Formalisation of a new weak Semantics for AuDaLa

Gijs P. Leemrijse, Tom T.P. Franken[0000−0002−1168−5450], and Thomas Neele[0000−0001−6117−9129]

Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. The *Autonomous Data Language* (AuDaLa) is a recently introduced programming language and is supported by an operational semantics. This work presents a new operational semantics for AuDaLa with relaxed memory consistency and incoherent memory, with the goal of allowing more compiler optimisations. We show that both semantics are equivalent under the absence of read/write conflicts. Furthermore, we translate our operational semantics into an axiomatic memory consistency model and show how the memory operations of our semantics can be mapped onto the NVIDIA PTX virtual ISA.

1 Introduction

Now that single core performance improvements are running out of steam [39], there is increasing focus on multicore platforms such as GPUs. While orchestrating program execution on such a device is a complex task, many purpose-made languages and frameworks for parallel processing exist [26, 48, 51, 54]. 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) [20]. 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. Data elements can store references to other elements, enabling communication. AuDaLa presents a significant abstraction from parallel hardware and instead concentrates on data and its computations.

To be able to make use of existing methods and optimizations for concurrency and high-performance computing, we have created a compiler prototype from AuDaLa to CUDA [36]. However, the *operational semantics* defined in [20] is relatively *strict* when it comes to ordering of memory operations, compared to PTX [42] (the *instruction set architecture* underlying CUDA). We argue that AuDaLa can benefit from *weaker* semantics such that more optimisations can be performed, like reordering of memory operations [45]. This intuition is supported by test results from tests performed with the compiler [36]. In this work, we investigate how AuDaLa's semantics can be weakened.

In literature, two formalisation techniques for *memory consistency models* (MCM) are popular: the *axiomatic* [4,33,42,59] and the *operational* [4,25,31,46] 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, each execution is represented as a graph. 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. Legality of executions is captured in axioms. The operational approach is often more intuitive, while the axiomatic approach lends itself better to automated verification [3]. Ideally, both approaches are used and proven equivalent, as is done in [4, 16, 46, 49], leaving the choice up to the user.

In this work, we formalise a *new* 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 enables use to compare with the axiomatic MCM of PTX [42], which powers NVIDIA GPUs. Our contributions are:

– We provide an alternative *operational* semantics for AuDaLa with a relaxed and incoherent memory model. The new semantics aims for a more performant implementation by exploiting the reorderings that become available in the relaxed and incoherent memory model [45]. We prove that our new semantics behaves the same as the original semantics [20] under the absence of read/write data races (Thm. 1). This is thus an *SC-DRF guarantee* [1]: sequentially consistent assuming (read/write) data race freedom.
– We translate our operational semantics into an *axiomatic* MCM and show their equivalence (Thm. 2). Using Alloy [28], we formally check for a bounded program size its correspondence with PTX's MCM [42]. We show that the long-standing *out-of-thin-air problem* [14] 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 (Thm. 3).

While these contributions are in the scope of AuDaLa, the methodology of working with multiple semantics can be generalized to any language that is conceptually distant from the hardware, like Ly [55], swarm programming [2] and graph programming [61,62]. By linking multiple semantics together formally, such languages can aim for intuitive analysis and high performance with separate semantics.

The structure is as follows: Sect. 2 provides preliminaries on AuDaLa and memory models. Then, Sect. 3 presents our alternative operational semantics in detail. We formalise and check the correctness of our mapping from AuDaLa to PTX in Sect. 4. In Sect. 5, we highlight related work and we conclude in Sect. 6.

2 Preliminaries

In this section, we briefly introduce the background of our work: AuDaLa, memory consistency models, their axiomatic treatment and the out-of-thin-air problem. First, we introduce some notation. The set of finite A-sequences is A^*. The empty list is denoted by ε and $\ell_1; \ell_2$ is the concatenation of ℓ_1 and ℓ_2. We identify singleton lists with their element: $e_1; e_2$ is the list containing elements e_1 and e_2.

Given a function $f : A \to B$ and $a \in A, b \in B$, we define a *function update* as $f[a \mapsto b](a) = b$ and $f[a \mapsto b](x) = f(x)$ for all $x \neq a$. This is 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.

2.1 AuDaLa

The data structures in an AuDaLa program are defined using *structs*, of which the instances created at runtime are called *struct instances*. Each struct contains zero or more named data attributes called *parameters*. These have a type: `Bool`, `Int`, `Nat`, or a reference type to another struct. Furthermore, a struct contains definitions of operations called *steps*. Finally, an AuDaLa program has a *schedule* that forms the final program and dictates when steps are executed by all the struct instances. Sect. 3 provides an abstract syntax; for the full syntax, see [20].

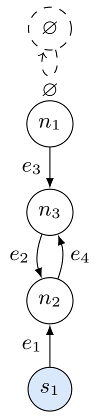
The semantics of an AuDaLa program are roughly as follows. When a step appears in the schedule, it is executed concurrently by all struct instances that contain a definition of that step. Only one step can be executed at a time: all struct instances must be finished with the previous step before computation can proceed past a barrier ('`<`' in the schedule). 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.

It is important to note that there are no loops in AuDaLa. Instead, AuDaLa relies on *fixpoints* in its schedule for iteration (keyword '`Fix`'). A fixpoint executes the schedule in its argument until a fixpoint is reached, that is, until no parameters change during an iteration. In this way, all parameters are treated as both the input and the output for the fixpoint. It follows that after the execution of a fixpoint, the entire parallel system is stable. The schedule supports nested fixpoints. At the start of the program, each struct 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 1. Fig. 1 implements the problem of reachability in a directed graph. We store for each node whether it is reachable (parameter `reachable` of `Node`); edges are simply combinations of a source and target node. The step `init` of struct `Node` constructs the graph the program operates on by using the constructors `Node()` and `Edge()`; in this case, only `s1` is initially reachable (Fig. 1b). By the schedule at the bottom, the `init` step is executed once (by the null instance of `Node`), after which the `reachability` step is executed in a fixpoint iteration. In each iteration the `propagate_reach` step is called and reachability is propagated once along all edges. The fixpoint will perform three iterations: two to propagate reachability to n_2 and n_3 and one iteration to conclude stability. □

```
struct Node (reachable : Bool) { // def. of Node struct
    init { // definition of the init step
        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);
    }
}
struct Edge (source : Node, target : Node) { // def. of Edge
    propagate_reach { // def. of the reachability step
        if source.reachable then {
            target.reachable := true;
        }
    }
}
init < Fix(propagate_reach) // The schedule
```

(a) An AuDaLa program for computing reachability.



(b) Instances after `init`.

Fig. 1: An example AuDaLa program specification. It computes the nodes reachable from `s1` via a parallel *breadth-first search*. In (b), blue indicates a node where `reachable` is true and the dashed node is a null instance.

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

(a) message passing

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

(b) Indep. reads of indep. writes [13]

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

(c) Load buffering [4]

Fig. 2: Litmus tests: `x` and `y` are variables in shared memory (initially 0) and `r1`...`r4` are thread-local registers.

2.2 Memory Consistency Models

A *memory consistency model* (MCM) defines what ordering and visibility guarantees exist between *memory operations* (*e.g.* loads and stores) in a shared-memory multiprocessor system. 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. We introduce three relevant MCMs and illustrate them with so called *litmus tests*.

A strong MCM is is *sequential consistency* (SC), where an execution appears as a single total order of reads and writes [34] and both execute atomically. Multiprocessor programs, therefore, behave as an interleaving of each program's instructions in program order. In the litmus test *message passing* of Fig. 2a, the printed value of `x` is always 1, since the instructions of the first thread may not be re-ordered under SC. Optimisations in compilers may also break SC [9,44,63]. If common sub-expression elimination is applied to eliminate the second read of `x` in `r1 = x; if (y == 1) { print x; }`, the print statement may yield 0, which is unexpected under SC. Enforcing SC carries a performance penalty [23,40].

The *Release/Acquire* (RA) MCM is intended for message passing, without having to resort to the expensive SC semantics. If an *acquiring* read `R` reads from a *releasing* write `W`, they *synchronise* [27]: operations before `W` in program order are visible to operations after `R` in program order. The litmus test in Fig. 2b allows the outcome `r1=1, r2=0, r3=1, r4=0` under RA but not under SC. Here, the middle two threads perceive a different order of writes to `x` and `y`.

The *relaxed* MCM only provides atomicity and a consistent total ordering of writes to *single* locations [27, 47]. Fig. 2c allows the outcome `r1 = r2 = 42` under relaxed consistency, showing that many operations may be re-ordered.

A related concept 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. Maintaining coherency between caches of different processors costs performance, so some systems are designed to allow incoherence.

Of the MCMs discussed in this section, the operational semantics of AuDaLa as introduced in [20] supports sequential consistency, while all MCMs introduced in this section are supported by PTX [47].

2.3 Out-of-thin-air problem

While weakening the MCM of a language allows more optimisations, weakening too much causes the MCM to allow nonsensical behaviour not observable in practice. This is called *out-of-thin-air* (OOTA) behaviour [9]. OOTA behaviour makes formal reasoning and compiler optimisation very difficult [8, 14, 56, 57]. 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 [29–31, 50].

The heart of the OOTA problem lies in the treatment of dependencies. For example in `r = y; if (r == 1)  x = 42;  else  x = 42;` , a compiler may optimise the if statement, because both branches are identical. This, however, breaks the dependency of `x = 42` on `r = y`, meaning that these instructions may now be reordered, resulting in different behaviour. The question is thus whether an MCM should include such dependencies or not. There are roughly three types of approaches to this [30]. With *syntactic dependencies* [14,32,33], all dependencies present in the source code are preserved. This disallows many compiler optimisations, but is useful for hardware memory models [4]. With *semantic dependencies* [17, 31, 35, 50], formalisations reason over all possible executions of a program in order to separate the true, semantic, dependencies from the false dependencies. Finally, 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 [9, 42, 59]. Each of these three approaches has its own drawbacks [30], and the OOTA problem is thus not fully solved yet.

Our definition of the operational semantics of AuDaLa effectively falls into the category of syntactic dependencies. In Sect. 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.

3 Operational Semantics

This section presents our new operational semantics of AuDaLa, improving on the original sequentially consistent semantics [20]. Our relaxed semantics are expected to offer more performance on modern hardware such as GPUs. We first introduce an abstract syntax for AuDaLa, show how it can be transformed into a sequence of atomic *commands* and give a formal definition of the semantic graph underlying an AuDaLa program.

The type of valid identifiers used within AuDaLa is denoted by $\mathbb{ID}$. For clarity, we refer to identifiers of steps with $\mathbb{ID}_{step}$, of structs with $\mathbb{ID}_{str}$ and of variables/parameters with $\mathbb{ID}_{var}$. 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 $\mathbb{T} \triangleq \{\texttt{Nat, Int, Bool}\} \cup \mathbb{ID}_{str}$. AuDaLa's abstract syntax is as follows.

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

$$\begin{aligned}
\mathbb{E} &::= \boldsymbol{Op}\ \mathbb{E} \times \mathbb{O} \times \mathbb{E} \mid \boldsymbol{Cons}\ \mathbb{ID}_{str} \times \mathbb{E}^* \mid \boldsymbol{Var}\ \mathbb{ID}^*_{var} \mid \boldsymbol{Not}\ \mathbb{E} \mid \boldsymbol{Lit}\ \mathbb{L} \\
\mathbb{S} &::= \boldsymbol{If}\ \mathbb{E} \times \mathbb{S}^* \times \mathbb{S}^* \mid \boldsymbol{Declare}\ \mathbb{T} \times \mathbb{ID}_{var} \times \mathbb{E} \mid \boldsymbol{Assign}\ \mathbb{ID}^*_{var} \times \mathbb{E} \\
\mathbb{L} &::= \boldsymbol{Nat}\ \mathbb{N} \mid \boldsymbol{Int}\ \mathbb{Z} \mid \boldsymbol{Bool}\ \mathbb{B} \mid \boldsymbol{Null}\ \mathbb{ID}_{str} \mid \boldsymbol{This} \\
\mathcal{SC} &::= \boldsymbol{Call}\ \mathbb{ID}_{str} \times \mathbb{ID}_{step} \mid \boldsymbol{CallAll}\ \mathbb{ID}_{step} \mid \boldsymbol{Fix}\ \mathcal{SC}^* \mid \boldsymbol{aFix}\ \mathcal{SC}^*
\end{aligned}$$

For brevity, we omit the parts of the AST that are irrelevant for our semantic discussion such as structure definitions. Furthermore, to keep the notation light, this abstract syntax is loosely typed and admits illegal constructions, which we assume do not occur.

We assume the function $Par \colon \mathbb{ID}_{str} \to \mathbb{ID}^*_{var}$, which is such that $Par(\vartheta)$ is the list of parameters of the struct type identified by $\vartheta \in \mathbb{ID}_{str}$. In addition, the function $TypeOf \colon \mathbb{ID}_{str} \times \mathbb{ID}_{var} \to \mathbb{T}$ is defined so that $TypeOf(\vartheta, v)$ is 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. We use **Call** to call a step for a specific struct type and **CallAll** to call a step for all the struct types in which it is defined. **aFix** is only used as a semantic symbol and is not present in the initial state of a program.

Within our semantics, we use *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 $\vartheta \in \mathbb{ID}_{str}$ 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 $\mathcal{L}^0 \subseteq \mathcal{L}$.

All *semantic values* are contained in the set $\mathcal{V}$, which contains 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 $defaultVal : \mathbb{T} \to \mathcal{V}$, such that $\mathit{defaultVal}(\texttt{Bool}) = \mathit{false}$, $\mathit{defaultVal}(T) = 0$ for $T \in \{\texttt{Nat}, \texttt{Int}\}$ and $\mathit{defaultVal}(T) = \ell^0_T$ for all $T \in \mathbb{D}_{str}$. We extract the semantic value from a literal $L \in \mathbb{L}$ via the function $\mathit{val}_\ell : \mathbb{L} \to \mathcal{V}$, where $\ell \in \mathcal{L}$, defined as:

$$\mathit{val}_\ell(\boldsymbol{Nat}(x)) = x \qquad \mathit{val}_\ell(\boldsymbol{Int}(x)) = x \qquad \mathit{val}_\ell(\boldsymbol{Bool}(x)) = x$$
$$\mathit{val}_\ell(\boldsymbol{Null}(\vartheta)) = \ell^0_\vartheta \qquad \mathit{val}_\ell(\boldsymbol{This}) = \ell$$

The semantics processes expressions much alike a stack machine does (although not necessarily in the same order). Therefore, we first translate AuDaLa code to atomic operations, called *semantic commands*.

Definition 2 (Semantic command). *The* command *type $\mathcal{C}$ is defined as:*

$$\begin{aligned}\mathcal{C} ::= \mathbf{rd}\ \mathbb{D}_{var} \mid \mathbf{wr}\ \mathbb{D}_{var} \mid \mathbf{wrP}\ \mathbb{D}_{var} \mid \mathbf{cons}\ \mathbb{D}_{str} \\ \mid \mathbf{if}\ (\mathcal{V} \cup \mathcal{C})^* \times (\mathcal{V} \cup \mathcal{C})^* \mid \mathbf{not} \mid \mathbf{op}\ \mathbb{O}\end{aligned}$$

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

The translation of AuDaLa code to commands is defined by the *interpretation function* $[\![\cdot]\!]_\ell$. The resulting list combines the commands and the values they operate on. The reasoning behind this is that our weak semantics does not necessarily execute commands in a sequential order, and we facilitate different execution orders by storing these values and commands in adjacent places.

Definition 3 (Interpretation function). *Let $v, v_1, \ldots, v_n \in \mathbb{D}_{var}$ 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, $\vartheta \in \mathbb{D}_{str}$ a struct type, $s \in \mathbb{S}$ a statement, $S, S_1, S_2 \in \mathbb{S}^*$ lists of statements, $\boldsymbol{op} \in \mathbb{O} \backslash \{\texttt{\&\&}, \texttt{||}\}$ an operator and let $\ell \in \mathcal{L}$ be a label. The* interpretation function *$[\![\cdot]\!]_\ell : \mathbb{S}^* \cup \mathbb{E} \to (\mathcal{V} \cup \mathcal{C})^*$ transforms statements and expressions into commands and values:*

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

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 4 (Struct instance). *A* struct instance *is a tuple* $\langle \vartheta, \chi, \xi \rangle$ *where* $\vartheta \in \mathbb{ID}_{str}$ *is the* struct type*,* $\chi \in (\mathcal{V} \cup \mathcal{C})^*$ *is a* value and command list *and* $\xi : \mathcal{L} \times \mathbb{ID}_{var} \to \mathcal{V} \cup \{\bot\}$ *is a* value cache*. The set of all possible struct instances is* $\mathcal{S}$*. 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 5 (State). *A* state *is a tuple* $\langle sc, \sigma, \mu, \iota, \delta \rangle$*, where:*

- $sc \in \mathcal{SC}^*$ *is a* schedule*,*
- $\sigma : \mathcal{L} \to \mathcal{S} \cup \{\bot\}$ *is a* struct environment*,*
- $\mu : \mathcal{L} \times \mathbb{ID}_{var} \to 2^{\mathcal{V} \times \mathcal{L}}$*, is a* global memory *of parameter values written during the current step paired with their writer's label,*
- $\iota : \mathcal{L} \times \mathbb{ID}_{var} \to \mathcal{V}$*, 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*.*

We 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* $\mu_\emptyset$*.*

All struct instances are uniquely labelled and contained within the labelled struct environment σ. During a step, struct instances communicate through the global memory μ, 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 6 (Initial state). *Let* $\mathcal{P}$ *be an AuDaLa program,* $\Theta \subseteq \mathbb{ID}_{str}$ *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:*

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

A command can only be executed if all preceding commands that reference the same parameter have been executed. This is expressed in the *Refs* predicate.

Definition 7 (The *Refs* predicate). *The predicate Refs* : $(\mathcal{V} \cup \mathcal{C})^* \times \mathcal{L} \times \mathbb{ID}_{var} \to \mathbb{B}$ *is defined as:*

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

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 $\mathbf{if}(\chi_2, \chi_3)$ are present in χ, *Refs* will recursively check both χ_2 and χ_3.

We next define the transition relation $\Rightarrow$ between states by listing a set of derivation rules in the following paragraphs. We use $\mathit{val} \in \mathcal{V}$, $v \in \mathbb{ID}_{var}$, $\chi_1, \chi_2 \in (\mathcal{V} \cup \mathcal{C})^*$, $\ell \in \mathcal{L}$ and $\mathit{op} \in \mathbb{O}$ as variables, possibly with subscripts or superscripts.

Computation on values is done via operators: the **not** command simply negates its operand and the $\mathbf{op}(\mathit{op})$ command applies the binary operator op to its two operands.

$$
\textbf{Not}\ \frac{\sigma(\ell) = \langle \vartheta, \chi_1; \mathit{val}; \mathbf{not}; \chi_2, \xi \rangle}{\langle sc, \sigma, \mu, \iota, \delta \rangle \Rightarrow \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; \neg \mathit{val}; \chi_2, \xi \rangle], \mu, \iota, \delta \rangle}
$$

$$
\textbf{Op}\ \frac{\sigma(\ell) = \langle \vartheta, \chi_1; \mathit{val}_a; \mathit{val}_b; \mathbf{op}(\mathit{op}); \chi_2, \xi \rangle}{\langle sc, \sigma, \mu, \iota, \delta \rangle \Rightarrow \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; (\mathit{val}_a\ \mathit{op}\ \mathit{val}_b); \chi_2, \xi \rangle], \mu, \iota, \delta \rangle}
$$

A read command can retrieve a value either from the initial value function ι, the value cache ξ or the global memory μ. This choice is non-deterministic. Reads from the same location are executed in program order, by the *Refs* function.

$$
\textbf{Rd-Init}\ \frac{\sigma(\ell) = \langle \vartheta, \chi_1; \ell'; \mathbf{rd}(v); \chi_2, \xi \rangle \qquad \xi(\ell', v) = \bot \qquad \neg \mathit{Refs}(\chi_1, \ell', v)}{\langle sc, \sigma, \mu, \iota, \delta \rangle \Rightarrow \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; \iota(\ell', v); \chi_2, \xi \rangle], \mu, \iota, \delta \rangle}
$$

$$
\textbf{Rd-Int}\ \frac{\sigma(\ell) = \langle \vartheta, \chi_1; \ell'; \mathbf{rd}(v); \chi_2, \xi \rangle \qquad \xi(\ell', v) \neq \bot \qquad \neg \mathit{Refs}(\chi_1, \ell', v)}{\langle sc, \sigma, \mu, \iota, \delta \rangle \Rightarrow \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; \xi(\ell', v); \chi_2, \xi \rangle], \mu, \iota, \delta \rangle}
$$

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

When writing, we identify three variants: writing local variables, null-instance parameters and parameters of regular instances. Local variables are stored in the instance's value cache ξ. Writes to null-instance parameters are treated as a no-op and, effectively, skipped. Regular parameter writes are split in two steps: first, the old value is 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.

$$\textbf{Wr-Local}\frac{\sigma(\ell) = \langle \vartheta, \chi_1; val; \ell; \mathbf{wr}(v); \chi_2, \xi\rangle \qquad v \notin Par(\vartheta) \qquad \neg Refs(\chi_1, \ell, v)}{\langle sc, \sigma, \mu, \iota, \delta\rangle \Rightarrow \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; \chi_2, \xi[(\ell, v) \mapsto val]\rangle], \mu, \iota, \delta\rangle}$$

$$\textbf{Wr-Null}\frac{\begin{array}{c}\sigma(\ell) = \langle \vartheta, \chi_1; val; \ell'; \mathbf{wr}(v); \chi_2, \xi\rangle \\ \sigma(\ell') = \langle \vartheta', \chi', \xi'\rangle \qquad v \in Par(\vartheta') \qquad \ell' \in \mathcal{L}^0\end{array}}{\langle sc, \sigma, \mu, \iota, \delta\rangle \Rightarrow \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; \chi_2, \xi\rangle], \mu, \iota, \delta\rangle}$$

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

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

Newly-constructed instances receive a fresh label and their parameters are populated atomically. Furthermore, creating instances clears the stability stack.

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

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.

$$\textbf{If-True}\frac{\sigma(\ell) = \langle \vartheta, \chi_1; true; \mathbf{if}(S_1, S_2); \chi_2, \xi\rangle}{\langle sc, \sigma, \mu, \iota, \delta\rangle \Rightarrow \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; S_1; \chi_2, \xi\rangle], \mu, \iota, \delta\rangle}$$

$$\textbf{If-False}\frac{\sigma(\ell) = \langle \vartheta, \chi_1; false; \mathbf{if}(S_1, S_2); \chi_2, \xi\rangle}{\langle sc, \sigma, \mu, \iota, \delta\rangle \Rightarrow \langle sc, \sigma[\ell \mapsto \langle \vartheta, \chi_1; S_2; \chi_2, \xi\rangle], \mu, \iota, \delta\rangle}$$

When only values remain in each instance's value and command list, a step is *done*. We define the predicates $End(\sigma) = \forall \ell.\, \sigma(\ell) = \langle \vartheta, \chi, \xi\rangle \Rightarrow \chi \in \mathcal{V}^*$ and $Done(\sigma, \mu) = End(\sigma) \wedge \mu = \mu_\emptyset$. When a step is finished executing ($End(\sigma)$ holds), all value caches are cleared and the initial value function is non-deterministically updated with a *final parameter value* $fin(\ell, v)$ for each parameter (ℓ, v) (or it retains its value if no write to (ℓ, v) occurred).

$$\textbf{Finish}\frac{\begin{array}{c}End(\sigma) \qquad \mu \neq \mu_\emptyset \\ W(\ell, v) = \{\xi(\ell, v) \mid \sigma(\ell') = \langle \vartheta, \chi, \xi\rangle\} \setminus \{\bot\} \\ fin(\ell, v) \in W(\ell, v) \vee (W(\ell, v) = \emptyset \wedge fin(\ell, v) = \iota(\ell, v))\end{array}}{\begin{array}{l}\langle sc, \sigma, \mu, \iota, \delta\rangle \Rightarrow \langle sc, \sigma[\{\ell \mapsto \langle \vartheta, \chi_\ell, \xi_\bot\rangle \mid \sigma(\ell) = \langle \vartheta, \chi_\ell, \xi_\ell\rangle\}], \\ \qquad \mu_\emptyset, \iota[\{(\ell, v) \mapsto fin(\ell, v) \mid (\ell, v) \in \mathcal{L} \times \mathbb{ID}_{var}\}], \delta\rangle\end{array}}$$

We denote the statements of step $F \in \mathbb{ID}_{step}$ of struct $\vartheta \in \Theta$ by $S^F_\vartheta \in \mathbb{S}^*$. If ϑ does not contain F, $S^F_\vartheta = \varepsilon$. When step F is called for ϑ, all ϑ instances' command lists are set to $[\![S^F_\vartheta]\!]_\ell$. Step F can be called for a single type ϑ or for all $\vartheta \in \Theta$.

$$\textbf{Step}\,\frac{Done(\sigma,\mu)}{\begin{array}{l}\langle \boldsymbol{Call}(\vartheta,F); sc,\sigma,\mu,\iota,\delta\rangle \Rightarrow \\ \qquad \langle sc, \sigma[\{\ell \mapsto \langle \vartheta, [\![S^F_\vartheta]\!]_\ell, \xi_\ell\rangle \mid \sigma(\ell) = \langle \vartheta, \chi_\ell, \xi_\ell\rangle\}], \mu, \iota, \delta\rangle\end{array}}$$

$$\textbf{StepAll}\,\frac{Done(\sigma,\mu)}{\begin{array}{l}\langle \boldsymbol{CallAll}(F); sc,\sigma,\mu,\iota,\delta\rangle \Rightarrow \\ \qquad \langle sc, \sigma[\{\ell \mapsto \langle \vartheta_\ell, [\![S^F_{\vartheta_\ell}]\!]_\ell, \xi_\ell\rangle \mid \sigma(\ell) = \langle \vartheta_\ell, \chi_\ell, \xi_\ell\rangle\}], \mu, \iota, \delta\rangle\end{array}}$$

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.

$$\textbf{Fix-Init}\,\frac{Done(\sigma,\mu)}{\langle \boldsymbol{Fix}(sc_1); sc_2,\sigma,\mu,\iota,\delta\rangle \Rightarrow \langle sc_1; \boldsymbol{aFix}(sc_1); sc_2,\sigma,\mu,\iota,\delta; true\rangle}$$

$$\textbf{Fix-Iter}\,\frac{Done(\sigma,\mu)}{\langle \boldsymbol{aFix}(sc_1); sc_2,\sigma,\mu,\iota,\delta; false\rangle \Rightarrow \langle sc_1; \boldsymbol{aFix}(sc_1); sc_2,\sigma,\mu,\iota,\delta; true\rangle}$$

$$\textbf{Fix-Resolve}\,\frac{Done(\sigma,\mu)}{\langle \boldsymbol{aFix}(sc_1); sc_2,\sigma,\mu,\iota,\delta; true\rangle \Rightarrow \langle sc_2,\sigma,\mu,\iota,\delta\rangle}$$

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 semantics:

Definition 8 (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 Def. 5, $\Rightarrow$ is the transition relation defined by the derivation rules as given above, and $P^0_\mathcal{P}$ is the initial state of $\mathcal{P}$ (Def. 6).*

The relation between these weak semantics and the sequentially consistent graph semantics defined by Franken *et al.* [20, Def. 7] is given in the following theorem:

Theorem 1. *For any AuDaLa program $\mathcal{P}$ without read-write data races, the sequentially consistent graph semantics for a program $\mathcal{P}$ and the weak graph semantics for a program according to Def. 8 are stutter trace equivalent [6, Def. 7.89].*

Proof (outline). We prove this by defining a labelling $\mathcal{C}$ for both semantics which makes the contents of the states comparable. Then, with $\mathcal{P}$ some program, $(\!|\mathcal{P}|\!)_{SC}$ being $\mathcal{P}$'s sequentially consistent semantics and $(\!|\mathcal{P}|\!)_{Wk}$ being $\mathcal{P}$'s weak semantics, we perform induction on the length of initial traces for $(\!|\mathcal{P}|\!)_{SC}$ and $(\!|\mathcal{P}|\!)_{Wk}$. Using this induction, we prove that for any initial trace in $(\!|\mathcal{P}|\!)_{SC}$ there exists a $\mathcal{C}$-stutter-equivalent initial trace for $(\!|\mathcal{P}|\!)_{Wk}$ and vice versa. In the proof, we consider only programs which takes the interpreter differences between the semantics into account; for programs that do not, we provide syntactic transformations to programs that do. $\square$

4 Axiomatic memory semantics

In this section, we compare AuDaLa's graph semantics with that of NVIDIA's PTX ISA [47], which has been formally defined as an axiomatic model [41, 42], to see whether AuDaLa's supports the Release/Acquire and Relaxed MCMs. We first transform AuDaLa's operational semantic model into an axiomatic model. Then, we compare AuDaLa's and PTX's axiomatic models with the Alloy analyser [28].

We adopt some terminology from [46]. Our axiomatic model reasons not on the program source code, but on *candidate executions* expressed as graphs. The basis of a candidate execution is formed by a *pre-execution*: a straight-line execution in which all addresses and control flow have been resolved. Therefore, different pre-executions represent different runs of the same program. In addition, a pre-execution models dependencies and threads.

A pre-execution does not contain information on how reads and writes interact; this is contained in an additional set of edges called the *execution witness*. An execution witness completes a pre-execution to form a candidate execution. Which candidate executions are *legal* is formalised in a set of *axioms*.

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:

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

4.1 AuDaLa candidate executions

In the graph that forms a candidate execution, 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.

To model the nodes of AuDaLa execution graphs, we define the following sets: `Loc` models the label-variable pairs (ℓ, v) that are read from or written to

Table 1: Overview of relations in AuDaLa pre-executions and witnesses.

	notation	name	subset of	description/definition
pre-execution	$\texttt{nxt}$	next command	$\texttt{Cmd} \times \texttt{Cmd}$	sequence of commands
	$\texttt{start}$	first command	$\texttt{Inst} \times \texttt{Cmd}$	first command of an instance
	$\texttt{loc}$	location	$\texttt{Cmd} \times \texttt{Loc}$	location of read or write
	$\texttt{dep}$	dependency	$\texttt{RdCmd} \times \texttt{Cmd}$	read dependency
	$\texttt{po}$	program order	$\texttt{Cmd} \times \texttt{Cmd}$	$\texttt{nxt}^+$
	$\texttt{inst}$	instance	$\texttt{Cmd} \times \texttt{Inst}$	$(\texttt{start}; \texttt{nxt}^*)^{-1}$
	$\texttt{same-loc}$	same location	$\texttt{Cmd} \times \texttt{Cmd}$	$\texttt{loc}; \texttt{loc}^{-1}$
	$\texttt{same-inst}$	same instance	$\texttt{Cmd} \times \texttt{Cmd}$	$\texttt{inst}; \texttt{inst}^{-1}$
	$\texttt{po-loc}$	pr. order by loc.	$\texttt{Cmd} \times \texttt{Cmd}$	$\texttt{po} \cap \texttt{same-loc}$
witness	$\texttt{rf}$	reads from	$\texttt{WrCmd} \times \texttt{RdCmd}$	origin of the value in a read op.
	$\texttt{co}$	coherence	$\texttt{WrCmd} \times \texttt{WrCmd}$	order of writes within an instance
	$\texttt{fr}_{\texttt{base}}$	from reads base	$\texttt{RdCmd} \times \texttt{WrCmd}$	$(\texttt{rf} \cap \texttt{same-inst})^{-1}; \texttt{co}^+$
	$\texttt{fr}_{\texttt{init}}$	from reads initial	$\texttt{RdCmd} \times \texttt{WrCmd}$	$((\texttt{RdCmd} \setminus (\texttt{WrCmd}; \texttt{rf})) \times \texttt{WrCmd}) \cap \texttt{same-inst} \cap \texttt{same-loc}$
	$\texttt{fr}$	from reads	$\texttt{RdCmd} \times \texttt{WrCmd}$	$\texttt{fr}_{\texttt{base}} \cup \texttt{fr}_{\texttt{init}}$

in the operational semantics. The set $\texttt{Inst}$ models the struct instances. Since we are interested in the memory model, we only model read and write commands, contained in the disjoint sets $\texttt{RdCmd}$ and $\texttt{WrCmd}$, respectively. We define $\texttt{Cmd} \triangleq \texttt{RdCmd} \cup \texttt{WrCmd}$ as the set of all commands. We do not consider the working of the schedule here, *i.e.*, the operational rules with the premise $Done(\sigma, \mu)$ also fall outside the axiomatic model.

Table 1 gives an overview of the relations within a pre-execution. Each struct instance carries a list χ of commands corresponding to a single step call. We model this in the *next command* relation $\texttt{nxt} \subseteq \texttt{Cmd} \times \texttt{Cmd}$, *e.g.*, $(a, b), (b, c) \in \texttt{nxt}$ is the list $a; b; c$. The relation $\texttt{start} \subseteq \texttt{Inst} \times \texttt{Cmd}$ associates each instance to the first command in its list. Each $\texttt{Cmd}$ writes or reads exactly one $\texttt{Loc}$, this is captured in $\texttt{loc} \subseteq \texttt{Cmd} \times \texttt{Loc}$. When (the execution of) a command depends on an earlier read command in the same struct instance, we model this via the $\texttt{dep} \subseteq \texttt{RdCmd} \times \texttt{Cmd}$ relation. We introduce a few derived relations to represent things such as "which instance does a command belong to" ($\texttt{inst}$) and "which commands operate on the same (memory) locations" ($\texttt{same-loc}$); see again Table 1.

Definition 9 (The $\texttt{WELLFORMED\_PREEX}$ predicate). *A pre-execution is* well-formed *iff the following all hold:*

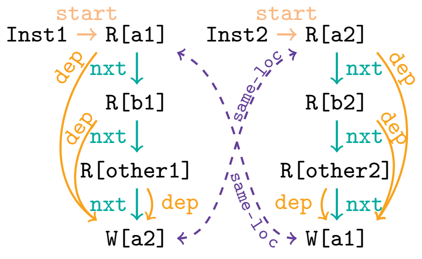
1. $\textbf{lone}(\texttt{nxt}) \wedge \textbf{lone}(\texttt{nxt}^{-1})$, *commands have at most one successor/predecessor.*
2. $\textbf{injective}(\texttt{start}, \texttt{Inst}, \texttt{Cmd})$, *each instance has exactly one* $\texttt{start}$ *successor and each command has at most one* $\texttt{start}$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

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

(a) AuDaLa source.

```
ℓ1;           ℓ2;
rd(a);        rd(a);
ℓ1;           ℓ2;
rd(b);        rd(b);
op(+);        op(+);
ℓ1;           ℓ2;
rd(other);    rd(other);
wr(a)         wr(a)
```

(b) commands



(c) pre-execution graph

Fig. 3: A program in three abstractions. We denote the locations in square brackets (instead of using Loc and loc). Dashed arrows indicate derived relations and we omit po, inst, and same-inst for clarity. The po-loc relation is empty.

6. *Cmd;* $\texttt{loc} = \texttt{Loc}$*, all locations are referenced (constrains the state space).*

Example 2. Fig. 3 presents the same program thrice: as AuDaLa source code, semantic commands, and as pre-execution. There are two S1 instances, 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. □

An execution witness captures the origin of reads in the *reads-from* relation rf: for every write, it provides all the reads that read from it. Some read operations read from the initial value function ι and do not occur in rf. The *coherence* relation co relates a WrCmd to another WrCmd that overwrites its value in the instance's value cache ξ. Our co relation is somewhat unconventional compared to other MCMs. Because a write can only be overwritten in the (local) cache, but not in the global memory, co only relates writes of the same instance. A derived relation is the *from-reads* relation fr, which indicates, given a read operation, which writes occur at a later moment. If R reads from W_1 and W_1 is overwritten by W_2 (*i.e.*, $(\texttt{W\_1}, \texttt{W\_2}) \in \texttt{co}$), then W_2 must have occurred after R so $(\texttt{R}, \texttt{W\_2}) \in \texttt{fr}$. This pattern is captured in $\texttt{fr}_{\texttt{base}}$. If R reads from the initial value function, then successive writes are captured in $\texttt{fr}_{\texttt{init}}$; see Table 1.

Definition 10 (The WELLFORMED_WITNESS predicate). *An execution witness is* well-formed *iff the following all hold:*

1. $\texttt{rf} \cup \texttt{co} \subseteq \texttt{same-loc}$*, reading/overwriting only occurs within a location.*
2. $\textbf{lone}(\texttt{rf}^{-1})$*, a read operation can read from at most one write operation.*
3. $\texttt{co} \subseteq \texttt{same-inst}$*, overwriting can only occur within the same instance.*
4. $\forall l \in \texttt{Loc}, i \in \texttt{Inst}.\, \textbf{total}(\texttt{co}, \texttt{WrCmd} \cap l; \texttt{loc}^{-1} \cap i; \texttt{inst}^{-1})$*, for every pair of location and instance, all corresponding writes are totally ordered by* co*.*

4.2 Deriving AuDaLa's axioms

In our axiomatic model, the RdCmd and WrCmd nodes correspond to the execution of the operational rules **Rd-Init**, **Rd-Int**, **Rd-Ext** and **Wr-II** for their respective instances. A derivation order is *legal* if the premise of each derivation rule is satisfied when following that order. Given a candidate execution, we can reason about the order of its nodes enforced by the operational rules. We capture this in the so called *semantic ordering* $\texttt{sem} \subseteq \texttt{Cmd} \times \texttt{Cmd}$.

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) [4, 33, 42]. Since AuDaLa's MCM is simple, we have only one axiom which forbids those executions in which an inconsistency between 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 2. *The semantic ordering of legal derivations is equivalent to the union of the program order by the location, reads-from, coherence, from-reads and dependency relations. Formally:* $\texttt{sem} = \texttt{po-loc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr} \cup \texttt{dep}$

Proof (outline). We separately consider $\texttt{po-loc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr} \cup \texttt{dep} \subseteq \texttt{sem}$ and $\texttt{po-loc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr} \cup \texttt{dep} \supseteq \texttt{sem}$. To prove the first, let $r \in \{\texttt{po-loc}, \texttt{rf}, \texttt{co}, \texttt{fr}, \texttt{dep}\}$. We take an arbitrary pair $(c_1, c_2) \in r$ and show that $(c_1, c_2) \in \texttt{sem}$. It turns out that all relations are necessary for the operational semantics to lead to a legal execution order. For example, the operational semantics defines that a value read from global memory or the value cache must have been generated from some write command executed before, and therefore, $\texttt{rf} \subseteq \texttt{sem}$.

To prove the second, we do the opposite. We take an arbitrary pair $(c_1, c_2) \in \texttt{sem}$ and, after some case distinctions, find an $r \in \{\texttt{po-loc}, \texttt{rf}, \texttt{co}, \texttt{fr}, \texttt{dep}\}$ such that $(c_1, c_2) \in r$. For the pairs (c_1, c_2) in $\texttt{sem}$, either the resolution of c_1 would be different if c_2 had been resolved before c_1, in which case $(c_1, c_2) \in \texttt{fr}$, or c_1 blocks c_2 in one of the following ways:

- c_1 belongs to the same expressions as c_2, then $(c_1, c_2) \in \texttt{dep}$;
- c_1 writes the value read during the resolution of c_2, then $(c_1, c_2) \in \texttt{rf}$; or
- c_1 blocks the resolution of c_2 through the *Refs*-predicate, then it holds that either $(c_1, c_2) \in \texttt{po-loc}$ or $(c_1, c_2) \in \texttt{dep}$. $\square$

Legal executions Using sem and CONSISTENCY, we define the predicate for a *legal AuDaLa execution*:

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

Given a well-formed pre-execution for which LEGAL_EXEC 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.

Observe that $\texttt{LEGAL\_EXEC} \wedge \texttt{WELLFORMED\_PREEX}$ implies $\texttt{co} \subseteq \texttt{po-loc}$, illustrating that our `co` is unconventional.

Since the interpretation function of the operational semantics (Def. 3) does not perform optimisations, any dependencies in the AuDaLa source code are also present in the semantic command list. An optimising compiler, however, might perform some optimisations and break some dependencies, but specifying this exactly is difficult (see Sect. 2.3). PTX also suffers from this problem. Until consensus is reached on how to treat dependencies in an MCM, the PTX memory model chooses to not respect any dependencies, except that reads can not read from writes that (transitively) depend 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 be able to compare both memory models modulo dependencies, we provide an alternative, weaker, legal execution predicate $\texttt{LEGAL\_EXEC}^{\star}$. It uses a weakened consistency axiom $\texttt{CONSISTENCY}^{\star}$ and introduces a new axiom `NO_THIN_AIR`:

$$\begin{aligned} \texttt{CONSISTENCY}^{\star} &\triangleq \textbf{acyclic}(\texttt{po-loc} \cup \texttt{rf} \cup \texttt{co} \cup \texttt{fr}) \\ \texttt{NO\_THIN\_AIR} &\triangleq \textbf{acyclic}(\texttt{rf} \cup \texttt{dep}) \\ \texttt{LEGAL\_EXEC}^{\star} &\triangleq \texttt{WELLFORMED\_WITNESS} \wedge \texttt{CONSISTENCY}^{\star} \wedge \texttt{NO\_THIN\_AIR} \end{aligned}$$

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 axiom $\texttt{CONSISTENCY}^{\star}$ is very similar to `CONSISTENCY`, except we exclude `dep` from `sem`.

Example 3. In Fig. 4 two instances execute two different programs represented as AuDaLa source code. Assume a shared reference to some `gm` struct with integer parameters `x` and `y`, both initially `0`. For brevity, we omit the reads of the `gm` reference so that the expression `gm.y` is a single read 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 Fig. 4b). The result `r1 = 42, r2 = 42` (depicted in Fig. 4c) is illegal for `LEGAL_EXEC` but legal for $\texttt{LEGAL\_EXEC}^{\star}$. It contains a `sem` cycle violating `CONSISTENCY`, but not $\texttt{CONSISTENCY}^{\star}$. One could argue that $\texttt{LEGAL\_EXEC}^{\star}$ 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. □

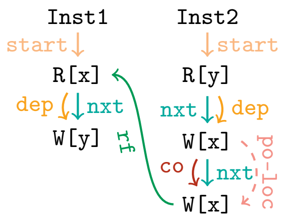
4.3 Mapping to and from PTX

We next compare the axiomatic MCMs of AuDaLa and PTX by mapping AuDaLa pre-executions onto PTX pre-executions and mapping PTX execution witnesses onto AuDaLa execution witnesses and checking their legality. This method

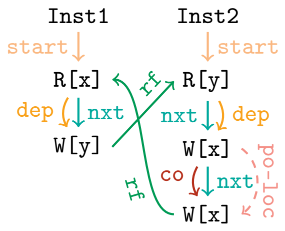
```
Inst1
Int r1 := gm.x;
gm.y := r1;

Inst2
Int r2 := gm.y;
gm.x := r2;
gm.x := 42;
```

(a) AuDaLa program.



(b) A legal execution.



(c) Controversial execution.

Fig. 4: A simple test program with two corresponding execution graphs. Fig. 4b shows a legal execution and Fig. 4c 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.

follows standard practice for this type of proof [10, 42, 60]. We use the formal PTX MCM provided in [42], we discuss its most relevant concepts below.

In the PTX model, the sets corresponding to Cmd, Loc and Inst are Op, Addr and Thread 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.

The non-derived relations in PTX are $\texttt{nxt}_{ptx} \subseteq \texttt{Op} \times \texttt{Op}$, $\texttt{start}_{ptx} \subseteq \texttt{Thread} \times \texttt{Op}$, $\texttt{loc}_{ptx} \subseteq \texttt{Op} \times \texttt{Addr}$ and $\texttt{dep}_{ptx} \subseteq \texttt{RdOp} \times \texttt{Op}$. We establish a mapping between AuDaLa and PTX pre-executions via the relations $\texttt{cmd-map} \subseteq \texttt{Cmd} \times \texttt{Op}$, $\texttt{loc-map} \subseteq \texttt{Loc} \times \texttt{Addr}$ and $\texttt{inst-map} \subseteq \texttt{Inst} \times \texttt{Thread}$. This mapping follows the MAP_PREEX predicate:

Definition 11 (The MAP_PREEX predicate). *A pre-execution is* correctly mapped *iff all of the following hold:*

- **bijective**(cmd-map, Cmd, Op), **bijective**(loc-map, Loc, Addr), *and* **bijective**(inst-map, Inst, Thread)*: mappings are one-to-one.*
- $\texttt{loc}_{ptx} = \texttt{cmd-map}^{-1}; \texttt{loc}; \texttt{loc-map}$*, 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.*

For execution witnesses, we reverse the direction of the mapping: we map a PTX execution witness back to an AuDaLa execution witness. The PTX counterparts of $\texttt{rf}$ and $\texttt{co}$ are denoted by $\texttt{rf}_{ptx} \subseteq \texttt{WrOp} \times \texttt{RdOp}$ and $\texttt{co}_{ptx} \subseteq \texttt{WrOp} \times \texttt{WrOp}$ respectively. PTX's analogue to `same-inst` is `same-thread`.

Definition 12 (The `MAP_WITNESS` predicate). *A PTX execution witness is* correctly mapped *iff the following all hold:*

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

4.4 Empirical correctness results

We formalise our model in the Alloy analyser [28], a relational modelling tool capable of reasoning over graphs, often used to analyse MCMs [24, 41–43, 59, 60]. Alloy's relational algebra is similar to our definitions. Our complete Alloy formalisation can be found on [37], a reproducibility package on [38].

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 correctness for our mapping:

Theorem 3 (Correctness of `MAP_PREEX`). *Given a valid AuDaLa pre-execution* $p_{\texttt{AuDaLa}}$*, suppose we map* $p_{\texttt{AuDaLa}}$ *onto a PTX pre-execution* $p_{\texttt{PTX}}$*, and suppose* $w_{\texttt{PTX}}$ *is a legal execution witness of* $p_{\texttt{PTX}}$*. If we interpret* $w_{\texttt{PTX}}$ *as an AuDaLa execution witness* $w_{\texttt{AuDaLa}}$*, then* $w_{\texttt{AuDaLa}}$ *is a legal execution witness of* $p_{\texttt{AuDaLa}}$*. Equivalently:*

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

We test Thm. 3 in Alloy 6 [28] using Z3 [19] version 4.13 as SAT solver and report its runtime on a system with an Intel i9-13900K CPU and 32GB RAM. We do this for small (4-8) pre-execution sizes, larger pre-executions quickly become intractable. However, some of the most important litmus tests for GPUs can be defined using six or fewer events [52]. 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 Thm. 3. 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 2.The table suggests that it is safe to compile to the Release/Acquire memory order. We discuss the counter-examples for the Relaxed memory order below. If we model modulo dependencies (**Rlx***), no counterexamples are found.

Two counter-examples of five events are found for Thm. 3 under the **Rlx** configuration; the larger counter-examples are variations of these. One counterexample is already discussed in Sect. 4.2 and shown in Fig. 4c. The other counterexample is similar.

Table 2: The verdict and solve time for Thm. 3 in three configurations: Release/Acquire semantics, Relaxed semantics, and Relaxed semantics with the `LEGAL_EXEC`* predicate. Solve time has been rounded up to the nearest second.

Size	4	5	6	7	8
RelAcq	✓(<1s)	✓(7s)	✓(1m6s)	✓(17m18s)	✓(6h27m40s)
Rlx	✓(<1s)	✗ (<1s)	✗ (12s)	✗ (12s)	✗ (<1s)
Rlx*	✓(<1s)	✓(2s)	✓(18s)	✓(4m37s)	✓(1h39m40s)

4.5 Discussion

We have demonstrated that the PTX MCM is susceptible to the OOTA problem. This makes it difficult to conclusively state the relation between AuDaLa and PTX 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 required are 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.

5 Related Work

Our approach of using Alloy to analyse MCMs is related to [60], which uses Alloy to derive litmus tests that distinguish MCMs. The PTX MCM is given in [42] and compared with the C++ MCM of [33]. Their approach of forward mapping pre-executions and reverse mapping execution witnesses is similar to ours, but they use both Alloy and Coq [53]. No operational model of the PTX MCM is given, however. A generic framework for MCMs, both axiomatic and operational, is given in [4], including a Coq proof that these models correspond.

The literature on relaxed memory semantics can be divided according to the approach to dependency tracking, using either strong dependencies [14,32,33] or no dependencies [11,42,59] (but allowing OOTA behaviour). Our two axiomatic models fall in either category. Many works aim to find a solution to the OOTA problem, such as *promosing semantics* [31] or *modular relaxed dependencies* [50].

Several parallel languages have similarities with AuDaLa. The *CHemical Abstract Machine (CHAM)* [12] is based on the *Γ-calculus* [7] in which *molecules* react with each other to form new molecules. Reactions can happen in parallel, making the Γ-calculus suitable for a GPU implementation [21, 22]. AuDaLa also has similarities to message-passing languages, such as the ParCel languages [15, 58], and *actor languages*, such as Ly [55] and A-NETL [5]. For a full overview, see [20].

6 Conclusion

We have investigated a relaxed memory semantics for AuDaLa based on a new operational semantics and a corresponding axiomatic semantics. Experiments with Alloy indicate that this semantics can be mapped to PTX's Release/Acquire and Relaxed operations (assuming no dependencies are broken by the compiler). Using this foundation, we aim to build a compiler from AuDaLa to CUDA, enabling its execution on GPUs.

References

1. Adve, S.V., Hill, M.D.: Weak ordering—a new definition. ACM SIGARCH Computer Architecture News **18**(2SI), 2–14 (1990). `https://doi.org/10.1145/325096.325100`
2. Aguzzi, G., Casadei, R., Viroli, M.: MacroSwarm: A Field-Based Compositional Framework for Swarm Programming. In: Coordination Models and Languages. pp. 31–51. Springer Nature Switzerland (2023). `https://doi.org/10.1007/978-3-031-35361-1_2`
3. Alglave, J., Kroening, D., Tautschnig, M.: Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. vol. 8044, pp. 141–157. Springer (2013). `https://doi.org/10.1007/978-3-642-39799-8_9`
4. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Transactions on Programming Languages and Systems **36**(2), 1–74 (2014). `https://doi.org/10.1145/2627752`, publisher: ACM
5. Baba, T., Yoshinaga, T.: A-NETL: a language for massively parallel object-oriented computing. In: PMMPC 1995. pp. 98–105. IEEE (1995). `https://doi.org/10.1109/PMMPC.1995.504346`
6. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
7. Banâtre, J.P., Métayer, D.L.: The gamma model and its discipline of programming. Science of Computer Programming **15**(1), 55–77 (1990). `https://doi.org/10.1016/0167-6423(90)90044-E`
8. Batty, M., Dodds, M., Gotsman, A.: Library Abstraction for C/C++ Concurrency. In: POPL 2013. pp. 235–248. ACM (Jan 2013). `https://doi.org/10.1145/2429069.2429099`
9. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The Problem of Programming Language Concurrency Semantics. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 283–307. Springer (2015). `https://doi.org/10.1007/978-3-662-46669-8_12`
10. Batty, M., Memarian, K., Owens, S., Sarkar, S., Sewell, P.: Clarifying and Compiling C/C++ Concurrency: From C++11 to POWER. In: POPL 2012. pp. 509–520. ACM (Jan 2012). `https://doi.org/10.1145/2103656.2103717`
11. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ Concurrency. In: POPL 2011. pp. 55–66. ACM (Jan 2011). `https://doi.org/10.1145/1926385.1926394`
12. Berry, G., Boudol, G.: The chemical abstract machine. Theoretical Computer Science **96**(1), 217–248 (1992). `https://doi.org/10.1016/0304-3975(92)90185-I`

13. Boehm, H.J., Adve, S.V.: Foundations of the C++ Concurrency Memory Model. In: PLDI 2008. pp. 68–78. ACM (Jun 2008). https://doi.org/10.1145/1375581.1375591
14. Boehm, H.J., Demsky, B.: Outlawing Ghosts: Avoiding out-of-Thin-Air Results. In: MSPC 2014. pp. 7:1–7:6. ACM (2014). https://doi.org/10.1145/2618128.2618134
15. Cagnard, P.J.: The ParCeL-2 Programming Language. In: Bode, A., Ludwig, T., Karl, W., Wismüller, R. (eds.) Euro-Par 2000 Parallel Processing. LNCS, vol. 1900, pp. 767–770. Springer (2000). https://doi.org/10.1007/3-540-44520-X_106
16. Cenciarelli, P., Knapp, A., Sibilio, E.: The Java Memory Model: Operationally, Denotationally, Axiomatically. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 331–346. Springer (2007). https://doi.org/10.1007/978-3-540-71316-6_23
17. Cho, M., Lee, S.H., Hur, C.K., Lahav, O.: Modular Data-Race-Freedom Guarantees in the Promising Semantics. In: PLDI 2021. pp. 867–882. ACM (2021). https://doi.org/10.1145/3453483.3454082, event-place: Virtual, Canada
18. Ciccozzi, F., Addazi, L., Asadollah, S.A., Lisper, B., Masud, A.N., Mubeen, S.: A Comprehensive Exploration of Languages for Parallel Computing. ACM Computing Surveys **55**(2), 24:1–24:39 (Jan 2022). https://doi.org/10.1145/3485008
19. De Moura, L., Bjørner, N.: Z3: An efficient SMT Solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
20. Franken, T.T.P., Neele, T., Groote, J.F.: An Autonomous Data Language. In: ICTAC 2023. LNCS, vol. 14446, pp. 158–177. Springer (2023). https://doi.org/10.1007/978-3-031-47963-2_11
21. Gannouni, S.: A Gamma-calculus GPU-based parallel programming framework. In: WSWAN 2015. pp. 1–4. IEEE (2015). https://doi.org/10.1109/WSWAN.2015.7210299
22. Gannouni, S., Touir, A., Mathkour, H.: Paradigma: A distributed framework for parallel programming. International Arab Journal of Information Technology **15**(5), 934–943 (Sep 2018)
23. Gharachorloo, K., Gupta, A., Hennessy, J.: Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In: ASPLOS 1991. pp. 245–257. ACM (Apr 1991). https://doi.org/10.1145/106972.106997
24. Goens, A., Chakraborty, S., Sarkar, S., Agarwal, S., Oswald, N., Nagarajan, V.: Compound Memory Models. In: PLDI 2023. Proc. ACM Program. Lang., vol. 7, pp. 1145–1168. ACM (Jun 2023). https://doi.org/10.1145/3591267
25. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java(TM) Language Specification, The (3rd Edition). Addison-Wesley Professional (2005)
26. Han, T.D., Abdelrahman, T.: hiCUDA: High-Level GPGPU Programming. IEEE Transactions on Parallel and Distributed Systems **22**(1), 78–90 (2011). https://doi.org/10.1109/TPDS.2010.62, publisher: IEEE
27. ISO/IEC: Programming Languages — C++. International Organization for Standardization, sixth edition edn. (2020), https://isocpp.org/files/papers/N4860.pdf
28. Jackson, D.: Alloy: a language and tool for exploring software designs. Communications of the ACM **62**(9), 66–76 (Aug 2019). https://doi.org/10.1145/3338843
29. Jagadeesan, R., Jeffrey, A., Riely, J.: Pomsets with Preconditions: A Simple Model of Relaxed Memory. In: OOPSLA 2020. Proc. ACM Program. Lang., vol. 4, pp. 194:1–194:30. ACM (Nov 2020). https://doi.org/10.1145/3428262
30. Jeffrey, A., Riely, J., Batty, M., Cooksey, S., Kaysin, I., Podkopaev, A.: The Leaky Semicolon: Compositional Semantic Dependencies for Relaxed-Memory Concur-

rency. In: POPL 2022. Proc. ACM Program. Lang., vol. 6, pp. 54:1–54:30. ACM (Jan 2022). `https://doi.org/10.1145/3498716`

31. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A Promising Semantics for Relaxed-Memory Concurrency. In: POPL 2017. pp. 175–189. ACM (Jan 2017). `https://doi.org/10.1145/3009837.3009850`
32. Kavanagh, R., Brookes, S.D.: A denotational account of C11-style memory. ArXiv **abs/1804.04214** (2018)
33. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in C/C++ 11. In: PLDI 2017. pp. 618–632. ACM (2017). `https://doi.org/10.1145/3062341.3062352`, publisher: ACM
34. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers **C-28**(9), 690–691 (Sep 1979). `https://doi.org/10.1109/TC.1979.1675439`
35. Lee, S.H., Cho, M., Podkopaev, A., Chakraborty, S., Hur, C.K., Lahav, O., Vafeiadis, V.: Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In: PLDI 2020. pp. 362–376. ACM (2020). `https://doi.org/10.1145/3385412.3386010`
36. Leemrijse, G.P.: The AuDaLaC compiler. (2023), `https://github.com/GPLeemrijse/AuDaLaC`, publication Title: GitHub repository
37. Leemrijse, G.P.: Towards relaxed memory semantics for the Autonomous Data Language [Supplemental Alloy Files] (Aug 2023). `https://doi.org/10.5281/zenodo.8244711`
38. Leemrijse, G.P., Franken, T.T.P., Neele, T.: Artifact - Formalisation of a new weak semantics for AuDaLa (Aug 2024). `https://doi.org/10.5281/zenodo.13354543`
39. Leiserson, C.E., Thompson, N.C., Emer, J.S., Kuszmaul, B.C., Lampson, B.W., Sanchez, D., Schardl, T.B.: There's plenty of room at the Top: What will drive computer performance after Moore's law? Science **368**(6495), eaam9744 (2020). `https://doi.org/10.1126/science.aam9744`
40. Liu, L., Millstein, T., Musuvathi, M.: Accelerating Sequential Consistency for Java with Speculative Compilation. In: PLDI 2019. pp. 16–30. ACM (2019). `https://doi.org/10.1145/3314221.3314611`
41. Lustig, D., Cooksey, S., Giroux, O.: Mixed-Proxy Extensions for the NVIDIA PTX Memory Consistency Model: Industrial Product. In: ISCA 2022. pp. 1058–1070. ACM (2022). `https://doi.org/10.1145/3470496.3533045`
42. Lustig, D., Sahasrabuddhe, S., Giroux, O.: A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In: ASPLOS 2019. pp. 257–270. ACM (2019). `https://doi.org/10.1145/3297858.3304043`
43. Lustig, D., Wright, A., Papakonstantinou, A., Giroux, O.: Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. In: ASPLOS 2017. pp. 661–675. ACM (2017). `https://doi.org/10.1145/3037697.3037723`
44. Marino, D., Singh, A., Millstein, T., Musuvathi, M., Narayanasamy, S.: A Case for an SC-Preserving Compiler. In: PLDI 2011. pp. 199–210. ACM (Jun 2011). `https://doi.org/10.1145/1993316.1993522`
45. Nagarajan, V., Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence. Springer International Publishing (2020). `https://doi.org/10.1007/978-3-031-01764-3`, `https://link.springer.com/10.1007/978-3-031-01764-3`
46. Nienhuis, K., Memarian, K., Sewell, P.: An Operational Semantics for C/C++11 Concurrency. In: OOPSLA 2016. pp. 111–128. ACM (Oct 2016). `https://doi.org/10.1145/2983990.2983997`, publisher: ACM

47. NVIDIA: PTX ISA (2023), https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, edition: 8.2
48. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 5.0 (Jan 2023), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
49. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_27
50. Paviotti, M., Cooksey, S., Paradis, A., Wright, D., Owens, S., Batty, M.: Modular Relaxed Dependencies in Weak Memory Concurrency. In: Müller, P. (ed.) ESOP 2020. LNCS, vol. 12075, pp. 599–625. Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_22
51. Rudy, G.: CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation (2010)
52. Sorensen, T., Donaldson, A.F.: Exposing Errors Related to Weak Memory in GPU Applications. In: PLDI 2016. pp. 100–113. ACM (Jun 2016). https://doi.org/10.1145/2908080.2908114
53. Sozeau, M., et al.: The Coq Proof Assistant (Jun 2023). https://doi.org/10.5281/zenodo.8161141, version Number: 8.17
54. Ueng, S.Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.m.W.: CUDA-Lite: Reducing GPU Programming Complexity. In: Amaral, J.N. (ed.) LCPC. vol. 5335, pp. 1–15. Springer (2008). https://doi.org/10.1007/978-3-540-89740-8_1
55. Ungar, D., Adams, S.S.: Harnessing emergence for manycore programming: early experience integrating ensembles, adverbs, and object-based inheritance. In: OOPSLA 2010. pp. 19–26. ACM (2010). https://doi.org/10.1145/1869542.1869546
56. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do about It. In: POPL 2015. pp. 209–220. ACM (Jan 2015). https://doi.org/10.1145/2676726.2676995
57. Vafeiadis, V., Narayan, C.: Relaxed Separation Logic: A Program Logic for C11 Concurrency. In: OOPSLA 2013. pp. 867–884. ACM (Oct 2013). https://doi.org/10.1145/2509136.2509532
58. Vialle, S., Cornu, T., Lallement, Y.: ParCeL-1: A Parallel Programming Language Based on Autonomous and Synchronous Actors. SIGPLAN Not. **31**(8), 43–51 (Aug 1996). https://doi.org/10.1145/242903.242945, https://doi.org/10.1145/242903.242945, publisher: ACM
59. Watt, C., Pulte, C., Podkopaev, A., Barbier, G., Dolan, S., Flur, S., Pichon-Pharabod, J., Guo, S.y.: Repairing and Mechanising the JavaScript Relaxed Memory Model. In: PLDI 2020. pp. 346–361. ACM (2020). https://doi.org/10.1145/3385412.3385973
60. Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically Comparing Memory Consistency Models. In: POPL 2017. pp. 190–204. ACM (2017). https://doi.org/10.1145/3009837.3009838
61. Yan, J., Tan, G., Mo, Z., Sun, N.: Graphine: Programming Graph-Parallel Computation of Large Natural Graphs for Multicore Clusters. IEEE Transactions on Parallel and Distributed Systems **27**(6), 1647–1659 (2016). https://doi.org/10.1109/TPDS.2015.2453978
62. Zhong, J., He, B.: Parallel graph processing on graphics processors made easy. Proc. VLDB Endow. **6**(12), 1270–1273 (2013). https://doi.org/10.14778/2536274.2536293

63. Ševčík, J.: Safe Optimisations for Shared-Memory Concurrent Programs. In: PLDI 2011. pp. 306–316. ACM (Jun 2011). `https://doi.org/10.1145/1993498.1993534`