EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Formal System Analysis

Towards relaxed memory semantics for the Autonomous Data Language

Master Thesis

Gijs Pieter Leemrijse

Committee:
dr. ir. Thomas Neele
dr. ir. Anton Wijs
dr. ir. Roel Jordans

August, 2023

Abstract

This work presents an alternative operational semantics for the Autonomous Data Language (AuDaLa) with relaxed memory consistency and incoherent memory. We show how the memory operations of our semantics can be safely mapped onto the NVIDIA PTX virtual ISA and demonstrate that our semantics performs faster than the original when executing AuDaLa programs on GPUs. We translate our operational semantics into an axiomatic memory consistency model and formally check, for a bounded program size, its correspondence with PTX's memory consistency model using the Alloy model finder. We conclude by presenting AuDaLaC, our compiler targeting the CUDA platform, with which we explore several different strategies to compile AuDaLa programs. We demonstrate in several case studies that AuDaLa implementations can perform faster than sequential implementations.

1. Introduction

Now that single core performance improvements are running out of steam [41], the field of high-performance computing is increasingly focused on multicore computing on platforms such as GPUs. A modern GPU contains thousands of cores, albeit simpler cores compared to a conventional CPU. These parallel capabilities make GPUs an attractive platform for massively parallel algorithms. However, orchestrating program execution on such a device is a complex task. Therefore, many purpose-made languages and frameworks for parallel processing exist [27, 54, 60, 64]. All of these attempt to hide the complex implementation details of parallel execution from the programmer. Most of these take a *task-parallel* or *data-parallel* approach [18]. A novel and different view, the *data-autonomous* view, is taken by the *Autonomous Data Language (AuDaLa)* due to Franken, Neele and Groote [21]. In the data-autonomous paradigm, data elements not only store their data but also perform computations on their data. These computations are carried out according to a schedule. Through references to other struct instances, communication can take place between struct instances. AuDaLa presents a significant abstraction from parallel hardware architectures and instead concentrates on data and their computations. It serves as a new mode of expressing parallel computations without having to know implementation details such as memory layouts and work division strategies. This makes it an attractive programming language and an interesting research topic. However, the *semantics* defined by Franken et al. [21] are relatively *strict*. We argue that AuDaLa can benefit from *weaker* semantics such that more optimisations can be performed. In this work, we investigate how AuDaLa's semantics can be weakened and what performance benefits this brings.

1.1. Research questions and motivation

It is essential for software optimisations to adhere to the semantics of the programming language. Otherwise, optimisations introduce unexpected behaviour. The semantics of the programming language, therefore, determine for a large part which optimisations can be performed. When semantics allow most memory operations to be reordered, it is said to have *relaxed memory consistency*. When semantics allow different processors to have different views of the memory, it is said to have *incoherent memory*. We argue that AuDaLa's semantics can have these qualities without sacrificing programmability. Both of these qualities provide opportunities for significant performance benefit. We are therefore interested to know:

> **RQ 1:** *How can semantics be defined for AuDaLa that allow relaxed memory consistency and incoherent memory?*

Once we have our semantics defined, the next question to ask is how they can be compiled correctly. GPUs are a natural target for an AuDaLa compiler due to their parallel computing capability. In our work, we use GPUs that are programmed with the PTX [53] Instruction Set Architecture (ISA). PTX comes with semantics of its own [46]. Therefore, we must make sure that our programs, when compiled to PTX, adhere to the semantics of AuDaLa. We are particularly interested in memory-related semantics, as that is the most error-prone. This is the focus of our second research question:

> **RQ 2:** *How can AuDaLa memory operations be compiled into PTX memory operations such that they adhere to the semantics of AuDaLa?*

Weaker semantics are harder to reason about than stricter semantics. To determine whether the runtime performance speedup outweighs this difficulty of reasoning, we need to quantify the speedup gained. We leave the performance benefits of incoherence for future work. Our third research question is thus:

> **RQ 3:** *What is the runtime speedup of relaxed memory consistency semantics over the semantics defined by Franken et al. for AuDaLa programs on GPUs?*

The simplest form of programming is sequential programming. However, sometimes better runtime performance is desired that only parallel computing can provide. Unfortunately, this comes at a cost of implementation complexity. AuDaLa attempts to reduce this complexity while still being more performant than sequential implementations. For our final research question we thus want to find out:

> **RQ 4:** *How does the runtime performance of algorithms implemented in AuDaLa compare to conventional sequential implementations?*

1.2. Contributions

To answer our research questions, we have made the following contributions:

1. We define an alternative *operational* semantics for AuDaLa, with relaxed memory consistency and incoherent memory for better performance.
2. We translate our operational semantics into an *axiomatic memory consistency model* and formally check, for a bounded program size, its correspondence with PTX's memory consistency model [46]. We show that the *out-of-thin-air problem* [13] causes both models to be incomparable when we model dependencies. When we model modulo dependencies, all possible PTX executions are allowed by AuDaLa's semantics.
3. We present *AuDaLaC*, our compiler that targets the CUDA platform, with which we explore several different strategies to compile AuDaLa programs.
4. We adapt for and implement in AuDaLa the *Small Progress Measures* algorithm for solving parity games [33], an explicit version of the *Supervisory Controller Synthesis* algorithm of [57], and the *Forward-Backward* [20] and *Colouring/Heads Off* [55] algorithms for extracting Strongly Connected Components (SCC).
5. We measure the runtime performance gain of our semantics over the original semantics. Furthermore, we compare the runtime performance of the different compilation strategies implemented by AuDaLaC with that of sequential implementations.

1.3. Report structure

The rest of this report is structured as follows: Section 2 provides all the theory and concepts required to read the other sections. Then, Section 3 presents our alternative operational semantics in detail. We formalise and check the correctness of our mapping from AuDaLa to PTX in Section 4. Next, we present the implementation details of AuDaLaC in Section 5. We continue by performing experiments using AuDaLaC and provide the results in Section 6. Then, we relate other work in the literature to our own in Section 7. We conclude our work and discuss future work opportunities in Section 8.

2. Preliminaries

This section presents the concepts and theory required to read the other chapters. We start by introducing AuDaLa, then we move on to define the notational conventions used throughout this work. We continue with a treatment of memory consistency models and how they can be formalised axiomatically. Next, we introduce the out-of-thin-air problem and the proposed solutions. We conclude with a section on the CUDA platform and the PTX ISA.

2.1. The Autonomous Data Language

AuDaLa programs are defined using *structs*, of which the instances are called *struct instances*. Each struct provides a kind of template for the data stored by each struct instance. It does so by defining named data attributes called *parameters*. Each parameter has a single type: `Bool`, `Int`, `Nat`, or a reference type to another struct. Furthermore, a struct can define its own operations called *steps*. These steps are autonomously executed by each struct instance according to some fixed *schedule* to form a program. Only one step can be executed at a time: all struct instances must be finished with the previous step before the next step can begin. Each step has a body, containing a sequence of *statements*. Statements are simple and are either a declaration, assignment, struct instance construction, or if statement. For the full syntax, we refer the reader to [21]. It is important to note that there are no loops in AuDaLa. Instead, AuDaLa relies on *fixpoints* in its schedule for iteration. The schedule is formed by a sequence of *step calls* and fixpoints containing subschedules. A fixpoint executes its subschedule until a fixpoint is reached, that is, until no parameters change during an iteration. The schedule supports nested fixpoints. At the start of the program, each structure has only one instance called the *null instance*. The parameters of the null instances are immutable. The null instances serve as a default value for reference types and to create other struct instances.

Example See Figure 1 for an example. There, two structs called `Node` and `Edge` are defined. The `Node` struct has a boolean parameter called `reachable`, and `Edge`'s parameters are called `source` and `target` and are of type `Node`. `Node` contains a step `init`, and `Edge` contains a step called `reachability`. The body of the `init` step contains only declarations that instantiate `Node` and `Edge` instances. At the bottom, the schedule is defined. The `init` step is executed once, after which the `reachability` step is executed in a fixpoint iteration. In the `reachability` step, all edges check if their source node is reachable and, if so, set their target node to be reachable as well. We illustrate the execution states of the program in Figure 2. Observe that initially only the null instances exist. Then, after calling `init`, the graph is instantiated and all nodes are unreachable, except for s_1. After the first iteration, n_2 becomes reachable and the fixpoint iteration continues. The second iteration changes n_3 to be reachable. Thus, the iteration continues. Finally, after the third iteration, nothing is changed and the fixpoint iteration terminates. As there are no steps left in the schedule, the program terminates.

```
// The definition of the Node struct
struct Node (reachable : Bool) {
    // The definition of the init step
    init {
        Node s1 := Node(true);
        Node n1 := Node(false);
        Node n2 := Node(false);
        Node n3 := Node(false);
        Edge e1 := Edge(s1, n2);
        Edge e2 := Edge(n3, n2);
        Edge e3 := Edge(n1, n3);
        Edge e4 := Edge(n2, n3);
    }
}
// The definition of the Edge struct
struct Edge (source : Node, target : Node) {
    // The def. of the reachability step
    reachability {
        if source.reachable then {
            target.reachable := true;
        }
    }
}
init < Fix(reachability) // The schedule
```

Figure 1: An example AuDaLa program specification. It computes the nodes reachable from `s1` via a parallel Breadth First Search.

2.2. Notational conventions

In this section we briefly introduce the notational conventions used in the rest of the work.

Lists, functions and sets Given a set D, we denote the powerset by 2^D and the cardinality of D by $|D|$. A list with elements in D is of the type D^*. The empty list is denoted by ε. Concatenation is denoted with a semicolon. By convention, concatenation with the empty list is omitted such that the list with two elements e_1 and e_2 is denoted by $e_1; e_2$ and the singleton list with element e is denoted by e.

Given a function $f : A \to B$, the update of f such that $a \in A$ maps to $b \in B$, is denoted by $f[a \mapsto b]$. For the updated function holds that $f[a \mapsto b](a) = b$, and $f[a \mapsto b](x) = f(x)$ for all $x \in A \setminus \{a\}$. The function update can be lifted to sets of updates, i.e. $f[\{a_1 \mapsto b_1, a_2 \mapsto b_2, \ldots\}] = f[a_1 \mapsto b_1][a_2 \mapsto b_2]\ldots$. This is only well defined if the domain of the updates $(a_1, a_2, \ldots)$ is pairwise distinct, as otherwise the order of updates becomes significant.

Relations Relations are represented as sets of tuples, i.e. a binary relation R with a domain in D and a co-domain in C is defined as $R \subseteq D \times C$. Given a binary relation R, we write R^* for its reflexive-transitive closure, R^+ for its transitive closure, and R^{-1} for its inverse. We write $R_1; R_2$ for the left composition of relations R_1 and R_2. We assume that $^{-1}$, $^+$, and * bind strongest and ; binds stronger than $\cup$ and $\setminus$. The identity relation is denoted by $\mathbb{I}$. We define the following predicates on relations:

- $\textbf{lone}(R) \triangleq (a, b), (a, b') \in R \Rightarrow b = b'$
- $\textbf{acyclic}(R) \triangleq R^+ \cap \mathbb{I} = \emptyset$
- $\textbf{total}(R, S) \triangleq \textbf{acyclic}(R) \wedge \forall s_1, s_2 \in S \wedge s_1 \neq s_2 : (s_1, s_2), (s_2, s_1) \in (R^+ \cup (R^{-1})^+)$
- $\textbf{injective}(R, S_1, S_2) \triangleq S_2; R^{-1} = S_1 \wedge \textbf{lone}(R^{-1})$.
- $\textbf{bijective}(R, S_1, S_2) \triangleq (\forall s \in S_1 : |s; R| = 1 \wedge s; R \in S_2) \wedge (\forall s \in S_2 : |s; R^{-1}| = 1 \wedge s; R^{-1} \in S_1)$.

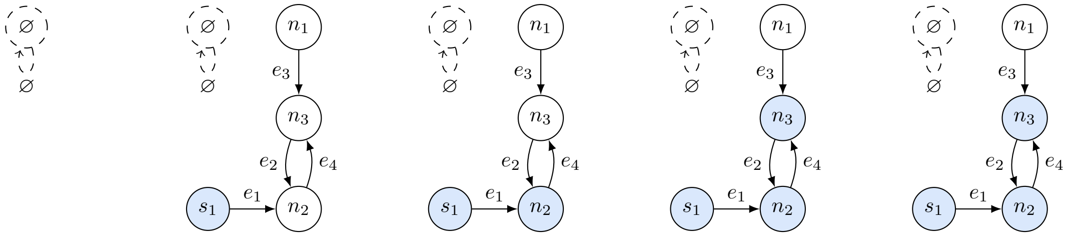
For a set A, we denote the identity relation of A with $[A]$. Which is defined as $[A] \triangleq (A \times A) \cap \mathbb{I}$. Specifically, note that $[A]; R; [B] = R \cap (A \times B)$.

2.3. Memory Consistency Models

A *Memory Consistency Model (MCM)* defines what ordering and visibility guarantees exist between *memory operations* in a shared-memory multiprocessor system. Loads and stores are examples of memory operations. The more guarantees an MCM provides, the *stronger* or *stricter* it is said to be. *Weaker* or *more relaxed* MCMs have fewer restrictions and therefore often provide more chances for optimisation. In this section, we discuss three classes of consistency models supported by C++ and PTX in decreasing order of strength. We illustrate the properties of each MCM via *litmus tests*, which are small test programs written in pseudocode. We use **`x`** and **`y`** to denote variables stored in memory and **`r1`**...**`r4`** for thread-local registers. We conclude this section by discussing the notion of *coherency*.

Sequential consistency The most intuitive memory consistency model is *sequential consistency* (SC). This is the MCM that is used in the semantics of Franken et al. [21]. In a sequentially consistent memory model, an execution appears as a single total order of reads and writes, where each read reads the last write to that location [37]. In addition, loads and stores execute atomically. Multiprocessor programs, therefore, behave as a simple interleaving of each program's instructions in program order. This makes formal reasoning about programs easy. Consider Figure 3 with the *Message Passing* (MP) litmus test [3]. There, two shared variables **`x`** and **`y`** are initially set to **`0`**. The first thread sets **`x`** to **`1`** and only then sets **`y`** to **`1`**. The other thread first reads **`y`** and checks if it is set to **`1`**, if so, it prints the value of **`x`**. With SC semantics, this value is always **`1`**.

Modern multiprocessors, however, do not exhibit SC behaviour by default. Hardware optimisations, such as write buffers, out-of-order execution, and non-atomic loads and stores violate SC [1, 7]. It is easy to see how out-of-order execution would violate SC in Figure 3 by



Before `init` is called.

After `init` is called.

After 1 iteration of `reachability`.

After 2 iterations of `reachability`.

After 3 iterations of `reachability`.

Figure 2: An illustration of the program states the example AuDaLa program of Figure 1 goes through. The dashed objects represent null instances and blue nodes are reachable. In the third iteration nothing changes and the fixpoint terminates.

Initially **`x, y = 0`**

`x = 1;`	**`if (y == 1)`**
`y = 1;`	**`print x;`**

Figure 3: The MP litmus test.

Initially **`x, y = 0`**

`x = 1;`	**`r1 = x;`**
`y = 1;`	**`if (y == 1)`**
	`print x;`

Figure 4: The MP litmus test with an earlier load of x stored in register **`r1`**.

Initially **`x, y = 0`**

`x = 1;`	**`r1 = x;`**	**`r3 = y;`**	**`y = 1;`**
	`r2 = y;`	**`r4 = x;`**	

Figure 5: The IRIW litmus test.

Initially **`x, y = 0`**

`r1 = y;`	**`r2 = x;`**
`x = 42;`	**`y = 42;`**

Figure 6: The LB litmus test.

swapping the first thread's stores. Not only hardware optimisations cause this violation, modern optimising compilers can also cause SC violations through their optimisations [7, 48, 69]. Figure 4 provides an example: many compilers apply common subexpression elimination and print the value of **`r1`** instead of loading **`x`** a second time. This violates SC and could print **`0`** instead of **`1`**. Through a combination of compiler restrictions and fence instructions, it is possible to maintain SC in modern multiprocessors, albeit at a performance cost [24, 43].

Release/Acquire consistency The *Release/Acquire* (RA) memory consistency model is specifically designed to perform "message passing" as in Figure 3, without having to resort to the expensive SC semantics. Consider a write **`W`** with *release semantics* and a read **`R`** with *acquire semantics*. If **`R`** reads from **`W`**, they are said to *synchronise* with each other [29]. If **`R`** and **`W`** synchronise, the release semantics of **`W`** guarantee that the effects of any operations before **`W`** in program order will be visible to any operations after **`R`** in program order. Furthermore, the acquire semantics of **`R`** guarantee that the operations following **`R`** in program order will not execute until **`R`** is completed.

Contrary to SC, not all threads have to agree on a total order of writes when using RA semantics, as is evident from the well-known Independent Reads Independent Writes (IRIW) litmus test [12] of Figure 5. If all operations are performed with RA semantics, the outcome **`r1 = 1, r2 = 0, r3 = 1, r4 = 0`** is allowed. This result indicates that the middle two threads perceive a different order of writes to **`x`** and **`y`**. With SC semantics, this outcome would be illegal, as only one order of writes exists.

Relaxed consistency The most relaxed model is fittingly named the *Relaxed* memory consistency model. Relaxed memory accesses do not synchronise threads or provide any ordering to concurrent accesses. However, it does provide atomicity and a consistent total ordering of writes to a **single** location [29, 53]. In Figure 6 the *Load Buffering* (LB) litmus test [3] is listed. Because each thread's operations are independent of each other and no ordering is imposed via relaxed loads and stores, the operations are allowed to be reordered. Therefore, the result **`r1 = r2 = 42`** is allowed for Relaxed MCM.

Coherency Another concept important for MCMs, but not a class of MCM in itself, is that of *coherency*. When a system's memory is coherent, all processors agree on the value stored at an address at all times: they have the same view of the memory. If a system has multiple processors with local caches and those caches are said to be *incoherent*, two processors could see different data stored at the same address. To maintain coherency, the caches would have to send invalidation signals to each other, which come at a performance cost. Therefore, some systems choose to have incoherent caches for performance benefit.

2.4. Axiomatic MCMs

In literature, two MCM formalisation techniques are popular: the *axiomatic* [3, 36, 46, 70] and the *opera-*

tional [3, 26, 34, 51] approach. In an operational MCM, an outcome is legal if it can be produced by an execution of some abstract machine that models the architecture or language. In an axiomatic MCM, executions are represented as graphs. Entities in the program, such as instructions and addresses, are represented as nodes while relations between these entities, such as "reading from", are modelled as edges. Defined axioms decide on the legality of the executions.

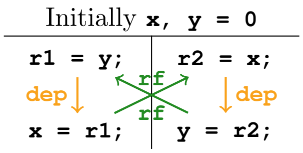
The operational approach is often more intuitive, while the axiomatic approach lends itself better to automated verification [2]. Ideally, both approaches are used and proven equivalent, as is done in [3, 16, 51, 58], leaving the choice up to the user. In this work, we formalise the complete operational semantics of AuDaLa, which already includes its MCM. Yet, we also provide AuDaLa's MCM in an axiomatic fashion, derived from the operational semantics. This makes a comparison with the PTX axiomatic MCM [46] easier.

Pre-executions and witnesses Instead of reasoning directly on the program source code, axiomatic models reason on program *candidate executions* expressed as graphs. The basis of a candidate execution is formed by a *pre-execution*. A pre-execution models a straight-line execution in which all control flow is completely unrolled (no loops are present) and all addresses have been resolved. Therefore, different pre-executions represent different runs of the same program. In addition, a pre-execution models dependencies and threads.

Although many aspects of an execution are already contained in a pre-execution, it is not yet complete. We are still missing which reads read from which writes, and which writes overwrite each other. This information is modelled as a set of edges called the *execution witness*. An execution witness completes a pre-execution to form a complete candidate execution. Not all candidate executions are *legal executions*. To filter out all legal executions, predicates are defined called *axioms*. A candidate execution is legal if and only if all axioms hold for that candidate execution.

2.5. Out-of-thin-air problem

The MCM of an optimised language strives to allow as many compiler optimisations and hardware reorderings as possible by weakening their ordering requirements. However, weakening too much causes the MCM to allow nonsensical behaviour not observeable in practice. This type of behaviour is called *out-of-thin-air* (OOTA) behaviour [7]. OOTA behaviour makes formal reasoning and compiler optimisation very difficult [6, 13, 65, 66]. How to define relaxed concurrency semantics that prevent OOTA behaviour, allow compiler optimisations, are suitable for formal verification, and allow (compositional) reasoning is an open problem [31, 32, 34, 59]. In Figure 7 the classical thin-air-execution example [11] is shown. A causal cycle is formed by the loads reading from the stores that transitively depend on them, allowing OOTA results such as **r1 = r2 = 42**.



```
Initially x, y = 0
r1 = y;   | r2 = x;
x = r1;   | y = r2;
```

Figure 7: The classical example of a thin-air-execution. The **rf** arrow indicates which load reads from which store and **dep** indicates a dependency between operations.

```
      Initially x, y = 0
r1 = x;  | r2 = y;
y = r1;  | if (r2 == 42)
         |   x = 42;
         | else
         |   x = 42;
```

Figure 8: An example illustrating compilers breaking dependencies.

Out-of-thin-air & dependencies The heart of the OOTA problem lies in the formalisation's treatment of dependencies. An MCM defines what ordering guarantees between instructions exist. A dependency between two instructions implies an ordering: If an instruction I_1 produces an operand used by another instruction I_2, then I_1 should execute before I_2. However, optimising compilers can break dependencies in favour of performance, destroying the ordering guarantee. It is impossible to define a per-candidate-execution condition that correctly decides which dependencies should be preserved and which could potentially be destroyed [7]. Consider the example of Figure 8. The allowed results are **r1 = r2 = 0** and **r1 = 42, r2 = 0**. The load of **y** to register **r2** cannot read **42** before executing the if statement. At the same time, the if statement cannot be executed before the load of **y**. Therefore, it is impossible that **r2** is assigned the value **42**. However, notice how the second thread stores **42** to **x** no matter the value of **r2**. Thus, an optimising compiler will replace the complete if statement with a store of **42** to **x**, breaking its dependency on the load of **y** and enabling the previously impossible result of **r1 = r2 = 42**.

Solutions There have been a few solutions proposed, each with their own deficiencies. They can be classified by their approach to dependency tracking in three categories [32]:

- With *syntactic dependencies* [13, 35, 36], all dependencies present in the source code are preserved. This disallows many compiler optimisations and the performance hit this incurs may be unacceptable. Hardware memory models also use syntactic dependencies [3].

- With *semantic dependencies* [17, 34, 38, 59], formalisations reason over all possible executions of a program in order to separate the true, semantic, dependencies from the false dependencies. These models are therefore called *multi-execution* models [50]. In addition, these approaches need to reason about values being read, which becomes intractable quickly.
- The last approach is to (largely) ignore dependencies. This allows OOTA behaviour but enables optimisations and is currently used by the C++, PTX, and JavaScript standards [7, 46, 70]. This approach to dependency modelling is "effectively incomplete, waiting for a better solution" [44].

Our definition of the operational semantics of AuDaLa effectively falls into the category of syntactic dependencies. In Section 4.2 we define two variants of our axiomatic MCM. One variant is true to the operational semantics and thus falls into the syntactic dependencies category, the other variant falls into the largely ignored dependencies category.

2.6. The CUDA platform

To simplify the interface with the GPU, NVIDIA created the CUDA platform for General-purpose computing on GPUs (GPGPU). In the CUDA programming model, a distinction is made between the *host* (CPU) and the *device* (GPU). The code for both the host and the device is usually written in C++ and is part of the same compilation unit. The device code is organised in *kernels*, which are specially annotated functions that are executed in parallel. The GPU has dedicated off-chip *device memory*. The host code is responsible for copying input and output data to/from device memory and launching kernels.

Execution hierarchy The smallest logical unit of computation in CUDA is the *thread*. When a kernel is launched, each thread executes the kernel's code in parallel, often with different data. On hardware, each thread is executed by one or more *CUDA cores*. Multiple threads are grouped in *thread blocks* which are mapped onto the *streaming multiprocessors (SMs)* of the GPU. Streaming multiprocessors contain a number of *CUDA cores*, various other types of cores depending on hardware generation, *caches*, and a *register file*. The register file is shared between all active threads of the SM. Therefore, *register pressure* can restrict the maximum number of threads active per SM and the maximum block size. The SMs execute the threads of the blocks assigned to them in batches of 32 threads called *warps*, which execute in lock-step fashion. The thread blocks are logically grouped into *thread grids*, which is the unit of work submitted to the device by the host.

Memory hierarchy CUDA GPUs have two additional levels of memory hierarchy between the register file and off-chip device memory. These are called the *L1* and *L2* caches. Each SM has a dedicated L1 cache and can optionally be configured as scratchpad memory called *Shared Memory*, which is only accessible by threads of the same block. The L1 caches are incoherent with respect to each other and are therefore not suitable for storing *strong* memory variables (see the **PTX memory orders** paragraph). Each GPU has a single L2 cache shared by all SMs, which is therefore suitable for storing strong memory variables with a device scope. L1 and L2 memory is on-chip memory and therefore has a much lower latency than the off-chip global memory. L1 has an even lower latency than L2. However, the capacity of L1 is usually limited to less than 100 KB/SM. The L2 capacity on consumer GPUs is in the order of a few megabytes.

Memory Coalescing The device memory is capable of performing very large memory transactions of 128 consecutive bytes. This means that 32 four-byte words can be read in parallel. By ensuring that all 32 threads in a warp read 32 consecutive words, this read can be combined into a single transaction. In addition, cache lines are 128 bytes wide and can thus also be filled completely with a single transaction. This is called *memory coalescing* and significantly speeds up performance.

Latency hiding and occupancy GPUs employ a technique called *latency hiding*, which is the practise of doing other useful work while waiting on the latency of an instruction or memory. This is achieved by the *warp schedulers*, of which each SM has one or more. These maintain a set of *resident warps*. Each cycle, the schedulers pick an *eligible* warp and issue one or more instructions for that warp. A warp is called eligible if it is not stalled. Latency hiding works best if all schedulers are completely saturated with warps. The fraction of actual resident warps over the maximum number of possible resident warps is called the *occupancy*.

CUDA graphs The *CUDA graphs API* is an alternative method of submitting work to the GPU, which is available since CUDA version 10. With CUDA graphs, a series of kernel launches is represented as a directed acylic graph. Once instantiated and uploaded to the GPU, a graph can be launched repeatedly with much lower overhead than regular kernel launches. Making it an ideal choice for short kernels that are repeatedly launched. In addition, as the complete sequence of operations is known when instantiating the graph, the CUDA driver can perform more optimisations than it could with a per-kernel view. Furthermore, graphs uploaded to the GPU can be launched from within a kernel on the GPU.

PTX CUDA programs are compiled using the *NVIDIA CUDA Compiler (NVCC)* into an instruction set called *Parellel Thread Execution (PTX)* [53]. It is important to note that PTX is not a classical hardware *Instruction Set*

Architecture (ISA), instead it functions more as a compatibility layer between architecture generations. The PTX source code produced by NVCC is optimised and compiled by the NVIDIA driver into NVIDIA's proprietary hardware ISA called *Source And Assembly (SASS)*, which can differ per architecture generation.

PTX memory orders Within PTX, memory operations can be tagged with several *qualifiers*. An important qualifier for this work is the *memory order qualifier*. PTX's MCM is designed to be mapped to in a straightforward manner from C++'s MCM [46]. Therefore, the available qualifiers are `.relaxed`, `.acquire`, and `.release`. Sequentially consistent operations are mapped to Release/Acquire operations preceded by a `fence.sc` instruction. Operations with these qualifiers are called *strong* operations. The *weak* operations have the `.weak` qualifier. These translate to non-atomic loads and stores and are the most performant type of operation since they need less synchronisation and more optimisations are allowed. They are not guaranteed to be coherent and, without other means of synchronisation, weak operations are unreliable for inter-thread communication.

PTX scopes Strong operations in PTX provide their atomicity and ordering guarantees only in a certain *scope*. PTX defines three scopes: *System scope* includes all threads in the current program, including those on the host CPU or other GPUs. *Device scope* contains only the threads of the GPU that executes the operation. *Block scope* contains all threads that execute in the same block as the executed operation.

3. Operational Semantics

This section presents our operational semantics for AuDaLa. Our semantics is an adaptation of the operational semantics due to Franken et al. [21]. Where Franken et al.'s semantics makes use of a sequentially consistent memory model, we use a more relaxed and incoherent memory consistency model, to better fit the execution model of GPUs and increase performance.

We start by introducing the Abstract Syntax Tree (AST) used to represent an AuDaLa program. Then, we show how the AST is transformed into a sequence of semantic commands that correspond to atomic steps of the semantics. Subsequently, we introduce the concepts that make up the semantic representation of the program state space. Next, we list the derivation rules that define the transition relation used to express AuDaLa's graph semantics. We conclude this section with a discussion of our semantics.

3.1. Formal program definitions

The type of valid identifiers used within AuDaLa is denoted by $\mathbb{ID}$ and the supported binary operators are of type $\mathbb{O}$, which contains at least the logical AND and OR operators which are denoted by `&&` and `||` respectively. Furthermore, the supported syntactic types are contained in $\mathbb{T}$, which is defined as $\mathbb{T} \triangleq \{\texttt{Nat, Int, Bool}\} \cup \mathbb{ID}$. An AuDaLa program is parsed into an AST by our implementation:

Definition 1 (Abstract syntax tree)**.** *Let $\mathbb{Z}$ be the integer type, $\mathbb{N}$ the natural number type and let $\mathbb{B}$ be the boolean type, then the* expression ($\mathbb{E}$)*,* statement ($\mathbb{S}$) *and* literal ($\mathbb{L}$) *types are defined by the following abstract data types:*

$$\begin{aligned}
\mathbb{E} &::= \boldsymbol{Op}\ \mathbb{E} \times \mathbb{O} \times \mathbb{E} \mid \boldsymbol{Cons}\ \mathbb{ID} \times \mathbb{E}^* \\
&\quad \mid \boldsymbol{Var}\ \mathbb{ID}^* \mid \boldsymbol{Not}\ \mathbb{E} \mid \boldsymbol{Lit}\ \mathbb{L} \\
\mathbb{S} &::= \boldsymbol{If}\ \mathbb{E} \times \mathbb{S}^* \times \mathbb{S}^* \\
&\quad \mid \boldsymbol{Declare}\ \mathbb{T} \times \mathbb{ID} \times \mathbb{E} \\
&\quad \mid \boldsymbol{Assign}\ \mathbb{ID}^* \times \mathbb{E} \\
\mathbb{L} &::= \boldsymbol{Nat}\ \mathbb{N} \mid \boldsymbol{Int}\ \mathbb{Z} \mid \boldsymbol{Bool}\ \mathbb{B} \\
&\quad \mid \boldsymbol{Null}\ \mathbb{ID} \mid \boldsymbol{This}
\end{aligned}$$

For brevity, we omit the parts of the AST that are irrelevant for our semantic discussion such as structure definitions. We do define the function $Par(\vartheta) : \mathbb{ID} \to \mathbb{ID}^*$ that returns the list of parameters of the struct type identified by $\vartheta \in \mathbb{ID}$. In addition, the function $TypeOf(\vartheta, v) : \mathbb{ID} \times \mathbb{ID} \to \mathbb{T}$ returns the type of the parameter v of struct ϑ.

The schedule to be executed is represented by a list of the schedule type $\mathcal{SC}$, which is either a step call or a fixpoint iteration:

Definition 2 (Schedule)**.** *The* schedule type $\mathcal{SC}$ *is defined by the following abstract data type:*

$$\mathcal{SC} ::= \boldsymbol{Call}\ \mathbb{ID} \times \mathbb{ID} \mid \boldsymbol{Fix}\ \mathcal{SC}^* \mid \boldsymbol{aFix}\ \mathcal{SC}^*$$

Note that **aFix** is only used as a semantic symbol and is not present in the initial state of a program.

Within our semantics, we make use of *labels* to reference concrete struct instances. We define $\mathcal{L}$ as the set of all labels, which contains sufficiently many elements to uniquely identify each struct instance. In addition, for each structure type $\vartheta \in \mathbb{ID}$ we also define the *null-label*, ℓ^0_ϑ which serves as the default value for a reference to ϑ. The set of all null-labels is defined as $\mathcal{L}^0$, such that $\mathcal{L}^0 \subseteq \mathcal{L}$ and $\ell^0_\vartheta \in \mathcal{L}^0$.

All *semantic values* are contained in the $\mathcal{V}$ set, which is defined as the union of the natural numbers, integers, booleans and labels, i.e. $\mathcal{V} \triangleq \mathbb{N} \cup \mathbb{Z} \cup \mathbb{B} \cup \mathcal{L}$. We define a *default value* for each syntactic type $T \in \mathbb{T}$ via the function $defaultVal : \mathbb{T} \to \mathcal{V}$, defined as:

$$defaultVal(T) \triangleq \begin{cases} 0 & \textbf{if } T \in \{\texttt{Nat}, \texttt{Int}\} \\ \mathit{false} & \textbf{if } T = \texttt{Bool} \\ \ell^0_T & \textbf{if } T \in \mathbb{ID} \end{cases}$$

We extract the semantic value from a literal $L \in \mathbb{L}$ via the function $val_\ell : \mathbb{L} \to \mathcal{V}$, where $\ell \in \mathcal{L}$, defined as:

$$val_\ell(L) \triangleq \begin{cases} x & \textbf{if } L \in \{\boldsymbol{Nat}(x), \boldsymbol{Int}(x), \boldsymbol{Bool}(x)\} \\ \ell^0_\vartheta & \textbf{if } L = \boldsymbol{Null} \\ \ell & \textbf{if } L = \boldsymbol{This} \end{cases}$$

The atomic operations within the semantics are represented as *semantic commands* of type $\mathcal{C}$:

Definition 3 (Semantic command)**.** *The* command *type* $\mathcal{C}$ *is defined by the following algebraic data type:*

$$\begin{aligned} \mathcal{C} ::= {} & \textbf{rd } \mathbb{ID} \mid \textbf{wr } \mathbb{ID} \mid \textbf{wrP } \mathbb{ID} \mid \textbf{cons } \mathbb{ID} \\ & \mid \textbf{if } (\mathcal{V} \cup \mathcal{C})^* \times (\mathcal{V} \cup \mathcal{C})^* \mid \textbf{not} \mid \textbf{op } \mathbb{O} \end{aligned}$$

The $\textbf{rd}(v)$ and $\textbf{wr}(v)$ commands respectively read and write variable v. The $\textbf{wrP}(v)$ command is a special case that writes a parameter variable v. A new struct instance of type ϑ can be constructed with the $\textbf{cons}(\vartheta)$ command. Conditional execution of either branch S_1 or S_2 is performed using the command $\textbf{if}(S_1, S_2)$. Finally, the **not** and $\textbf{op}(\mathit{op})$ commands respectively perform the negation and op operation.

Semantic commands are placed together with their operand values in a list, the initial configuration of this list is generated from a step's AST by the *interpretation function*:

Definition 4 (Interpretation function)**.** *Let* $v, v_1, \ldots, v_n \in \mathbb{ID}$ *be variables,* $e, e_1, \ldots, e_m \in \mathbb{E}$ *expressions,* $T \in \mathbb{T}$ *a syntactic type,* $lit \in \mathbb{L}$ *a literal,* $\vartheta \in \mathbb{ID}$ *a struct type,* $s \in \mathbb{S}$ *a statement,* $S, S_1, S_2 \in \mathbb{S}^*$ *lists of statements,* $\mathit{op} \in \mathbb{O} \setminus \{\texttt{\&\&}, \texttt{||}\}$ *an operator and let* $\ell \in \mathcal{L}$ *be a label. We define the* interpretation function $\llbracket \cdot \rrbracket_\ell : \mathbb{S}^* \cup \mathbb{E} \to (\mathcal{V} \cup \mathcal{C})^*$ *transforming a list of statements and expressions into a list of commands and values:*

$$\begin{aligned} \llbracket \boldsymbol{Op}(e_1, \texttt{\&\&}, e_2) \rrbracket_\ell &= \llbracket e_1 \rrbracket_\ell ; \textbf{if}(\llbracket e_2 \rrbracket_\ell, \mathit{false}) \\ \llbracket \boldsymbol{Op}(e_1, \texttt{||}, e_2) \rrbracket_\ell &= \llbracket e_1 \rrbracket_\ell ; \textbf{if}(\mathit{true}, \llbracket e_2 \rrbracket_\ell) \\ \llbracket \boldsymbol{Op}(e_1, \mathit{op}, e_2) \rrbracket_\ell &= \llbracket e_1 \rrbracket_\ell ; \llbracket e_2 \rrbracket_\ell ; \textbf{op}(\mathit{op}) \\ \llbracket \boldsymbol{Cons}(\vartheta, e_1; \ldots; e_m) \rrbracket_\ell &= \llbracket e_1 \rrbracket_\ell ; \ldots ; \llbracket e_m \rrbracket_\ell ; \textbf{cons}(\vartheta) \\ \llbracket \boldsymbol{Var}(v_1; \ldots; v_n) \rrbracket_\ell &= \ell ; \textbf{rd}(v_1); \ldots ; \textbf{rd}(v_n) \\ \llbracket \boldsymbol{Not}(e) \rrbracket_\ell &= \llbracket e \rrbracket_\ell ; \textbf{not} \\ \llbracket \boldsymbol{Lit}(lit) \rrbracket_\ell &= val_\ell(lit) \\ \llbracket \boldsymbol{If}(e, S_1, S_2) \rrbracket_\ell &= \llbracket e \rrbracket_\ell ; \textbf{if}(\llbracket S_1 \rrbracket_\ell, \llbracket S_2 \rrbracket_\ell) \\ \llbracket \boldsymbol{Declare}(T, v, e) \rrbracket_\ell &= \llbracket \boldsymbol{Assign}(v, e) \rrbracket_\ell \\ \llbracket \boldsymbol{Assign}(v_1; \ldots; v_n; v, e) \rrbracket_\ell &= \llbracket e \rrbracket_\ell ; \llbracket \boldsymbol{Var}(v_1; \ldots; v_n) \rrbracket_\ell ; \textbf{wr}(v) \\ \llbracket \varepsilon \rrbracket_\ell &= \varepsilon \\ \llbracket s; S \rrbracket_\ell &= \llbracket s \rrbracket_\ell ; \llbracket S \rrbracket_\ell \end{aligned}$$

Note how we apply short circuit evaluation to expressions with the `&&` and `||` operators to prevent side effects of the right-hand side by transforming the expressions into conditional statements.

Multiple instances of a struct definition may exist during execution, these are called the *struct instances*:

Definition 5 (Struct instance)**.** *A* struct instance *is a tuple* $\langle \vartheta, \chi, \xi \rangle$ *where:*

- $\vartheta \in \mathbb{ID}$ *is the* struct type*,*
- $\chi \in (\mathcal{V} \cup \mathcal{C})^*$ *is a* value and command list*,*
- $\xi : \mathcal{L} \times \mathbb{ID} \to \mathcal{V} \ \cup \{\bot\}$ *is a* value cache*,*

We define $\mathcal{S}$ *as the set of all possible struct instances. We denote an* empty cache *– in which all elements map to* $\bot$ *– with* $\xi_\bot$*.*

Each struct instance independently executes commands from its list. When it writes a value, it records a copy of that value in its value cache. Each type of structure has at least one instance, called the *null-instance*, which is labelled with the null-label ℓ^0_ϑ. The null-instance's parameters cannot be written to.

While each struct instance captures its local state, the global state is captured by the program execution *state*:

Definition 6 (State)**.** *A* state *is a tuple* $\langle sc, \sigma, \gamma, \iota, \delta \rangle$*, where:*

- $sc \in \mathcal{SC}^*$ *is a* schedule*,*
- $\sigma : \mathcal{L} \to \mathcal{S} \cup \{\bot\}$ *is a* struct environment*,*
- $\gamma : \mathcal{L} \times \mathbb{ID} \to 2^{\mathcal{V} \times \mathcal{L}}$*, is a* global memory *of parameter values written during the current step paired with their writer's label,*
- $\iota : \mathcal{L} \times \mathbb{ID} \to \mathcal{V}$*, is a* initial value function *of parameter values recording the value of each parameter before the start of the current step,*
- $\delta \in \mathbb{B}^*$ *is a* stability stack*.*

Let define $\mathcal{ST}$ *as the set of all possible states. We denote an* empty global memory *– in which all label-variable pairs map to the empty set – by* $\gamma_\emptyset$*.*

All struct instances are uniquely labelled and contained within the labelled struct environment σ. Communication between struct instances during a step occurs through the global memory γ, which keeps track of the written values and the instance responsible for writing each value. When a step is called, all instances initially agree on the parameter values and these values are recorded in the initial value function ι. The stability stack δ is used to determine stability of fixpoint iterations.

In the *initial state*, all instances are null-instances with an empty command list and value cache. Their parameters are initialised to their default values, recorded in the initial value function.

Definition 7 (Initial state)**.** *Let* $\mathcal{P}$ *be an AuDaLa program,* $\Theta \subseteq \mathbb{ID}$ *a corresponding set of struct types defined in* $\mathcal{P}$*, and* $sc_\mathcal{P} \in \mathcal{SC}^*$ *the schedule of* $\mathcal{P}$*. We define* $\sigma_\bot$ *to be a struct environment that maps to* $\bot$ *for all elements,* *and* $\iota_?$ *to be an arbitrary initial value function. We define the* initial struct environment *(*$\sigma^0_{\mathcal{P}}$*),* initial-initial value function *(*$\iota^0_{\mathcal{P}}$*) and* initial state *(*$P^0_{\mathcal{P}}$*) as follows:*

$$\begin{aligned}
\sigma^0_{\mathcal{P}} &\triangleq \sigma_\bot[\{\ell^0_\vartheta \mapsto \langle \vartheta, \varepsilon, \xi_\bot\rangle \mid \vartheta \in \Theta\}]\\
\iota^0_{\mathcal{P}} &\triangleq \iota_?[\{(\ell^0_\vartheta, v) \mapsto \mathit{defaultVal}(\mathit{TypeOf}(\vartheta, v)) \mid\\
&\qquad\qquad \vartheta \in \Theta, v \in \mathit{Par}(\vartheta)\}]\\
P^0_{\mathcal{P}} &\triangleq \langle sc_{\mathcal{P}}, \sigma^0_{\mathcal{P}}, \gamma_\emptyset, \iota^0_{\mathcal{P}}, \varepsilon\rangle
\end{aligned}$$

A command is only allowed to be executed if all commands before it that reference the same parameter have been executed. This is expressed in the *Refs* predicate:

Definition 8 (The *Refs* predicate)**.** *Let* $\chi \in (\mathcal{V} \cup \mathcal{C})^*$ *be a command and value list,* $\ell \in \mathcal{L}$ *a label,* $v \in \mathbb{ID}$ *a variable,* $val \in \{\ell\} \cup \mathcal{C}$ *a command or the value* ℓ *and let* $cmd \in \{\mathbf{rd}(v), \mathbf{wr}(v), \mathbf{wrP}(v)\}$ *be a read or write command referencing* v*. Then, we define the predicate* $\mathit{Refs} : (\mathcal{V} \cup \mathcal{C})^* \times \mathcal{L} \times \mathbb{ID} \to \mathbb{B}$ *as follows:*

$$\begin{aligned}
&\mathit{Refs}(\chi, \ell, v) \triangleq\\
&\qquad \chi = \chi_1; val; cmd; \chi_2 \vee\\
&\qquad (\chi = \chi_1; \mathbf{if}(\chi_2,\ \chi_3); \chi_4\\
&\qquad\quad \wedge \mathit{Refs}(\chi_2, \ell, v) \vee \mathit{Refs}(\chi_3, \ell, v))\\
&\qquad \textbf{\textit{for some}}\ \chi_1, \chi_2, \chi_3, \chi_4 \in (\mathcal{V} \cup \mathcal{C})^*
\end{aligned}$$

The $\mathit{Refs}(\chi, \ell, v)$ predicate checks if a read or write command is present in χ that references, i.e. potentially reads or writes, v of ℓ. When commands of the form $\mathbf{if}(\chi_1, \chi_2)$ are present in χ, *Refs* will recursively check both χ_1 and χ_2.

3.2. Derivation rules

Now that our notion of state and the initial state is complete, we are ready to define the transition relation $\Rightarrow$ between states. We do so by listing a set of derivation rules in the following paragraphs. We use $val \in \mathcal{V}$, $v \in \mathbb{ID}$, $\chi_1, \chi_2 \in (\mathcal{V} \cup \mathcal{C})^*$, $\ell \in \mathcal{L}$ and $op \in \mathbb{O}$ as variables, possibly with subscripts or superscripts.

Operators Computation on values is done via operators, for which two derivation rules related to the *not* operator and binary operators are defined. The **not** command simply negates its operand and the $\mathbf{op}(op)$ command applies the binary operator op to its two operands.

`Command Not:`

$$\frac{\sigma(\ell) = \langle \vartheta, \chi_1; val; \mathbf{not}; \chi_2, \xi\rangle}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta\rangle \Rightarrow\\ \quad \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; \neg val; \chi_2, \xi\rangle], \gamma, \iota, \delta\rangle\end{array}}$$

`Command Operator:`

$$\frac{\sigma(\ell) = \langle \vartheta, \chi_1; val_a; val_b; \mathbf{op}(op); \chi_2, \xi\rangle}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta\rangle \Rightarrow \langle sc, \sigma[\ell \mapsto\\ \quad \langle \vartheta, \chi_1; (val_a\ op\ val_b); \chi_2, \xi\rangle], \gamma, \iota, \delta\rangle\end{array}}$$

Reading A read command can retrieve a written value in three ways. It can read the value from the initial value function ι, the value cache ξ or from the global memory γ. Between each of the applicable options is chosen nondeterministically. Read commands to the same location are executed in program order, as per the *Refs* function.

`Command Read Initial:`

$$\frac{\begin{array}{c}\sigma(\ell) = \langle \vartheta, \chi_1; \ell'; \mathbf{rd}(v); \chi_2, \xi\rangle\\ \xi(\ell', v) = \bot\\ \neg\mathit{Refs}(\chi_1, \ell', v)\end{array}}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta\rangle \Rightarrow\\ \quad \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; \iota(\ell', v); \chi_2, \xi\rangle], \gamma, \iota, \delta\rangle\end{array}}$$

`Command Read Internal:`

$$\frac{\begin{array}{c}\sigma(\ell) = \langle \vartheta, \chi_1; \ell'; \mathbf{rd}(v); \chi_2, \xi\rangle\\ \xi(\ell', v) \neq \bot\\ \neg\mathit{Refs}(\chi_1, \ell', v)\end{array}}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta\rangle \Rightarrow\\ \quad \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; \xi(\ell', v); \chi_2, \xi\rangle], \gamma, \iota, \delta\rangle\end{array}}$$

`Command Read External:`

$$\frac{\begin{array}{c}\sigma(\ell) = \langle \vartheta, \chi_1; \ell'; \mathbf{rd}(v); \chi_2, \xi\rangle\\ (val, \ell^w) \in \gamma(\ell', v) \wedge \ell^w \neq \ell\\ \neg\mathit{Refs}(\chi_1, \ell', v)\end{array}}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta\rangle \Rightarrow\\ \quad \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; val; \chi_2, \xi\rangle], \gamma, \iota, \delta\rangle\end{array}}$$

Writing When writing, we identify three variants of the written variable: a local variable, a null-instance parameter, and a genuine instance parameter. Local variables are stored in the instance's value cache ξ. Writes to a null-instance parameter are skipped. A write to a genuine parameter is split in two steps: First, the old value must be retrieved by spawning a read operation. Then, the write completes once the old value is read, clearing the stability stack if the old and new value differ.

`Command Write Local Variable:`

$$\frac{\begin{array}{c}\sigma(\ell) = \langle \vartheta, \chi_1; val; \ell; \mathbf{wr}(v); \chi_2, \xi\rangle\\ v \notin \mathit{Par}(\vartheta)\\ \neg\mathit{Refs}(\chi_1, \ell, v)\end{array}}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta\rangle \Rightarrow\\ \quad \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; \chi_2, \xi[(\ell, v) \mapsto val]\rangle], \gamma, \iota, \delta\rangle\end{array}}$$

`Command Write Parameter Null-Skip:`

$$\frac{\begin{array}{c}\sigma(\ell) = \langle \vartheta, \chi_1; val; \ell'; \mathbf{wr}(v); \chi_2, \xi\rangle\\ \sigma(\ell') = \langle \vartheta', \chi', \xi'\rangle\\ v \in \mathit{Par}(\vartheta') \wedge \ell' \in \mathcal{L}^0\end{array}}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta\rangle \Rightarrow\\ \quad \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; \chi_2, \xi\rangle], \gamma, \iota, \delta\rangle\end{array}}$$

Command Write Parameter I:

$$\frac{\begin{array}{c}\sigma(\ell) = \langle \vartheta, \chi_1; val; \ell'; \mathbf{wr}(v); \chi_2, \xi \rangle \\ \sigma(\ell') = \langle \vartheta', \chi', \xi' \rangle \\ v \in Par(\vartheta') \wedge \ell' \notin \mathcal{L}^0\end{array}}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta \rangle \Rightarrow \langle sc, \\ \quad \sigma[\ell \mapsto \langle \vartheta, \chi_1; \ell'; \mathbf{rd}(v); val; \ell'; \mathbf{wrP}(v); \chi_2, \\ \quad\quad \xi \rangle], \gamma, \iota, \delta \rangle\end{array}}$$

Command Write Parameter II:

$$\frac{\begin{array}{c}\sigma(\ell) = \langle \vartheta, \chi_1; val_o; val_n; \ell'; \mathbf{wrP}(v); \chi_2, \xi \rangle \\ stable = (val_o = val_n) \\ \neg Refs(\chi_1, \ell', v)\end{array}}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta \rangle \Rightarrow \langle sc, \\ \quad \sigma[\ell \mapsto \langle \vartheta, \chi_1; \chi_2, \xi[(\ell', v) \mapsto val_n] \rangle], \\ \quad \gamma[(\ell', v) \mapsto \gamma(\ell', v) \cup \{(val_n, \ell)\}], \\ \quad \iota, \delta_1 \wedge stable; \ldots; \delta_{|\delta|} \wedge stable \rangle\end{array}}$$

Constructing Instantiated instances receive a fresh label and their parameters are populated atomically. Furthermore, creating instances clears the stability stack.

Command Constructor:

$$\frac{\begin{array}{c}\sigma(\ell) = \langle \vartheta, \chi_1; val_1; \ldots; val_n; \mathbf{cons}(\vartheta'); \chi_2, \xi \rangle \\ Par(\vartheta') = v_1; \ldots; v_n \\ \sigma(\ell') = \bot\end{array}}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta \rangle \Rightarrow \langle sc, \\ \quad \sigma[\{\ell' \mapsto \langle \vartheta', \varepsilon, \xi_\bot \rangle, \ell \mapsto \langle \vartheta, \chi_1; \ell'; \chi_2, \xi \rangle\}], \\ \quad \gamma, \iota[\{(\ell', v_i) \mapsto val_i \mid 1 \leq i \leq n\}], false^{|\delta|} \rangle\end{array}}$$

Control flow AuDaLa supports simple control flow through if statements. Depending on the condition, either one of its branches is taken and placed on the command list.

Command If:

$$\frac{\begin{array}{c}\sigma(\ell) = \langle \vartheta, \chi_1; val; \mathbf{if}(S_1, S_2); \chi_2, \xi \rangle \\ (val = true \wedge b = S_1) \vee (val = false \wedge b = S_2)\end{array}}{\begin{array}{l}\langle sc, \sigma, \gamma, \iota, \delta \rangle \Rightarrow \\ \quad \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; b; \chi_2, \xi \rangle], \gamma, \iota, \delta \rangle\end{array}}$$

Step-calls When only values remain in each instance's value and command list, a step is *done*. We define the predicate $Done(\sigma) = \forall \ell : \sigma(\ell) = \langle \vartheta, \chi, \xi \rangle \Rightarrow \chi \in \mathcal{V}^*$. And we denote the statements of step $F \in \mathbb{ID}$ of struct $\vartheta \in \Theta$ by $S^F_\vartheta \in \mathbb{S}^*$. When step F is called for ϑ, all ϑ instances' command lists are set to the interpretation of S^F_ϑ, the value cache is cleared and the initial value function is updated with the *final parameter values* of the last step. The final value of parameter (ℓ, v) is denoted with $fin(\ell, v)$, whose value is nondeterministically chosen from the set $W(\ell, v)$ containing all values each instance wrote last to the parameter (ℓ, v) during the last step. If no instance wrote to a parameter, it retains its initial value.

Call Step:

$$\frac{\begin{array}{c}Done(\sigma) \\ W(\ell, v) = \{\xi(\ell, v) \mid \sigma(\ell') = \langle \vartheta, \chi, \xi \rangle\} \setminus \{\bot\} \\ (fin(\ell, v) \in W(\ell, v) \vee \\ (W(\ell, v) = \emptyset \wedge fin(\ell, v) = \iota(\ell, v)))\end{array}}{\begin{array}{l}\langle \boldsymbol{Call}(\vartheta, F); sc, \sigma, \gamma, \iota, \delta \rangle \Rightarrow \langle sc, \\ \quad \sigma[\{\ell \mapsto \langle \vartheta, [\![S^F_\vartheta]\!]_\ell, \xi_\bot \rangle \mid \sigma(\ell) = \langle \vartheta, \chi_\ell, \xi_\ell \rangle\}], \\ \quad \gamma_\emptyset, \iota[(\ell, v) \mapsto fin(\ell, v) \mid (\ell, v) \in \mathcal{L} \times \mathbb{ID}], \delta \rangle\end{array}}$$

Fixpoint iteration A fixpoint $\boldsymbol{Fix}(sc)$ is initiated by pushing *true* on the stability stack and placing sc on the schedule list followed by $\boldsymbol{aFix}(sc)$. Once $\boldsymbol{aFix}(sc)$ is encountered again, the topmost value of the stability stack is checked: if *true*, the fixpoint is resolved, if *false*, another iteration is executed.

Fixpoint Initiate:

$$\frac{Done(\sigma)}{\begin{array}{l}\langle \boldsymbol{Fix}(sc_1); sc_2, \sigma, \gamma, \iota, \delta \rangle \Rightarrow \\ \quad \langle sc_1; \boldsymbol{aFix}(sc_1); sc_2, \sigma, \gamma, \iota, \delta; true \rangle\end{array}}$$

Fixpoint Iterate:

$$\frac{Done(\sigma)}{\begin{array}{l}\langle \boldsymbol{aFix}(sc_1); sc_2, \sigma, \gamma, \iota, \delta; false \rangle \Rightarrow \\ \quad \langle sc_1; \boldsymbol{aFix}(sc_1); sc_2, \sigma, \gamma, \iota, \delta; true \rangle\end{array}}$$

Fixpoint Resolve:

$$\frac{Done(\sigma)}{\begin{array}{l}\langle \boldsymbol{aFix}(sc_1); sc_2, \sigma, \gamma, \iota, \delta; true \rangle \Rightarrow \\ \quad \langle sc_2, \sigma, \gamma, \iota, \delta \rangle\end{array}}$$

Graph semantics Our definition of the transition relation $\Rightarrow$ is now complete. We combine $\Rightarrow$ with our earlier definitions of the state space to formally define AuDaLa's graphs semantics:

Definition 9 (AuDaLa's graph semantics). *Let $\mathcal{P}$ be an AuDaLa program. We define the graph semantics of $\mathcal{P}$ as a tuple $(\!|\mathcal{P}|\!) = \langle \mathcal{ST}, \Rightarrow, P^0_{\mathcal{P}} \rangle$, where $\mathcal{ST}$ is the set of states as defined in Definition 6, $\Rightarrow$ is the transition relation defined by the derivation rules as given above, and $P^0_{\mathcal{P}}$ is the initial state of $\mathcal{P}$ as defined in Definition 7.*

3.3. Discussion

We have now defined our operational semantics with a relaxed memory consistency model and incoherent memory. We have thus answered RQ 1. However, relaxed memory consistency makes communication between threads unreliable due to reorderings, as shown in the preliminaries via the MP litmus test (Figure 3). It would seem that AuDaLa therefore loses some programmability. However, we argue that communication *within* a step is already unreliable in AuDaLa because steps contain no loops and, therefore, instances cannot wait for communication to have happened. In addition, instances execute steps autonomously in no particular order. In contrast, communication *across* steps is reliable, even with relaxed memory consistency, as the previous step is guaranteed to have been completed before the next one begins. Therefore, we can benefit from the increased performance of relaxed memory consistency without sacrificing programmability. On the same note, we argue that the burden of maintaining coherence within steps can also be removed from AuDaLa's semantics, as long as all instances agree on all values at the start/end of a step. As is done through our initial value function. This allows instances, or sets of instances, to work on local copies of data and only at the end of a step communicate their results to the other instances, improving performance. This is similar to the "temporary view" of memory that OpenMP employs [54].

Our global memory is now modelled as a set containing values and their writer's label. Instances select the writes they read non-deterministically and can thus read a value even though they have already observed a value written by a later write from the same instance to the same parameter. To make the semantics more predictable for the programmer, while still maintaining incoherence, future work could employ a system similar to the "timemaps" of the *promising semantics* [34]. Then, the global memory would also contain timestamps and each instance would keep track of the latest timestamp they have observed from each parameter and instance. We have chosen not to include this in our semantics in favour of simplicity.

4. Axiomatic memory semantics

We wish to compare AuDaLa's MCM with that of NVIDIA's PTX ISA [53]. In recent years, PTX's MCM has been formally defined as an axiomatic model, through the work of Lustig et al. [45, 46]. Since AuDaLa's semantics are expressed in an operational fashion, we first transform the previously presented operational semantic model into an axiomatic MCM. Then, both models can be compared using a model finding tool such as the Alloy analyser [30]. We compare with PTX even though our compiler targets the CUDA programming language. It is safe to do so since the mapping from CUDA/C++ `std::atomic` operations to PTX instructions is 1-to-1 and given in [46].

We start this section by modelling AuDaLa candidate executions and subsequently define what axioms should hold for such an execution to be a legal AuDaLa execution. Next, we show how an AuDaLa pre-execution can be mapped to a PTX pre-execution and how a PTX execution witness can be mapped back into an AuDaLa execution witness. We conclude this section with a discussion of the model-finding results and discuss how compatible both models are.

4.1. AuDaLa candidate executions

Recall that in an axiomatic MCM candidate executions are modelled as graphs. Program entities and events form the nodes, while their relations and interactions form the edges. This set of edges can be divided into the pre-execution and execution witness components. This section presents our model of these components.

Execution graph nodes To model the nodes of AuDaLa execution graphs, we define the following sets: The set $\texttt{Loc}$ models the label-variable pairs (ℓ, v) that are read from or written to in the operational semantics. The set $\texttt{Inst}$ models the struct instances. Since we are only interested in creating a memory model, we only model read and write commands. We define the disjoint sets $\texttt{RdCmd}$ and $\texttt{WrCmd}$ containing the read and write commands, respectively. We define $\texttt{Cmd} \triangleq \texttt{RdCmd} \cup \texttt{WrCmd}$ as the set of all commands.

Pre-execution relations Within an AuDaLa program, commands are part of a list (χ). We model this via the *next command* relation $\texttt{nxt} \subseteq \texttt{Cmd} \times \texttt{Cmd}$. For example, the list $a; b; c$ is represented as $(a, b), (b, c) \in \texttt{nxt}$. Struct instances and their commands are related by the $\texttt{start} \subseteq \texttt{Inst} \times \texttt{Cmd}$ relation that relates the instance to the first command in its list. Each $\texttt{Cmd}$ writes or reads exactly one $\texttt{Loc}$, this relation is modelled as the $\texttt{loc} \subseteq \texttt{Cmd} \times \texttt{Loc}$ relation. When a command depends on the result of a read command, we model this via the $\texttt{dep} \subseteq \texttt{RdCmd} \times \texttt{Cmd}$ relation.

We introduce a few derived relations to allow for a compact presentation. The *program-order* relation $\texttt{po} \subseteq \texttt{Cmd} \times \texttt{Cmd}$ orders commands in program order and is defined as: $\texttt{po} \triangleq \texttt{nxt}^{+}$. The *instance* relation $\texttt{inst} \subseteq \texttt{Cmd} \times \texttt{Inst}$ relates commands to their instance. It is defined as $\texttt{inst} \triangleq (\texttt{start}; \texttt{nxt}^{*})^{-1}$. The *same location* relation $\texttt{same\_loc} \subseteq \texttt{Cmd} \times \texttt{Cmd}$ relates all commands that reference the same location. It is defined as $\texttt{same\_loc} \triangleq \texttt{loc}; \texttt{loc}^{-1}$. Similarly, for commands of the same instance, the *same instance* relation $\texttt{same\_inst} \subseteq \texttt{Cmd} \times \texttt{Cmd}$ is defined: $\texttt{same\_inst} \triangleq \texttt{inst}; \texttt{inst}^{-1}$. Finally, we define the *program order by location* relation $\texttt{po\_loc} \subseteq \texttt{Cmd} \times \texttt{Cmd}$ as the subset of $\texttt{po}$ that relates commands referencing the same location: $\texttt{po\_loc} \triangleq \texttt{po} \cap \texttt{same\_loc}$.

Not all pre-executions are considered *well-formed*, only those that satisfy the $\texttt{WELLFORMED\_PREEX}$ predicate are:

Definition 10 (The $\texttt{WELLFORMED\_PREEX}$ predicate)**.** *A pre-execution is* well-formed*, if all requirements on the pre-execution relations listed below are met:*

1. $\textbf{lone}(\texttt{nxt}) \wedge \textbf{lone}(\texttt{nxt}^{-1})$*, each command has at most one* $\texttt{nxt}$ *successor and predecessor.*
2. $\textbf{injective}(\texttt{start}, \texttt{Inst}, \texttt{Cmd})$*, each instance has exactly one* $\texttt{start}$ *successor and each command has at most one* $\texttt{start}$ *predecessor.*

3. **acyclic**($\texttt{nxt}$), *no circular command lists.*
4. $\texttt{Cmd} \setminus \texttt{Cmd}; \texttt{nxt} = \texttt{Inst}; \texttt{start}$, *the commands without* $\texttt{nxt}$ *predecessor have an incoming* $\texttt{start}$ *relation.*
5. $\texttt{dep} \subseteq \texttt{po}$, *dependencies follow program order.*
6. $\texttt{Cmd}; \texttt{loc} = \texttt{Loc}$, *all locations are referenced (constrains the state space).*

The conjunction of these predicates is defined as the **WELLFORMED_PREEX** *predicate.*

Pre-execution example Consider Figure 9. There, the same program is represented in three abstractions: as AuDaLa source code, semantic commands, and as a pre-execution graph. There are two `S1` instances that are labelled ℓ_1 and ℓ_2. Each instance has the other's label as its `other` parameter. Only read and write commands are used in the pre-execution, but dependencies carried through other commands are preserved. Note that the write to the other instance's `a` is dependent on the reads of its own `a` and `b`, but those are not dependent on each other. The semantics can therefore freely choose between reading `a` or `b` first.

Execution witness To model the candidate execution witness of a program we model which reads read from what writes and we model a coherence order between writes. The *reads-from* relation $\texttt{rf} \subseteq \texttt{WrCmd} \times \texttt{RdCmd}$ relates a **WrCmd** to its readers. And the *coherence* relation $\texttt{co} \subseteq \texttt{WrCmd} \times \texttt{WrCmd}$ relates a **WrCmd** to another **WrCmd** that overwrites its value in the instance's value cache ξ. Note that our **co** relation is somewhat unconventional compared to other MCMs. Because a write can only be overwritten in the cache, but not in the global memory, and the cache is local to an instance, **co** only relates writes of the same instance.

Not all execution witnesses are considered *well-formed*, only those that satisfy the **WELLFORMED_WITNESS** predicate are:

Definition 11 (The **WELLFORMED_WITNESS** predicate)**.** *A execution witness is* well-formed, *if all requirements on the execution relations listed below are met:*

1. $\texttt{rf} \cup \texttt{co} \subseteq \texttt{same\_loc}$, *reading or overwriting can only occur between commands sharing a location.*
2. **lone**($\texttt{rf}^{-1}$), *a read can only read from at most one write.*
3. $\texttt{co} \subseteq \texttt{same\_inst}$, *overwriting can only occur between commands executed by the same instance.*
4. $\forall l \in \texttt{Loc}, i \in \texttt{Inst} \mid \textbf{total}(\texttt{co}, \texttt{WrCmd} \cap l; \texttt{loc}^{-1} \cap i; \texttt{inst}^{-1})$, *for every pair of location and instance, all belonging writes are related by* **co** *in a total order.*

The conjunction of these predicates is defined as the **WELLFORMED_WITNESS** *predicate.*

4.2. Deriving AuDaLa's axioms

Before we can define the axioms of AuDaLa, we first need to introduce an important relation called the from-reads relation.

The fr relation The *from-reads* relation orders reads with respect to writes. We define a *base from-reads* relation and a *from-initial-reads* relation that together form the from-reads relation. Consider the example of Figure 10. The topmost write, **W_1**, is overwritten by **W_2** (as indicated by **co**), yet **R** reads from **W_1**. This must indicate that **R** is executed before **W_2**, otherwise it would have read from **W_2**. This ordering is called the $\texttt{fr}_{\texttt{base}}$ ordering, which is formally defined as $\texttt{fr}_{\texttt{base}} \triangleq \texttt{rfi}^{-1}; \texttt{co}^{+}$, where **rfi** is the *internal reads-from* relation defined as $\texttt{rfi} \triangleq \texttt{rf} \cap \texttt{same\_inst}$. We use **rfi** instead of **rf** because external reads do not need to follow coherence order in our semantics, as the value is picked non-deterministically from global memory.

We extend the $\texttt{fr}_{\texttt{base}}$ relation with the from-initial-reads relation $\texttt{fr}_{\texttt{init}}$. In our axiomatic model, a read that reads from no write is considered to read from the initial value function. Looking at the derivation rule **Command Read Initial**, a read is only allowed to read from the initial value function if that read's instance's value cache is empty. Therefore, a read that does not read from any write must occur before any writes from the same instance to the same location. We capture this ordering in the $\texttt{fr}_{\texttt{init}}$ relation defined as $\texttt{fr}_{\texttt{init}} \triangleq [\texttt{RdCmd} \setminus \texttt{WrCmd}; \texttt{rf}]; (\texttt{same\_inst} \cap \texttt{same\_loc}); [\texttt{WrCmd}]$. We define the complete from-reads relation $\texttt{fr} \subseteq \texttt{RdCmd} \times \texttt{WrCmd}$ as: $\texttt{fr} \triangleq \texttt{fr}_{\texttt{base}} \cup \texttt{fr}_{\texttt{init}}$

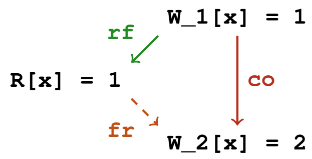
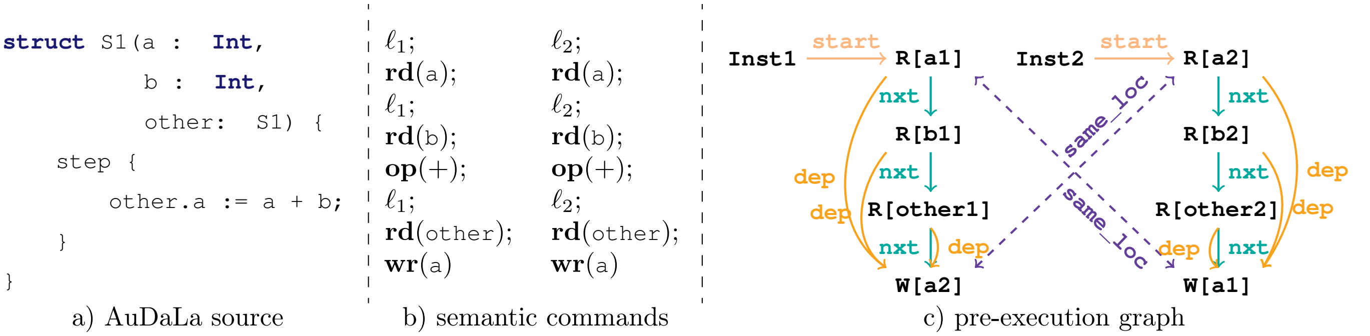


Figure 10: The classical example of the **fr** relation.

The CONSISTENCY axiom Most common MCMs have a handful of axioms, usually related to its methods of synchronisation (e.g. fences) or type of forbidden behaviour (e.g. thin-air) [3, 36, 46]. Since we have kept AuDaLa's MCM deliberately simple, we have only one axiom which we call **CONSISTENCY**.

In our axiomatic model, the **RdCmd** and **WrCmd** nodes correspond to the executions of the operational rules **Command Read Initial/Internal/External** and **Command Write Parameter II** for their respective instances. A derivation order is *legal* if the premise of each derivation rule is satisfied when following that order. Given an execution graph, we can establish in what order the derivation rules corresponding to the graph's nodes should be executed. We call this ordering the *semantic ordering* $\texttt{sem} \subseteq \texttt{Cmd} \times \texttt{Cmd}$. The **CONSISTENCY** axiom forbids those executions in which an inconsistency between



```
struct S1(a :  Int,
          b :  Int,
          other:  S1) {
    step {
        other.a := a + b;
    }
}
```

a) AuDaLa source

ℓ_1;	ℓ_2;
rd(a);	**rd**(a);
ℓ_1;	ℓ_2;
rd(b);	**rd**(b);
op(+);	**op**(+);
ℓ_1;	ℓ_2;
rd(other);	**rd**(other);
wr(a)	**wr**(a)

b) semantic commands

c) pre-execution graph

Figure 9: The same program represented in three abstractions. For the sake of presentation, we omit the **Loc** nodes and the `loc` relation. Instead, we denote the locations in square brackets. Note that we denote derived relations with dashed arrows and that we omit `po`, **inst**, and **same_inst** for clarity. The `po_loc` relation is empty.

the order of derivation steps exists, i.e. cyclic orderings:

$$\texttt{CONSISTENCY} \triangleq \textbf{acyclic}(\texttt{sem})$$

The semantic order is a partial ordering as the same candidate execution can correspond to multiple derivation sequences. We derive the relations that the semantic ordering necessarily needs to follow:

Theorem 1 (Necessity of `po_loc`, **rf**, **co**, **fr** and **dep** in the semantic ordering)**.** *The semantic ordering must necessarily follow the program order by location, reads-from, coherence, from-reads and dependency relations in order for the corresponding derivation orders to be* legal*. Formally:* $\texttt{po\_loc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr} \cup \texttt{dep} \subseteq \texttt{sem}$

Proof sketch. In the operational semantics, a read or write derivation step is only allowed to be taken if there are no other read or write commands preceding in program order that share the same location, as is enforced by the *Refs* predicate. Therefore, we require $\texttt{po\_loc} \subseteq \texttt{sem}$. Whether a read reads from the global memory (γ) or value cache (ξ), the read value is always generated from some write command executed before; hence we require $\texttt{rf} \subseteq \texttt{sem}$. A write can only be overwritten by a write executed after itself; hence, we require $\texttt{co} \subseteq \texttt{sem}$. A read executes before any writes that overwrite its read value; hence, we require $\texttt{fr}_{\texttt{base}} \subseteq \texttt{sem}$. A read of the initial value function (ι) occurs before any local write to the same location; hence, we require $\texttt{fr}_{\texttt{init}} \subseteq \texttt{sem}$, and thus $\texttt{fr} \subseteq \texttt{sem}$. Finally, if there is a dependency between commands, the command on which the other is dependent should execute before the dependent command in the semantic ordering; hence, we require $\texttt{dep} \subseteq \texttt{sem}$. We combine each of the listed requirements, giving: $\texttt{po\_loc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr} \cup \texttt{dep} \subseteq \texttt{sem}$. □

We have diligently studied the orderings implied by the derivation rules and are confident that the orderings listed in Theorem 1 are not only necessary, but also sufficient. However, as of yet, we do not have a formal proof and their sufficiency therefore remains conjecture:

Conjecture 1 (Sufficiency of `po_loc`, **rf**, **co**, **fr** and **dep** in the semantic ordering)**.** *It is sufficient for the semantic ordering to follow the program order by location, reads-from, coherence, from-reads and dependency relations in order for the corresponding derivation orders to be legal. Formally:* $\texttt{po\_loc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr} \cup \texttt{dep} \supseteq \texttt{sem}$

In accordance with Theorem 1 and Conjecture 1, we define:

$$\texttt{sem} \triangleq \texttt{po\_loc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr} \cup \texttt{dep}$$

Legal execution predicate Now that **sem**, and thereby **CONSISTENCY**, is defined, we are ready to define the predicate for a *legal AuDaLa execution*:

$$\texttt{LEGAL\_EXEC} \triangleq \texttt{WELLFORMED\_WITNESS} \land \texttt{CONSISTENCY}$$

Given a wellformed pre-execution, if the **LEGAL_EXEC** predicate holds a derivation order in the operational semantics exists such that the outcome (what values are read and stored) is the same for both the axiomatic and operational model.

Alternative legal execution predicate In the interpretation function defined during our discussion of the operational semantics, no optimisations occur. Therefore, any dependencies present in the AuDaLa source code will also be present in the semantic command list. An optimising compiler, however, might perform some optimisations and break some dependencies. As discussed in Section 2.5, specifying which dependencies are allowed to be broken and which are not is a difficult problem. Since PTX is an optimised language, it suffers from the same problems. Until consensus is reached on how to treat dependencies in an MCM, the PTX memory model chooses to not respect any dependencies at all, except for specifying that reads can not read from writes that are (transitively) dependent on them. However, some dependencies are essential to maintain program semantics when run in parallel with another program. Therefore, the PTX MCM still suffers from thin-air-executions.

To still be able to compare both memory models modulo dependencies, we provide an alternative, weaker, legal execution predicate called **LEGAL_EXEC***. It uses a weakened consistency axiom called **CONSISTENCY*** and introduces a new axiom called **NO_THIN_AIR**:

$$\begin{aligned}
\texttt{CONSISTENCY}^\star &\triangleq \text{acyclic}(\texttt{po\_loc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr}) \\
\texttt{NO\_THIN\_AIR} &\triangleq \text{acyclic}(\texttt{rf} \cup \texttt{dep}) \\
\texttt{LEGAL\_EXEC}^\star &\triangleq \texttt{WELLFORMED\_WITNESS} \land \\
&\quad\ \texttt{CONSISTENCY}^\star \land \\
&\quad\ \texttt{NO\_THIN\_AIR}
\end{aligned}$$

Our **NO_THIN_AIR** axiom is identical to PTX's no-thin-air axiom and prevents reads from reading writes that are (transitively) dependent on them. The **CONSISTENCY*** axiom is very similar to **CONSISTENCY**, except we exclude **dep** from **sem**.

Execution example Consider Figure 11. There, two instances execute two different programs represented as AuDaLa source code. Assume that both instances communicate through a shared reference to some **gm** struct with integer parameters **x** and **y**, both initially set to **0**. For brevity, we omit the reads of the **gm** reference so that the expression **gm.y** is transformed into a single read event in the execution graph. When using the **LEGAL_EXEC** predicate, two results are allowed: **r1 = 0, r2 = 0** (not shown) or **r1 = 42, r2 = 0** (depicted in Figure 11b). The result **r1 = 42, r2 = 42** (depicted in Figure 11c) is illegal for **LEGAL_EXEC** but legal for **LEGAL_EXEC***. It contains a **sem** cycle violating **CONSISTENCY**, but not **CONSISTENCY***. One could argue that **LEGAL_EXEC*** correctly recognises this execution as legal, arguing that a valid compiler optimisation would be to remove the store of **r2** to **gm.x**, as it will be overwritten immediately with **42** anyway. However, the same execution graph can correspond to a program in which the store of **r2** to **gm.x** can absolutely not be removed. For example, if its address depends on the load of **gm.y** the compiler would not be able to recognise that **gm.x** will be overwritten immediately and the ordering should be maintained. Thus, the execution would be considered an OOTA execution.

4.3. Mapping to and from PTX

Now that we have defined AuDaLa's axiomatic MCM, we can compare it with PTX's axiomatic MCM by mapping AuDaLa pre-executions onto PTX pre-executions and mapping PTX execution witnesses onto AuDaLa execution witnesses and checking their legality. This method follows standard practice for this type of proof [8, 46, 71].

We use the formal PTX MCM provided in [46]. We do not discuss all of its details here. Instead, we introduce its definitions as needed in the following section.

Mapping pre-executions The nodes of an AuDaLa execution graph are formed by the **Cmd**, **Loc** and **Inst** sets. In the PTX model, these correspond to the **Op**, **Addr** and **Thread** sets respectively. We further divide **Cmd** into **RdCmd** and **WrCmd**, which PTX does in a similar manner through the sets **RdOp** and **WrOp**. Unlike AuDaLa, PTX divides **RdOp** and **WrOp** further by their memory order semantics: PTX defines acquire-reads as **RdAcq**, relaxed-reads as **RdRlx**, release-writes as **WrRel** and relaxed-writes as **WrRlx**. This means that we have to choose the memory semantics to which we map AuDaLa's operations. Additionally, PTX operations have a scope attribute, which we fix to a single Device scope.

An AuDaLa pre-execution is formed by the **nxt**, **start**, **loc** and **dep** relations. Their PTX counterparts are denoted by $\texttt{nxt}_{ptx} \subseteq \texttt{Op} \times \texttt{Op}$, $\texttt{start}_{ptx} \subseteq \texttt{Thread} \times \texttt{Op}$, $\texttt{loc}_{ptx} \subseteq \texttt{Op} \times \texttt{Addr}$ and $\texttt{dep}_{ptx} \subseteq \texttt{RdOp} \times \texttt{Op}$ respectively. We establish a mapping between the nodes of an AuDaLa pre-execution and a PTX pre-execution via the relations $\texttt{cmd\_map} \subseteq \texttt{Cmd} \times \texttt{Op}$ which relates commands to operations, $\texttt{loc\_map} \subseteq \texttt{Loc} \times \texttt{Addr}$ which relates locations to addresses and $\texttt{inst\_map} \subseteq \texttt{Inst} \times \texttt{Thread}$ which relates instances to threads. We now use these relations to map an AuDaLa pre-execution to a PTX pre-execution. We do this by defining the **MAP_PREEX** predicate:

Definition 12 (The **MAP_PREEX** predicate)**.** *A pre-execution is* correctly mapped *if all requirements on the mapping relations listed below are met:*

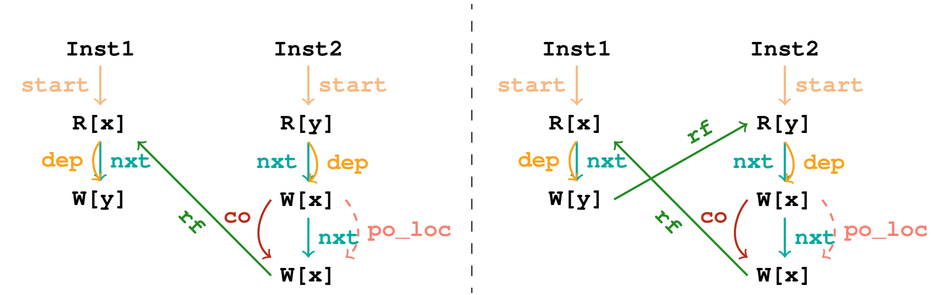
- bijective(**cmd_map**, **Cmd**, **Op**)*, every* ***Op*** *is derived from exactly one* ***Cmd****.*
- bijective(**loc_map**, **Loc**, **Addr**)*, every* ***Addr*** *is derived from exactly one* ***Loc****.*
- bijective(**inst_map**, **Inst**, **Thread**)*, every* ***Thread*** *is derived from exactly one* ***Cmd****.*
- $\texttt{loc}_{ptx} = \texttt{cmd\_map}^{-1}; \texttt{loc}; \texttt{loc\_map}$*, we map* **loc** *onto* $\texttt{loc}_{ptx}$*.*
- $\texttt{nxt}_{ptx} = \texttt{cmd\_map}^{-1}; \texttt{nxt}; \texttt{cmd\_map}$*, we map* **nxt** *onto* $\texttt{nxt}_{ptx}$*.*
- $\texttt{start}_{ptx} = \texttt{inst\_map}^{-1}; \texttt{start}; \texttt{cmd\_map}$*, we map* **start** *onto* $\texttt{start}_{ptx}$*.*
- $\texttt{dep}_{ptx} = \texttt{cmd\_map}^{-1}; \texttt{dep}; \texttt{cmd\_map}$*, we map* **dep** *onto* $\texttt{dep}_{ptx}$*.*
- *Finally, we decide which memory order semantics to use via either:*
  - $\texttt{cmd\_map} \subseteq \texttt{RdCmd} \times \texttt{RdAcq} \cup \texttt{WrCmd} \times \texttt{WrRel}$*, i.e. using Release/Acquire semantics. Or,*
  - $\texttt{cmd\_map} \subseteq \texttt{RdCmd} \times \texttt{RdRlx} \cup \texttt{WrCmd} \times \texttt{WrRlx}$*, i.e. using Relaxed semantics.*

The conjunction of these predicates is defined as the ***MAP_PREEX*** *predicate.*

Inst1	Inst2

Int r1 := gm.x;
gm.y := r1;
	Int r2 := gm.y;
	gm.x := r2;
	gm.x := 42;

a) An example AuDaLa program.



b) A legal execution.

c) A controversial execution.

Figure 11: A simple test program with two corresponding execution graphs. Figure 11b shows a legal execution and Figure 11c shows a controversial execution that could also be interpreted as an OOTA execution. We again omit **Loc**, **loc**, **po**, **inst**, **same_inst**, and **same_loc**. The **fr** relation is empty.

```
pred bijective[R: univ->univ,
                S1: univ, S2: univ] {
    (all s : S1 | (one s.R) and (s.R in S2))
    (all s : S2 | (one s.~R) and (s.~R in S1))
   )
}
```

Figure 12: An example showing the **bijective**(R, S_1, S_2) predicate (Section 2.2) defined in Alloy.

Mapping witnesses For execution witnesses, we reverse the direction of the mapping: we map a PTX execution witness back to an AuDaLa execution witness. Recall that an AuDaLa execution witness is formed by the **rf** and **co** relations. Their PTX counterparts are denoted by $\texttt{rf}_{ptx} \subseteq \texttt{WrOp} \times \texttt{RdOp}$ and $\texttt{co}_{ptx} \subseteq \texttt{WrOp} \times \texttt{WrOp}$ respectively. We also make use of the **same_thread** relation, which is PTX's analogue to AuDaLa's **same_inst** relation. A correctly mapped execution witness satisfies the **MAP_WITNESS** predicate:

Definition 13 (The **MAP_WITNESS** predicate)**.** *A PTX execution witness is* correctly mapped *if all requirements on the execution witness relations listed below are met:*

- $\texttt{rf} = \texttt{cmd\_map}; \texttt{rf}_{ptx}; \texttt{cmd\_map}^{-1}$, *we map* $\texttt{rf}_{ptx}$ *onto* **rf***.*
- $\texttt{co} = \texttt{cmd\_map}; (\texttt{co}_{ptx}{}^{+} \cap \texttt{same\_thread}); \texttt{cmd\_map}^{-1}$, *we map the subset of* $\texttt{co}_{ptx}{}^{+}$ *that relates writes by the same thread onto* **co***.*

The conjunction of these predicates is defined as the **MAP_WITNESS** *predicate.*

4.4. Formalising in Alloy

We formalise our model in the Alloy analyser [30]. Alloy is a relational modeling tool capable of reasoning over graphs. Therefore, Alloy is often used to analyse MCMs [25,45–47,70,71]. Alloy's syntax is very similar to the relational algebra syntax we have been using in our definitions. Figure 12 shows an example of how definitions can be translated into Alloy syntax in a straightforward manner. Each node in an Alloy graph has a type *signature*. A signature defines what type of outgoing edges each node has. Signatures can also be specialisations of other signatures through inheritance. Figure 13 shows how the **Cmd**, **RdCmd**, and **WrCmd** sets are defined in Alloy as well as their non-derived relations. In Appendix A as well as in [40] our complete Alloy formalisation can be found.

```
abstract sig Command {
  nxt : lone Command,
  loc : one Location,
  cmd_map : one hwEvent
}
sig ReadCommand extends Command {
  dep : set Command
}
sig WriteCommand extends Command {
  rf : set ReadCommand,
  co : set WriteCommand
}
```

Figure 13: The defined signatures representing the **Cmd**, **RdCmd** and **WrCmd** sets and their non-derived relations with their multiplicity.

4.5. Empirical correctness results

We wish to assert that when an AuDaLa program is compiled to a PTX program according to the mapping defined by **MAP_PREEX**, any *legal PTX execution* is also legal when interpreted as an AuDaLa execution. The PTX axiomatic MCM defines the predicate **PTX_MM**, which holds if and only if the PTX execution is legal. We use this predicate to define a theorem of correctness for our mapping:

Theorem 2 (Correctness of **MAP_PREEX**)**.** *Given a valid AuDaLa pre-execution* p_{AuDaLa}*, suppose we map* p_{AuDaLa} *onto a PTX pre-execution* p_{PTX}*, and suppose* w_{PTX} *is a legal execution witness of* p_{PTX}*. If we interpret* w_{PTX} *as an AuDaLa execution witness* w_{AuDaLa}*, then* w_{AuDaLa} *is a legal execution witness of* p_{AuDaLa}*. Equivalently, in predicate* *logic:*

$$\texttt{WELLFORMED\_PREEX} \land \texttt{MAP\_PREEX} \land \\ \texttt{PTX\_MM} \land \texttt{MAP\_WITNESS} \Rightarrow \texttt{LEGAL\_EXEC}$$

We test Theorem 2 in Alloy 4.2 [30] using the included MiniSat solver [19] and report the solving time returned by Alloy on a system with an AMD Ryzen 5 5600x CPU and 32GB RAM. We do this for small (4-8) pre-execution sizes. Due to state space explosion, larger pre-executions become intractable. However, some of the most important litmus tests for GPUs can be defined using six or fewer events [61]. We test three configurations: **RelAcq**, **Rlx** and **Rlx***. The first two configurations test both options of the **`MAP_PREEX`** predicate, i.e. with Release/Acquire semantics and with Relaxed semantics. The third configuration, **Rlx***, uses the Relaxed version of **`MAP_PREEX`** but uses **`LEGAL_EXEC`*** as conclusion of the implication of Theorem 2. Therefore, **Rlx*** tests a weaker version of the axiomatic model in which not all dependencies need to be respected. The results of our empirical testing are presented in Table 1, in which a timeout (TO) of 48 hours is used. The table demonstrates that it is safe to compile to the Release/Acquire memory order. For the Relaxed memory order, counterexamples were found that we will discuss in the next paragraph. If we model modulo dependencies (**Rlx***), no counterexamples were found.

Counterexamples As can be seen from Table 1, Theorem 2 does not hold for the **Rlx** configuration. Two counterexamples are found for a size of 5 events. The larger counterexamples are variations of these smaller counterexamples. One counterexample is already discussed in Section 4.2 and shown in Figure 11c. The other counterexample is similar and is shown in Figure 14. There, a **`sem`** cycle is formed because in **`Inst2`** the second read connects the first read through a **`dep`**; **`fr`**/`po_loc` edge with the write. This edge might not be preserved by the compiler if, for example, the second read is unused and, therefore, removed as dead code. The **Rlx** configuration does not take into account these optimisations and, therefore, disallows this execution, while PTX, accounting for possible optimisations, allows it. Possibly, the edge cannot be broken by any compiler optimisation and forms a causal cycle, thus resulting in OOTA behaviour.

4.6. Discussion

We have demonstrated that the PTX MCM is susceptible to the OOTA problem. This makes it difficult to answer RQ 2 conclusively for the Relaxed memory order. When dependencies are not completely modelled (**Rlx***), the Relaxed memory order is shown to be safe to compile AuDaLa's memory operations to. This does mean, however, that we should *trust* the compiler not to break any dependencies that are essential for an ordering assumed to hold by the programmer essential for program correctness. We use the word "trust", because the current PTX MCM does not clearly define which dependencies are inviolable. To be completely certain that no thread will observe the compiler breaking a dependency, and thus violating our operational semantics in which no dependencies can be broken, the Release/Acquire memory order can also be used as an alternative. In Section 6 we will discuss what kind of performance penalty this incurs.

Size	RelAcq		Rlx		Rlx*	
4	✓	(1s)	✓	(1s)	✓	(1s)
5	✓	(15s)	✗	(2s)	✓	(2s)
6	✓	(6.5m)	✗	(7s)	✓	(32s)
7	✓	(8h)	✗	(1s)	✓	(21.5m)
8	?	(TO)	✗	(35s)	✓	(30h)

Table 1: The verdict and solve time for Theorem 2 in three configurations: Release/Acquire semantics, Relaxed semantics, and Relaxed semantics with the **`LEGAL_EXEC`*** predicate.

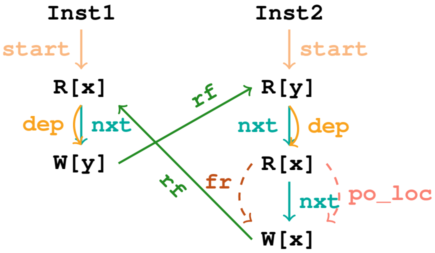


Figure 14: The other counterexample found for **Rlx**.

5. The AuDaLaC compiler

This section presents our experimental AuDaLa compiler called *AuDaLaC*. The full source code of AuDaLaC is provided in [39]. We start this section by providing an overview of the compilation process and the components involved. We continue by discussing two major design choices: how we translate an AuDaLa schedule into a (series of) kernel launch(es), and how we synchronise between fixpoint iterations and communicate their stability to all threads. These choices correspond to what we call *schedule strategies* and *fixpoint strategies*, respectively. We refer to both types of strategy when we speak about the *compilation strategy*. Then, we discuss some erroneous behaviour that can occur when creating instances with relaxed memory semantics, and how we prevent it. We conclude this section by detailing our approach to automated data race detection, which we use to replace unnecessary strong-memory operations with weak-memory operations.

5.1. Overview

AuDaLaC is written in the Rust programming language. The *frontend* parses AuDaLa source code files (`.adl`) in an AST and performs some validation and analysis on it. Then, the *backend* translates the AST into CUDA source code files (`.cu`). A number of components form AuDaLa's runtime and are compiled separately. Together with the runtime components, the resulting CUDA files are compiled with NVCC into an executable file with embedded PTX code. When executed, it invokes the NVIDIA driver that just-in-time compiles the embedded PTX into native SASS code. The complete compilation process is visualised in Figure 15.

Frontend AuDaLaC uses the LALRPOP parser generator [49] to parse AuDaLa files into an AST. We use the grammar as given by Franken et al. in [21]. There are a few differences, however. AuDaLaC does not support string values or operations. In addition, where Franken et al. did not specify operator precedence, AuDaLaC uses the same operator precedence as in C++.

After parsing the input, the resulting AST is validated. Basic rules like unique names and variable scoping are enforced at compile time. Variable shadowing is also not allowed. For a complete list of error codes and their causes, we refer to Appendix B.

Initialisation files Instead of explicitly instantiating each struct instance in some initialisation step, we use *initialisation files*. These contain all initial struct instances and their parameter values. This allows us to separate the input data from the algorithm. Each AuDaLa program executable takes an initialisation file (`.init`) as argument. This file is parsed by the `InitFile` runtime component that is compiled with each AuDaLa executable.

The `Struct` runtime component For each struct of the program an object is created that inherits from the `Struct` runtime component. This object holds all parameter data and is responsible for tracking the number of active and created struct instances (see Section 5.4). In addition, it checks the initialisation file for compatibility with the compiled program and migrates its data onto the GPU.

Data layout The parameter data is stored in the global memory of the GPU. Each parameter is stored as a structure of arrays, as opposed to an array of structures, such that coalesced memory accesses are enabled. Struct instance labels are represented as indices of these parameter arrays. The parameter values are stored as atomic objects with a size of 32 bits. Our operational semantics uses numbers of unbounded size, therefore, the use of larger numbers violates the semantics.

Launch dimensions AuDaLa programs typically contain short steps with relatively light computations. In addition, all parameter data must be retrieved from global memory. Therefore, most of the time is usually spent waiting on memory latencies. Latency hiding can mitigate the detrimental effects of latency. Higher occupancy can increase the effectiveness of latency hiding. AuDaLaC therefore uses CUDA's built-in `cudaOccupancyMaxPotentialBlockSize` function to dynamically calculate launch dimensions that achieve maximum occupancy for our kernels.

Backend Since AuDaLa expressions and statements are very similar to C++ expressions and statements, their compilation is straightforward. However, special care must be taken with parameter assignments and constructor expressions. For parameter assignments, the old value must first be retrieved and compared with the new value. If they differ, the fixpoint stability stack needs to be cleared. This is done using the `SetParam` function. In addition, `SetParam` only writes the parameter if its owner is not a null instance. The compilation of constructor expressions is discussed in Section 5.4.

5.2. Schedule strategies

We have implemented three approaches to compiling schedules in AuDaLaC. We call these *schedule strategies*. We detail each strategy in this section.

On-host schedule The most straightforward schedule strategy is the *on-host schedule*. In the on-host strategy, all steps are compiled as separate kernels. The host launches these kernels with one thread per instance. After a fixpoint iteration, the host and device must synchronise: the host needs to decide, based on the stability stack, whether to continue with the rest of the schedule or to run another fixpoint iteration. A benefit of this strategy is that it is relatively simple to implement and that the grid size can scale dynamically at runtime. A major drawback of this strategy is the cost of synchronising between host and device and the overhead costs of repeated kernel launches. This performance hit is especially severe for fixpoints with many short iterations, where the synchronisation time and launch latency are large compared to the kernel execution time.

Graph schedule We attempt to remedy the drawbacks of the on-host strategy by using CUDA graphs. Recall that a sequence of kernel launches can be represented as a CUDA graph and that the overhead of launching a CUDA graph is much lower than a regular kernel launch. In addition, CUDA graphs can be (re)launched from within a kernel, avoiding having to synchronise with the host. The *graph schedule* strategy first transforms the schedule into a set of CUDA graphs. Then it uploads the graphs to the GPU and launches the first graph in the schedule. Each graph launches the next graph in the schedule directly from the GPU as needed. After a

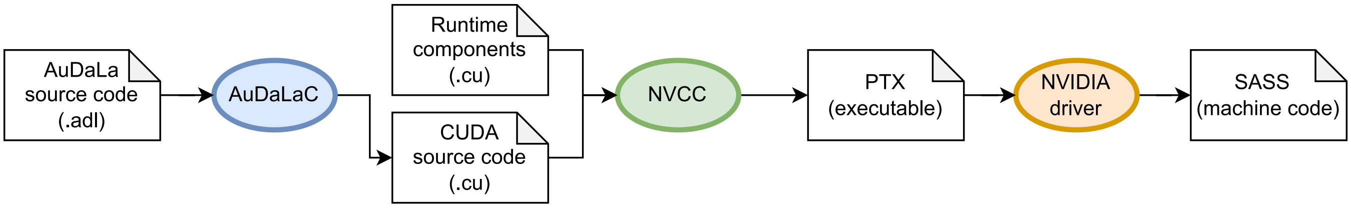


Figure 15: A visualisation of the compilation process from AuDaLa source code to SASS.

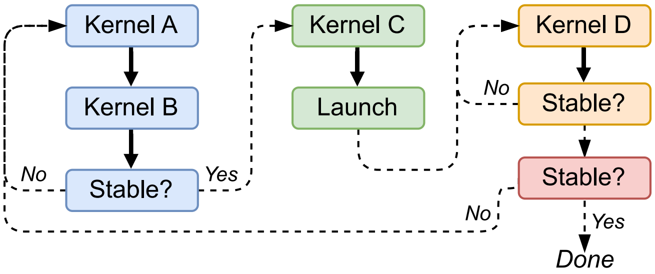


Figure 16: A schematic representation of the CUDA graphs generated by the graph schedule strategy for the schedule `Fix(Fix(A < B) < C < Fix(D))`. Each block represents a kernel, graph edges are depicted as solid arrows, in-kernel graph launches are depicted as dashed arrows, and each color represents a CUDA graph.

fixpoint iteration, the *relaunch kernel* reads the stability stack and determines which graph to launch next. See Figure 16 for an example of the graphs representing the schedule `Fix(Fix(A < B) < C < Fix(D))`, note that an additional launch kernel is needed for an in-kernel launch of the next fixpoint graph after step C.

In-kernel schedule The final schedule strategy is the *in-kernel* strategy. Instead of defining a kernel per step, the in-kernel strategy compiles the entire schedule into a single kernel. The benefit of this strategy is that it eliminates launch overhead altogether. A drawback of this strategy is that it requires a grid-wide synchronisation between steps to ensure that all instances execute the schedule in lockstep. This decreases the maximum grid size, as all blocks participating in a grid-wide synchronisation need to be resident on the device simultaneously. Therefore, a single thread sometimes needs to execute multiple instances, which are then divided over the threads in such a way that accesses are still coalesced. Another drawback is the increase in register pressure as a result of the larger kernel. Both of these drawbacks result in less utilisation of the available parallelism on the GPU.

5.3. Fixpoint strategies

Fixpoints are compiled as do-while loops with loop guards that read the Top Of the Stability Stack (TOSS). The synchronisation of the TOSS between threads can be compiled in a number of ways. We call these approaches *fixpoint strategies* and have detailed each type of strategy in this section.

Simple strategy With the *simple* strategy, each iteration starts with *resetting* the TOSS. As the iteration is executed, the TOSS can be *cleared*. Once the iteration is finished, the TOSS is *read*, determining whether another iteration is needed or if the schedule continues. To avoid race conditions, synchronisation must be performed between these actions. This means that the grid must be synchronised three times per fixpoint iteration for the in-kernel schedule strategy. However, with the on-host and graph schedule strategies, two synchronisation points are already provided by the launching of a kernel and its termination. The third synchronisation is not needed as both the host and the relaunch kernel execute on a single thread.

Rotating strategy The simple fixpoint strategy prevents race conditions by separating accesses to the stability stack in time, via synchronisation. The *rotating* fixpoint strategy prevents race conditions by separating accesses in space, over three separate stability stacks. Exactly one action is performed on each stack, after synchronisation the actions rotate: the stack that was potentially cleared is read, the stack that was read is reset, and the stack that was reset will potentially be cleared. This requires only one synchronisation per iteration, compared to three for the simple strategy. The pseudocode for the rotating strategy on the in-kernel schedule is given in Figure 17.

Only the on-host and in-kernel schedule strategies have a rotating fixpoint strategy defined. The relaunch kernel of the graph schedule strategy executes both read and reset actions on a single thread, already from the device. It provides negligible, if any, performance gain to do so in parallel on separate stacks. The on-host strategy clears the stability stack via a `memset` operation from the host to the device. This is an expensive operation, and, as shown in Section 6, it pays off to perform this in parallel on a separate CUDA stream.

5.4. Constructing new instances

A constructor expression is compiled into a call of the `create_instance` function, implemented by each `Struct` object. This function atomically increments the number of created instances, creating a fresh label. Then it writes the corresponding parameter data passed as arguments

```
$clear(x) = x \mod 3$
$reset(x) = x + 1 \mod 3$
$read(x) = x + 2 \mod 3$
$\delta \leftarrow [[true, true, true], \ldots]$;
procedure Fix($S$, $lvl$)
    $i \leftarrow 0$; // Each thread stores $i$ in a register
    do
        if grid.thread_rank() == 0 then
            $\delta[lvl][reset(i)] = true$;
        end if
        $S()$; // Potentially clears $\delta[lvl][clear(i)]$
        $i \leftarrow i + 1$;
        grid.sync();
    while $\neg\delta[lvl][read(i)]$;
end procedure
```

Figure 17: Pseudocode for the in-kernel schedule's rotating fixpoint strategy. The stability stack is represented by δ and the nesting depth of the fixpoint by lvl.

and returns the label.

Preventing uninitialised data reads When the parameters are initialised with relaxed writes, it can cause a read of unitialised data. Consider the following scenario: a thread T_c creates a new instance with fresh label ℓ. When T_c communicates ℓ to another thread T_r, the parameter writes performed by T_c in the `create_instance` function might not have become visible to T_r. If T_r now reads a parameter of the object labelled with ℓ, it could read uninitialised data. To prevent this, AuDaLaC performs a data flow analysis of fresh labels. An assignment statement that (possibly) stores a fresh label in a parameter is compiled as a write with release semantics. A read expression of a parameter (possibly) holding a fresh label is compiled as a read with acquire semantics. Now, whenever two threads communicate a fresh label, they synchronise and all parameter writes are guaranteed to be visible to the receiving thread.

Updating instance counters Newly created instances do not execute the step in which they were created. They only execute the next steps. This creates another synchronisation problem, as each thread must read the number of created instances at the end of the last step before the next step can start increasing the number of created instances again. We solve this differently for each schedule strategy. For the on-host schedule strategy, we synchronise the host with the device after each step that potentially creates instances and update the counters from the host. For the in-kernel schedule strategy, we prevent having to do two synchronisations between steps by employing a strategy similar to the rotating stability stack: we use two alternating active instance counters, one is being read while the other increases together with the created instance counter. Finally, for the graph schedule strategy, we update the counters in a kernel inserted into the graph after each step that potentially creates new instances.

5.5. Automatic race detection

Recall that the most performant type of memory operations in PTX are weak memory operations. However, weak memory operations are neither atomic nor coherent and, therefore, can only be safely used for data-race-free addresses. A data-race occurs if two memory operations, performed by different threads, access the same address simultaneously and at least one of the operations is a write. Because of AuDaLa's use of a schedule, we at compile time know exactly which parameters, and therefore addresses, are (potentially) being accessed simultaneously. In AuDaLa, instances can only write their own parameters or the parameters of instances they have a reference to. Therefore, it is impossible for two instances to access the same instance's parameter without at least one of the instances accessing that parameter via a reference. So, we say that a parameter is *racing* during a step if that parameter is both being written, either through an assignment or via a constructor expression, and accessed via a reference during that step. As an optimisation, we compile accesses to non-racing parameters into weak memory operations. We call this Optimised Non-Racing Parameters (ONRP). In addition, stores of fresh labels (see Section 5.4) to non-racing parameters do not need release semantics and can also be compiled as weak operations.

6. Experiments

In this section, we quantify the performance impact of the compilation methods discussed in the previous section. The objective is to measure the impact of the memory order, our ONRP optimisation, schedule strategy, and fixpoint strategy. Additionally, we compare our fastest implementation with a sequential implementation. We start by introducing the algorithms used in our benchmarks. Then we detail our methodology. We conclude by discussing the obtained results of our benchmarks.

6.1. The algorithms

We have implemented five algorithms in AuDaLa for our benchmarks. We introduce each in the following paragraphs.

Prefix Sum Our first test algorithm is the parallel *Prefix Sum* algorithm due to Hillis et al. [28]. Franken et al. translated the program into AuDaLa and explained its functioning [21]. The full source code is listed in Section C.1. The algorithm calculates the prefix sum of n elements in $O(\log n)$ time, provided that at least n processing elements are used. The reading and writing of

```
// Trimming step
Fix(set_inc_outg < trim < reset_inc_outg) <
// Main algorithm
Fix( pivot_nominate < allocate_sets <
    Fix(compute_fwd_bwd) < divide_into_sets )
```

Figure 18: The schedule of the Forward-Backward algorithm.

```
init_cols < Fix(
    Fix(prop_fwd) < del_fwd_frontier_edges <
    Fix(prop_bwd) < del_bwd_frontier_edges <
    reset_fwd_bwd )
```

Figure 19: The schedule of the Colouring/Heads Off algorithm.

```
Fix( Fix(expand_m_routes) < del_blocking <
    Fix(del_if_blocking_reachable) <
    Fix(collapse_m_routes) ) <
init_supervisor < Fix(expand_supervisor)
```

Figure 20: The schedule of the Supervisory Controller Synthesis algorithm.

the parameters is completely separated in distinct steps. Therefore, with ONRP the program can be compiled using only weak memory operations.

Forward-Backward We have implemented two SCC extraction algorithms in AuDaLa. The first is the *Forward-Backward (FB)* algorithm originally due to Fleischer et al. [20]. The full AuDaLa source code can be found in Section C.2. In the algorithm, a *pivot* is randomly selected from the set of nodes V. From the pivot, the set of forward reachable nodes F and the set of backward reachable nodes B are calculated. Naturally, the set $F \cap B$ is an SCC. The key insight of the algorithm is that the subsets $F \setminus B$, $B \setminus F$, and $V \setminus (B \cup F)$ can be solved recursively in parallel, as all remaining SCCs are contained in exactly one subset.

Similarly to [72], we added an iterative *trimming* step to remove trivial SCCs from the graph before running the main algorithm. First, edges between nodes in the same set write their label in the `inc` and `outg` parameters of their source and target node. Then, nodes without an `inc` or `outg` edge form a trivial SCC and thus create a new set for themselves. Finally, any `inc` or `outg` parameter that references a trimmed edge is reset. This process repeats until a fixpoint is reached.

The main algorithm also runs in a fixpoint. First, all nodes nominate themselves as their set's pivot. Only one node per set wins the data race and becomes the pivot. Then, non-empty sets (those with pivots) create three new sets, one for each subset described above. Now, the forward and backward sets starting from the pivots are computed in a fixpoint. Finally, each node enters the subset to which it belongs, depending on whether it is forward and/or backward reachable. The complete schedule is shown in Figure 18.

Colouring/Heads Off The second algorithm we implemented to extract SCCs is the *Colouring/Heads Off (CH)* algorithm due to Orzan [56]. The AuDaLa source code is listed in Section C.3. The algorithm uses *colours* related in a total order. We use the instances' labels as colours and use the instantiation order as our total order. Each node stores a *forward colour* and a *backward colour*, both initially referencing itself.

The main fixpoint of the algorithm is as follows: First, the edges between nodes propagate their source's forward colour forwards if it is lower than the target's forward colour, until a fixpoint is reached. Then, edges between nodes with different forward colours are deleted, as they are guaranteed to be between different SCCs. Now, the same is done for the backward colour, only with reversed edges. After this step, the nodes with the same forward and backward colour are part of an SCC. Nodes with two different colours reset their colours to reference itself again. This process repeats itself until no nodes change colours. All nodes in an SCC now have the same colour. See Figure 19 for the complete schedule.

Supervisory Controller Synthesis In the *Supervisory Controller Synthesis (SCS)* algorithm due to Ouedraogo et al. [57], a system is modelled as an automaton. Events are divided between *controllable* and *uncontrollable* events. Some states are *initial* states, and some states are *marked* states. The goal is to synthesise a *supervisor* automaton in which it is impossible to reach, through a sequence of uncontrollable events, a *blocked* state. A state is blocked if there is no sequence of events that can reach a marked state. The complete AuDaLa source code can be found in Section C.5.

The original algorithm computes which states to delete through a series of fixpoints. In doing so, it first resets all states marked as nonblocking, and then recalculates the nonblocking states with a nested fixpoint iteration. However, in AuDaLa all parameters must be stable throughout an iteration for a fixpoint to terminate. Therefore, it is not possible to first reset a parameter and then calculate its new value in a fixpoint. This will never terminate. Therefore, we not only calculate which states are nonblocking, but also keep track of the route through which a marked state can be reached (through the `m_route` parameter). After states are deleted, we collapse those routes that depend on a now-deleted state. This way, only the parameters that need to be reset are reset, and the program terminates again. The final step of the algorithm calculates the supervisor states: the non-deleted states reachable from the initial state. The complete schedule is shown in Figure 20.

Small Progress Measures The final algorithm we implemented is the *Small Progress Measures (SPM)* algorithm for solving parity games due to Jurdziński [33]. The details of the algorithm are beyond the scope of this work. Instead, we will discuss the key adaptations needed for an implementation in AuDaLa. The complete AuDaLa source code can be found in Section C.4.

In the algorithm, each edge computes a *measure*, which in our implementation is a vector of type $\mathbb{B} \times \mathbb{N} \times \mathbb{N}$. Then, each node must select either the minimum or maximum measure, lexicographically, from each of its outgoing edges. This operation is carried out by the edges as nodes do not have any information about their outgoing edges. However, AuDaLa does not support atomic min or max operations. Instead, we run a fixpoint for each element of the measure in which an edge nominates itself as a *candidate* minimum or maximum if it is lower or greater than the current candidate for that element. Because the boolean measure element only has two values, and thus terminates within two iterations, we can replace that fixpoint with two consecutive step calls.

Our use of candidate edges is similar to the `inc`, `outg` and `m_route` parameters of the FB and SCS algorithms. If the candidate edge's measure changes, it can remove itself as the candidate of its source node so that a new selection can take place without unnecessarily resetting the candidate during a fixpoint iteration.

6.2. Method

In this section we detail how we generated the test cases for each algorithm, which sequential implementations we compare to the AuDaLa implementations, and our benchmarking setup.

Test cases We run the Prefix Sum algorithm on a list of random numbers. Both of the SCC algorithms run on the same graphs instances, which are random directed graphs. The probability P of an edge between each pair of nodes is $P = \frac{1.3}{N}$, where N is the number of nodes. This gives relatively sparse graphs with a lot of SCCs of various sizes. Higher probabilities tend to result in graphs with a single SCC. For the SCS algorithm, we also use $P = \frac{1.3}{N}$ random graphs in which 60% of the edges are controllable, 5% of the nodes are initial states, and 20% are marked states. Finally, for the SPM algorithm, we use a model of the Dining Philosophers problem with varying numbers of philosophers. We test two types of parity games corresponding to the *Invariantly Inevitably Eat (IIE)* and *Invariantly Plato Starves (IPS)* modal formulas. We vary the *problem size* of our test cases between approximately 10^3 and 10^7. We take the maximum number of instances of a struct that executes steps as the problem size: the number of edges for the SCC and SPM algorithms, the number of states for the SCS algorithm, and the number of list elements for Prefix Sum.

Sequential implementations We compare our algorithms to the following sequential implementations. The Prefix Sum algorithm is compared to a simple sequential for loop that stores the prefix sum into a presized vector. The SCC algorithms are compared to Tarjan's algorithm [62], whose implementation is taken from the mCRL2 toolset [14]. We implemented the sequential SCS algorithm ourselves. It uses a depth-first search to perform the required reachability analysis. The SPM algorithm is compared to `pbespgsolve` from the mCRL2 toolset with the loop elimination option enabled.

Correctness We have tested the correctness of our algorithms, on all memory orders, by comparing the results with sequential implementations. For the SCC algorithms we compared the number of components returned, for SPM we looked at the partition between odd-won and even-won vertices, for Prefix Sum we checked every 5000th prefix sum, and for the SCS algorithm we looked at the number of remaining states. We ran each test multiple times and no differences were found.

Setup All of our GPU benchmarks are run on a RTX 3070. It contains 46 SMs, each with 128 KiB L1 cache and 128 CUDA cores, giving it a total of 5888 CUDA cores. Per SM, at most $1,536$ threads can be resident at any time. Therefore, at maximum occupancy, at most $46 \cdot 1,536 = 70,656$ threads can be resident on the device. The size of the L2 cache is 4 MiB. We measure the total execution time of the kernel(s) with CUDA events and do not measure the time it takes to copy the initial instances onto the device. All experiments are repeated 30 times, except for the SPM algorithm which is repeated 10 times due to its longer running time. We take the average of all measurements as the *runtime*. Its relative standard deviation is observed to be low on the larger test cases ($\approx 1\%$ or lower). Due to the pivot in the FB algorithm, the relative standard deviation is a bit higher, generally $< 3\%$. The GPU programs are compiled by NVCC version 12.2 with the `-O3` option. Our sequential implementations are run on an AMD Ryzen 5 5600x CPU with 32GB RAM. Runtime is measured through C++'s `high_resolution_clock` or the solving time reported by mCRL2. The sequential programs are compiled by GCC version 11.3 with the `-O3` option.

6.3. Results

In this section, we present and discuss the speedups gained through each of our compilation methods. After we find the optimal setting for each category, we use that setting for each of the subsequent measurements. Until we arrive at the optimal AuDaLaC configuration. We start with ONRP and then continue with the memory order. We move on with the fixpoint and schedule strategies, and we conclude by comparing the optimal AuDaLaC

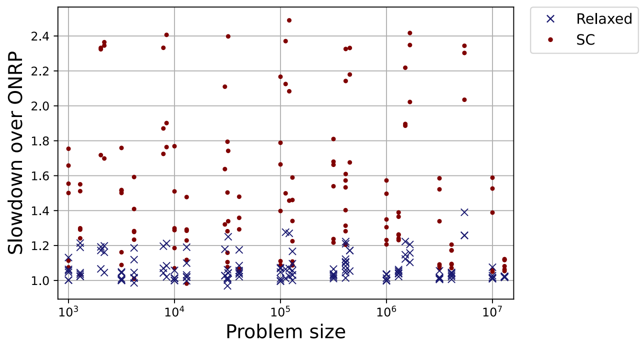


Figure 21: The slowdown over ONRP for the Relaxed and SC memory orders across all benchmarked algorithms, problem sizes and compilation strategies.

configuration with the sequential implementations.

ONRP The speedup gained through ONRP depends on how many accesses are non-racing and on how much the runtime is dominated by strong memory operation latencies. For example, when a kernel is mainly compute-bound, ONRP will make relatively little difference. Therefore, we cannot give a single speedup value. Instead, we plot the speedup of ONRP (or, equivalently, the slowdown without) measured for each combination of algorithm, problem size, and compilation strategy in Figure 21. We distinguish between measurements with the Relaxed memory order and the SC memory order. It can be seen that with the SC memory order, in the best case, ONRP runs approximately 2.5 times faster than without. In the worst case, the difference is negligible. For the relaxed memory order, the speedup is less extreme. In the best case, ONRP runs approximately 1.4 times faster than without. In the worst case, the difference is again negligible.

Memory order Similarly to ONRP, the speedup gained through the Relaxed memory order also depends on how much the runtime is dominated by strong memory operation latencies. We want to investigate the relationship between the algorithm, schedule strategy, problem size, and the Relaxed memory order speedup. Therefore, we plot the speedup against the problem size separately for each combination of algorithm and schedule strategy. We compare the runtime of the Relaxed memory order with both the SC and RA memory orders. We fix the fixpoint strategies to the simple strategies and compile with ONRP enabled. We omit the prefix sum algorithm because, after ONRP, it no longer contains any strong operations. The graphs are shown in Figure 23. Notice that the RA memory order behaves similarly to the SC memory order, only a bit faster. For FB, the speedup of the Relaxed memory order peaks at 1.65 compared to SC and 1.48 compared to RA, for CH at 1.69 and 1.45, for SCS at 1.22 and 1.15, and for SPM at 2.12 and 1.79 for the IPS game.

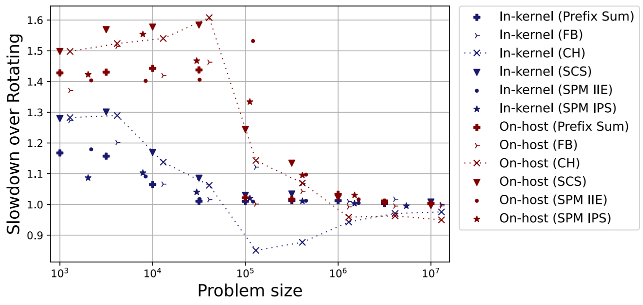


Figure 22: The measured slowdowns of the simple fixpoint strategy over the rotating fixpoint strategy for the in-kernel and on-host schedule strategies. The CH algorithm measurements are connected with a dotted line.

Across all algorithms and schedules, the speedup of the Relaxed memory order starts to increase after the problem size reaches approximately $10^{4.5}$. This seems to roughly correspond to the maximum number of resident threads possible on our GPU ($70,656 \approx 10^{4.85}$). We plot the maximum number of resident threads as a vertical dashed line in Figure 23. On the in-kernel schedule, this number is sometimes smaller due to register pressure limiting the block size as a result of the larger kernel. It seems that the impact of the memory order starts to grow when more threads than can be resident on the device are launched. This correlation could also be incidental: We cannot find any single profiler metric (such as the number of eligible warps per cycle) whose difference with the Relaxed memory order increases as sharply as the speedup after the maximum number of resident threads is surpassed. Therefore, the speedup increase seems to be caused by a summation of various effects.

With larger problem sizes, the speedup drops again. We explain this in Section 6.4. However, we can conclude that the Relaxed memory order will always perform fastest, followed by RA and then SC.

Fixpoint strategy We now compare the simple fixpoint strategy with the rotating fixpoint strategy. In Figure 22 we have plotted the speedup gained by the rotating strategy for both the on-host and in-kernel schedules. For the smaller problem sizes, the speedup can be as large as 1.6 on the on-host strategy and 1.3 on the in-kernel strategy. Interestingly, for the CH algorithm, the simple strategy outperforms the rotating strategy for larger problem sizes. We have marked those measurements with a dotted line in Figure 22. We believe that this happens because clearing the stability stack on the rotating strategy is relatively more expensive compared to the simple strategy. This is because it needs to take into account which stack element is being cleared at each fixpoint level. Because the CH algorithm assigns colours instead of boolean values, as is for example done in the FB algorithm, parameters change values relatively often,

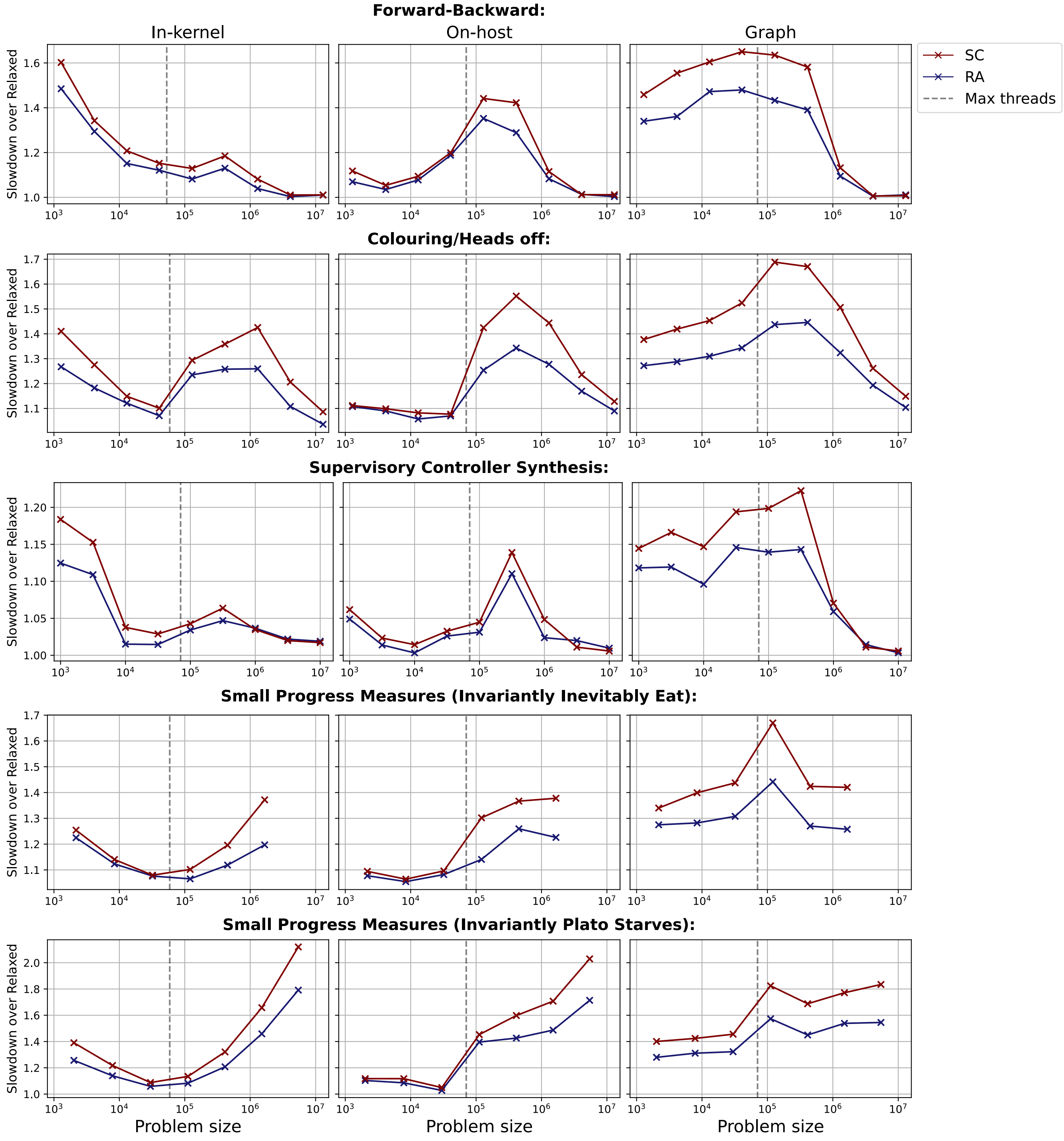


Figure 23: The slowdown over the Relaxed memory order for the SC and RA memory orders. Schedule strategies are divided over columns, and algorithms over rows. The maximum number of resident threads is marked by the vertical dashed line.

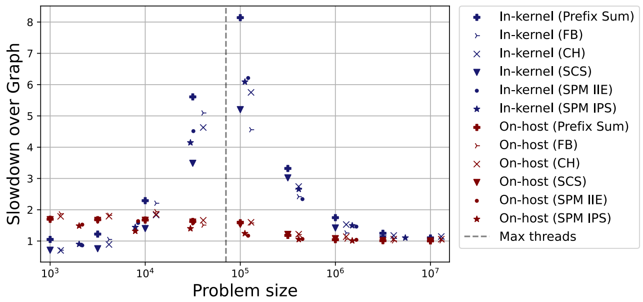


Figure 24: The measured slowdowns of the in-kernel and on-host schedule strategies over the graph schedule strategy.

resulting in more stability stack clearings. It could be the case that the extra runtime as a result of this outweighs the runtime saved by needing less synchronisation. Another thing we notice in Figure 22 is that, as the problem size increases, the speedup decreases until the difference is negligible. We discuss the cause of this in Section 6.4. However, we can conclude that for most cases on the on-host and in-kernel schedule strategies, the rotating fixpoint strategies are better than the simple strategies.

Schedule strategy Now that we have seen the speedup gained through ONRP, memory order, and fixpoint strategy, we make our final comparison between the three schedule strategies to select the configuration that we will compare with the sequential implementations. In Figure 24 we have plotted the slowdown as a result of using the on-host and in-kernel strategies over the graph strategy against the problem size. The in-kernel strategy performs rather poorly. Only for very small problem sizes does it outperform the other strategies. In the worst case, it runs more than 8 times slower than the graph schedule. We believe that grid-wide synchronisation takes more time when more blocks are participating. This would explain the increase in slowdown of the in-kernel schedule as the problem size grows to the maximum number of resident threads. The on-host schedule strategy is closer to the graph schedule strategy. In the worst case, it runs just under 2 times slower than the graph schedule. The slowdown of both schedules decreases when the problem size increases beyond 10^5. We give an explanation for this in Section 6.4. However, we can conclude that the graph schedule strategy is the best of our three schedule strategies.

Comparing with sequential implementations We have now verified that the graph schedule strategy with relaxed memory order and ONRP is the best-performing configuration. Therefore, we compare this configuration with the sequential implementations detailed in Section 6.2. The result is shown in Figure 25. We observe that the clearest winner over the sequential implementation is the SCS algorithm. In the best case, it executes

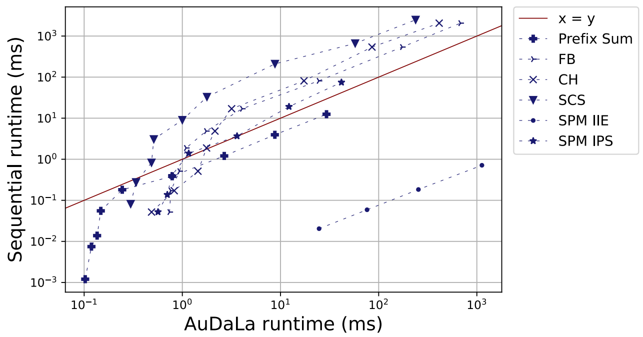


Figure 25: The runtime of the sequential implementations compared to the AuDaLa implementations. We have cropped off the last two measurements of the SPM IIE testcase for clarity.

23.9 times faster than the sequential implementation. The runner-up is the CH algorithm followed by the FB algorithm. Both algorithms beat Tarjan's algorithm running on a CPU. In the best case, CH runs 6.3 times faster and FB 4.2 times. With the SPM algorithm, AuDaLa beats mCRL2 for the Invariantly Plato Starves parity game by at most 1.8 times. However, this parity game contains only one vertex of priority 3 for all problem sizes, which means that fewer operations are required to reach a fixpoint. A more complex parity game to solve is the Invariantly Inevitably Eat game. For that game, AuDaLa performs at best $1,212$ times slower and at worst $7,944$ times. We believe that this inefficiency is mainly due to having to calculate the minimum or maximum measures in a fixpoint, instead of with an atomic minimum or maximum operation. The Prefix Sum algorithm comes close, initially lagging behind the sequential implementation but quickly catching up. However, as the number of resident threads on the GPU runs out, it just loses out by running at best 1.3 times slower.

6.4. The impact of the L2 cache

The speedup of many of our optimisations diminishes as the problem size grows beyond $\approx 10^5$. This can be seen in Figure 23, 22, and 24. This indicates that another phenomenon occurs that dominates the runtime so much that our optimisations make little difference anymore. It turns out that the L2 cache is the culprit. We have used NVIDIA's Nsight Compute profiler [52] to track the L2 cache hit rate of the CH algorithm in Figure 26. Clearly, the L2 cache hit rate drops dramatically with a problem size of $10^{5.5}$ or larger. The L2 cache of our GPU has a capacity of 4 MiB $= 4 \cdot 2^{20}$ bytes $\approx 3.3 \cdot 10^{5.5}$ parameter values. Therefore, the parameter data need to be retrieved from the off-chip device memory, which has very high latency and thus increases runtime significantly.

It can be seen from Figure 23 that the SPM algorithm does not suffer as much from L2 cache misses as the other

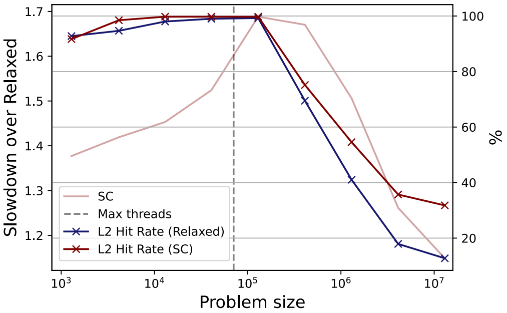


Figure 26: The slowdown of SC over the Relaxed memory order (left axis) plotted against the problem size for the CH algorithm on the graph schedule, superimposed with the L2 cache hit rate as returned by the Nsight Compute profiler (right axis).

algorithms. This is due to the nature of the parity game graphs and how they are generated. Compared to the random graphs of the other algorithms, parity games are much more clustered in connected components. The nodes of these components are usually defined close to each other in the `.init` file and are thus often mapped to the same thread block. Caches can leverage this locality if threads of the same block request data from the same memory region. To show the role data locality plays in the SPM algorithm, we shuffle the input order of the parity games' edges and run the benchmark again. The result of the shuffled SPM algorithm is shown in Figure 27, where the decrease in speedup is clearly visible this time.

6.5. Discussion

In this section, we have shown that our best-performing compilation method is the graph schedule strategy, with ONRP, and the Relaxed memory order. We have also shown that the speedup gained through the memory order differs per algorithm, compilation strategy, and problem size. Therefore, the most complete answer to RQ 3 we can provide is shown in Figure 23. In the most extreme case, the relaxed memory order outperforms SC by 2.1 times. On the graph schedule, the Relaxed memory order outperforms SC by at most 1.7 times. We leave it up to the user to decide whether or not this speedup is worth the added difficulty of reasoning that comes with our semantics. In addition, the speedups gained through our optimisations and the memory order quickly diminish for larger problem sizes due to L2 cache misses dominating the runtime. Improving data locality is thus an important future optimisation.

With regard to RQ 4, we have shown that AuDaLa can be used successfully as a method of parallelising algorithms. We have shown that AuDaLa beats the sequential implementations of the SCS, CH, and FB algorithms. AuDaLa beats the SPM algorithm only for the IPS parity game. On the IIE parity game, the AuDaLa implementation loses by at least a factor of $1,212$ times. This shows that not all algorithms are well suited for an implementation in AuDaLa. The sequential Prefix Sum implementation also outperforms the AuDaLa implementation, even though its asymptotic runtime is worse ($O(n)$ vs $O(\log n)$, with n processors). This shows that the overhead of an AuDaLa implementation can be significant. In addition, due to our mapping of instances to threads, the Prexix Sum algorithm does not divide its work well over the SMs of the GPU: The final steps of the algorithm are all executed in a single block and eventually even in a single warp. However, many algorithms naturally align with AuDaLa's programming model. Especially for those algorithms, AuDaLa proves to be a useful programming language that abstracts from parallel hardware architectures while still benefitting from a high degree of parallel execution.

7. Related Work

In this section, work related to ours is discussed. We have grouped the related work by the themes discussed throughout this work.

Comparisons of memory models The first work that uses Alloy to compare MCM is due to Wickerson et al [71]. In addition to comparing MCMs, they also use Alloy to derive useful litmus tests that can distinguish given MCMs. Furthermore, they formalise an unofficial MCM for PTX. In the work of Lustig et al. [46], the official formalisation of the PTX MCM is presented and compared with the formalisation of the C++ MCM due to Lahav et al. [36]. Both are formalised as axiomatic models. They take a similar approach to ours by forward mapping pre-executions and reverse mapping execution witnesses. They do not map any dependencies, as the used C++ model does include them. In addition to using Alloy for bounded program sizes, they, unlike our work, use Coq [63] to formally prove that their mapping is correct for all program sizes. However, where we formalise AuDaLa's MCM both operationally and axiomatically, they only formalise PTX as an axiomatic MCM. In [3], Alglave et al. provide a general framework for MCMs, formalised both in an axiomatic and operational fashion. The correspondence between their two models is formally proven via a mechanised Coq proof.

Relaxed memory semantics Formalisations of relaxed memory semantics can be classified by their approach to dependency tracking. Some have very strong dependencies, which disallow common compiler optimisations but forbid OOTA behaviour [13, 35, 36]. Others do not bother with dependencies at all or only in a very limited fashion, allowing for optimisations, but also OOTA behaviour [9, 46, 70]. We have presented two axiomatic MCMs for AuDaLa, one in each of the previously men-

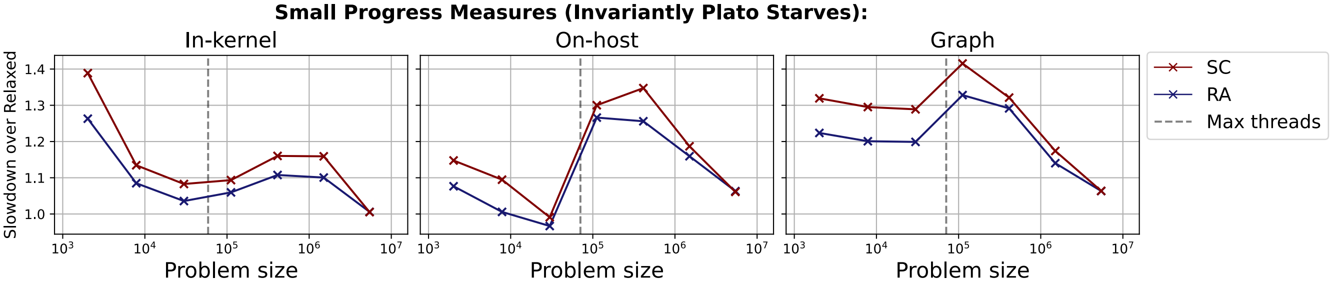


Figure 27: The slowdown over the Relaxed memory order for the SC and RA memory orders on the SPM algorithm, with its edges randomly shuffled in the `.init` file.

tioned categories. The C++ standard [29] forbids OOTA behaviour only informally: *"Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation."*. This requirement is too informal to build analysis on. Therefore, much work has gone into finding a formalism that finds the actual dependencies, instead of an over- or under-approximation. Several solutions have been proposed: An influential proposal is the *promising semantics* due to Kang et al. [34]. It provides an elegant operational semantics for relaxed memory. However, a high level of nondeterminism makes it intractable for automated verification techniques. Furthermore, Paviotti et al. [59] reason that its speculative operational model departs too much from the axiomatic style adopted by the C++ standard and makes mature analysis tools obsolete. Instead, they propose *Modular Relaxed Dependencies (MRD)*. In MRD the true dependency relation is calculated using *event structures*, which, unlike axiomatic candidate executions, contain information about all execution traces. This true dependency relation can then be incorporated into existing axiomatic models or used to define new operational models [73]. In addition, they provide a refinement relation that identifies a program as a refinement of another if it is a permissible optimisation of the other. These properties have resulted in a proposal to use MRD to fix the thin-air problem in C++ [5].

Programming models A programming model similar to AuDaLa is that of the *CHemical Abstract Machine (CHAM)* [10]. The CHAM is based on the Γ-*calculus* [4] in which *molecules* react with each other to form new molecules. The molecules carry data and can only participate in a single reaction at a time. Because of this, their data are never shared, thus preventing the shared-memory concurrency issues we have discussed in this work. Similarly to AuDaLa's fixpoint termination, the computation ends when no more reactions can occur.

Since multiple reactions can occur simultaneously, there is a high degree of parallelism in the Γ-calculus. Therefore, it is also an attractive target for GPU implementation, as has been done in the work of Gannouni et al. [22, 23]. It shows results of a CUDA implementation of a parallel programming framework based on the Γ-calculus. In the work of Lin et al. [42] a Γ-calculus implementation on various computing platforms is presented, including a work-in-progress CUDA implementation.

An AuDaLa struct instance that has a reference to another struct instance can mutate the other's data. This can be viewed as a message-passing mechanism between concurrent actors. There are many message-passing languages in which concurrent processes are used to perform computations. Some example languages are the ParCel languages [15, 67, 68].

8. Conclusion and future work

In this work we have investigated the implications of relaxed memory semantics for the AuDaLa programming language, both formally and experimentally. We defined an alternative operational semantics with relaxed memory consistency semantics and incoherent memory. Subsequently, we used Alloy to verify the correctness of mapping AuDaLa memory operations to PTX memory operations with either the Relaxed memory order or the Release/Acquire memory order. We showed that mapping to the Release/Acquire memory order can be done safely. Using the Relaxed memory order comes with a warning: Mapping to the Relaxed memory order is only safe when it is certain that the compiler will not break any dependencies that are critical in an ordering assumed to hold by the programmer that is essential for program corectness. However, due to the out-of-thin-air problem, the PTX MCM cannot satisfactorily specify which dependencies are inviolable and which are not.

An opportunity for future work is to mechanise a proof of correctness of our translation from the operational model to the axiomatic in a proof assistant such as Coq, as is commonly done in the literature. The mapping to PTX could then also be proven for unbounded program sizes. The improvements to our operational semantics discussed at the end of Section 3.3 could also be the subject of future work. To date, no real consensus has emerged on ways to address the lack of rigorous dependency modelling in programming language MCMs. When a solution does become widely accepted, future work could revist AuDaLa's relaxed memory semantics to see how such a new formalism can be incorporated into the semantics.

On the experimental side, we presented the AuDaLa compiler AuDaLaC. We adapted a range of algorithms for AuDaLa and used AuDaLaC to benchmark and verify the implementations. We showed how AuDaLa's use of a schedule makes it possible to identify race-free accesses, which can be replaced with weak memory operations. This resulted in a best-case speedup of 2.5 when using the SC memory order and 1.4 when using the Relaxed memory order. We identified three strategies to run the schedule on a GPU. The most performant uses the CUDA Graphs API to minimise kernel launch and host-device synchronisation overhead, resulting in a speedup of up to 1.9 compared to conventional kernel launches. Although running the complete schedule in a single kernel eliminates launch overhead, it still performs worse as the grid synchronisation costs outweigh this benefit. The impact of the Relaxed memory order can be high. We have measured speedups of 1.7 times compared with the SC memory order on the graph schedule strategy. However, once the L2 cache miss rate starts to increase, these speedups become insignificant. Future work should thus focus on methods to improve the data locality during execution. The incoherent memory semantics we have defined for AuDaLa should help in this regard.

Finally, we have shown that most of our tested AuDaLa implementations beat their sequential counterparts. This result is promising and demonstrates that AuDaLa successfully allows programmers to focus on their algorithms rather than the implementation details while still taking advantage of the parallel execution capabilities that GPUs have to offer.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Thomas Neele. I am grateful for all of our long meetings in which, between my ramblings, you were able to help me structure my thoughts and prioritise. Your ideas were invaluable and you always provided me with quick feedback, for which I am grateful.

My thanks goes out to Daniel Lustig for helping me understand the PTX MCM and answering all my emails in detail.

I am also deeply indebted to Julia, who has given me irreplaceable support, gave me all the time and space I needed, helped me make schedules, and even managed to let me keep some of them ;)

My final word of thanks is for my parents, who have nurtured my curiosity from a young age.

References

[1] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, Dec 1996.

[2] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 141–157, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):1–74, 2014.

[4] Jean-Pierre Banâtre and Daniel Le Métayer. The gamma model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, 1990.

[5] Mark Batty, Simon Cooksey, Scott Owens, Anouk Paradis, Marco Paviotti, and Daniel Wright. ISO/IEC JTC1/SC22/WG21: C++ standard draft proposal D1780R0, 2019.

[6] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. *SIGPLAN Not.*, 48(1):235–248, jan 2013.

[7] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In Jan Vitek, editor, *Programming Languages and Systems*, pages 283–307, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[8] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to power. *SIGPLAN Not.*, 47(1):509–520, jan 2012.

[9] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. *SIGPLAN Not.*, 46(1):55–66, jan 2011.

[10] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[11] Hans Boehm, Mark Batty, Brian Demsky, Olivier Giroux, Paul McKenney, Peter Sewell, and Francesco Zappa Nardelli. ISO/IEC JTC1/SC2/WG21: C++ standards committee paper N3710, 2013.

[12] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, jun 2008.

[13] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mcrl2 toolset for analysing concurrent systems. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39, Cham, 2019. Springer International Publishing.

[15] Paul-Jean Cagnard. The ParCeL-2 Programming Language. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000 Parallel Processing*, pages 767–770, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[16] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 331–346, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[17] Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. Modular data-race-freedom guarantees in the promising semantics. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 867–882, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Federico Ciccozzi, Lorenzo Addazi, Sara Abbaspour Asadollah, Björn Lisper, Abu Naser Masud, and Saad Mubeen. A

Comprehensive Exploration of Languages for Parallel Computing. *ACM Computing Surveys*, 55(2):24:1–24:39, January 2022.

[19] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[20] Lisa K. Fleischer, Bruce Hendrickson, and Ali Pınar. On identifying strongly connected components in parallel. In José Rolim, editor, *Parallel and Distributed Processing*, pages 505–511, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[21] Tom Franken, Thomas Neele, and Jan Friso Groote. An autonomous data language. *Unpublished manuscript*, 2023.

[22] Sofien Gannouni. A gamma-calculus GPU-based parallel programming framework. *2015 2nd World Symposium on Web Applications and Networking, WSWAN 2015*, 08 2015.

[23] Sofien Gannouni, A. Touir, and H. Mathkour. Paradigma: A distributed framework for parallel programming. *International Arab Journal of Information Technology*, 15:934–943, 09 2018.

[24] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. *SIGPLAN Not.*, 26(4):245–257, apr 1991.

[25] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. Compound memory models. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[26] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[27] T. D. Han and T. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[28] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, dec 1986.

[29] ISO/IEC. *Programming Languages — C++*. International Organization for Standardization, sixth edition edition, 2020.

[30] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, apr 2002.

[31] Radha Jagadeesan, Alan Jeffrey, and James Riely. Pomsets with preconditions: A simple model of relaxed memory. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[32] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. The leaky semicolon: Compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[33] Marcin Jurdziński. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS 2000*, pages 290–301, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[34] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. *SIGPLAN Not.*, 52(1):175–189, jan 2017.

[35] Ryan Kavanagh and Stephen D. Brookes. A denotational account of C11-style memory. *ArXiv*, abs/1804.04214, 2018.

[36] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++ 11. *ACM SIGPLAN Notices*, 52(6):618–632, 2017.

[37] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sep. 1979.

[38] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0: Global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 362–376, New York, NY, USA, 2020. Association for Computing Machinery.

[39] G. P. Leemrijse. The AuDaLaC compiler. https://github.com/GPLeemrijse/AuDaLaC, 2023.

[40] G. P. Leemrijse. Towards relaxed memory semantics for the Autonomous Data Language [Supplemental Alloy Files]. https://doi.org/10.5281/zenodo.8244711, August 2023.

[41] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[42] Hong Lin, Jeremy Kemp, and Padraic Gilbert. Computing gamma calculus on computer cluster. *IJTD*, 1:42–52, 10 2010.

[43] Lun Liu, Todd Millstein, and Madanlal Musuvathi. Accelerating sequential consistency for java with speculative compilation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 16–30, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Daniel Lustig. Personal communication, May 2023.

[45] Daniel Lustig, Simon Cooksey, and Olivier Giroux. Mixed-proxy extensions for the nvidia PTX memory consistency model: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1058–1070, New York, NY, USA, 2022. Association for Computing Machinery.

[46] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 257–270, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated synthesis of comprehensive memory model litmus test suites. *SIGARCH Comput. Archit. News*, 45(1):661–675, apr 2017.

[48] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A case for an SC-preserving compiler. *SIGPLAN Not.*, 46(6):199–210, jun 2011.

[49] Niko Matsakis and contributors. LALRPOP. https://lalrpop.github.io/lalrpop/, 2023.

[50] Evgenii Moiseenko, Michalis Kokologiannakis, and Viktor Vafeiadis. Model checking for a multi-execution memory model. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022.

[51] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for C/C++11 concurrency. *SIGPLAN Not.*, 51(10):111–128, oct 2016.

[52] NVIDIA. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.

[53] NVIDIA. PTX ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, 2023.

[54] OpenMP Architecture Review Board. OpenMP application program interface version 5.0, January 2023.

[55] S.M. Orzan. *On distributed verification and verified distribution*. PhD thesis, Centrum voor Wiskunde en Informatica, Vrije Universiteit Amsterdam, 2004.

[56] S.M. Orzan. *On Distributed Verification and Verified Distribution*. PhD thesis, Vrije Universiteit Amsterdam, 2004.

[57] Lucien Ouedraogo, Ratnesh Kumar, Robi Malik, and Knut Åkesson. Symbolic approach to nonblocking and safe control of extended finite automata. In *2010 IEEE International Conference on Automation Science and Engineering*, pages 471–476, Aug 2010.

[58] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[59] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. Modular relaxed dependencies in weak memory concurrency. In Peter Müller, editor, *Programming Languages and Systems*, pages 599–625, Cham, 2020. Springer International Publishing.

[60] Gabe Rudy. CUDA-CHiLL: A programming language interface for gpgpu optimizations and code generation, 2010.

[61] Tyler Sorensen and Alastair F. Donaldson. Exposing errors related to weak memory in GPU applications. *SIGPLAN Not.*, 51(6):100–113, jun 2016.

[62] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[63] The Coq Development Team. The coq proof assistant, June 2023.

[64] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-mei W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. Series Title: Lecture Notes in Computer Science.

[65] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. *SIGPLAN Not.*, 50(1):209–220, jan 2015.

[66] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. *SIGPLAN Not.*, 48(10):867–884, oct 2013.

[67] Stéphane Vialle, Makram Bouzid, Vincent Chevrier, and François Charpillet. ParCeL-3: A Parallel Programming Language Based on Concurrent Cells and Multiple Clocks. In *International Conference on Software Engineering Applied to Networking and Parallel/Distributed Computing - SNPd'00*, page 4 p, Reims, France, 2000. Colloque avec actes et comité de lecture. internationale.

[68] Stéphane Vialle, Thierry Cornu, and Yannick Lallement. Parcel-1: A parallel programming language based on autonomous and synchronous actors. *SIGPLAN Not.*, 31(8):43–51, aug 1996.

[69] Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. *SIGPLAN Not.*, 46(6):306–316, jun 2011.

[70] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. Repairing and mechanising the javascript relaxed memory model. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 346–361, New York, NY, USA, 2020. Association for Computing Machinery.

[71] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 190–204, New York, NY, USA, 2017. Association for Computing Machinery.

[72] Anton Wijs, Joost-Pieter Katoen, and Dragan Bošnački. Efficient GPU algorithms for parallel decomposition of graphs into strongly connected and maximal end components. *Formal Methods in System Design*, 48(3):274–300, April 2016.

[73] Daniel Wright, Sadegh Dalvandi, Mark Batty, and Brijesh Dongol. Mechanised operational reasoning for c11 programs with relaxed dependencies. *Form. Asp. Comput.*, 35(2), jun 2023.

A. Alloy files

```
module ADL[hwEvent, hwAddress, hwThread]
open util

/** Signatures and helper functions */
sig Location {
  loc_map : one hwAddress
}

some sig Instance {
  start : one Command,
  inst_map : one hwThread
}

abstract sig Command {
  nxt : lone Command,
  loc : one Location,
  cmd_map : one hwEvent
}

sig ReadCommand extends Command {
  dep : set Command
}

sig WriteCommand extends Command {
  rf : set ReadCommand,
  co : set WriteCommand
}

fun inst : Command->Instance { ~(start.*nxt) }
fun po : Command->Command { ^nxt }
fun po_loc : Command->Command { po & same_loc }
fun rfi : WriteCommand->ReadCommand { rf & same_instance }
fun fr : ReadCommand->WriteCommand { ~rfi.^co + fr_init }
fun fr_init : ReadCommand->WriteCommand { ident[ReadCommand - WriteCommand.rf].((same_instance & same_loc) :> WriteCommand) }
fun same_instance : Command->Command { inst.~inst }
fun same_loc : Command->Command { loc.~loc }
fun other_loc : Command->Command { Command->Command - same_loc }

/** Pre-execution correctness predicates */
pred at_most_one_incoming_cmd { all c : Command | lone c.~nxt }
pred at_most_one_incoming_start { all c : Command | lone c.~start }
pred only_start_commands_have_no_predecessor { Instance.start = (Command - Command.nxt) }
pred use_all_locations { Command.loc = Location }
pred dep_in_po { dep in po }
pred wellformed_adl_preex {
  at_most_one_incoming_cmd
  at_most_one_incoming_start
  only_start_commands_have_no_predecessor
  acyclic[nxt]
  use_all_locations
  dep_in_po
}

/** Execution witness correctness predicates */
pred rf_and_co_of_same_location { rf + co in same_loc }
pred co_of_same_instance { co in same_instance }
pred reads_read_from_at_most_one_write { all r : ReadCommand | lone r.~rf }
pred co_total_per_loc_inst {
  all l : Location, i : Instance |
    // Intersection of all commands with l and i as location and instance, with writes
    total[co, l.~loc & i.~inst & WriteCommand]
}

/* CONSISTENCY Axiom: Semantic ordering is consistent */
pred CONSISTENCY { acyclic[po_loc + co + rf + fr + dep] }

/* CONSISTENCY* Axiom: Semantic ordering is consistent, without dep.*/
pred CONSISTENCY_STAR { acyclic[po_loc + co + rf + fr] }

/* NO_THIN_AIR Axiom: No reads from dependants. */
pred NO_THIN_AIR { acyclic[rf + dep] }

pred legal_adl_execution {
  rf_and_co_of_same_location
  co_of_same_instance
  reads_read_from_at_most_one_write
  co_total_per_loc_inst
  //CONSISTENCY

  CONSISTENCY_STAR
  NO_THIN_AIR
}
```

Listing 1: The AuDaLa axiomatic MCM formalised in Alloy.

```
module compile_adl2ptx
open util as util
open ptx as hw
open ADL[hw/Event, hw/Address, hw/Thread] as sem

fun Device : Scope { System.subscope }
pred map_preex {
  // cmd_map is a 1-to-1 map between sem/Command and hw/Event
  bijective[cmd_map, sem/Command, hw/Event]
  // inst_map is a 1-to-1 map between sem/Instance and hw/Thread
  bijective[inst_map, sem/Instance, hw/Thread]
  // loc_map is a 1-to-1 map between sem/Location and hw/Address
  bijective[loc_map, sem/Location, hw/Address]
  // map next_cmd onto po
  nxt = cmd_map.po.~cmd_map
  // map loc onto address
  loc = cmd_map.address.~loc_map
  // map sem/start onto hw/start
  (sem/Instance <: start) = inst_map.(hw/Thread <: start).~cmd_map
  // map sem/dep onto hw/dep
  (sem/Command <: dep) = cmd_map.(hw/Event <: dep).~cmd_map

  // Set scopes and rmw properly
  set_scopes_and_rmw

  // Pick one:
  // Relaxed
  map_preex_rlx
  // Acquire/Release
  //map_preex_acqrel
}


pred map_preex_rlx {
  // map sem/ReadCommand to hw/Read, but not acquire
  (sem/ReadCommand).cmd_map in (hw/Read - hw/Acquire)
  // map sem/WriteCommand to hw/Write, but not release
  (sem/WriteCommand).cmd_map in (hw/Write - hw/Release)
}

pred map_preex_acqrel {
  // map sem/ReadCommand to hw/Acquire
  (sem/ReadCommand).cmd_map in (hw/Acquire)
  // map sem/WriteCommand to hw/Release
  (sem/WriteCommand).cmd_map in (hw/Release)
}

pred set_scopes_and_rmw {
  no rmw // No read-modify-write
  one Device // one device
  Device.subscope = hw/Thread // Thread scope is one lvl lower
  (hw/Event).scope = Device // all events have device scope
}

pred reverse_map {
  (sem/WriteCommand <: rf) = cmd_map.(hw/Write <: rf).~cmd_map
  (sem/WriteCommand <: co) = cmd_map.(hw/Write <: (same_thread[^co])).~cmd_map
}

assert mapping_correct {
  (wellformed_adl_preex and
  map_preex and
  ptx_mm and
  reverse_map) implies legal_adl_execution
}

check mapping_correct for 8 but 0 hw/Fence, 0 hw/Barrier
```

Listing 2: The mapping between AuDaLa and PTX formalised in Alloy.

```
module util

fun symmetric[r: univ->univ] : univ->univ { r & ~r }
fun optional[f: univ->univ] : univ->univ  { iden + f }
pred irreflexive[rel: univ->univ]         { no iden & rel }
pred acyclic[rel: univ->univ]             { irreflexive[^rel] }
pred total[rel: univ->univ, bag: univ] {
  all disj e, e’: bag | e->e’ + e’->e in ^rel + ^~rel
  acyclic[rel]
}

fun ident[e: univ] : univ->univ       { iden & e->e }
fun imm[rel: univ->univ] : univ->univ { rel - rel.rel }
pred transitive[rel: univ->univ]      { rel = ^rel }
pred strict_partial[rel: univ->univ] {
  irreflexive[rel]
  transitive[rel]
}

pred bijective[R: univ->univ, S1: univ, S2: univ] {
  (all s : S1 | (one s.R) and (s.R in S2))
  (all s : S2 | (one s.~R) and (s.~R in S1))
}
```

Listing 3: Utility functions formalised in Alloy.

B. AST validation errors

Error code	**Name**	**Cause**
E001	StructDefinedTwice	Two structs share a name.
E002	StepDefinedTwice	Two steps, of the same struct, share a name.
E003	ParameterDefinedTwice	Two parameters of a struct share a name.
E004	VariableAlreadyDeclared	A variable is declared with a name that is already in use.
E005	UndefinedType	An undefined type is referenced.
E006	UndefinedParameter	An undefined parameter is referenced.
E007	UndefinedStep	An undefined step is called.
E008	UndefinedVariable	An undefined variable is referenced.
E009	InvalidNumberOfArguments	A struct is constructed with the wrong number of parameters.
E010	TypeMismatch	A supplied type does not match, or cannot be coerced into, the expected type.
E011	InvalidTypesForOperator	The supplied operands of an operator are of invalid types for that operator.
E012	ReservedKeyword	A reserved keyword is used as an ID.

Table 2: The AST validation errors and their cause.

C. Benchmarked AuDaLa programs

C.1. Prefix Sum

```
struct Position(val: Int, prev: Position, auxval: Int, auxprev: Position) {
    read {
        auxval := prev.val;
        auxprev := prev.prev;
    }
    write {
        val := val + auxval;
        prev := auxprev;
    }
}
```

```
Fix(read < write)
```

Listing 4: The Prefix Sum algorithm from [21].

C.2. Forward-Backward

```
struct NodeSet(pivot : Node, f_and_not_b : NodeSet,
               not_f_and_b : NodeSet, not_f_and_not_b : NodeSet) {

    allocate_sets {
        if pivot != null then {
            if f_and_not_b == null then {
                f_and_not_b := NodeSet(null, null, null, null);
                not_f_and_b := NodeSet(null, null, null, null);
                not_f_and_not_b := NodeSet(null, null, null, null);
            }
            pivot.fwd := true; pivot.bwd := true;
        }
    }
}

struct Node(set : NodeSet, fwd : Bool, bwd : Bool,
    inc: Edge, outg: Edge) {
    pivot_nominate {
        if set.pivot == null then {
            set.pivot := this;
        }
    }

    divide_into_sets {
        Bool f := fwd; Bool b := bwd;
        if f  && !b then { set := set.f_and_not_b; }
        if !f &&  b then { set := set.not_f_and_b; }
        if !f && !b then { set := set.not_f_and_not_b; }

        if !(f && b) then { fwd := false; bwd := false; }
    }

    trim {
        if this != null && !fwd && (inc == null || outg == null) then {
            set := NodeSet(this, null, null, null);
            set.f_and_not_b := set;
            set.not_f_and_b := set;
            set.not_f_and_not_b := set;
            fwd := true; bwd := true;
        }
    }
}

struct Edge(s : Node, t : Node){
    compute_fwd_bwd {
        if t.set == s.set then {
            if s.fwd then { t.fwd := true; }
            if t.bwd then { s.bwd := true; }
        }
    }

    set_in_out {
        if t.set == s.set then {
            if t.inc == null then { t.inc := this; }
            if s.outg == null then { s.outg := this; }
        }
    }

    reset_inc_outg {
        if t.set != s.set then {
            if t.inc == this then { t.inc := null; }
            if s.outg == this then { s.outg := null; }
        }
    }
}

Fix( set_inc_outg < trim < reset_in_out ) <
Fix( pivot_nominate < allocate_sets <
    Fix(compute_fwd_bwd) < divide_into_sets
)
```

Listing 5: The Forward-Backward algorithm implemented in AuDaLa.

C.3. Colouring/Heads-off

```
struct Edge(s : Node, t : Node, deleted: Bool) {
    propagate_fwd {
        if !deleted then {
            if s.fwd < t.fwd then { t.fwd := s.fwd; }
        }
    }

    propagate_bwd {
        if !deleted then {
            if t.bwd < s.bwd then { s.bwd := t.bwd; }
        }
    }

    del_fwd_frontier_edges {
        if !deleted then { deleted := s.fwd != t.fwd; }
    }

    del_bwd_frontier_edges {
        if !deleted then { deleted := s.bwd != t.bwd; }
    }
}

struct Node(fwd : Node, bwd : Node) {
    init { fwd := this; bwd := this; }

    reset_fwd_bwd {
        if fwd != bwd then { fwd := this;  bwd := this; }
    }
}
```

```

init <
Fix(
    Fix(propagate_fwd) < del_fwd_frontier_edges <
    Fix(propagate_bwd) < del_bwd_frontier_edges <
    reset_fwd_bwd
)
```

Listing 6: The Colouring/Heads-off algorithm implemented in AuDaLa.

C.4. Small Progress Measures

```
struct Node(p: Nat, is_odd: Bool, dirty: Bool, rho_top: Bool,
            rho_1: Nat, rho_3: Nat, cand: Edge) {
    lift {
        dirty := false;
        if cand != null then {
            // Copy if cand is lexicographically greater
            if (cand.prog_top != rho_top && cand.prog_top) ||
               (cand.prog_top == rho_top && cand.prog_1 > rho_1) ||
               (cand.prog_top == rho_top && cand.prog_1 == rho_1 && cand.prog_3 > rho_3) then {
                    rho_top := cand.prog_top;
                    rho_1 := cand.prog_1;
                    rho_3 := cand.prog_3;
                    dirty := true;
            }
        }
    }
}

struct Edge(v: Node, w: Node, max1: Nat, max3: Nat, prog_top: Bool, prog_1: Nat, prog_3: Nat) {
    prog {
        if w.dirty then {
            Bool m_top := w.rho_top; Nat m_1 := 0; Nat m_3 := 0;
            if v.p >= 1 then { m_1 := w.rho_1; }
            if v.p >= 3 then { m_3 := w.rho_3; }

            if v.p % 2 == 1 then {
                Bool increased := m_top;
                if v.p >= 3 && !increased then {
                    Nat new_3 := 0;
                    if m_3 < max3 then { new_3 := m_3 + 1; increased := true; }
                    m_3 := new_3;
                }
                if v.p >= 1 && !increased then {
                    Nat new_1 := 0;
                    if m_1 < max1 then { new_1 := m_1 + 1; increased := true; }
                    m_1 := new_1;
                }
                if !increased then { m_top := true; }
            }

            // Reset cand if it changed
            if v.cand == this then {
                Bool changed := m_top != prog_top || prog_1 != m_1 || prog_3 != m_3;
                if changed then { v.cand := null; }
            }
            prog_top := m_top; prog_1 := m_1; prog_3 := m_3;
        }
    }

    minmax_top1 {
        if v.cand == null then { v.cand := this; }
    }

    minmax_top2 {
        if  /*min*/ (!v.is_odd &&  v.cand.prog_top && !prog_top) ||
            /*max*/ ( v.is_odd && !v.cand.prog_top &&  prog_top)
             then {
                v.cand := this;
        }
    }

    minmax_1 {
        if v.cand.prog_top == prog_top && (
            /*min*/ (!v.is_odd && v.cand.prog_1 > prog_1) ||
            /*max*/ ( v.is_odd && v.cand.prog_1 < prog_1)) then {
            v.cand := this;
        }
    }

    minmax_3 {
        if v.cand.prog_top == prog_top &&
           v.cand.prog_1   == prog_1   && (
            /*min*/ (!v.is_odd && v.cand.prog_3 > prog_3) ||
            /*max*/ ( v.is_odd && v.cand.prog_3 < prog_3)) then {
            v.cand := this;
        }
    }

    self_loops_to_top {
        if v == w && v.is_odd && v.p % 2 == 1 then { v.rho_top := true; }
    }
}

self_loops_to_top <
Fix(
    prog < minmax_top1 < minmax_top2 <
    Fix(minmax_1) < Fix(minmax_3) < lift
)
```

Listing 7: The Small Progress Measures algorithm implemented in AuDaLa.

C.5. Supervisory Controller Synthesis

```
struct State(marked : Bool, initial : Bool, deleted : Bool, m_route : ControllableEvent,
    supervisor: Bool) {
    del_blocking {
        if m_route == null && !marked then { deleted := true; }
    }

    collapse_m_routes {
        if m_route != null &&
           (m_route.y.deleted || (m_route.y.m_route == null && !m_route.y.marked)) then {
            m_route := null;
        }
    }

    init_supervisor { supervisor := initial && !deleted; }
}

struct ControllableEvent(x : State, y : State){
    expand_m_route {
        if !x.deleted && !y.deleted then {
            if x.m_route == null && (y.m_route != null || y.marked) then {
                x.m_route := this;
            }
        }
    }

    expand_supervisor {
        if !x.deleted && !y.deleted then {
            if x.supervisor then { y.supervisor := true; }
        }
    }
}

struct UncontrollableEvent(x : State, y : State) {
    del_if_blocking_reachable {
        if y.deleted then { x.deleted := true; }
    }
}

Fix(
    Fix(expand_m_route) < del_blocking <
    Fix(del_if_blocking_reachable) <
    Fix(collapse_m_routes)
) < init_supervisor < Fix(expand_supervisor)
```

Listing 8: The Supervisory Controller Synthesis algorithm implemented in AuDaLa.