it.get_range(x)` is encoded into `seq<int>{dim0, dim1}[x]`.
- Invocations to methods `get_id(x)`, `get_local_id(x)`, and `get_group(x)` return a subscript `[x]` on a sequence containing all basic, local, or group iteration variables, respectively. An example can be seen in Figure 6.17 on line 2 in both listings.
- Invocations to methods `get_linear_id()`, `get_local_linear_id()`, and `get_group_linear_id()` are encoded into a reference to the only present iteration variable in a case of a 1D kernel, or an invocation of the `linearize2` or `linearize3` function with all iteration variables as `id` arguments and dimension size fields as dimension size arguments. An example of this encoding can be seen in Figure 6.17 on line 3 in both listings.

The encodings `nd_item` methods concerning the global index-space and ids are more involved. No corresponding dimension size fields and iteration variables are generated when encoding ND-range kernels, as they are not used in the definition of the *kernel-parblocks*. So, instead, these are calculated from the local and group dimensions sizes and ids as follows:

- Invocations to the method `it.get_global_range(x)` are encoded into the invocation of `get_global_range` with as arguments the encodings of `get_local_range(x)` and `get_group_range(x)`. An example of this encoding can be seen in Figure 6.17 on lines 5-7 in both listings.
- Invocations to the method `it.get_global_id(x)` are encoded as a linearization of the corresponding local and group ids. This is done with an invocation of `linearize2` with as arguments the encodings of `get_local_id(x)`, `get_group(x)`, `get_local_range(x)` and `get_group_range(x)`. An example of this encoding can be seen in Figure 6.17 on lines 8-11 in both listings.

- Invocations to the method `get_global_linear_id()` are encoded into an invocation of `get_global_id(0)` in a case of a 1D kernel, or an invocation of the `linearize2` or `linearize3` function with the encodings of the invocations `get_global_id(x)` for each dimension `x` as id arguments and `get_global_range(x)` for each dimension `x` as dimension size arguments. An example of this encoding can be seen in Figure 6.17 on lines 12-16 in both listings.

1	<code>it.get_range(x);</code>	1	<code>seq&lt;int&gt;{dim0, dim1}[x];</code>
2	<code>it.get_id(x);</code>	2	<code>seq&lt;int&gt;{id0, id1}[x];</code>
3	<code>it.get_linear_id();</code>	3	<code>sycl::linearize2(id0, id1, dim0, dim1);</code>
4		4	
5	<code>it.get_global_range(x);</code>	5	<code>sycl::ndItem::get_global_range(</code>
6		6	<code>it.get_local_range(x), it.get_group_range(x)</code>
7		7	<code>);</code>
8	<code>it.get_global_id(x);</code>	8	<code>sycl::linearize2(</code>
9		9	<code>it.get_local_id(x), it.get_group(x),</code>
10		10	<code>it.get_local_range(x), it.get_group_range(x)</code>
11		11	<code>);</code>
12	<code>it.get_global_linear_id();</code>	12	<code>sycl::linearize2(</code>
13		13	<code>it.get_global_id(0), it.get_global_id(1),</code>
14		14	<code>it.get_global_range(0),</code>
15		15	<code>it.get_global_range(1)</code>
16		16	<code>);</code>

(a) Invocations of all supported (nd\_)item methods.

(b) The COL encodings of the method invocations in Figure 6.17a. The method invocations in gray are not actually inserted themselves, their encodings are inserted instead.

Figure 6.17: Encodings of invocations of 2D (nd\_)item methods into COL.

### 6.3.3.3 Thrown errors

For the (nd\_)item methods that take a `dimension` parameter, this `dimension` should be within the range of the number of dimensions in the kernel index-space. As mentioned in the previous subsection, the methods that take a `dimension` parameter are encoded into a sequence with the same length as the number of dimensions in the index-space, which is subscripted with the users' argument for `dimension`. This means that if the user uses as argument a value outside the range of the number of dimensions, an error is thrown automatically by VerCors stating the subscript is outside the bounds of the sequence. This error is caught and then re-thrown with an error message stating that the requested `dimension` in the invocation of the (nd\_)item method is outside the range of the number of dimensions of the index-space.

## 7 Added support for SYCL's data sharing between devices and the host

The third research question (RQ3, see subsection 2.1 (page 7)) states that SYCL's buffers and data accessors should be supported to be able to share data between devices and the host. In this chapter, the added support to VerCors for these components is discussed.

### 7.1 Buffers

This section first shares the definition of the constructor of SYCL's `buffer` class for which support was added to VerCors. Then, the semantics and the encoding into VerCors for declarations of buffers are discussed.

#### 7.1.1 Definition of the supported constructor

The definition of the constructor of SYCL's `buffer` class described below is supported in VerCors. The definition of the constructor is a simplified version of the description in the SYCL specification [72] (on page 112).

```
buffer(T* hostData, sycl::range<S>& bufferRange) → sycl::buffer<T, S>
```

**In class:** `sycl::buffer<T, S>`

**Arguments:**

- `hostData`: A pointer to the data of type `T` the buffer should manage.
- `bufferRange`: A `sycl::range` object which describes how the buffer's copy of the `hostData` will structured.

**Returns:** A new `sycl::buffer` object.

**Description:** Creates a buffer which contains a copy of the `hostData` transformed into a (multi-)dimensional array, whose structure is described by `bufferRange`.

#### 7.1.2 Semantics

As mentioned in subsubsection 3.2.3.1 (page 22), a SYCL buffer can be seen as an array with one, two, or three dimensions, which can be accessed in command groups through accessors. These buffers are typically declared in the application scope. A buffer can be constructed with a pointer to the data the buffer should hold, called the `hostData`, together with a `sycl::range` object. An example of such a construction can be seen on line 3 in Listing 7.1, where a buffer is constructed with `arr` as its `hostData` and a 2-by-5 two-dimensional range. It was chosen to only implement support for such buffer constructors, because they are the most simple way for the user to declare an initialized buffer, since it only requires a pointer and a `range` object as arguments. The definition of the chosen constructor can be found in the previous subsection. For the rest of this chapter, when buffer declarations are mentioned, only the declarations chosen to be supported should be considered.

```

1  // @ context \pointer(arr, 10, write);
2  void buffer_declaration(int* arr) {
3      sycl::buffer<int, 2> buff = sycl::buffer(arr, sycl::range<2>(2,5));
4  }

```

Listing 7.1: Example of a buffer declaration in SYCL.

When a buffer is constructed, a copy is made of the `hostData`, but not necessarily with the same one-dimensional structure as the `hostData`: the copy will have the same number of dimensions with the same sizes as the provided range object. This does put a restriction on the provided range object: its total size must not be greater than the size of the `hostData`. How buffers structure their data copy based on the provided range is illustrated in Figure 7.2 on the next page, where the same `hostData`, which is an array of items `a` to `h`, is given to three buffers with different ranges. As can be seen, each range arranges the elements of the linear `hostData` differently. This feature is useful when a user, for instance, wants to access linear data from a multi-dimensional kernel, as it enables the user to match the structure of the buffer with the structure of the kernel's index-space, which allows for easier indexing of the data.

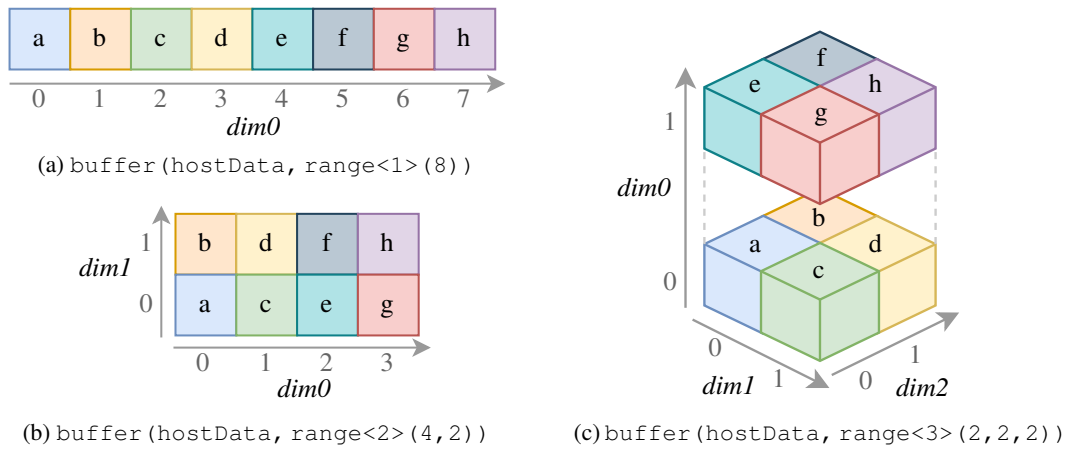


Figure 7.2: Example of how the same `hostData` is arranged in buffers which different ranges. The arrows indicate the index in every dimension of the buffer.

In Listing 7.3, the lifecycle of a buffer and its effects on its `hostData` are shown. According to SYCL's specification [72] (page 128), after the buffer has been initialized, it should be assumed that the contents of the `hostData` are unspecified for the entire lifetime of the buffer. It further states that this means that when `hostData` has been put in a buffer, reading it, writing to it, or putting it in another buffer gives undefined results. However, when a buffer is destroyed, its contents at that point in time are copied to the `hostData` (if necessary) to make sure its values match the buffer's contents. This also means that after buffer destruction, the contents of the `hostData` are defined and can be used again without undefined behaviour. As mentioned in subsection 3.2.4.1 (page 24), the destruction of a buffer also causes the host to wait on all submitted kernels with access to the buffer for which the contents need to be copied back to the buffer's `hostData` to finish executing.

```

1  void test(int* arr) {
2      {
3          // Create a buffer with arr as its hostData
4          sycl::buffer<int, 1> buff = sycl::buffer(arr, sycl::range<1>(10));
5          // Contents of arr are undefined for the rest of this scope
6          ...
7      } // The buffer is destroyed here and its contents are copied to arr
8      // Contents of arr are defined again
9  }

```

Listing 7.3: Illustration of the lifecycle of a buffer and its effects on its `hostData`.

### 7.1.2.1 Exclusive access to hostData

On page 112 of the SYCL specification [72], the following is stated in the description of the buffer constructor that was chosen to be supported:

*The buffer is initialized with the memory specified by `hostData`, and the buffer assumes exclusive access to this memory for the duration of its lifetime.*

It is nowhere stated in the specification whether the phrase "*this memory*" means the entire size of the `hostData` or only the part that is within the bounds of the buffer's range parameter. Knowing the exact meaning matters in this case because the total size of the range parameter is allowed to be smaller than the size of the `hostData`. Attempts were made to solve this uncertainty by testing when errors would be thrown by the OneAPI DPC++ SYCL compiler [40], and by asking about it on Kronos' forum [43]. However, these attempts did not yield any answer to this matter. This meant that an assumption had to be made about the meaning in order to implement support for buffers. The second suggested definition, that only the part that is within the bounds of the buffer's range parameter is claimed, seemed more logical than the first, because claiming exclusive access to a part of data that is not included in the buffer seemed excessive. So in this thesis, it is assumed that "*this memory*" means *the contents of `hostData` that are within the bounds of the range parameter*.

This means that the code below, where an element outside the bounds of the buffer's range is re-assigned, is considered valid:

```
1 // @ context \pointer(arr, 5, write);
2 void buffer_declaration(int* arr) {
3     // Create a buffer containing the first 4 items
4     sycl::buffer<int, 1> buff = sycl::buffer(arr, sycl::range<1>(4));
5     // Write to the 5th item
6     arr[4] = 5;
7 }
```

Listing 7.4: Example of writing to a part of a pointer outside the buffer's range.

## 7.1.3 Encoding into VerCors

To implement support for buffer declarations and destructions, additions were made to VerCors' transformation phase. These additions are described in this subsection.

### 7.1.3.1 Buffer declarations

Buffer declarations are detected in the transformation phase, where every variable initialization to a buffer constructor is seen as a buffer declaration. For each of these declarations, several COL constructs are generated, which are described in this subsubsection.

**Generated predicate and methods** For all detected buffer declarations, one predicate and two global methods are generated, unless they were already generated for another buffer with the same `hostData` type and range, in which case the already generated predicate and methods are reused.

The predicate `exclusive_hostdata_access`, which can be seen in Listing 7.5, takes a pointer with the same base-type `T` as `hostData` and an integer representing the requested size. When folding and unfolding the predicate, it takes away and returns, respectively, write permission to `hostData` from the pointer location up to, but not including, the requested size.

```
1 resource exclusive_hostdata_access(T* hostData, int size) =
2     ⇨ \pointer(hostData, range, size);
```

Listing 7.5: The generated predicate `exclusive_hostdata_access`, for `hostData` of type `T*`.



The abstract method `copy_hostdata_to_buffer`, which can be seen in Listing 7.6, models `hostData` being copied to a new COL array, without actually implementing the copying process. The `size` parameter represents the amount of data that is copied from the `hostData`. First, it is checked that `size` is a positive number, that `hostData` is at least as big as `size` and that there is write permission for that part of `hostData`. If this is all true, it is ensured that write permission for `hostData` is given back when the method returns. It is further ensured that the return value, indicated by `\result`, contains `size` items and with write permission to all of them, and that it contains the same values on the same indices as `hostData`.

```

1 requires size >= 0;
2 context \pointer(hostData, size, write);
3 ensures \array(\result, size) ** Perm(\result[*], write);
4 ensures (\forallall int i; i >= 0 && i < size ==> {: \result[i] :} == hostData[i]);
5 T[] copy_hostdata_to_buffer(T* hostData, int size);

```

Listing 7.6: The abstract method `copy_hostdata_to_buffer`, for `hostData` of type `T*`.

The abstract method `copy_buffer_to_hostdata`, which can be seen in Listing 7.7, models the contents of an array called `buffer` being copied to `hostData`, without actually implementing the copying process. First, it is checked that there is read permission on all elements of `buffer`. Then it is made sure that `exclusive_hostdata_access` is folded, which means that there is write permission to the elements of `hostdata` for the length of the buffer, but that it is not part of the current program state (so users cannot alter the `hostData` at this moment). This is also ensured in the post-condition. It is further ensured that `hostData` contains the same values on the same indices as `buffer` when the method terminates. To be able to ensure this, the predicate `exclusive_hostdata_access` is temporarily unfolded.

```

1 context \array(buffer, buffer.length);
2 context Perm(buffer[*], read);
3 context exclusive_hostdata_access(hostdata, buffer.length);
4 ensures unfolding exclusive_hostdata_access(hostdata, buffer.length) \in
5   ⇐ (\forallall int i; i >= 0 && i < buffer.length ==> {: hostdata[i] :} ==
6     ⇐ buffer[i]);
7 void copy_buffer_to_hostdata(T* hostdata, T[] buffer);

```

Listing 7.7: The generated abstract method `copy_buffer_to_hostdata`, for `hostData` of type `T*`.

**Encoding of buffer declarations** Listing 7.9 shows a template in yellow for the COL statements that are generated for the generalized buffer declaration in Listing 7.8. As can be seen in Listing 7.9, for each buffer declaration, a new COL array is created and the part of data within the provided range object, which has as upper bound `dim0 * dim1`, is copied to that new array using the `copy_hostdata_to_buffer` method. Then the `exclusive_hostdata_access` predicate is folded, which takes away the write permission to the just copied part of data. This ensures that the buffer has exclusive access to this part of data, as it can no longer be read or written to, and disallows the construction of another buffer with that part of data. This construction of another buffer is prevented by the fact that the `copy_hostdata_to_buffer` method that is called at the start of the generated code requires write permission, which was just removed. This technique to claim exclusive access does have as a result that write permission to the `hostData` is required at the point the buffer is constructed, even if the buffer, and thus the underlying `hostData`, is never written to.

```

1 someType someMethod() {
2     {
3         sycl::buffer<T,2> buff = sycl::buffer(data, sycl::range<2>(dim0,dim1));
4         ...
5     } // the buffer is destroyed here
6 }

```

Listing 7.8: General pattern of declaring and destroying a two-dimensional buffer.

```

1 someType someMethod() {
2     T[] buff;
3     buff = copy_hostdata_to_buffer(data, dim0 * dim1);
4     fold exclusive_hostdata_access(data, dim0 * dim1);
5     ...
6     copy_buffer_to_hostdata(data, buff);
7     unfold exclusive_hostdata_access(data, dim0 * dim1);
8 }

```

Listing 7.9: Excerpt of the COL code that is generated for the code in Listing 7.8.

**Always generating a linear COL array** As might have been noticed, a one-dimensional COL array is generated as a copy of a buffer's `hostData`, regardless of how many dimensions the `bufferRange` has, even though it was explained in subsection 7.1.2 (page 76) that the copy the buffer makes has the same number of dimensions as the provided range object. However, this is done deliberately: proving that values are copied correctly between a (one-dimensional) pointer and one-dimensional array is easier to prove for Z3. This does mean that when, for instance, a "three-dimensional" buffer is indexed through a data accessor as follows: `my_accessor[x][y][z]`, the three-dimensional index needs to be converted to a one-dimensional index. However, proving that that conversion is done correctly is easier to prove for Z3 than proving that a (one-dimensional) pointer is copied correctly to a multi-dimensional array.

**Necessary modifications if the assumption of exclusive access is wrong** If the assumption made in subsection 7.1.2.1 later turns out to be wrong, and the buffer does claim the entirety of its `hostData`, VerCors' support for buffers would have to be modified as follows: The predicate `exclusive_hostdata_access` described in Listing 7.5 should not take a `size` parameter, but use the size of the `hostData` pointer instead. At the time of writing, querying a pointer's size is not supported in VerCors, so support for that would have to be implemented as well.

**Manually unfolding `exclusive_hostData_access`** A concern one could have is that a user could decide to manually unfold the `exclusive_hostdata_access` predicate by writing:

```

unfold exclusive_hostdata_access(hostData, size);

```

This would cause the write permission to the `hostData` to be given back before the buffer holding it goes out of scope. This would allow for `hostData` to be read, written to, and to be put in a new buffer, while the current buffer is still in use, making the generated code unsound. However, when a user tries to use the `exclusive_hostdata_access` predicate, VerCors throws an error that it cannot find it. This is caused by the fact that predicates used by the user are resolved in the resolution phase, at which point the predicate does not exist yet, because it is generated in the transformation phase that comes after it. So it is not possible for the user to use this predicate.

**No reassigning of buffer variables** A restriction was added to variables holding a buffer: they are not allowed to be reassigned. This restriction was added because there was uncertainty about the effects of reassigning on the exclusive access to the `hostData` of the previous buffer value and the currently executing kernels with buffer accesses through that variable. This restriction is implemented as a syntactic check which checks for every variable re-assign that the variable does not reference a buffer object.

This does have as a result that if a user wants to verify their code with VerCors, they must rewrite every reassignment of a buffer variable to multiple buffer variable declarations instead, which might make their code less compact. However, the parameters of a buffer constructor are allowed to be dynamic. This can be used when rewriting code that initializes a buffer variable using reassignments based on a condition, such as shown in Figure 7.10a. Such code can be rewritten to use dynamic parameters in the buffer constructor in variable's declaration, as can be seen in Figure 7.10b.

```

1  sycl::buffer<int,1> b;
2  if (condition) {
3    b = sycl::buffer(dataA, rangeA);
4  } else {
5    b = sycl::buffer(dataB, rangeB);
6  }

```

(a) Example code of initializing a buffer variable using reassignments.

```

1  sycl::buffer<int,1> b = sycl::buffer(
2    condition ? dataA : dataB,
3    condition ? rangeA : rangeB
4  );

```

(b) Example code of Figure 7.10a rewritten to use dynamic parameters in the declaration constructor.

Figure 7.10: Example of rewriting a buffer variable initialization that uses reassignments to use dynamic parameters in the declaration constructor instead.

### 7.1.3.2 Buffer destructions

When the scope in which a buffer was declared ends, the buffer is destroyed. As mentioned in subsection 7.1.2, when a buffer is destroyed, the host waits on all submitted kernels with data accessors to the buffer, for which the contents need to be copied back to the buffer's `hostData`, to finish executing. When those kernels have terminated, the buffer's (altered) values are copied back to the `hostData`.

Listing 7.9 shows a template in blue for the COL statements that are generated for the scope of the buffer ending in Listing 7.8. To make the host wait on the kernels using the buffer to finish, a search is done internally to find all running kernels that have write access to the buffer. For each of the found kernels the same code is generated as for invocations of the `wait()` method on the kernel's event (which is described in section 6.2). After that code has been added, code is generated to copy the buffer's data back to the `hostData`. This code can be seen on lines 6-7 in Listing 7.9. First the method `copy_hostdata_to_buffer` is called to write the buffer's (altered) COL array to `data`. Then the predicate `exclusive_hostData_access` is unfolded to get the write permission for `data` back, which allows `data` to be used again by the user.

## 7.2 Data accessor declarations

This section first shares the definitions of the constructor of SYCL's `accessor` class and the values of the `access_mode` enum for which support was added to VerCors. Then, the semantics and the encoding into VerCors for declarations of data accessors is discussed.

### 7.2.1 Definitions of supported constructor and access modes

The constructor of the `accessor` class described below is supported in VerCors. The definition of the constructor is a simplified version of the description in the SYCL specification [72] (on page 164).

```
accessor(sycl::buffer<T, S> bufferRef, sycl::handler& cgh,  
↪ sycl::access_mode accessMode) → sycl::accessor<T, S, access_mode>
```

**In class:** `sycl::accessor<T, S, access_mode>`

**Arguments:**

- `bufferRef`: A reference to the buffer object the accessor should provide access to.
- `cgh`: A command group handler object, which determines what command group the accessor belongs to.
- `access_mode`: A supported value from the `sycl::access_mode` enum, which determines what kind of access the command group will have to the underlying buffer.

**Returns:** A new `sycl::accessor` object.

**Description:** Creates a data accessor which gives the command group represented by `cgh` `access_mode` access to the buffer object in `bufferRef`.

The access modes described below are supported in VerCors. The definition of these modes were taken from the SYCL specification [72] (on page 155).

**read\_only**

**In enum:** `sycl::access_mode`

**Description:** Provides *read-only* access to the provided buffer.

**read\_write**

**In enum:** `sycl::access_mode`

**Description:** Provides *read-write* access to the provided buffer.

### 7.2.2 Semantics

As mentioned in subsection 3.2.3.1 (page 22), data accessors give kernels access to memory objects on the host, such as buffers. They are usually declared inside command group scopes. A data accessor can be constructed with a `sycl::buffer` object, together with the current command group handler and an access mode. The access mode determines whether a user can read and/or write to the buffer the accessor wraps around. An example of such a data accessor construction can be found in yellow on line 2 in Listing 7.11, where a data accessor is constructed with *read-write* access to the buffer `buff`. It was chosen to only implement support for such accessor constructions, because they are the most simple way for the user to declare a (non-placeholder) data accessor, since it only requires a `buffer`, a command group handler and a access mode as arguments.

```
1  [&](sycl::handler& cgh) {  
2    sycl::accessor<int, 2> acc = sycl::accessor(buff, cgh, sycl::read_write);  
3  }
```

Listing 7.11: Example code where a data accessor is declared inside a command group. *The surrounding host code and the required call to the `parallel_for` method have been omitted here.*

For the remainder of this chapter, when accessor declarations are mentioned, only the declarations chosen to be supported should be considered. Furthermore, in this chapter and the ones that follow, when reading or writing *to the accessor* is mentioned, it will actually mean reading/writing *to the buffer object the accessor wraps around*.

### 7.2.2.1 No support for other access modes

Only support for the access modes *read-only* and *read-write* was added to VerCors. Permission-based separation logic (see subsection 3.3.5.4 (page 34)), only allows to express either read permission or write permission which also allows reading. Therefore, support for the *write-only* access mode would require the ability to semantically search kernel bodies for read-accesses to the accessor. Implementing this ability did not fit in the time-frame of this thesis, so no support was added for *write-only* access. The *no-init-write-only* access mode was omitted from support for the same reason. Lastly, the only extra feature the *no-init-read-write* access mode adds over the *read-write* access mode is that it avoids copying the accessor's data in some cases. So it was decided not worth the effort to implement it, as it would add little extra value to the supported subset of SYCL.

The unsupported access modes are used in the SYCL examples in the teaching materials in [20, 21, 36]. However, in those examples the unsupported access modes could be replaced by supported ones whilst maintaining a similar enough behaviour. However, the code might then run slightly less efficient in some cases, as the usage of *no-init-...* access modes might avoid the copying of some data. But this was not seen as enough reason to add support for the unsupported access modes.

### 7.2.3 Encoding into VerCors

Data accessor declarations are detected in the transformation phase, where every variable initialization to a data accessor constructor is seen as a data accessor declaration. For each of these declarations, several COL statements are generated and added to various sections of the generated COL code. What COL statements are added to each section of the generated code is described in the next subsections.

#### 7.2.3.1 Additions to the *host code*

A template of what COL statements are generated in the *host code* for the generalized data accessor declaration in Listing 7.12 are highlighted in yellow in Listing 7.13. When the *event-constructor* is called, which can be seen on line 5 in Listing 7.13, the following arguments are inserted: each data accessor's buffer array and the dimension sizes of that buffer.

```
1 someType someMethod() {
2   sycl::buffer<T,2> buff = sycl::buffer(data, sycl::range<2>(dim0, dim1));
3
4   queueObject.submit (
5     [&](sycl::handler& cgh) {
6       sycl::accessor<T,2> acc = sycl::accessor(buff, cgh, access_mode);
7
8       cgh.parallel_for(...);
9     }
10  );
11 }
```

Listing 7.12: General pattern of declaring a two-dimensional data accessor inside a command group.

```
1 someType someMethod() {
2   ...
3   T[] buff;
4   ...
5   sycl_event_ref = constructor(buff, dim0, dim1);
6   fork sycl_event_ref;
7   ...
8 }
```

Listing 7.13: Generalized view of the COL code that is added to the generated *host code* for the data accessor declaration in Listing 7.12.

### 7.2.3.2 Additions to the *event-class*

A template of what fields are added to the *event-class* for the generalized accessor declaration in Listing 7.12 are highlighted in yellow on lines 10-12 in Listing 7.14. As mentioned in subsection 6.1.3, code inside the kernel scope is inserted into the *kernel-parblock* in the generated COL code. Because the *kernel-runner* in which the *kernel-parblock* resides cannot take any parameters or return any value, the buffer that will be accessed cannot be passed directly to it. Therefore, for every data accessor declaration a field is added with the same type as the variable generated for the accessor's buffer, which is referenced to as the *buffer-field* for the remainder of this chapter. This field can be seen on line 10. Furthermore, for each dimension of the accessor's buffer an integer field is added, holding the size of the dimension. These fields will be referenced to as *buffer-range-fields* for the remainder of this chapter. These fields can be seen on lines 11 and 12. These fields can then be used in the *kernel-parblock* and read when the kernel terminates.

```
9  class SYCL_EVENT_CLASS {
10  T[] acc;
11  int acc_dim0;
12  int acc_dim1;
13
14  context Perm(this.acc_dim0, read);
15  context Perm(this.acc_dim1, read);
16  context Perm(this.acc, read);
17  context \array(this.acc, this.acc_dim0 * this.acc_dim1);
18  context Perm(this.acc[*], accessMode == read_write ? write : read );
19  run {
20    par SYCL_BASIC_KERNEL(ranges in index-space)
21    context Perm(this.acc_dim0, read);
22    context Perm(this.acc_dim1, read);
23    context Perm(this.acc, read);
24    context \array(this.acc, this.acc_dim0 * this.acc_dim1);
25    { kernel body }
26  }
27 }
```

Listing 7.14: Generalized view of the COL code that is added to the generated *event-class* for the data accessor declaration in Listing 7.12. Code inside gray frames is conditional: based on the condition, one of the yellow parts is inserted. The generated kernel dimension fields and their specifications have been ommitted here.

### 7.2.3.3 Additions to the *kernel-runner* and *kernel-parblock*

For every data accessor declaration, permission statements are added to the pre- and post-conditions of the *kernel-runner* and the *kernel-parblock*. As can be seen in yellow on lines 14-18 and lines 21-24 in Listing 7.14, there are four statements that are the same. The first three of these statements make sure that all accessor-related fields in the *event-class* can be read, but not altered, by each work-item. The fourth statement makes sure that the length of each accessor's *buffer-field* equals the multiplication of all the corresponding *buffer-range-fields*, which is the same length as the array that is generated in the *host code* when a buffer is declared. This statement was added because when indexing the *buffer-field*, the length of it needs to be known to verify that the index is within its bounds. The *kernel-runner* contains one additional permission statement which gives the body of the *kernel-runner* write permission to all elements of the *buffer-field* if the accessor's access mode is *read-write*, or read permission if the accessor's access mode is *read-only*. These permission statements are added before any user-defined pre- and post-conditions, to allow the user-defined conditions to access the accessor-related fields.

To interact with the elements of the *buffer-field* array inside the *kernel-parblock*, the user needs to specify what elements of the *buffer-field* array each work-item is allowed to access and/or alter. This should be done in the contract of the kernel declaration, as that contract is merged with the contract of the *kernel-parblock*. An example of this will be shown in section 7.3, which covers data accessor usage.

### 7.2.3.4 Additions to the *event-constructor*

A template of what statements are added to the contract of the *event-constructor* for the generalized data accessor declaration in Listing 7.12 can be seen in yellow on lines 30-37 in Listing 7.15. As can be seen, the first five statements added to the constructor contract for each accessor are the same as the statements added to the *kernel-runner*'s contract. These were added to make sure that pre-conditions of the *kernel-runner* can be satisfied. The only difference is that they are only added as post-conditions here, as in the constructor's pre-state, the new instance of the *event-class* does not exist yet. Then, as can be seen on lines 35-37, it is ensured that each of the generated *buffer-fields* and *buffer-range-fields* in the *event-class* have same values the constructor's parameters, which will contain the actual buffer array and buffer ranges from the *host code*, as explained in subsubsection 7.2.3.1. Since the constructor's body is not implemented, these post-conditions will all automatically be assumed to be true.

```

28 ensures \result != null;
29 ensures \typeof(\result) == \type(SYCL_EVENT_CLASS);
30 ensures Perm(\result.acc_dim0, read);
31 ensures Perm(\result.acc_dim1, read);
32 ensures Perm(\result.acc, read);
33 ensures \array(\result.acc, \result.acc_dim0 * \result.acc_dim1);
34 ensures Perm(\result.acc[*], access_mode == read_write ? write : read);
35 ensures \result.acc == acc;
36 ensures \result.acc_dim0 == acc_dim0;
37 ensures \result.acc_dim1 == acc_dim1;
38 ensures idle(\result);
39 SYCL_EVENT_CLASS event_constructor(T[] acc, int acc_dim0, int acc_dim1);

```

Listing 7.15: Generalized view of the COL code that is added to the generated *event-constructor* for the data accessor declaration in Listing 7.12. Code inside gray frames is conditional: based on the condition, one of the yellow parts is inserted. The generated dimension parameters and specifications have been omitted here.

### 7.2.3.5 Kernel execution order


As mentioned in subsubsection 3.2.2.1 (page 18), when multiple kernels are submitted to a queue, their order of execution is determined by their requisites at runtime, which, in turn, are (partially) determined by their data accessors. Support for the influence of data accessors on the execution order of kernels was added to VerCors as well. The kernel execution order is determined by the rules below, where accessors are considered equivalent if they wrap around the same buffer object.

If a new kernel is submitted...

- ... and it has a disjoint set of data accessors with the currently running kernels, it is immediately started after its submission and executed at the same time as those kernels. This is similar to SYCL's actual kernel execution ordering.
- ... and it has an intersection of data accessors with the currently running kernels, where all data accessors in the intersection provide *read-only* access, the kernels are ordered the same as in the first rule. This is also similar to SYCL's actual kernel execution ordering.
- ... and it has an intersection of data accessors with the currently running kernels, where one or more data accessors in the intersection provide *read-write* access, then all the running kernels with those data accessors are joined (which also happens when `wait()` is called on their linked event, see subsection 6.2.3 (page 70)) before starting the new kernel.

This is mostly similar to SYCL's actual kernel execution ordering, except that in VerCors, the currently running kernels are waited on before any other host code is executed, whereas in SYCL the currently running kernels can finish later. This is illustrated in Listing 7.16, where the  $CG_b$  needs to wait on  $CG_a$  to finish executing and before starting to execute itself, which we will call the *turnover point*. In VerCors, the *turnover point* is at the pink marker, and in SYCL, the *turnover point* can be at any position indicated by the blue marker.

```

1      {
2          sycl::buffer<int, 1> buffer1 = sycl::buffer(hostData, range);
3          queue.submit(CG_a(buffer1_*,...));
4          host code
5          queue.submit(CG_b(buffer1_RW,...));
6          
7          host code
8      }
9      host code

```

Listing 7.16: Indications of where the *turnover point* can be in VerCors’ encoding and SYCL. *RW* stands for read-write access and *\** stands for any access mode.

This would make the verification of kernels unsound if the position of the *turnover point* influences the results of host code on line 7 in Listing 7.16, as those influences would then not be present in VerCors encoding, which means that VerCors could prove a certain condition to be true even though it is not true in reality, and vice versa, which would make the encoding unsound.

However, this is not the case for the following reason: Running kernels have a mostly disjoint heap from the host code. The only data the heap of a kernel and the heap of the host code can share are buffer objects. However, only kernels can read and write to them: the host code can neither read nor write to them and their `hostData`. So the buffers and their underlying `hostData` are unusable parts of the host code’s heap. The host code can submit new kernels that use the buffer, however, since those kernels also adhere to the aforementioned kernel ordering, they cannot cause an unsound verification either. This means that the ‘useable’ parts of the heaps of kernels and host code are always disjoint, which means that the results of host code cannot be influenced by the aforementioned kernel ordering.

A buffer’s `hostData` becomes accessible to the host code after it has been destroyed, such as on line 9 in Listing 7.16. However, the destruction of the buffer waits on all kernels using that buffer to terminate, so after that point only the host code is the only running component that has access to the buffer, which means its results still cannot be influenced.

Another interesting point is that if, for example,  $CG_a$  also required access to any buffers that  $CG_b$  did not require access to, then in the encoding those buffers could be released sooner than in SYCL. However, as mentioned previously, the host code cannot access the contents of the buffer till it is destroyed, at which point  $CG_a$  has terminated. Also, kernels using those buffers could be started sooner in the encoding than in SYCL, but their ordering stays the same, so this is not an issue either.

### 7.2.3.6 Multiple accessors with same buffer object

It is possible in SYCL to declare multiple data accessors in the same kernel for the same buffer objects. This has an effect on the kernel’s requested access to the buffer: the requested (and supported by VerCors) access modes are combined into a single access mode in SYCL and also in the encoding using the following rules (see [72], page 26-27):

Access mode 1	Access mode 2	Combined access mode
<i>read_only</i>	<i>read_only</i>	<i>read_only</i>
<i>read_only</i>	<i>read_write</i>	<i>read_write</i>
<i>read_write</i>	<i>read_write</i>	<i>read_write</i>

For these multiple data accessors a single *buffer-field* and a single set of *buffer-range-fields* are created, and the permission given to the elements of the *buffer-field* is the permission that corresponds with the combined access mode. Then all usages of the data accessors variables are encoded to use that single *buffer-field*. This special treatment of multiple data accessors for the same buffer is required, as changes written to one accessor should be present in all the other accessors as well. With multiple *buffer-fields* this would require applying the change to every *buffer-field* every time they are written to, which is more complicated than creating a single *buffer-field*.



One downside to this single *buffer-field* approach is that if a kernel contains a data accessor `a` with *read-only* access to a buffer and one or more data accessors to the same buffer with *read-write* access, users will be allowed to write to `a` in the kernel body, because the kernel body has *read-write* access to the *buffer-field* due to the access mode combination. So no error will be shown to the user in that case. However, this does not make the encoding unsound, as the kernel overall is allowed to write to the buffer, and the resulting state of the buffer is not affected by what accessor was used to write to it.

#### 7.2.3.7 Recurring statements in contracts

As was noted in the previous subsections, four statements are repeated in the contracts of the *kernel-runner*, *kernel-parblock*, and *event-constructor*. An inline predicate could have been added with those four statements as its body. And then this predicate could have been called in the contracts to avoid repetition and to increase readability. However, the decision was made not to do this for the following reason: All calls to inline predicates in COL code are replaced with the bodies of these statements further along in the transformation phase. This means the generated COL code, after all transformation passes have been executed, ends up with the same code as when inline predicates are not used. So using inline predicates just adds an extra transformation task. Moreover, the intermediate COL code generated by the C++ transformation is usually not read by humans (apart from the snippets shown in this report), so readability cannot be used as a justification for using inline predicates in this case.

#### 7.2.3.8 Lambda captures of command groups and kernel declarations

The following restrictions, which were found by testing for what lambda captures the OneAPI DPC++ SYCL compiler [40] throws compilation errors, are present for lambda captures in command groups and kernel declarations:

If there are no data accessors present, the captures of the command group lambda have no effects, as local variable usage is not allowed in them (see subsection 6.1.3.3). Furthermore, if data- and local accessors are not used inside the kernel declaration lambda, the captures of the kernel declaration lambda have no effects either, for the same reason.

The `bufferRef` argument of accessor is passed-by-reference. This means that the command group declaration which, as mentioned in the previous chapter, is a lambda expression, must capture the `bufferRef` argument as pass-by-reference (`[&]`). The kernel declaration also is also a lambda expression. When data- or local accessors (see subsection 8.2.2.1) are used in its scope, they must be captured as pass-by-value (`[=]`).

Since lambda captures are ignored in VerCors, lambdas with wrong captures are considered valid. However, SYCL compilers, such as the DPC++ SYCL compiler [40], throws errors when the lambda captures are wrong in the aforementioned lambda expressions. So as long as the user only verifies SYCL programs in VerCors that can be compiled without errors by a SYCL compiler, this should not be an issue.

## 7.3 Accessing elements of a buffer through a data accessor

This section first shares the definitions of the SYCL methods related to data accessors for which support was added to VerCors. Then the semantics and the encoding into VerCors for accessing elements of a buffer through SYCL's accessors is discussed.

### 7.3.1 Definitions of supported methods

The method `get_range` in SYCL's `accessor` class described below is supported in VerCors. The definition is a simplified version of the method description in the SYCL specification [72] (on page 196).

**`get_range()`  $\rightarrow$  `sycl::range<S>`**

**In class:** `sycl::accessor<_, S, _>`

**Returns:** A `sycl::range` object where each dimension contains the size of the corresponding dimension of the accessor's buffer.

**Description:** Returns a `sycl::range` object where each dimension contains the size of the corresponding dimension of the accessor's buffer. This method can only be invoked in VerCors if on the result of this method the method below is called. The definition is a simplified version of the method description in the SYCL specification [72] (on page 247).

**`get(int dimension)`  $\rightarrow$  `int`**

**In class:** `sycl::range<S>`

**Arguments:**

- `dimension`: What dimension of the range to query.

**Returns:** The size of of the requested dimension in the range.

**Description:** Returns the size of of the requested dimension in the range.

### 7.3.2 Semantics

The elements of an accessor's buffer can be read and written to by using subscripts on the data accessor object, in the same way conventional C++ arrays are accessed. How many subscripts are required to access an element of the buffer through a data accessor `acc` depends on the number of dimensions in the buffer's range:

- `range<1>(dim0)  $\implies$  use one subscript: acc[x],  
where x is an integer and  $0 \leq x < \text{dim0}$ .`
- `range<2>(dim0, dim1)  $\implies$  use two subscripts: acc[x][y],  
where x and y are integers and  $0 \leq x < \text{dim0}$  and  $0 \leq y < \text{dim1}$ .`
- `range<3>(dim0, dim1, dim2)  $\implies$  use three subscripts: acc[x][y][z],  
where x, y and z are integers and  $0 \leq x < \text{dim0}$ ,  $0 \leq y < \text{dim1}$  and  $0 \leq z < \text{dim2}$ .`

To give an example, on line 8 in Listing 7.17 elements are accessed using two subscripts, because the data accessor `acc` holds a two-dimensional buffer. Users can also subscript the data accessor in their kernel contracts. An example of this can be seen on line 5 in Listing 7.17. However, if the data accessor provides *read-only* access, statements that require write access to the data accessor, such as acquiring and releasing write permission to an element on line 5, are not allowed.

```

1  [&](sycl::handler& cgh){
2      sycl::accessor<T,2> acc = sycl::accessor(buff, cgh, access_mode);
3
4      cgh.parallel_for(sycl::range<2>(2, 5),
5          /*@
6              context idx0 < acc.get_range().get(0);
7              context idx1 < acc.get_range().get(1);
8              context Perm(acc[idx0][idx1], write);
9          */
10         [=](sycl::item<2> it) {
11             acc[idx0][idx1] = value;
12         }
13     );
14 }

```

Listing 7.17: General pattern of subscripting a two-dimensional data accessor. *The surrounding method invocation to submit has been omitted here.*

### 7.3.3 Encoding into VerCors

To implement support for accessing elements of buffers through data accessors, additions were made to VerCors' transformation phase. These additions are described in this subsection.

#### 7.3.3.1 Additions to the transformation phase

When a data accessor is subscripted, it is encoded into a one-dimensional subscript on the data accessor's *buffer-field*. If the subscript is multi-dimensional, the method `linearize2` or `linearize3` (defined in subsubsection 6.3.3.1) is called to linearize the multi-dimensional subscript to a one-dimensional subscript. An example of an encoded data accessor subscript can be seen in Listing 7.18, where the encoded subscript is given for the two-dimensional subscript on the data accessor `acc` on line 11 of Listing 7.17.

```

this.acc[sycl::linearize2(idx0, idx1, this.acc_dim0, this.acc_dim1)] =
                                                                    ↪ value;

```

Listing 7.18: Generalized view of the generated COL code for the two-dimensional subscript on the data accessor `acc` on line 11 of Listing 7.17

**Bound-checking on subscripts** When subscripting a data accessor, VerCors checks whether these subscripts are within the range of the data accessor's *buffer-field*. However, due to the complexity of the generated code, VerCors cannot infer this automatically from the generated code and permissions statements. So, to be able to verify that the subscripts are within range, the user often needs to state in the kernel's contract, for each subscript that is used, that it is within the bounds of the dimension it indexes. An example of such an addition can be seen on lines 6-7 of Listing 7.17, where for the subscripts `[idx0]` and `[idx1]`, which are used on lines 8 and 11, it is stated that `idx0` is smaller than the size of the first dimension, and `idx1` is smaller than the size of the second dimension. This extra information is then checked by VerCors, and if it is true, it is added to its knowledge base. And with that extra information VerCors can successfully verify that the subscripts on `acc` are within range.

**Retrieving the sizes of the dimensions** Support was added for the `get_range()` method of the accessor class in combination with the `get(int dimension)` method of the `range` class to be able to retrieve the size of each dimension of an accessor. This support was added because the user might need to state that a subscript is smaller than the size of the corresponding dimension, as described in the previous paragraph. It is implemented as follows: when the pattern `acc.get_range().get(i)` is found, where `acc` is an accessor and `i` the index of a dimension, it is replaced by a subscript `[i]` on the COL sequence holding the *buffer-range-fields* of `acc`, which selects the *buffer-range-field* corresponding to the requested dimension.

## 7.4 Updating a buffer's contents

In this section, the semantics and the encoding into VerCors for data accessors updating the contents of a buffer is discussed.

### 7.4.1 Semantics

When a user updates an element of a buffer through a data accessor, the data accessor writes this update to the buffer. However, this is not the case in the generated COL code, where all updates are written to the data accessor's *buffer-field* in the *event-class*. The contents of the data accessor's *buffer-field* need to be copied to the actual buffer array in the *host code* to actually update the buffer. This needs to happen before another kernel attempts to read the buffer, otherwise it might read old values.

### 7.4.2 Encoding into VerCors

It was chosen to copy the updated data accessor's *buffer-field* back to the actual buffer array when the kernel is terminating. At that point in time no further updates will be made to the data accessor's *buffer-field*, and other kernels cannot access the buffer yet, as they need to wait till the kernel has finished terminating.

To show how the contents of *buffer-field* are copied back, a general template of what COL code is generated for the kernel termination in Listing 7.19 is shown in yellow in Listing 7.20. First the *kernel-runner* is joined, as was explained in subsection 6.2.3 (page 70). Then, immediately after that, each buffer to which the kernel had *read-write* access is updated with the current contents of the corresponding *buffer-field*.

```
1 someType someMethod(sycl::queue q) {
2     sycl::event ev = q.submit(
3         [&](sycl::handler& cgh) {
4             sycl::accessor<T,2> acc = sycl::accessor(buff, cgh, access_mode);
5             ...
6         }
7     );
8     ev.wait();
9 }
```

Listing 7.19: General pattern of the host waiting on a kernel containing a two-dimensional data accessor to finish executing. *The buffer declaration and required call to the `parallel_for` method have been omitted here.*

```
1 someType someMethod() {
2     T[] buff;
3     ...
4     join sycl_event_ref;
5     access_mode == read_write ? buff = sycl_event_ref.acc : ;
6     ...
7 }
```

Listing 7.20: Generalized view of the COL code added to the *host code* for the terminating kernel in Listing 7.19.

## 8 Added support for SYCL's address spaces

The fourth research question (RQ4, see subsection 2.1 (page 7)) states that SYCL's global, local, and private address spaces should be supported to be able to control what work-items share what data. In this chapter, the added support to VerCors for these components is discussed.

### 8.1 Global and private address spaces

VerCors support for the global and private address spaces was already implemented and discussed in the previous research questions. Data accessors, for which support was added in RQ3 (chapter 7 (page 76)), provide access to global memory, as the contents of the buffer it holds are copied to the device's global memory. All work-items in all work-groups can access it, and the data is also persistent across kernel executions, which was also a requirement, because data accessors write the modified data back to their buffers when kernels terminates. These buffers can be shared used in multiple kernels. As mentioned in, subsection 3.2.3.2 (page 23) a variable is automatically allocated in private memory when declared in the `sycl::parallel_for` scope. Support for `parallel_for` invocations was added in RQ2 (chapter 6 (page 58)), where all code inside the `parallel_for`'s lambda parameter is copied to the innermost generated *kernel-parblock*.

### 8.2 Local address space

In this section, the addition of support for reasoning about SYCL's local address space is described. This section first shares the definition of the constructor of SYCL's `local_accessor` class for which support was added to VerCors. Then the semantics and the encoding into VerCors for declarations and usage of SYCL's local accessors are discussed.

#### 8.2.1 Definition of supported constructor

The constructor of the `local_accessor` class described below is supported in VerCors. The definition of the constructor is a simplified version of the description in the SYCL specification [72] (on page 193).

```
local_accessor<T>(sycl::range<S>& allocationSize, sycl::handler& cgh)
    → sycl::local_accessor<T, S>
```

**In class:** `sycl::local_accessor<T, S>`

**Arguments:**

- `T`: A data type that will be the type of each element in the resulting array.
- `allocationSize`: A `sycl::range` object which describes how the resulting array will structured.
- `cgh`: A command group handler object, which determines what command group the local accessor belongs to.

**Returns:** A new `sycl::local_accessor` object.

**Description:** Creates a new uninitialized (multi-)dimensional array, whose structure is described by `bufferRange`, where each work-group in the command group `cgh` has its own separate copy of the array.

The `get_range` method in SYCL's `local_accessor` class is supported in VerCors as well. It has the same definition as the `get_range` method in SYCL's `accessor` class, which is described in subsection 7.3.1. Similar to the `accessor` class, `get_range` can only be invoked in VerCors if on the result of this method the method `get(int dimension)` is called.

## 8.2.2 Semantics

There are two common methods to declare variables in the local address space. The first method is to declare local accessors. The second method is to split up a kernel declaration into a work-group section and a work-item section and declare the variable inside the work-group section. Both these methods are described in this subsection.

### 8.2.2.1 Local accessors

Local accessors create an uninitialized (multi-)dimensional array in each work-group's local memory. They are usually declared inside command group scopes. A local accessor can be constructed with a `sycl::range` object, together with the current command group handler. The user must also specify the type of the elements as a generic template parameter. An example of such a local accessor construction can be found in yellow on lines 2-3 in Listing 8.1, where a local accessor is constructed with a two-dimensional range, whose elements are integers.

```

1  [&](sycl::handler& cgh) {
2      sycl::local_accessor<int, 2> local_acc =
3          ↪ sycl::local_accessor<int>(sycl::range<2>(2,4), cgh);
4      cgh.parallel_for(...);
5  }
```

Listing 8.1: Example code where a two-dimensional local accessor is declared inside a command group. *The surrounding host code has been omitted here.*

### 8.2.2.2 Split-up kernel declaration

A kernel declaration can be split up into a work-group section and a work-item section. This split can be done in the command group scope by, instead of invoking the `parallel_for` method, invoking the `parallel_for_work_group` method, which in turn invokes the `parallel_for_work_item` method. Then, every variable declared inside the work-group section will be added to each work-group's local memory. An example of such a variable declaration can be found in yellow on line 4 in Listing 8.2, where an integer array is constructed in the local address space with a two-dimensional range.

```

1  [&](sycl::handler& cgh) {
2      cgh.parallel_for_work_group(sycl::range<1>(2), sycl::range<1>(4),
3          [=] (sycl::group<1> g) {
4              int local_array[2][4];
5
6              g.parallel_for_work_item(...);
7          }
8      );
9  }
```

Listing 8.2: Example code where a two-dimensional array is declared inside a `parallel_for_work_group` method, which declares it in each work-group's local memory. *The surrounding host code has been omitted here.*

### 8.2.2.3 The declaration approach that was chosen

Declaring variables in the local address space using the split-up kernel method has two advantages over using the local accessor method: First of all, the separation into a work-group section and a work-item section makes it more clear to the user in what address space they are declaring a variable: variables in work-group sections are declared in local memory and variables inside work-item sections are declared in private memory. Secondly, it allows more flexibility in the type of the declared local variable than local accessors, as local accessors always create an array.

Despite these advantages over the local accessor method, it was chosen to support the local accessor method instead of the split-up kernel method. The reason for this is that it was already decided in RQ2 to not support alternate kinds of kernel declarations (see subsection 6.1.2.1 (page 60)). Support for this new kernel declaration structure would have to be implemented from scratch, which would have taken a considerable amount of time. Support was only added for the local accessor constructions described in subsection 8.2.2.1, because they are the most simple way for the user to declare a local accessor, since it only requires the type of the elements, a `range` object and the command group handler as arguments. For the remainder of this chapter, when local accessor declarations are mentioned, only the declarations chosen to be supported should be considered.

**Local accessor usage** Since local accessors are in essence arrays, they can also be subscripted as such. An example of this can be seen on line 13 in Listing 8.3, where each work-item of the work-group reads an element of the two-dimensional `local_acc` local accessor.

## 8.2.3 Encoding into VerCors

To implement support for local accessors, additions were made to VerCors' transformation phase. These additions are described in this subsection.

### 8.2.3.1 Local accessor declarations

Local accessor declarations are detected in the transformation phase, where every variable initialization to a local accessor constructor inside the command group for an ND-range kernel is seen as a local accessor declaration. For each of these declarations, a COL array variable with the same structure as the provided range is declared in the outer *kernel-parblock*, right above the declaration of the inner *kernel-parblock*. This gives every inner *kernel-parblock*, which represents the work-items in a single work-group, its own separate version of that array variable. Listing 8.4 shows a template in yellow for the COL array variable that is generated for the generalized local accessor declaration in Listing 8.3.

```
1  [&](sycl::handler& cgh) {
2      sycl::local_accessor<T, 2> local_acc =
3          ↪ sycl::local_accessor<T>(sycl::range<2>(50,20), cgh);
4
5      cgh.parallel_for(
6          sycl::nd_range<2>(sycl::range(250,80), sycl::range(50,20)),
7          /*@
8              context it.get_local_id(0) < local_acc.get_range().get(0);
9              context it.get_local_id(1) < local_acc.get_range().get(1);
10             context Perm(local_acc[it.get_local_id(0)][it.get_local_id(1)], read);
11             */
12             [=] (sycl::nd_item<2> it) {
13                 T x = local_acc[it.get_local_id(0)][it.get_local_id(1)];
14             }
15         );
16 }
```

Listing 8.3: General pattern of declaring a two-dimensional local accessor inside a command group. *The surrounding host code has been omitted here.*

```

1 class SYCL_EVENT_CLASS {
2     /*@ kernel contract */
3     run {
4         par SYCL_ND_RANGE_KERNEL_WORKGROUPS (
5             int GROUP_ID0 = 0 .. group_dim0, int GROUP_ID1 = 0 .. group_dim1
6         ) /*@ kernel contract */ {
7             T [][] local_acc;
8             local_acc = new T [50][20];
9             par SYCL_ND_RANGE_KERNEL_WORKITEMS (
10                int LOCAL_ID0 = 0 .. local_dim0, int LOCAL_ID1 = 0 .. local_dim1
11            ) /*@ kernel contract */ { /* kernel body */ }
12        }
13    }
14 }

```

Listing 8.4: Generalized view of the COL code that is added to the generated *event-class* for local accessor declarations. *The generated kernel dimension fields and their specifications have been omitted here.*

**Similarities to the CUDA and OpenCL implementations** When inspecting the implementation of CUDA and OpenCL support in VerCors, it appears to handle variable declarations for the local address space the same as the implemented SYCL support described in this subsection: the variables are inserted in that implementation’s generated outer *kernel-parblock*, right before the declaration of its generated inner *kernel-parblock*. This approach is also shown in Listing 16 on page 114 of [27], which shows the results of translating an OpenCL kernel to PVL.

### 8.2.3.2 Local accessor usage

No code had to be added to add support for subscribing local accessors. This is caused by the fact that they are encoded into a locally declared COL array, for which subscripts were already supported.

**Local accessors declarations in command groups for basic kernels** The SYCL specification ([72], page 191) states that local accessors must not be used in `parallel_for` methods that take a `range` parameter, i.e. inside basic kernels. When this happens, SYCL compilers are supposed to throw an exception when that kernel is submitted to a queue. In VerCors, a similar mechanism was implemented which shows an error to the user when a local accessor variable is used inside a basic kernel. Since the specification only mentions local accessor *usage*, and not local accessors *declarations*, it was decided to not throw an error when finding local accessor declarations, but to simply not generate any COL code for them.

**Bound-checking on subscripts** When subscribing a local accessor, VerCors checks whether these subscripts are within the range of the generated COL array. When an `(nd_)item` method from subsection 6.3.1 is used in the subscript VerCors cannot infer the actual value of the subscript due to the complexity of the generated code. So, to be able to verify that the subscripts are within range, the user often needs to state in the kernel’s contract, for each subscript that is used, that it is within the bounds of the dimension it indexes, similar to how it is often necessary for data accessors, as described in subsection 7.3.3.1.

**Retrieving the sizes of the dimensions** Support was added for the `get_range()` method of the `local_accessor` class in combination with the `get(int dimension)` method of the `range` class to be able to retrieve the size of each dimension of a local accessor. This support was added because the user might want to state for a subscript that it is smaller than the size of the corresponding dimension. It is implemented as follows: when the pattern `local_acc.get_range().get(i)` is found, where `local_acc` is a local accessor and `i` the index of a dimension, it is replaced by a subscript `[i]` on a COL sequence holding the dimension sizes of `local_acc`, which selects the requested dimension size.



## 9 Validation of added support

It is important that the encoding of SYCL into VerCors is sound, where *sound* should be understood as: If VerCors successfully verifies that a SYCL program adheres to its specifications, this should actually be true, and if VerCors states that the specifications cannot be satisfied, there must exist at least one reachable program state in the SYCL program for which the specifications are not satisfied. When the verifier of a program is unsound, its results should be considered unsound as well. Therefore, an effort was made to verify the soundness of the encoding of SYCL into VerCors. This effort is discussed in this chapter.

### 9.1 Formal proof

The soundness of the encoding of SYCL into VerCors has not been proven formally. Formally proving this soundness would take a considerable amount of effort, as there are quite a number of nuances in the way SYCL executes kernels and handles memory, for which the soundness of the encoding into VerCors should be proven. Furthermore, the SYCL specification, as indicated in various sections in this thesis, is at times unclear about how a certain construct behaves or is allowed to behave. This also makes any formal proof about the soundness of the encoding not entirely formal, as the encoding would partially be verified against assumptions about some behaviours in SYCL, instead of its actual (undocumented) behaviour. Therefore it was decided not to formally proof the soundness of the encoding, but to use the techniques described in the next sections to informally verify the encoding's soundness instead.

### 9.2 Integration testing

The implementation of the encoding described in this thesis was tested with integration tests. VerCors already had a framework for this, which takes program files and their expected results, which can be a successful verification or a certain error code being thrown, and tests whether verifying the program file with VerCors gives the expected result. In this section, first an overview of the entire test-set used in this thesis to investigate the soundness of the implementation is given. Then some examples of these tests are shown. Finally, the time it took to execute the tests is discussed.

#### 9.2.1 Overview of tests

For every SYCL concept in the research questions, several 'positive' and 'negative' tests were added. The positive tests are program files that should verify successfully. The negative tests are program files which should fail to verify or cause VerCors to throw an error.

The number of positive and negative tests per construct can be found in Table 9.1. There are a few positive tests for every construct which test whether the declarations and behaviours of the constructs that are described in this thesis are actually supported by the implementation. However, not every variation of each declaration and behaviour are covered, such as all combinations of possible data types and generic template arguments for declarations, as that would be a substantial task. Secondly, there are tests for each `(nd_)item` method that returns a (linearized) id, where they are used as subscripts for data- or local accessors in permission statements to make sure that no injectivity errors are thrown. Furthermore, for every indication given in this thesis that a certain error should be shown to the user in a certain situation, an integration test was added to ensure this is the case. An overview of these expected errors can be found in Appendix C. Last of all, some tests were added that test whether errors are thrown when the generic template arguments of buffers and data- and local accessors do not match the actual type returned by their constructors. All these tests pass.

RQ	Construct	Nr. of positive tests	Nr. of negative tests
1	Basic C++ constructs	18	1
2	Basic and ND-range kernels	3	12
2	Item methods (injectivity)	12	0
3	Buffers	3	13
3	Data accessors	8	15
4	Local accessors	2	10
	<b>Total</b>	<b>46</b>	<b>51</b>

Table 9.1: Number of positive and negative tests per supported construct.

### 9.2.1.1 Test files

All the test files used in this test-set can be found in the VerCors source code [75]. The test files for basic C++ constructs, can be found in the folder `examples/concepts/cpp/` and the expected results in `test/main/vct/test/integration/examples/CPPSpec.scala`. The test files for the SYCL constructs can be found in `examples/concepts/sycl/` and the expected results in `test/main/vct/test/integration/examples/SYCLSpec.scala`.

## 9.2.2 Examples

In this subsection, first the expected result descriptions in VerCors' test framework are explained. After this four tests taken from the tests for buffers are discussed.

### 9.2.2.1 Test descriptions

To test that a program stored in the file `example.cpp` verifies successfully, one would have the following expected result description in the VerCors' test framework:

```
vercors should verify using silicon example "example.cpp"
```

To test that VerCors throws an error with code `code` when parsing, transforming, or verifying a program stored in the file `example.cpp`, one would have the following expected result description in the framework:

```
vercors should error withCode "code" example "example.cpp"
```

To test that the verification of a program stored in the file `example.cpp` (which could successfully be transformed) fails with a code `code`, one would have the following expected result description in the framework:

```
vercors should fail withCode "code" using silicon example "example.cpp"
```

### 9.2.2.2 Tests from the buffer test-set

In this subsection, two positive and two negative tests taken from the tests for buffers are explained.

**Positive test 1** The first positive test is shown in Listing 9.2. It tests whether the supported variants of buffer declarations are accepted by VerCors and do not cause any errors or fails. It declares three buffers, each with a different type, and different generic template arguments and ranges with different numbers of dimensions in the constructor.

```

vercors should verify using silicon example
    ⇨ "concepts/sycl/buffers/BufferDeclarations.cpp"
1  #include <sycl/sycl.hpp>
2
3  /*@
4   requires \pointer(a, 10, write);
5   requires \pointer(b, 10, write);
6   requires \pointer(c, 10, write);
7  */
8  void test(bool[] a, int* b, float[] c) {
9      sycl::buffer<bool, 1> aBuffer = sycl::buffer<bool>(a, sycl::range<1>(10));
10     sycl::buffer<int, 2> bBuffer = sycl::buffer<int, 2>(b, sycl::range<2>(2, 5));
11     sycl::buffer<float, 3> cBuffer = sycl::buffer(c, sycl::range<3>(2, 5, 1));
12 }

```

Listing 9.2: Test program BufferDeclarations, which should verify successfully.

**Positive test 2** The second positive test is shown in Listing 9.3. It tests whether writing to a buffer's `hostData` after the buffer has been destroyed by its scope ending is accepted by VerCors and does not cause any errors or fails. It declares a buffer with `a` as its `hostData` in the scope on line 6. Then on line 9, where the buffer has been destroyed by the scope ending after line 7, it assigns a value to an element of `a`.

```

vercors should verify using silicon example
    ⇨ "concepts/sycl/buffers/ReleaseDataFromBuffer.cpp"
1  #include <sycl/sycl.hpp>
2
3  //@ requires \pointer(a, 10, write);
4  void test(bool* a) {
5      {
6          sycl::buffer<bool, 1> aBuffer = sycl::buffer(a, sycl::range<1>(10));
7      }
8      // Buffer should be destroyed now, so permissions should have been given back
9      a[5] = true;
10 }

```

Listing 9.3: Test program ReleaseDataFromBuffer, which should verify successfully.

**Negative test 1** The first negative test is shown in Listing 9.4. It tests whether reading a buffer's `hostData` whilst the buffer still exists causes the verification of the program to fail. It declares a buffer with `a` as its `hostData`. Then on line 6, `a` is subscripted, and thus read, which is not allowed. So the verification process should fail because of insufficient permission to `a`. This kind of failure has the `ptrPerm` error code so an error with that code should be thrown.

```

vercors should fail withCode "ptrPerm" using silicon example
    ⇨ "concepts/sycl/buffers/ReadDataInBufferScope.cpp"
1  #include <sycl/sycl.hpp>
2
3  //@ requires \pointer(a, 10, write);
4  void test(bool* a) {
5      sycl::buffer<bool, 1> aBuffer = sycl::buffer(a, sycl::range<1>(10));
6      bool x = a[5]; // Not allowed
7  }

```

Listing 9.4: Test program ReadDataInBufferScope, for which verification should fail with the code `ptrPerm`.

**Negative test 2** The second negative test is shown in Listing 9.5. It tests whether declaring a buffer with a range larger than VerCors’ inferred size of the `hostData` parameter causes VerCors to throw an error to the user that the buffer cannot be constructed. Pointer `a` is assumed to have at least size 10, because that is what the pre-condition of the method states on line 3. It is unknown how much bigger `a`’s size is than 10, or whether it is longer at all. However, on line 6, a buffer is declared with `a` as its `hostData`, and range with total size 11. So the VerCors should throw an error with the `syclBufferConstructionFailed` error code, as that is the error that should be thrown when a buffer cannot be constructed.

```

vercors should error withCode "syclBufferConstructionFailed"
    ⇨ "example concepts/sycl/buffers/TooBigBuffer.cpp"
1 #include <sycl/sycl.hpp>
2
3 //@ requires \pointer(a, 10, write);
4 void test(bool* a) {
5     // a is only guaranteed to have 10 elements, so cannot make buffer with 11
6     sycl::buffer<bool, 1> aBuffer = sycl::buffer(a, sycl::range<1>(11));
7 }

```

Listing 9.5: Test program `TooBigBuffer`, for which VerCors should throw an error with the code `syclBufferConstructionFailed`.

### 9.2.3 Testing speed

The tests for all supported C++ and SYCL components were run using the ScalaTest [8] test runner in IntelliJ [42] on a laptop with Fedora 38 as its operating system on an Intel Core i7-1165G7 processor (4 cores, 2 threads per core, avg. 2.8GHz, max. 4.7 GHz) and 16 GB RAM (DDR4, max. 3200MT/s). The ScalaTest test runner reports for each run of a test set how long it took to execute that set of tests. When averaging 5 runs, running all tests concerning C++ constructs took 44 seconds in total, and running all tests concerning SYCL constructs took 9 minutes and 30 seconds in total.

The time it takes for each construct to be tested separately was not measured. It was chosen not to measure them as the results would not be accurate: some tests require other constructs to be present, such as the tests for data accessors, which require buffers, so the test times between constructs are not independent of each other.

## 9.3 Adhering to SYCL’s behaviour

An effort was made to model the behaviour of the encoding of a SYCL program as closely as possible to the behaviour of the SYCL program itself. The SYCL specification is extensive. This meant that the behaviour of most SYCL constructs was clearly enough defined, which made it clear how VerCors’ encoding should behave as well. However, for some constructs the specification was unclear about their behaviour. For example, in what cases a command group fails to be enqueued (see subsection 6.1.2.2), or how extensive the *exclusive access* of buffer to its `hostData` is (see subsection 7.1.2.1). In such cases, assumptions had to be made about a SYCL construct’s behaviour to be able to encode that construct. These assumptions are documented in this thesis, so if one finds that the behaviour of VerCors’ encoding differs from SYCL, it can be checked whether one of these assumptions is invalid or whether the encoding is unsound. There were also cases where the encoding behaves differently from SYCL on purpose, such as the ordering of kernel executions (see subsection 7.2.3.5). In these cases, the reason why it behaves differently and why it can still be considered a sound encoding was explained.

## 9.4 Full SYCL programs

The integration tests show that the C++ and SYCL constructs behave as described in this thesis. However, they do not show whether the supported features allow users to actually verify statements about SYCL programs performing a full task. To show that this is possible, two full SYCL programs are given in Appendix A.

The first program represent vector addition: adding the elements of two arrays and storing the results in a different array. This program was created because the SYCL teaching materials in [20, 21, 36] all contain their own version of 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 a SYCL program performing a task they already know in VerCors through this example.

In the second program a function is applied to each element of a matrix and then the results of that function application is transposed. This program uses all of the SYCL features for which support was added, but not all possible variations of each feature, as that would create an incomprehensibly large program. This program shows that programs containing multiple SYCL features working together to accomplish a task can be verified as well<sup>1</sup>.

The time it takes to verify these programs in the way that users will typically verify a program VerCors, i.e. by executing VerCors in a terminal, was recorded using the following command:

```
time ./bin/vct --no-infer-heap-context-into-frame filepath/to/program
```

Averaged over 5 executions, verifying the vector addition program took 2 minutes and 26 seconds, and verifying the matrix mapping and transposing program took 5 minutes and 19 seconds.

---

<sup>1</sup>Note should be taken that for about 1 in 5 verification attempts the pre-conditions of the second kernel cannot be established, even though they are sound. Normally non-deterministic results should be impossible for formal verifiers. However, as described in subsection 3.4.4.4, VerCors uses heuristics and randomisation to instantiate quantified expressions without triggers. This can, in some cases, cause programs that contain quantified expressions without triggers, such as the quantified contracts above the *kernel-runner* and the outer *kernel-parblock* in the encoding of this program, to fail every once in a while. This is most likely the cause of the occasional failure to establish pre-conditions of the second kernel.

## 10 Conclusion

In this thesis, the goal was to be able to deductively verify a subset of SYCL programs using the VerCors toolset. To accomplish this, four research questions were investigated, in which support was added to VerCors for various C++ and SYCL constructs.

For RQ1 support was added for the basic C++ constructs that are used to express SYCL constructs. First support was added to VerCors' initial phases by copying and altering an already existing C++ 14 lexer and parser and by copying and altering VerCors' conversion to COL and resolution phases for C. Support for most of C++ basic constructs was implemented in VerCors by copying the implementation for the equivalent C constructs. For the remaining constructs support was implemented from scratch.

For RQ2 it was investigated how reasoning about SYCL's basic and ND-range kernels could be supported in VerCors. First, the semantics of declaring and submitting kernels in SYCL were discussed, after which the implemented encoding into VerCors for those kernels was explained. This implementation made use of VerCors' built-in parblock and fork/join constructs to encode that SYCL kernels execute multiple work-items in parallel, asynchronously from the host. It was also shown how the specifications for a kernel given by the user are incorporated in the encoding. After this, the semantics and encoding of a SYCL statement that forces the host to wait on a kernel to finish executing, which was rather trivial, were discussed. Finally, the encoding of several SYCL instance methods, which can be used in kernel bodies to query a work-item's id and the kernel's index-space, was described.

For RQ3 the addition of support for reasoning about SYCL's buffers and data accessors to VerCors was explored. The semantics of buffer declarations were discussed first, after which the implemented encoding into VerCors for them was explained. This implementation copies the data a buffer constructor is given to a new COL array and then claims exclusive access to the data. The encoding of buffer destructions is also shown, which copies the buffer's contents back to its data and releases its exclusive access to it. After this, the semantics of data accessor declarations were explored, after which the implemented encoding into VerCors for them was discussed. This implementation creates for every accessor declared for a kernel a field in the class generated for that kernel. Then, permissions to access and/or alter these fields are automatically added in various places in the encoding. The encoding of SYCL's kernel ordering was also discussed. Finally, the encodings of subscribing data accessors and updates to the data accessor's underlying buffer were given.

For RQ4 it was investigated how reasoning about SYCL's address spaces could be supported in VerCors. It was revealed that support for the global and private address spaces was already implemented implicitly in the previous research questions. However, support for the local address space was not added yet. It was chosen to add support for local accessors, which allow uninitialized arrays to be declared in the local address space. It was shown that the encoding of local accessors was rather trivial, as a local accessor declaration could simply be encoded as the declaration a new COL array above the work-group parblock that is generated for each kernel.

Finally, the soundness of the added support for the SYCL constructs in the research questions was discussed. It was explained that integration testing was used, in combination with mirroring the behaviours described in SYCL specification in the encoding as closely as possible. The usefulness of the supported subset of SYCL constructs was also shown. Vector addition, which is used as an example in most SYCL teaching materials could be programmed using the supported constructs. A program performing a more complex task: applying a function on all elements of a matrix and then transposing the entire matrix, could be manufactured using the supported SYCL constructs as well.

All in all, it can be concluded that a subset of SYCL programs, where this subset consists of declarations and usage of kernels, buffers, data- and local accessors, can now be deductively verified in the VerCors toolset.

## 10.1 Future work

In this thesis, only the subset of SYCL constructs for which support was added to VerCors was shown. However, the SYCL language contains more features, some of which can be quite useful to the user. Therefore, the future work first describes what three SYCL features are the most useful and interesting to research adding support to VerCors for in the future. After that, the two C++ constructs which would be the most useful to add support to VerCors for to aid the implementation of SYCL support are discussed.

### 10.1.1 Extending SYCL support

There are multiple SYCL features that are useful and interesting to research adding VerCors support for. This subsection explains the three SYCL features that are considered the most useful and interesting to research in the future.

#### 10.1.1.1 Adding support for SYCL's group barriers and memory fences

SYCL's group barriers and memory fences are quite useful, as they can be used to control the reordering of memory loads and stores in a kernel and allow work-items to synchronise with each other (see sub-subsection 3.2.4.2). Because they influence how memory is reordered, it also has effects on permissions declared in specifications, which makes this interesting to research.

#### 10.1.1.2 Generic address space

How to add support to VerCors for SYCL's generic address space was not researched in this thesis. The generic space is quite different from the global, local, and private address spaces, as it is a virtual address space, which overlaps the other three address spaces and has pointers which can be converted to an address in one of the other three spaces. This makes it interesting to research how to support and reason about it in VerCors.

#### 10.1.1.3 Split-up kernel declarations

As discussed in RQ2 and RQ4, there are multiple ways to declare kernels in SYCL, but support was only added for one of them. Adding support for the split-up kernel declaration described in RQ4 could be useful, as its explicit separation into a work-group section and work-item section makes it clear to the user in what address space their variables are declared. Furthermore, it allows more flexibility in the type of the variables declared in the local address space than local accessors, which could be useful.

### 10.1.2 Extending C++ support

There are multiple C++ features that are useful to add VerCors support for to aid the implementation of SYCL support. This subsection explains the two C++ features that are considered the most useful to add support for in the future.

#### 10.1.2.1 Templates

Many of SYCL's instance methods are templated. For the constructors and methods in the subset of supported SYCL features, their template syntax could be avoided by not declaring the constructors and methods that require them in the SYCL header, were the other SYCL methods are declared, but to embed them in encoding in the transformation phase instead. However, it means that for every newly added template method, an embedding into the encoding has to be implemented as well. Therefore, it would be useful to investigate how C++ templates can be supported natively in VerCors instead.

#### 10.1.2.2 Classes

As mentioned in RQ1, SYCL's class instance methods were declared as methods in namespaces, and class constructor invocations are generated in the transformation phase because adding support for C++ classes was considered out of scope. This was adequate for adding support for the SYCL features in this thesis. However, this form of support might not be adequate enough when adding support for other SYCL constructs in the future. So, it would be useful to investigate how C++ classes can be supported natively in VerCors instead.

## Bibliography

- [1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt and Matthias Ulbrich, eds. *Deductive Software Verification – The KeY Book. From Theory to Practice*. 1st ed. Lecture Notes in Computer Science. Springer Cham, 19th Dec. 2016. ISBN: 978-3-319-49812-6. DOI: <https://doi.org/10.1007/978-3-319-49812-6>.
- [2] Anoud Alshnakat, Dilian Gurov, Christian Lidström and Philipp Rümmer. ‘Constraint-Based Contract Inference for Deductive Verification’. In: *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*. Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle and Matthias Ulbrich. Cham: Springer International Publishing, 2020, pp. 149–176. ISBN: 978-3-030-64354-6. DOI: [10.1007/978-3-030-64354-6\\_6](https://doi.org/10.1007/978-3-030-64354-6_6). URL: [https://doi.org/10.1007/978-3-030-64354-6\\_6](https://doi.org/10.1007/978-3-030-64354-6_6).
- [3] AMD. *HIP: Frequently asked questions*. 28th Nov. 2022. URL: [https://github.com/ROCm-Developer-Tools/HIP/blob/develop/docs/user\\_guide/faq.md](https://github.com/ROCm-Developer-Tools/HIP/blob/develop/docs/user_guide/faq.md).
- [4] AMD. *HIP Programming Guide*. Sept. 2019. URL: [https://sep5.readthedocs.io/en/latest/Programming\\_Guides/HIP-GUIDE.html](https://sep5.readthedocs.io/en/latest/Programming_Guides/HIP-GUIDE.html).
- [5] AMD. *HIP: What is this repository for?* 21st July 2022. URL: <https://github.com/ROCm-Developer-Tools/HIP>.
- [6] AMD. *vectoradd\_hip.cpp*. July 2022. URL: [https://github.com/ROCm-Developer-Tools/HIP-Examples/blob/master/vectorAdd/vectoradd\\_hip.cpp](https://github.com/ROCm-Developer-Tools/HIP-Examples/blob/master/vectorAdd/vectoradd_hip.cpp).
- [7] Afshin Amighi, Saeed Darabi, Stefan Blom and Marieke Huisman. ‘Specification and Verification of Atomic Operations in GPGPU Programs’. In: *Software Engineering and Formal Methods*. Ed. by Radu Calinescu and Bernhard Rumpe. Cham: Springer International Publishing, 2015, pp. 69–83. ISBN: 978-3-319-22969-0.
- [8] Artima. *ScalaTest support in the IntelliJ Scala plugin*. Accessed on 30-12-2023. URL: [https://www.scalatest.org/user\\_guide/using\\_scalatest\\_with\\_intellij](https://www.scalatest.org/user_guide/using_scalatest_with_intellij).
- [9] Vytautas Astrauskas, Aurel Bily, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli and Alexander J. Summers. ‘The Prusti Project: Formal Verification for Rust (invited)’. In: *NASA Formal Methods (14th International Symposium)*. Springer, 2022, pp. 88–108. URL: [https://link.springer.com/chapter/10.1007/978-3-031-06773-0\\_5](https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5).
- [10] Bernhard Beckert and Reiner Hähnle. ‘Reasoning and Verification: State of the Art and Current Trends’. *IEEE Intelligent Systems* 29.1 (2014), pp. 20–29. DOI: [10.1109/MIS.2014.3](https://doi.org/10.1109/MIS.2014.3).
- [11] Simon Bliudze, Petra van den Bos, Marieke Huisman, Robert Rubbens and Larisa Safina. ‘JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java’. In: *26th International Conference on Fundamental Approaches to Software Engineering*. Cham, Apr. 2023. URL: <https://hal.inria.fr/hal-03911393>.
- [12] Stefan Blom, Saeed Darabi, Marieke Huisman and Wytse Oortwijn. ‘The VerCors Tool Set: Verification of Parallel and Concurrent Software’. In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 102–110. DOI: [10.1007/978-3-319-66845-1\\_7](https://doi.org/10.1007/978-3-319-66845-1_7).
- [13] Stefan Blom, Saeed Darabi, Marieke Huisman and Mohsen Safari. ‘Correct program parallelisations’. *International Journal on Software Tools for Technology Transfer* 23.5 (Oct. 2021), pp. 741–763. ISSN: 1433-2787. DOI: [10.1007/s10009-020-00601-z](https://doi.org/10.1007/s10009-020-00601-z). URL: <https://doi.org/10.1007/s10009-020-00601-z>.



- [14] Stefan Blom, Marieke Huisman and Matej Mihelčič. ‘Specification and verification of GPGPU programs’. *Science of Computer Programming* 95 (2014). Special Section: ACM SAC-SVT 2013 + Bytecode 2013, pp. 376–388. ISSN: 0167-6423. DOI: <https://doi.org/10.1016/j.scico.2014.03.013>.
- [15] Richard Bornat, Cristiano Calcagno, Peter W. O’Hearn and Matthew Parkinson. ‘Permission accounting in separation logic’. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2005, pp. 259–270. URL: <https://dl.acm.org/doi/pdf/10.1145/1040305.1040327>.
- [16] James Bornholt. *Memory Consistency Models: A Tutorial*. 17th Feb. 2016. URL: [https://en.cppreference.com/w/cpp/atomic/memory\\_order](https://en.cppreference.com/w/cpp/atomic/memory_order).
- [17] Bernhard F. Brodowsky. ‘Translating Scala to SIL’. MA thesis. Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2013.
- [18] Stephen Brookes and Peter W. O’Hearn. ‘Concurrent separation logic’. *ACM SIGLOG News* 3.3 (2016), pp. 47–65. URL: <https://read.seas.harvard.edu/~kohler/class/cs260r-17/brookes16concurrent.pdf>.
- [19] Codeplay Software Ltd. *ComputeCPP Community Edition*. Accessed on 14-12-2023. URL: <https://developer.codeplay.com/products/computecpp/ce/home>.
- [20] Codeplay Software Ltd. *Introduction to SYCL on Tech.io*. Accessed on 14-12-2023. URL: <https://tech.io/playgrounds/48226/introduction-to-sycl/hello-world>.
- [21] Codeplay Software Ltd. *SYCL Academy on GitHub*. Accessed on 14-12-2023. URL: <https://github.com/codeplaysoftware/syclacademy>.
- [22] Codeplay Software Ltd. *SYCL Guide*. Accessed on 14-12-2023. URL: <https://developer.codeplay.com/products/computecpp/ce/2.11.0/guides/sycl-guide>.
- [23] Codeplay Software Ltd. *The Future of ComputeCpp*. Accessed on 07-07-2023. URL: <https://codeplay.com/portal/news/2023/07/07/the-future-of-computecpp>.
- [24] David R. Cok. ‘OpenJML: JML for Java 7 by Extending OpenJDK’. In: *NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann and Rajeev Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 472–479. ISBN: 978-3-642-20398-5.
- [25] cppreference.com. *Order of evaluation*. 12th Feb. 2023. URL: [https://en.cppreference.com/w/cpp/language/eval\\_order](https://en.cppreference.com/w/cpp/language/eval_order).
- [26] cppreference.com. *std::memory\_order*. 24th Jan. 2023. URL: [https://en.cppreference.com/w/cpp/atomic/memory\\_order](https://en.cppreference.com/w/cpp/atomic/memory_order).
- [27] Saeed Darabi. ‘Verification of program parallelization’. English. IPA Dissertation Series No. 2018-02 IDS PhD. Thesis Series No. 18-458. PhD thesis. University of Twente, Mar. 2018. ISBN: 978-90-365-4484-9. DOI: 10.3990/1.9789036544849.
- [28] Leonardo De Moura and Nikolaj Bjørner. ‘Z3: An efficient SMT solver’. In: *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer. 2008, pp. 337–340.
- [29] Marco Eilers and Peter Müller. ‘Nagini: A Static Verifier for Python’. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 596–603. ISBN: 978-3-319-96145-3.
- [30] ENCCS. *OpenMP for GPU offloading*. Aug. 2022. URL: <https://enccs.github.io/openmp-gpu>.
- [31] Reiner Hähnle. ‘Introduction Deductive Software Verification’ (n.d.). URL: [https://citesee.rx.ist.psu.edu/doc\\_view/pid/709ebdd322866dc312e3b7056db8fe9c523d9d2c](https://citesee.rx.ist.psu.edu/doc_view/pid/709ebdd322866dc312e3b7056db8fe9c523d9d2c).
- [32] Vincent Hindriksen. *Difference between CUDA and OpenCL 2010*. Apr. 2010. URL: <https://streamhpc.com/blog/2010-04-22/difference-between-cuda-and-opencl/>.

- [33] Tony Hoare. ‘An Axiomatic Basis for Computer Programming’. *Commun. ACM* 12.10 (Oct. 1969), 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.
- [34] Petr Hudeček. ‘Soothsharp: A C#-to-Viper translator’ (2017). URL: [https://dspace.cuni.cz/bitstream/handle/20.500.11956/90340/DPTX\\_2016\\_1\\_11320\\_0\\_464227\\_0\\_183413.pdf](https://dspace.cuni.cz/bitstream/handle/20.500.11956/90340/DPTX_2016_1_11320_0_464227_0_183413.pdf).
- [35] Marieke Huisman, Wolfgang Ahrendt, Daniel Grahl and Martin Hentschel. ‘Formal Specification with the Java Modeling Language’. In: *Deductive Software Verification – The KeY Book: From Theory to Practice*. Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt and Matthias Ulbrich. Cham: Springer International Publishing, 2016, pp. 193–241. ISBN: 978-3-319-49812-6. DOI: 10.1007/978-3-319-49812-6\_7.
- [36] Intel. *Explore SYCL with Samples from Intel*. 2023. URL: <https://www.intel.com/content/www/us/en/docs/oneapi/code-samples-dpcpp/2023-1/overview.html>.
- [37] Intel. *Five Outstanding Additions Found in SYCL 2020*. 22nd Apr. 2022. URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/five-outstanding-additions-sycl2020.html>.
- [38] Intel. *How 12th Gen’s Intel® Core™ Hybrid Technology Works*. Sept. 2022. URL: <https://www.intel.com/content/www/us/en/support/articles/000091896/processors.html?wapkw=how%2012th%20gen%20intel%20core%20processors%20work>.
- [39] Intel. *Intel Unveils 12th Gen Intel Core, Launches World’s Best Gaming Processor, i9-12900K*. 27th Oct. 2021. URL: <https://www.intel.com/content/www/us/en/newsroom/news/12th-gen-core-processors.html>.
- [40] Intel®. *Intel® oneAPI DPC++/C++ Compiler*. Accessed on 07-11-2023. URL: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html>.
- [41] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx and Frank Piesens. ‘VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java.’ *NASA Formal Methods* 6617 (2011), pp. 41–55.
- [42] JetBrains s.r.o. *IntelliJ IDEA – the Leading Java and Kotlin IDE*. Accessed on 30-12-2023. URL: <https://www.jetbrains.com/idea/>.
- [43] Khronos Community. *Definition of ‘exclusive access’ of a buffer to its hostData*. 24th Oct. 2023. URL: <https://community.khronos.org/t/definition-of-exclusive-access-of-a-buffer-to-its-hostdata/110182>.
- [44] Khronos® OpenCL Working Group. *The OpenCL™ Specification*. 6th Feb. 2023. URL: [https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL\\_API.html](https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html).
- [45] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman and Werner Dietl. *JML Reference Manual*. DRAFT, Revision 2344. 31st May 2013. URL: <http://www.jmlspecs.org/refman/jmlrefman.pdf>.
- [46] Rustan Leino. ‘Accessible Software Verification with Dafny’. *IEEE Software* 34.6 (2017), pp. 94–97. DOI: 10.1109/MS.2017.4121212.
- [47] Sparsh Mittal and Jeffrey S. Vetter. ‘A survey of CPU-GPU heterogeneous computing techniques’. *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–35.
- [48] Peter Müller, Malte Schwerhoff and Alexander J. Summers. ‘Viper: A Verification Infrastructure for Permission-Based Reasoning’. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and Klas R. M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 41–62. ISBN: 978-3-662-49122-5.
- [49] NVIDIA. *CUDA C++ Programming Guide*. Apr. 2023. URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.

- [50] NVIDIA. *CUDA Toolkit Documentation 12.1*. 30th Jan. 2023. URL: <https://docs.nvidia.com/cuda/index.html>.
- [51] NVIDIA. *Your GPU Compute Capability*. Accessed on 06-03-2023. URL: <https://developer.nvidia.com/cuda-gpus>.
- [52] Peter O’Hearn. ‘Separation Logic’. *Commun. ACM* 62.2 (Jan. 2019), 86–95. ISSN: 0001-0782. DOI: 10.1145/3211968. URL: <https://doi.org/10.1145/3211968>.
- [53] OpenMP ARB. *OpenMP Compilers & Tools*. Nov. 2022. URL: <https://www.openmp.org/resources/openmp-compilers-tools/#compilers>.
- [54] OpenMP ARB. *OpenMP FAQ*. June 2018. URL: <https://www.openmp.org/about/openmp-faq>.
- [55] Oracle. *Class ArithmeticException*. Accessed on 27-03-2023. URL: <https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/ArithmeticException.html>.
- [56] Steve Oualline. *Practical C++ programming*. " O’Reilly Media, Inc.", 1995, pp. 3–4. ISBN: 9781565921399.
- [57] Terence Parr. *The definitive ANTLR 4 reference*. The Pragmatic Bookshelf, 2013. ISBN: 978-1934356999.
- [58] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook and Xinmin Tian. *Data Parallel C++*. Apress, 2021. DOI: 10.1007/978-1-4842-5574-2.
- [59] John C. Reynolds. ‘An introduction to separation logic (preliminary draft)’. *Course notes, October* (2008). URL: <http://flint.cs.yale.edu/cs428/doc/seplogic08.pdf>.
- [60] John C. Reynolds. ‘Separation logic: A logic for shared mutable data structures’. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 55–74. URL: <https://www.cs.cmu.edu/~jcr/seplogic.pdf>.
- [61] Robert Rubbens. ‘Improving Support for Java Exceptions and Inheritance in VerCors’. MA thesis. May 2020. URL: <http://essay.utwente.nl/81338/>.
- [62] Robert Rubbens, Sophie Lathouwers and Marieke Huisman. ‘Modular Transformation of Java Exceptions Modulo Errors’. In: *Formal Methods for Industrial Critical Systems*. Ed. by Alberto Lluch Lafuente and Anastasia Mavridou. Cham: Springer International Publishing, 2021, pp. 67–84. ISBN: 978-3-030-85248-1.
- [63] Mohsen Safari and Marieke Huisman. ‘Formal verification of parallel prefix sum and stream compaction algorithms in CUDA’. *Theoretical Computer Science* 912 (2022), pp. 81–98. ISSN: 0304-3975. DOI: <https://doi.org/10.1016/j.tcs.2022.02.027>.
- [64] Ömer F. O. Şakar. ‘Extending support for axiomatic data types in VerCors’. MA thesis. Apr. 2020. URL: <http://essay.utwente.nl/80892/>.
- [65] Wolfram Schulte. ‘VCC: Contract-based Modular Verification of Concurrent C’. In: *31st International Conference on Software Engineering, ICSE 2009*. IEEE Computer Society, Jan. 2008. URL: <https://www.microsoft.com/en-us/research/publication/vcc-contract-based-modular-verification-of-concurrent-c/>.
- [66] Howard J. Siegel, John K. Antonio, Richard C. Metzger, Min Tan and Yan A. Li. ‘Heterogeneous computing’. *ECE Technical Reports* (1994), p. 206.
- [67] Jan Smans, Bart Jacobs and Frank Piessens. ‘Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic’. In: *ECOOP 2009 – Object-Oriented Programming*. Ed. by Sophia Drossopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 148–172. ISBN: 978-3-642-03013-0.
- [68] Stack Overflow. *How to compile OpenCL project with kernels*. Oct. 2014. URL: <https://stackoverflow.com/questions/26517114/how-to-compile-opencl-project-with-kernels>.

- [69] The Khronos® Group. *Conformant Products*. 2023. URL: <https://www.khronos.org/conformance/adopters/conformant-products/opengl>.
- [70] The Khronos® Group. *OpenCL Guide*. 2020. URL: <https://github.com/KhronosGroup/OpenCL-Guide>.
- [71] The Khronos® SYCL™ Working Group. *SYCL*. Accessed on 15-02-2023. URL: [https://www.khronos.org/api/index\\_2017/sycl](https://www.khronos.org/api/index_2017/sycl).
- [72] The Khronos® SYCL™ Working Group. *SYCL™ 2020 Specification (revision 7)*. 18th Apr. 2023. URL: <https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf>.
- [73] Top500. *Top500 Supercomputers*. Nov. 2022. URL: <https://www.top500.org/>.
- [74] UL HPC Platform. *Introduction to OpenCL Programming (C/C++)*. Nov. 2021. URL: <https://ulhpc-tutorials.readthedocs.io/en/latest/gpu/opengl>.
- [75] UTwente FMT Group. *VerCors' source code on GitHub*. Accessed on 23-11-2023. URL: <https://github.com/utwente-fmt/vercors>.
- [76] Various, ISO/IEC JTC 1/SC 22. *ANTLR lexer and parser for C++ 14 on GitHub*. Accessed on 06-07-2023. URL: <https://github.com/antlr/grammars-v4/tree/master/cpp>.
- [77] VerCors. *VerCors wiki*. 20th Mar. 2023. URL: <https://github.com/utwente-fmt/vercors/wiki>.
- [78] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira and Peter Müller. 'Gobra: Modular Specification and Verification of Go Programs'. In: *Computer Aided Verification (CAV)*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. LNCS. Springer International Publishing, 2021, pp. 367–379. URL: [https://link.springer.com/chapter/10.1007/978-3-030-81685-8\\_17](https://link.springer.com/chapter/10.1007/978-3-030-81685-8_17).
- [79] Mohamed Zahran. 'Heterogeneous Computing: Here to Stay: Hardware and Software Perspectives'. *Queue* 14.6 (Dec. 2016), 31–42. ISSN: 1542-7730. DOI: 10.1145/3028687.3038873.

## Glossary

- accelerator** An accelerator is a computing unit in a heterogeneous system which is not the host.
- address space** An address space is a wrapper around a region of memory in a system and determines how this region of memory is accessed. Different address spaces can have different methods of accessing the data in their memory region.
- backend** In SYCL a backend is an implementation of SYCL's programming model, which exposes one or multiple platforms. An example is the OpenCL backend, which can expose multiple OpenCL platforms.
- basic kernel** A basic kernel is a kernel with a one-, two, or three-dimensional index space, where for every point in the index-space a work-item is executed.
- buffer** A SYCL buffer defines a shared array of one, two, or three dimensions. They allow the same object to be shared between devices with different contexts, platforms or backends.
- computing unit** A computing unit is a physical component in a system that can perform calculations. Examples of computing units are (cores of) CPUs, GPUs, and FGPAs.
- data accessor** A SYCL data accessor gives a kernel on a device access to a memory object on the host, such as a buffer.
- device** A device is a computing unit in a heterogeneous system which is not the host.
- formal verification** Formal verification is a verification method that verifies, using mathematical techniques, whether every possible program behaviour adheres to a set of given specifications.
- generic address space** The generic address space is a virtual address space which overlaps the global, local, and private address spaces.
- global address space** Memory in the global address space is accessible to all work-items in all work-groups. This address space is persistent across kernel executions.
- group barrier** A group barrier can be placed in a kernel to synchronise work-items in the same work- or sub-group.
- heap** The set of data stored in a system's heap memory, such as class- and global variables. Data in this set can be accessed by multiple processes.
- host** The host is the computing unit on which the code that manages and assigns tasks to other computing units is run. Usually the main CPU in the system is the host.
- kernel** A kernel contains code, and executes that code in parallel on a computing unit. Different kernels can be executed concurrently on multiple computing units.
- local address space** Memory in the local address space is accessible to all work-items in the same work-group.

**memory fence** A memory fence can be placed in a kernel to control the reordering of memory loads and stores. The memory fence's behaviour is dependent on its memory order. What work-items are affected by the fence is determined by the fence's memory scope.

**ND-range kernel** An ND-range kernel has a one-, two-, or three-dimensional index-space, where for every point in the index-space a work-item is executed. These work-items are organised into work-groups, where work-items in the same work-group execute concurrently on the processing elements of a single compute unit.

**private address space** Memory in the private address space is accessible to a single work-item.

**static analysis** Analysis of a program at compilation time, by, for instance, analyzing its source code.

**unified shared memory** Unified shared memory (USM) uses a single unified address space across the host and the devices. This means that pointers to USM allocations are consistent across the host and devices and can be directly passed to kernels as arguments.

**work-group** Work-items can be organized into work-groups, which have a one-, two-, or three-dimensional index space. Each work-group gets a unique work-group id, and each work-item is assigned a local id, which is unique inside its work-group. All work-items in the same work-group execute concurrently on the processing elements of a single compute unit.

**work-item** A single instance of a kernel function, which is executed on a device. Each work-item resides with other work-items in an index space, where each work-item is assigned a unique global id.

OpenCL	CUDA
Computing unit	Multiprocessor
Device	GPU
Global memory	Global memory
Local memory	Shared memory
Private memory	Local memory
Program / Kernel	Kernel
Work-group	Block
Work-item	Thread

Table 10.1: CUDA equivalences of OpenCL terms. *Based on the table in [32].*

## A Examples of VerCors-verifiable SYCL programs

In this appendix, two full SYCL programs that are verifiable by VerCors are shown and discussed.

### A.1 Vector addition

The first program can be seen in Listing A.1. This program can also be found in the VerCors' source code [75] in the file `examples/concepts/sycl/fullExamples/VectorAdd.cpp`. This program can be seen as a generalization of the program described in subsection 3.1.3.2 (page 10). It has a method `vector_add` which takes two arrays, `a` and `b`, of size `size` and puts the results of adding their elements in array `c`, which also has size `size`.

```
1 #include <sycl/sycl.hpp>
2
3 /*@
4 context size >= 0;
5 context \pointer(a, size, write);
6 context \pointer(b, size, write);
7 context \pointer(c, size, write);
8 ensures (\forallall int i; 0 <= i && i < size; c[i] == a[i] + b[i]);
9 @*/
10 void vector_add(sycl::queue q, int size, int* a, int* b, int* c) {
11     sycl::buffer<int, 1> a_buf = sycl::buffer(a, sycl::range<1>(size));
12     sycl::buffer<int, 1> b_buf = sycl::buffer(b, sycl::range<1>(size));
13     sycl::buffer<int, 1> c_buf = sycl::buffer(c, sycl::range<1>(size));
14
15     q.submit([&](sycl::handler& h) {
16         sycl::accessor<int, 1, sycl::access_mode::read> a_acc =
17             ↪ sycl::accessor(a_buf, h, sycl::read_only);
18         sycl::accessor<int, 1, sycl::access_mode::read> b_acc =
19             ↪ sycl::accessor(b_buf, h, sycl::read_only);
20         sycl::accessor<int, 1> c_acc = sycl::accessor(c_buf, h, sycl::read_write);
21
22         h.parallel_for(sycl::range<1>(size),
23             /*@
24             context it.get_id(0) < a_acc.get_range().get(0);
25             context Perm(a_acc[it.get_id(0)], read);
26             context it.get_id(0) < b_acc.get_range().get(0);
27             context Perm(b_acc[it.get_id(0)], read);
28             context it.get_id(0) < c_acc.get_range().get(0);
29             context Perm(c_acc[it.get_id(0)], write);
30             ensures c_acc[it.get_id(0)] == a_acc[it.get_id(0)] + b_acc[it.get_id(0)];
31             @*/
32             [=](sycl::item<1> it) {
33                 c_acc[it.get_id(0)] = a_acc[it.get_id(0)] + b_acc[it.get_id(0)];
34             }
35         );
36     });
37 }
```

Listing A.1: A SYCL program that performs vector addition of two arrays.

## A.2 Matrix mapping and transposition

The second program can be seen in Listing A.2. This program can also be found in the VerCors source code [75] in the file `examples/concepts/sycl/fullExamples/MatrixTransposeWithF.cpp`. This program uses all of the SYCL features for which support was added, but not all possible variations of each feature, as that would create an incomprehensibly large program. This program has a method `transpose_with_f` which takes a matrix `matrix` (stored as a 1-dimensional array), with `rows` rows and `cols` columns, applies unimplemented function `f` to every element and then transposes the resulting matrix, and stores the results of this entire process (as a 1-dimensional array) in `result`.

```
1 #include <sycl/sycl.hpp>
2
3 // Helper function which calculates a linear index out of a row and a column
4 /*@
5   requires row >= 0 && row < rows && col >= 0 && col < cols;
6   ensures \result == col + (row * cols);
7   ensures \result >= 0 && \result < rows * cols;
8   pure int calc_index(int row, int col, int rows, int cols);
9 */
10
11 // The function to apply to every element of matrix
12 /*@ pure */ float f(float value);
13
14 /*@
15   context rows >= 0 && cols > 0;
16   context (rows * cols) % cols == 0;
17   context \pointer(matrix, rows * cols, write);
18   context \pointer(result, cols * rows, write);
19   ensures (\forall int r, int c;
20     r >= 0 && r < rows && c >= 0 && c < cols;
21     result[calc_index(c, r, cols, rows)] ==
22     f(matrix[calc_index(r, c, rows, cols)])
23   );
24 */
25 void transpose_with_f(float matrix[], int rows, int cols, float* result) {
26   sycl::queue myQueue;
27
28   sycl::buffer<float, 2> matrix_buffer =
29     ↪ sycl::buffer(matrix, sycl::range<2>(rows, cols));
30   // Create a buffer to share results of the first kernel with the second one
31   float temp[rows * cols];
32   sycl::buffer<float, 1> temp_buffer =
33     ↪ sycl::buffer(temp, sycl::range<1>(rows * cols));
34
35   // Apply f to all elements of matrix and store results in temp_buffer
36   myQueue.submit(
37     [&](sycl::handler& cgh) {
38       sycl::accessor<float, 2, sycl::access_mode::read> matrix_acc =
39         ↪ sycl::accessor(matrix_buffer, cgh, sycl::read_only);
40       sycl::accessor<float, 1> temp_acc =
41         ↪ sycl::accessor(temp_buffer, cgh, sycl::read_write);
42
43       cgh.parallel_for(sycl::range<2>(rows, cols),
44         /*@
45           context it.get_id(0) < matrix_acc.get_range().get(0);
46           context it.get_id(1) < matrix_acc.get_range().get(1);
47           context Perm(matrix_acc[it.get_id(0)][it.get_id(1)], read);
48           context it.get_linear_id() < temp_acc.get_range().get(0);
49           context Perm(temp_acc[it.get_linear_id()], write);
```



```

50     ensures temp_acc[it.get_linear_id()] ==
51                                     ⇨ f(matrix_acc[it.get_id(0)][it.get_id(1)]);
52 */
53 [=] (sycl::item<2> it) {
54     temp_acc[it.get_linear_id()] = f(matrix_acc[it.get_id(0)][it.get_id(1)]);
55 }
56 );
57 }
58 );
59
60 bool done = false;
61 {
62     sycl::buffer<float, 2> result_buffer =
63                                     ⇨ sycl::buffer(result, sycl::range<2>(cols, rows));
64
65     // Transpose the results of the first kernel and store it in result_buffer
66     // Implicitly waits on previous kernel as that kernel writes to temp_buffer
67     // and this kernel needs read access to temp_buffer
68     sycl::event transpose_event = myQueue.submit(
69         [&](sycl::handler& cgh) {
70             sycl::accessor<float, 1, sycl::access_mode::read> temp_acc =
71                                     ⇨ sycl::accessor(temp_buffer, cgh, sycl::read_only);
72             sycl::accessor<float, 2> result_acc =
73                                     ⇨ sycl::accessor(result_buffer, cgh, sycl::read_write);
74
75             // Cannot use cols variable here, so use result_acc.get_range().get(0)
76             sycl::local_accessor<float, 1> temp_local_acc = sycl::local_accessor<float,
77                                     ⇨ 1>(sycl::range<1>(result_acc.get_range().get(0)), cgh);
78
79             cgh.parallel_for(sycl::nd_range(sycl::range(rows*cols), sycl::range(cols)),
80                 /*@
81                 context it.get_global_id(0) < temp_acc.get_range().get(0);
82                 context Perm(temp_acc[it.get_global_id(0)], read);
83                 context it.get_local_id(0) < result_acc.get_range().get(0);
84                 context it.get_group(0) < result_acc.get_range().get(1);
85                 context Perm(result_acc[it.get_local_id(0)][it.get_group(0)], write);
86                 context it.get_local_linear_id() < temp_local_acc.get_range().get(0);
87                 context Perm(temp_local_acc[it.get_local_linear_id()], write);
88                 ensures result_acc[it.get_local_id(0)][it.get_group(0)] ==
89                                     ⇨ temp_acc[it.get_global_id(0)];
90                 */
91                 [=] (sycl::nd_item<1> it) {
92                     int ll_id = it.get_local_linear_id();
93                     temp_local_acc[ll_id] = temp_acc[it.get_global_id(0)];
94                     result_acc[it.get_local_id(0)][it.get_group(0)] = temp_local_acc[ll_id];
95                 }
96             );
97         });
98     // Explicitly wait till the whole matrix has been transposed
99     transpose_event.wait();
100     done = true;
101 }
102 // result_buffer has been destroyed so result is accessible again
103 //@ assert rows > 0 ==> result[calc_index(0, 0, cols, rows)] ==
104                                     ⇨ f(matrix[calc_index(0, 0, rows, cols)]);
105 }

```

Listing A.2: A SYCL program that applies function  $f$  to each element of a matrix and then transposes the result.

## B Overview of all assumptions and limitations

In this appendix, an overview is given for each C++ and SYCL construct of what assumptions were made in the encoding of the construct into VerCors and the limitations of the encoding.

### Basic C++ constructs

- **Limitation:** When a (lambda) method parameter contains a single addressing operator `&`, the parameter is processed as if `&` was not there. This workaround might not be entirely accurate behaviour-wise, as using `&` in a method parameter could change it from pass-by-value to pass-by-reference.
- **Limitation:** Two methods with an identical signature are allowed to be created. When a user invokes one of those methods, the one that was declared first is called. This is not allowed in C++.
- **Limitation:** Lambda captures, such as `[=]` and `[&]`, which define what variables declared outside the lambda expression are accessible in what way inside the lambda body, are ignored.

### Basic and ND-range kernels

- **Assumption:** In SYCL, enqueueing a command group, or executing it, can fail. In VerCors, enqueueing a command group to a queue or executing it cannot fail if the user has declared a valid command group. Here 'valid' means that the command group declaration passes the checks mentioned throughout this thesis, such as not having negative range dimensions, and the global range dimensions being divisible by the local range dimensions for `nd_range` objects.
- **Limitation:** In SYCL, a command group can statically contain more than one kernel declaration, as long as at runtime only one of them is executed. In VerCors a command group can statically only contain one kernel declaration.
- **Limitation:** In SYCL, submission of command groups without a kernel declaration inside them is allowed. This is not allowed in VerCors.
- **Limitation:** In SYCL, it is allowed for the command group scope to contain code other than invocations to the `parallel_for` method and declarations of data- and local accessors. This is not supported in VerCors.
- **Limitation:** In SYCL, local variables that have been declared outside the command group scope of the submit method invocation are allowed to be used inside the command group scope and kernel scope. In VerCors local variables are only allowed to be used in the range parameters of `parallel_for` methods.

### Buffers

- **Assumption:** In a buffer constructor, the buffer is initialized with the memory specified by its `hostData` parameter, and the buffer assumes exclusive access to this memory for the duration of its lifetime. It is assumed that "this memory" means the contents of `hostData` that are within the bounds of the constructor's range parameter.
- **Limitation:** Variables holding a buffer are not allowed to be reassigned in VerCors.

### Data accessors

- **Limitation:** The access modes *write-only*, *no-init-write-only*, and *no-init-read-write* are not supported in VerCors.
- **Limitation:** If a kernel contains a data accessor `a` with *read-only* access to a buffer and one or more data accessors to the same buffer with read-write access, users will be allowed to write to `a` in the kernel body. This is not allowed in SYCL.
- **Limitation:** To be able to verify that a subscript on a data accessor is within the data accessor's range, the user often needs to state in the kernel's contract that it is within the bounds of the dimension it indexes.

### Local accessors

- **Limitation:** To be able to verify that a subscript on a local accessor is within the local accessor's range, the user often needs to state in the kernel's contract that it is within the bounds of the dimension it indexes.

## C Overview of errors that can be thrown

In this appendix, an overview is given of what SYCL-specific situations cause VerCors to throw a (custom) error when trying to verify a SYCL program because the user made a mistake in their input or because the program fails to verify.

### Basic and ND-range kernels

- More than one `parallel_for` invocation is found in the command group scope.
- No `parallel_for` invocation is found in the command group scope.
- Non-SYCL code is found in the command group scope.
- SYCL code which is not a `parallel_for` invocation or a data- or local accessor declaration is found in the command group scope.
- Using a local variable from host code inside the command group scope or kernel scope.
- The type of the first argument of a `parallel_for` invocation is a `range`, and the parameter of the kernel lambda method in the second argument is not of type `item`.
- The type of the first argument of a `parallel_for` invocation is an `nd_range`, and the parameter of the kernel lambda method in the second argument is not of type `nd_item`.
- The range dimension checks described in subsection 6.1.3.4 fail to verify.

### Item methods

- The dimension parameter of an `(nd_)item` method is not within the bounds of the number of dimensions in the kernel's index-space.

### Buffers

- The total size of the `bufferRange` parameter of a buffer constructor is negative or larger than the size of the constructor's `hostData` parameter.
- There is no write permission for the requested range on the `hostData` parameter of a buffer constructor.
- Using the `exclusive_hostdata_access` predicate.
- Reassigning a variable holding a buffer object.

### Data accessors

- Subscripting a data accessor with a subscript outside its range.
- Invoking `get_range().get(i)` on an `accessor` object when `i` greater or equal to the number of dimensions in the data accessors' range or negative.

### Local accessors

- Using a local accessor variable inside a basic kernel.
- Subscripting a local accessor with a subscript outside its range.
- Invoking `get_range().get(i)` on a `local_accessor` object when `i` greater or equal to the number of dimensions in the local accessors' range or negative.

## D Proofs of linearization methods

In this appendix, proofs are given for the assumptions made in the functions `linearize2` and `linearize3` in subsubsection 6.3.3.1.

### D.1 Linearization of two ids

The linearization of two ids is defined as follows in the SYCL specification [72] (section 3.11.1):

$$id1 + (id0 * r1) \tag{D.1}$$

The `linearize2` function in the SYCL header uses this formula, as described in subsubsection 6.3.3.1. It ensures three assumptions about the results, which are proven in the next subsections.

The `linearize2` method requires the bounds to the variables listed below, so it is never called with values outside these bounds. Therefore, these bounds on the variables will also be assumed in the proofs.

$$id0 \in [0, r0) \quad \wedge \quad id1 \in [0, r1) \quad \wedge \quad r0 > 0 \quad \wedge \quad r1 > 0$$

#### D.1.1 Assumption 1: Minimum value

**Assumption 1**  $id1 + (id0 * r1) \geq 0$

Because all variables are  $\geq 0$  and only  $+$  and  $*$  are used, the result of Equation D.1 decreases when  $id0$  or  $id1$  decreases. This means that we can find the minimum value by using the smallest possible values for  $id0$  and  $id1$ :

$$id0 = 0 \quad \wedge \quad id1 = 0$$

Filling in those values in Equation D.1 gives us a minimum value of:

$$0 + (0 * r1) \Leftrightarrow 0 + 0 \Leftrightarrow 0$$

#### D.1.2 Assumption 2: Maximum value

**Assumption 2**  $id1 + (id0 * r1) < r0 * r1$

Because all variables are  $\geq 0$  and only  $+$  and  $*$  are used, the result of Equation D.1 increases when  $id0$  or  $id1$  increases. This means that we can find the maximum value by using the biggest possible values for  $id0$  and  $id1$ :

$$id0 = (r0 - 1) \quad \wedge \quad id1 = (r1 - 1)$$

Filling in those values in Equation D.1 gives us a maximum value of:

$$\begin{aligned} (r1 - 1) + ((r0 - 1) * r1) &\Leftrightarrow \\ r1 - 1 + r0 * r1 - r1 &\Leftrightarrow \\ r0 * r1 - 1 \end{aligned}$$

which is less than  $r0 * r1$ .

### D.1.3 Assumption 3: Injectivity

Equation D.1 is injective with regards to  $id0$  and  $id1$ :

**Assumption 3**  $id1 + (id0 * r1) \neq id1' + (id0' * r1)$  where  $id0 \neq id0' \vee id1 \neq id1'$

If Equation D.1 would **not** be injective with regards to  $id0$  and  $id1$ , then the following must be true for at least one set of values for  $id0$ ,  $id0'$ ,  $id1$ , and  $id1'$ :

$$id1 + (id0 * r1) = id1' + (id0' * r1) \quad \text{where } id0 \neq id0' \vee id1 \neq id1' \quad (D.2)$$

Suppose  $p$  denotes the difference between  $id0$  and  $id0'$ , such that  $id0 + p = id0'$ . Then Equation D.2 can be rewritten to:

$$\begin{aligned} id1 + (id0 * r1) &= id1' + ((id0 + p) * r1) \Leftrightarrow \\ id1 + id0 * r1 &= id1' + id0 * r1 + p * r1 \Leftrightarrow \\ id1 &= id1' + id0 * r1 + p * r1 - id0 * r1 \Leftrightarrow \\ id1 &= id1' + p * r1 \end{aligned} \quad (D.3)$$

Now we can distinguish three cases:

1.  $p = 0$ : In this case  $id0 = id0'$  so  $id1 \neq id1'$  must be true for Equation D.2 to be valid. Filling in  $p = 0$  in Equation D.3 gives:

$$\begin{aligned} id1 &= id1' + 0 * r1 \Leftrightarrow \\ &= id1' \end{aligned}$$

This is invalid as we had just determined  $id1 \neq id1'$ . So Equation D.2 is invalid for  $p = 0$ .

2.  $p > 0$ : The left-hand-side of Equation D.3 is greater or equal to the smallest value the right-hand-side can attain. In this case the right-hand-side is the smallest when  $p = 1$ . Using this value in Equation D.3 gives:

$$\begin{aligned} id1 &\geq id1' + 1 * r1 \Leftrightarrow \\ &\geq id1' + r1 \Leftrightarrow \quad (\text{the smallest value } id1' \text{ can attain is } 0) \\ &\geq r1 \end{aligned}$$

This is invalid as  $id1 \in [0, r1)$ , so Equation D.2 is invalid for  $p > 0$ .

3.  $p < 0$ : The left-hand-side of Equation D.3 is less or equal to the biggest value the right-hand-side can attain. In this case the right-hand-side is the biggest when  $p = -1$ . Using this value in Equation D.3 gives:

$$\begin{aligned} id1 &\leq id1' - 1 * r1 \Leftrightarrow \\ &\leq id1' - r1 \Leftrightarrow \quad (\text{division by } -1) \\ -id1 &\geq -id1' + r1 \Leftrightarrow \quad (\text{swap } id1 \text{ and } id1') \\ id1' &\geq id1 + r1 \Leftrightarrow \quad (\text{the smallest value } id1 \text{ can attain is } 0) \\ id1' &\geq r1 \end{aligned}$$

This is invalid as  $id1' \in [0, r1)$ , so Equation D.2 is invalid for  $p < 0$ .

This means that Equation D.2 is always invalid, so its inverse must be true, which means that Equation D.1 must be injective.

## D.2 Linearization of three ids

The linearization of three ids is defined as follows in the SYCL specification [72] (section 3.11.1):

$$id2 + (id1 * r2) + (id0 * r1 * r2) \quad (D.4)$$

The `linearize3` function in the SYCL header uses this formula, as described in subsection 6.3.3.1. It ensures three assumptions about the results, which are proven in the next subsections.

The `linearize3` method requires the bounds to the variables listed below, so it is never called with values outside these bounds. Therefore, these bounds on the variables will also be assumed in the proofs.

$$id0 \in [0, r0) \quad \wedge \quad id1 \in [0, r1) \quad \wedge \quad id2 \in [0, r2) \quad \wedge \quad r0 > 0 \quad \wedge \quad r1 > 0 \quad \wedge \quad r2 > 0$$

### D.2.1 Assumption 1: Minimum value

**Assumption 4**  $id2 + (id1 * r2) + (id0 * r1 * r2) \geq 0$

Because all variables are  $\geq 0$  and only  $+$  and  $*$  are used, the result of Equation D.4 decreases when  $id0$ ,  $id1$ , or  $id2$  decreases. This means that we can find the minimum value by using the smallest possible values for  $id0$ ,  $id1$ ,  $id2$ :

$$id0 = 0 \quad \wedge \quad id1 = 0 \quad \wedge \quad id2 = 0$$

Filling in those values in formula Equation D.1 gives us a minimum value of:

$$0 + (0 * r2) + (0 * r1 * r2) \quad \Leftrightarrow \quad 0 + 0 + 0 \quad \Leftrightarrow \quad 0$$

### D.2.2 Assumption 2: Maximum value

**Assumption 5**  $id2 + (id1 * r2) + (id0 * r1 * r2) < r0 * r1 * r2$

Because all variables are  $\geq 0$  and only  $+$  and  $*$  are used, the result of Equation D.1 increases when  $id0$ ,  $id1$ , or  $id2$  increases. This means that we can find the maximum value by using the biggest possible values for  $id0$ ,  $id1$ , and  $id2$ :

$$id0 = (r0 - 1) \quad \wedge \quad id1 = (r1 - 1) \quad \wedge \quad id2 = (r2 - 1)$$

Filling in those values in Equation D.1 gives us a maximum value of:

$$\begin{aligned} (r2 - 1) + ((r1 - 1) * r2) + ((r0 - 1) * r1 * r2) &\Leftrightarrow \\ r2 - 1 + r1 * r2 - r2 + r0 * r1 * r2 - r1 * r2 &\Leftrightarrow \\ r0 * r1 * r2 - 1 \end{aligned}$$

which is less than  $r0 * r1 * r2$ .

### D.2.3 Assumption 3: Injectivity

Equation D.4 is injective with regards to  $id0$ ,  $id1$ , and  $id2$ :

**Assumption 6**  $id2 + (id1 * r2) + (id0 * r1 * r2) \neq id2' + (id1' * r2) + (id0' * r1 * r2)$   
 where  $id0 \neq id0' \vee id1 \neq id1' \vee id2 \neq id2'$

If Equation D.4 would **not** be injective with regards to  $id0$ ,  $id1$ , and  $id2$ , then the following must be true for at least one set of values for  $id0$ ,  $id0'$ ,  $id1$ ,  $id1'$ ,  $id2$ , and  $id2'$ :

$$id2 + (id1 * r2) + (id0 * r1 * r2) = id2' + (id1' * r2) + (id0' * r1 * r2) \quad (D.5)$$

where  $id0 \neq id0' \vee id1 \neq id1' \vee id2 \neq id2'$

Suppose  $p$  denotes the difference between  $id0$  and  $id0'$ , such that  $id0 + p = id0'$ , and  $q$  denotes the difference between  $id1$  and  $id1'$ , such that  $id1 + q = id1'$ . Then Equation D.5 can be rewritten to:

$$\begin{aligned} id2 + (id1 * r2) + (id0 * r1 * r2) &= id2' + ((id1 + q) * r2) + ((id0 + p) * r1 * r2) \Leftrightarrow \\ id2 + id1 * r2 + id0 * r1 * r2 &= id2' + id1 * r2 + q * r2 + id0 * r1 * r2 + p * r1 * r2 \Leftrightarrow \\ id2 &= id2' + id1 * r2 + q * r2 + id0 * r1 * r2 + p * r1 * r2 - id1 * r2 - id0 * r1 * r2 \Leftrightarrow \\ id2 &= id2' + q * r2 + p * r1 * r2 \end{aligned} \quad (D.6)$$

Now we can distinguish nine cases:

- $p = 0 \wedge q = 0$ : In this case  $id0 = id0'$  and  $id1 = id1'$ , so  $id2 \neq id2'$  must be true for Equation D.5 to be valid. Filling in  $p = 0$  and  $q = 0$  in Equation D.6 gives:

$$\begin{aligned} id2 &= id2' + 0 * r2 + 0 * r1 * r2 \Leftrightarrow \\ &= id2' \end{aligned}$$

This is not allowed as we had just determined  $id2 \neq id2'$ . So Equation D.5 is invalid for  $p = 0 \wedge q = 0$ .

- $p = 0 \wedge q > 0$ : The left-hand-side of Equation D.6 is greater or equal to the smallest value the right-hand-side can attain. In this case the right-hand-side is the smallest when  $p = 0$  and  $q = 1$ . Using these values in Equation D.6 gives:

$$\begin{aligned} id2 &\geq id2' + 1 * r2 + 0 * r1 * r2 \Leftrightarrow \\ &\geq id2' + r2 \Leftrightarrow & (\text{the smallest value } id2' \text{ can attain is } 0) \\ &\geq r2 \end{aligned}$$

This is invalid as  $id2 \in [0, r2)$ , so Equation D.5 is invalid for  $p = 0 \wedge q > 0$ .

- $p = 0 \wedge q < 0$ : The left-hand-side of Equation D.6 is less or equal to the biggest value the right-hand-side can attain. In this case the right-hand-side is the biggest when  $p = 0$  and  $q = -1$ . Using these values in Equation D.6 gives:

$$\begin{aligned} id2 &\leq id2' - 1 * r2 + 0 * r1 * r2 \Leftrightarrow \\ &\leq id2' - r2 \Leftrightarrow & (\text{division by } -1) \\ -id2 &\geq -id2' + r2 \Leftrightarrow & (\text{swap } id2 \text{ and } id2') \\ id2' &\geq id2 + r2 \Leftrightarrow & (\text{the smallest value } id2 \text{ can attain is } 0) \\ &\geq r2 \end{aligned}$$

This is invalid as  $id2' \in [0, r2)$ , so Equation D.5 is invalid for  $p = 0 \wedge q < 0$ .



- $p > 0 \wedge q = 0$ : The left-hand-side of Equation D.6 is greater or equal to the smallest value the right-hand-side can attain. In this case the right-hand-side is the smallest when  $p = 1$  and  $q = 0$ . Using these values in Equation D.6 gives:

$$\begin{aligned}
id2 &\geq id2' + 0 * r2 + 1 * r1 * r2 \Leftrightarrow \\
&\geq id2' + r1 * r2 \Leftrightarrow && \text{(the smallest value } id2' \text{ can attain is 0)} \\
&\geq r1 * r2 \Leftrightarrow && \text{(the smallest value } r1 \text{ can attain is 1)} \\
&\geq r2
\end{aligned}$$

This is invalid as  $id2 \in [0, r2)$ , so Equation D.5 is invalid for  $p > 0 \wedge q = 0$ .

- $p > 0 \wedge q > 0$ : The left-hand-side of Equation D.6 is greater or equal to the smallest value the right-hand-side can attain. In this case the right-hand-side is the smallest when  $p = 1$  and  $q = 1$ . Using these values in Equation D.6 gives:

$$\begin{aligned}
id2 &\geq id2' + 1 * r2 + 1 * r1 * r2 \Leftrightarrow \\
&\geq id2' + r2 + r1 * r2 \Leftrightarrow && \text{(the smallest value } id2' \text{ can attain is 0)} \\
&\geq r2 + r1 * r2 \Leftrightarrow && \text{(the smallest value } r1 \text{ can attain is 1)} \\
&\geq r2 + r2
\end{aligned}$$

This is invalid as  $id2 \in [0, r2)$ , so Equation D.5 is invalid for  $p > 0 \wedge q > 0$ .

- $p > 0 \wedge q < 0$ : The left-hand-side of Equation D.6 is less or equal to the biggest value the right-hand-side can attain.  $p$  equals  $id0' - id0$ , so the biggest value  $p$  can attain can be found by filling in the biggest possible value for  $id0'$ , which is  $r0 - 1$  and the smallest possible value for  $id0$ , which is 0. This means the biggest value  $p$  can attain is  $r0 - 1 - 0 = r0 - 1$ . In this case the right-hand-side is the biggest when  $p = r0 - 1$  and  $q = -1$ . Using these values in Equation D.6 gives:

$$\begin{aligned}
id2 &\leq id2' - 1 * r2 + (r0 - 1) * r1 * r2 \Leftrightarrow \\
&\leq id2' - r2 + r0 * r1 * r2 - r1 * r2 \Leftrightarrow && \text{(division by } -1) \\
-id2 &\geq -id2' + r2 - r0 * r1 * r2 + r1 * r2 \Leftrightarrow && \text{(swap } id2 \text{ and } id2') \\
id2' &\geq id2 + r2 - r0 * r1 * r2 + r1 * r2 \Leftrightarrow && \text{(the smallest value } id2 \text{ can attain is 0)} \\
&\geq r2 - r0 * r1 * r2 + r1 * r2 \Leftrightarrow && \text{(add } r0 * r1 * r2 \text{ to both sides)} \\
id2' + r0 * r1 * r2 &\geq r2 + r1 * r2 \Leftrightarrow && \text{(the smallest value } r0 \text{ can attain is 1)} \\
id2' + r1 * r2 &\geq r2 + r1 * r2 \Leftrightarrow && \text{(subtract } r1 * r2 \text{ from both sides)} \\
id2' &\geq r2
\end{aligned}$$

This is invalid as  $id2' \in [0, r2)$ , so Equation D.5 is invalid for  $p > 0 \wedge q < 0$ .

- $p < 0 \wedge q = 0$ : The left-hand-side of Equation D.6 is less or equal to the biggest value the right-hand-side can attain. In this case the right-hand-side is the biggest when  $p = -1$  and  $q = 0$ . Using these values in Equation D.6 gives:

$$\begin{aligned}
id2 &\leq id2' + 0 * r2 - 1 * r1 * r2 \Leftrightarrow \\
&\leq id2' - r1 * r2 \Leftrightarrow && \text{(division by } -1) \\
-id2 &\geq -id2' + r1 * r2 \Leftrightarrow && \text{(swap } id2 \text{ and } id2') \\
id2' &\geq id2 + r1 * r2 \Leftrightarrow && \text{(the smallest value } id2 \text{ can attain is 0)} \\
&\geq r1 * r2 \Leftrightarrow && \text{(the smallest value } r1 \text{ can attain is 1)} \\
&\geq r2
\end{aligned}$$

This is invalid as  $id2' \in [0, r2)$ , so Equation D.5 is invalid for  $p < 0 \wedge q = 0$ .

- $p < 0 \wedge q > 0$ : The left-hand-side of Equation D.6 is greater or equal to the smallest value the right-hand-side can attain.  $p$  equals  $id0' - id0$ , so the smallest value  $p$  can attain can be found by filling in the smallest possible value for  $id0'$ , which is 0 and the biggest possible value for  $id0$ , which is  $r0 - 1$ . This means the smallest value  $p$  can attain is  $0 - (r0 - 1) = 1 - r0$ . In this case the right-hand-side is the smallest when  $p = 1 - r0$  and  $q = 1$ . Using these values in Equation D.6 gives:

$$\begin{aligned}
id2 &\geq id2' + 1 * r2 + (1 - r0) * r1 * r2 \Leftrightarrow \\
&\geq id2' + r2 + r1 * r2 - r0 * r1 * r2 \Leftrightarrow \text{(the smallest value } id2' \text{ can attain is 0)} \\
&\geq r2 + r1 * r2 - r0 * r1 * r2 \Leftrightarrow \text{(add } r0 * r1 * r2 \text{ to both sides)} \\
id2 + r0 * r1 * r2 &\geq r2 + r1 * r2 \Leftrightarrow \text{(the smallest value } r0 \text{ can attain is 1)} \\
id2 + r1 * r2 &\geq r2 + r1 * r2 \Leftrightarrow \text{(subtract } r1 * r2 \text{ from both sides)} \\
id2 &\geq r2
\end{aligned}$$

This is invalid as  $id2 \in [0, r2)$ , so Equation D.5 is invalid for  $p < 0 \wedge q > 0$ .

- $p < 0 \wedge q < 0$ : The left-hand-side of Equation D.6 is less or equal to the biggest value the right-hand-side can attain. In this case the right-hand-side is the biggest when  $p = -1$  and  $q = -1$ . Using these values in Equation D.6 gives:

$$\begin{aligned}
id2 &\leq id2' - 1 * r2 - 1 * r1 * r2 \Leftrightarrow \\
&\leq id2' - r2 - r1 * r2 \Leftrightarrow \text{(division by } -1) \\
-id2 &\geq -id2' + r2 + r1 * r2 \Leftrightarrow \text{(swap } id2 \text{ and } id2') \\
id2' &\geq id2 + r2 + r1 * r2 \Leftrightarrow \text{(the smallest value } id2 \text{ can attain is 0)} \\
&\geq r2 + r1 * r2 \Leftrightarrow \text{(the smallest value } r1 \text{ can attain is 1)} \\
&\geq r2 + r2
\end{aligned}$$

This is invalid as  $id2' \in [0, r2)$ , so Equation D.5 is invalid for  $p < 0 \wedge q < 0$ .

This means that Equation D.5 is always invalid, so its inverse must be true, which means that Equation D.4 must be injective.