CAAT: Consistency as a Theory

THOMAS HAAS, TU Braunschweig, Germany
ROLAND MEYER, TU Braunschweig, Germany
HERNÁN PONCE DE LEÓN*, Huawei Dresden Research Center, Germany

We propose a family of logical theories for capturing an abstract notion of consistency and show how to build a generic and efficient theory solver that works for all members in the family. The theories can be used to model the influence of memory consistency models on the semantics of concurrent programs. They are general enough to precisely capture important examples like TSO, Power, ARMv8, RISC-V, RC11, IMM, and the Linux kernel memory model. To evaluate the expressiveness of our theories and the performance of our solver, we integrate them into a lazy SMT scheme that we use as a backend for a bounded model checking tool. An evaluation against related verification tools shows, besides flexibility, promising performance on challenging programs under complex memory models.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Concurrency**; *Verification by model checking*.

Additional Key Words and Phrases: Weak memory models, program verification, bounded model checking.

ACM Reference Format:
Thomas Haas, Roland Meyer, and Hernán Ponce de León. 2022. CAAT: Consistency as a Theory. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 129 (October 2022), 31 pages. https://doi.org/10.1145/3563292

1 INTRODUCTION

Modern programming languages like C11, Rust, and Kotlin are difficult to analyze for state-of-the-art verification technology. The problem is not merely one of verification lagging behind, but a shift on the programming language side that verification has a hard time to reflect. Modern languages move towards programming abstractions that give the programmer detailed control over the execution environment. Co-routines can modify the scheduling, C11 atomics come with ordering and visibility guarantees about their execution, and the memory is managed by reclamation schemes [Dang et al. 2020; Elizarov et al. 2021]. Classically, verification technology ignores all this: the scheduling is modeled by non-determinism, instructions are executed in program order, and there is a garbage collector reclaiming unused memory. When using this verification technology to analyze a program in a modern language, we may miss bugs or, less critically, judge correct programs as faulty. The problem becomes even more pressing as the new programming abstractions have turned out to be difficult to use (as witnessed by the repeated corrections to the C11 standard [Batty et al. 2016; Lahav et al. 2016, 2017; Manerkar et al. 2016; Vafeiadis et al. 2015; Vafeiadis and Narayan 2013]), and thus verification technology is badly needed.

*This research was done while the author was at Bundeswehr University Munich, Germany.

Authors' addresses: Thomas Haas, TU Braunschweig, Germany, t.haas@tu-braunschweig.de; Roland Meyer, TU Braunschweig, Germany, roland.meyer@tu-bs.de; Hernán Ponce de León, Huawei Dresden Research Center, Germany, hernanl.leon@huawei.com.

2475-1421/2022/10-ART129
https://doi.org/10.1145/3563292

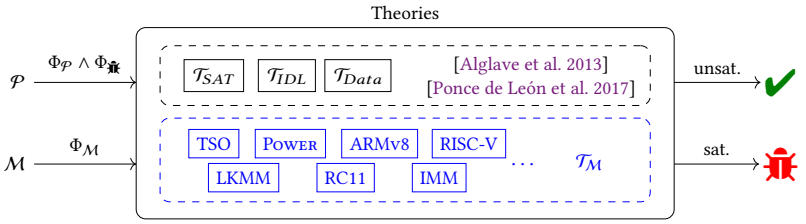

Fig. 1. The BMC problem with memory models.

The verification community has taken up the challenge and made considerable progress on incorporating aspects of the execution environment into the verification technology. For asynchronous programming, formal techniques are emerging [Koval et al. 2021]. In the realm of memory consistency, there are now tools that model the semantics of programs under Intel's, ARM's, and IBM's memory models [Abdulla et al. 2015, 2016; Alglave et al. 2013; Ponce de León et al. 2020]. Also, the popular causal consistency as well as the release-acquire fragment of C11 have found their way into tools [Abdulla et al. 2018]. For memory reclamation, reductions to verification under garbage collection have been proposed [Meyer and Wolff 2019, 2020].

We argue that these efforts are not enough because the number of execution environments grows much faster than the verification community can provide technology. The reason is that the execution environment consists of several components (the scheduler, the consistency model, the reclamation scheme), each under active development, and every combination yields a new setting. State-of-the-art verification tools are holistic objects that have to carefully integrate every single aspect of the program semantics.

We believe there is a need for verification technology that can be easily adapted to the execution environment. There is a corner of verification where we find a promising flexibility in the program semantics: bounded model checking (BMC) [Clarke et al. 2001]. BMC tools have been developed for various classes of programs, from hardware to software and from timed systems [Audemard et al. 2002] to heap-manipulating programs [Ponzio et al. 2021]. Their flexibility in the semantics stems from flexible backend technology. They translate verification problems to satisfiability modulo theories (SMT) queries. SMT solvers [Barrett et al. 2009] integrate an arsenal of logical theories that turned out to be useful to capture the program semantics.

Problem Statement. Our goal is to contribute bounded model checking technology that can prove the correctness of a given concurrent program for an also given execution environment. We assume correctness is formulated as a set of assertions that are part of the program. The task is thus to either find an assertion violation or to prove that they all hold.

We will focus on the memory (consistency) model of the environment [Adir et al. 2003; Adve and Gharachorloo 1996; Alglave et al. 2012, 2014; Batty et al. 2012; Boehm and Adve 2008; Collier 1992; Manson et al. 2006; Pulte et al. 2018; Sarkar et al. 2011; Shasha and Snir 1988; Sindhu et al. 1992]. Traditional verification techniques for concurrent programs assume sequential consistency (SC) [Lamport 1979], where the program executions are just the interleavings of the instructions in each thread. Modern memory models add to this the possibility of reordering, dropping, and forwarding instructions to different threads at different moments in time (called non-multi-copy atomicity), resulting in a perception of the program execution that is only weakly consistent among the threads. The memory model depends on the environment. Moreover, it changes as the desirable notion of consistency evolves. For example, Intel and ARM processors have different memory models, and ARMv8 (the ARM memory model in version 8) is multi-copy atomic while ARMv7 is not [Pulte et al. 2018]. Verification techniques shall not only take into account the notion of consistency, but

also be easily adaptable in case this notion changes. We thus target verification technology that works relative to a given memory model $\mathcal{M}$. The model defines, independently of the program, a set of executions $[\![\mathcal{M}]\!]$ that it considers consistent.

The program $\mathcal{P}$, in turn, defines basic properties of the control and data flow that are independent of the memory model. These properties are enough to identify the executions that fail an assertion, denoted by $[\![\mathcal{P}_{🐞}]\!]$. Then, a given program $\mathcal{P}$ is correct under a given memory model $\mathcal{M}$, if

$$[\![\mathcal{P}_{🐞}]\!] \cap [\![\mathcal{M}]\!] = \emptyset \,.$$

Our goal is to check this emptiness. If $[\![\mathcal{P}_{🐞}]\!]$ is finite, which we assume throughout the paper, one can compute this set of executions and filter out the ones that are inconsistent with $\mathcal{M}$. This is the approach followed by Herd [Alglave et al. 2014], which is great to understand the corner cases of $\mathcal{M}$. Unfortunately, it does not scale to large programs.

A more scalable approach is bounded model checking (BMC), Figure 1. It is well known that $[\![\mathcal{P}_{🐞}]\!]$ can be represented as a logical formula $\Phi_{\mathcal{P}} \wedge \Phi_{🐞}$ over a theory $\mathcal{T}_{Data}$ [Alglave et al. 2013].[1] Here, $\Phi_{\mathcal{P}}$ encodes the program's control and data flow, while $\Phi_{🐞}$ encodes the violation of an assertion. Also $[\![\mathcal{M}]\!]$ can be encoded as a formula $\Phi_{\mathcal{M}}$ over $\mathcal{T}_{SAT} + \mathcal{T}_{IDL}$.[2] The BMC approach to verification then amounts to checking

$$\mathcal{T}_{Data} + \mathcal{T}_{SAT} + \mathcal{T}_{IDL} \models \Phi_{\mathcal{P}} \wedge \Phi_{🐞} \wedge \Phi_{\mathcal{M}} \,.$$

If the formula is satisfiable, then some execution leads to an assertion violation. Otherwise, the program is correct. The approach has been implemented in Dartagnan [Ponce de León et al. 2017]. While more scalable than enumeration, Dartagnan still has a performance problem. The SMT encoding is *eager*: all constraints of the memory model have to be translated into $\Phi_{\mathcal{M}}$ in one shot. In particular for complex models like Power or ARMv8 this results in a huge formula.

The idea behind our work is to use the flexibility of the SMT engine in the background theories to match the flexibility of the verification problem in the consistency model. We wonder if there is a family of theories $\mathcal{T}_{\mathcal{M}}$, each of them characterizing a memory model $\mathcal{M}$, that allows us to get rid of the eager encoding $\Phi_{\mathcal{M}}$. Instead of having $\Phi_{\mathcal{M}}$, the new theories will teach the SMT solver the meaning of consistency-related literals. The new BMC approach would then amount to checking

$$\mathcal{T}_{Data} + \mathcal{T}_{\mathcal{M}} \models \Phi_{\mathcal{P}} \wedge \Phi_{🐞} \,.$$

We refer to this as a *lazy* SMT encoding, because the theory literals are not eagerly compiled to other theories. The research problem we tackle is thus:

> *Can we devise a family of logical theories $\mathcal{T}_{\mathcal{M}}$ expressive enough to capture practical memory models and can we give a generic and efficient solver for all members in the family?*

Before we elaborate on our contribution, it is appropriate to compare this goal to related works in the literature. The recent [Fan et al. 2022; He et al. 2021] presents a novel logical theory that is expressive enough to reflect the consistency constraints of SC, TSO, and PSO, and at the same time admits very efficient theory solving. The theory solver is the basis for Deagle, the first bounded model checker that is memory model-aware and relies on a lazy encoding of the form we want to achieve. The performance is amazing: with its first participation, Deagle won the concurrency category in the competition for software verification [Beyer 2022].[3] Still, there is a crucial difference between our development and [Fan et al. 2022; He et al. 2021]. We do not want a solver for a

[1] *Data* might be, e.g., integer arithmetic or bitvectors.
[2] $\mathcal{T}_{IDL}$ is the theory of *integer difference logic*, which is commonly used to compactly formulate acyclicity/scheduling constraints.
[3] https://sv-comp.sosy-lab.org/2022/.

fixed theory, but we are interested in having a language of consistency theories that we can all solve. This makes the problem different in nature. The Deagle development can hardcode the consistency model. Our work is language driven, every computation has to work along the syntax of the consistency model at hand.

The second competitor is first-order relational logic [Jackson 2000]. It is a competitor because our language for consistency models will be based on recursively defined relations as first pioneered by Alglave et al. [2016]. Our algorithmic contributions all refer to this recursion, from the check for theory satisfiability to the theory explanation. The only recursive construct that has been studied in relational logic is transitive closure. While it is known that linear (in the relational composition operator) memory models can be translated to transitive closure, important models like the Linux kernel memory model [Alglave et al. 2018] and Power [Mador-Haim et al. 2012] are non-linear, and replacing the recursion by an unwinding has turned out highly impractical [Bornholt and Torlak 2017].

To sum up, despite the recent success of Deagle and the body of literature on relational logic, there is a strong motivation to pursue the research question we propose.

Contribution. We present a family of theories for capturing the influence of an execution environment on the program semantics (Section 2). We devise a subset of the family that is expressive enough in practice and algorithmically appealing (Section 3). We give algorithms to efficiently check theory satisfiability and generate theory explanations if satisfiability fails (Section 4). These two features are enough to integrate our theories into an offline SMT scheme, where a SAT solver computes satisfying assignments that theory solvers check for theory satisfiability. We further show how to balance the workload between the SAT and the theory solver and how to overcome limitations of our development (Section 5). We prove the point of flexibility in the context of memory consistency, where we show that our theories can capture TSO, Power, ARMv8, RISC-V, IMM, RC11, and the Linux kernel memory model (LKMM). We build a BMC tool (Section 6) on top of a lazy SMT encoding and demonstrate in an extensive evaluation (Section 7) that it performs favorably against competitors. Finally, we discuss related (Section 8) and future work (Section 9).

2 CONSISTENCY AS A THEORY

While our problem statement focuses on memory consistency models (i.e., $\mathcal{M}$), in this section we assume a more general setting and reason about abstract consistency models (denoted by *cm*). We define a language and logical theories for abstract consistency which, as we will show in Sections 6 and 7, can be used to formalize concrete memory models.

2.1 A Language for Consistency

Our language of consistency models is inspired by[4] CAT [Alglave et al. 2016] and defined by the grammar in Figure 2. A consistency model is a constraint system over so-called *predicates*. Predicates are either unary, called *sets*, or binary, called *relations*. New predicates can be built from named predicates using predefined operations over them. The consistency language supports union $\cup$, intersection $\cap$, difference $\setminus$, inverse $\bullet^{-1}$, transitive (and reflexive) closure $\bullet^{+}(\bullet^{*})$, relational composition ;, projections `domain` and `range`, cartesian product $\times$, and identities on sets $[\bullet]$. Cartesian products are restricted to sets so that all derivable predicates have arity at most two. While the intended semantics of most operators is obvious, the set identity operator may be less familiar: it is defined by $[\mathrm{s}] := \{(x, x) \mid x \in \mathrm{s}\}$. We use precedence rules to simplify the presentation: unary operators take precedence over binary operators, and among the binary operators the precedence order from high to low is cartesian product $\times$, composition ;, and then all Boolean operators.

[4]A discussion about similarities and differences to CAT can be found in Section 8.

$$
\begin{array}{rcl}
\langle cm \rangle & ::= & \langle axm \rangle \mid \langle rel \rangle \mid \langle set \rangle \mid \langle cm \rangle \langle cm \rangle \\
\langle axm \rangle & ::= & \mathbf{acyclic}(\langle dr \rangle) \mid \mathbf{irreflexive}(\langle dr \rangle) \mid \mathbf{empty}(\langle dr \rangle) \mid \mathbf{empty}(\langle ds \rangle) \\
\langle dr \rangle & ::= & \langle rname \rangle \mid \mathsf{id} \mid \langle dr \rangle \cup \langle dr \rangle \mid \langle dr \rangle \cap \langle dr \rangle \mid \langle dr \rangle \setminus \langle dr \rangle \\
& & \mid \langle dr \rangle ; \langle dr \rangle \mid \langle dr \rangle^{-1} \mid \langle dr \rangle^{+} \mid \langle dr \rangle^{*} \mid [\langle ds \rangle] \mid \langle ds \rangle \times \langle ds \rangle \\
\langle ds \rangle & ::= & \langle sname \rangle \mid \mathbb{D} \mid \texttt{domain}(\langle dr \rangle) \mid \texttt{range}(\langle dr \rangle) \\
& & \mid \langle ds \rangle \cap \langle ds \rangle \mid \langle ds \rangle \cup \langle ds \rangle \mid \langle ds \rangle \setminus \langle ds \rangle \\
\langle rel \rangle & ::= & \texttt{let}\ \langle rname \rangle = \langle dr \rangle \\
\langle set \rangle & ::= & \texttt{let}\ \langle sname \rangle = \langle ds \rangle
\end{array}
$$

Fig. 2. Grammar for consistency models.

Consistency is defined via unary predicates which we call *consistency axioms* over the other predicates. Axioms are acyclicity, irreflexivity, and emptiness constraints. Named predicates may have an associated defining equation, i.e., $\texttt{let}\ \langle rname \rangle = \langle dr \rangle$ and $\texttt{let}\ \langle sname \rangle = \langle ds \rangle$. Notably, the right-hand side may again contain named predicates, making the system of defining equations recursive. If a named predicate has no defining equation, we call it a *base* predicate, otherwise we call it *derived*. We reserve the names $\mathbb{D}$ and id for distinguished base predicates; $\mathbb{D}$ is the unary *domain* set and id is the binary *identity* relation which will have a special semantics. We use the symbols b, d, and p to refer to base, derived, and arbitrary predicates; s and r for sets and relations; and *axm* for axioms. Boldface symbols are used to denote vectors, e.g., $\mathbf{b}$ is a vector of base predicates.

Definition 2.1. A *consistency model cm* is an element of the consistency language in Figure 2.

We call a consistency model *normalized* if *(i)* every right-hand side of an equation contains exactly one operator and *(ii)* every axiom refers to a single named predicate. We can always achieve *(i)* by introducing fresh predicates for complex expressions. Consider the equation $\texttt{let}\ \mathsf{p}_1 = (\mathsf{p}_2; \mathsf{p}_3); \mathsf{p}_2$. We add an intermediate defining equation $\texttt{let}\ \mathsf{p}_4 = \mathsf{p}_2; \mathsf{p}_3$ and redefine p_1 using $\texttt{let}\ \mathsf{p}_1 = \mathsf{p}_4; \mathsf{p}_2$. Similarly, we achieve *(ii)* for, say, $\mathbf{acyclic}(\mathsf{p}_1 \cup \mathsf{p}_2)$ by introducing $\texttt{let}\ \mathsf{p}_3 = \mathsf{p}_1 \cup \mathsf{p}_2$ and using the axiom $\mathbf{acyclic}(\mathsf{p}_3)$ instead.

Figure 3 shows the memory consistency model ARMv8 from [Pulte et al. 2018] in our language. Note that relations like `rf`, `co`, and `po` do not have a defining equation; they are base relations. The program order `po` relates instructions from the same thread in the order they appear (syntactically) in the code; the read-from relation `rf` connects store instructions to the load instructions that read from them; the coherence order `co` is an arbitrary total order over stores to the same address, that is, every pair of stores accessing the same address is related by `co` and pairs of stores accessing disjoint memory addresses are unrelated. A typical interpretation of the coherence order `co` is the order in which stores are committed to main memory (i.e., moved from a thread-local store buffer to shared memory). This semantics has to get explicitly encoded into the memory consistency model via axioms. In contrast, `po` and `rf` have intrinsic semantics for the program, independent of the memory consistency model. Base sets include `W` and `R` representing store and load instructions, `dbm.full`, `dmb.ld`, and `dmb.st` representing memory fences, and `L` and `A` representing instructions with release and acquire semantics, respectively.

We explain the high-level idea of the ARMv8 memory model and how this is captured in our consistency language. The model defines two types of orderings, the first being related to inter-thread communication and the second to thread-local reorderings. These two types of orderings have to agree on a single acyclic ordering ob. In the first category, the relation `obs` links an

```
/* Coherence-after */
let ca = fr ∪ co

/* Internal visibility requirement */
acyclic (po ∩ loc) ∪ ca ∪ rf

/* Observed-by */
let obs = rfe ∪ fre ∪ coe

/* Dependency-ordered-before */
let dob = addr ∪ data
  ∪ ctrl; [W]
  ∪ (ctrl ∪ (addr; po)); [ISB]; po; [R]
  ∪ addr; po; [W]
  ∪ (ctrl ∪ data); coi
  ∪ (addr ∪ data); rfi

/* Atomic-ordered-before */
let aob = rmw ∪ [range(rmw)]; rfi;[A ∪ Q]

/* Barrier-ordered-before */
let bob = po; [dmb.full]; po
  ∪ [L]; po; [A]
  ∪ [R]; po; [dmb.ld]; po
  ∪ [A ∪ Q]; po
  ∪ [W]; po; [dmb.st]; po; [W]
  ∪ po; [L]
  ∪ po; [L]; coi

/* Ordered-before */
let ob = obs ∪ dob ∪ aob ∪ bob ∪ (ob; ob)

/* External visibility requirement */
irreflexive ob

/* Atomic: Basic LDXR/STXR constraint to
     forbid intervening writes. */
empty rmw ∩ (fre; coe)
```

Fig. 3. Memory consistency model ARMv8 [Pulte et al. 2018].

observed event of one thread with the observing event of another thread, e.g., a load observes a store if it reads its value (`rfe` stands for *read-from external*). ARMv8 is multi-copy atomic, meaning there is a single shared main memory and all threads agree on their view of this shared memory. This is witnessed by having the relations `coe` (*coherence external*) and `fre` (*from-read external*) as part of `obs`. In the second category, the dependency-induced ordering `dob` (`dob` stands for *dependency-ordered before*) captures data, address, and control dependencies that are preserved by the consistency model. The ARMv8 memory model supports load speculation, which means it can execute load instructions inside a conditional branch even before establishing whether the branch will get executed. As a consequence, in the absence of special barrier instructions, it only preserves control dependencies to store instructions, which is witnessed by the `ctrl; [W]` fragment of `dob`. Then we have the ordering `aob` for atomics, which makes sure that events coming from read-modify-write (rmw) operations are properly ordered. This is needed because a single rmw-operation is modeled via a load and a store and the two operations should not be reordered. Atomicity of the rmw-operation is guaranteed by the axiom **empty**(`rmw` ∩ `fre; coe`), which disallows memory operations of other threads to be ordered in between the load and the store. Finally, we have the barrier-induced ordering `bob` for explicitly placed barriers like `dbm.full` and acquire/release operations ([`A`]/[`L`]). The union of the four ordering relations (`ob`) has to be acyclic for an execution to be consistent under ARMv8, formulated here via transitive closure (in recursive form) and irreflexivity to illustrate aspects of our development.

Note that the above information about the semantics of the memory consistency model is irrelevant for our verification approach, which only cares about the syntactic structure of the model. This achieves a separation of concerns between systems engineering (coming up with the model) and verification (w.r.t. the model).

2.2 Semantics

The semantics $[\![cm]\!]$ of a consistency model *cm* is defined in terms of consistent *cm*-structures. Let $\mathbf{b}$ be the vector of base predicates. A *cm*-structure $\mathcal{H} = (D, \mathcal{I}_{\mathbf{b}})$ consists of a domain D and an interpretation $\mathcal{I}_{\mathbf{b}}$ of all base predicates over the domain D. The predicate $\mathbb{D}$ is such that it holds for the whole domain and id is the identity relation on D. Let $\mathbf{d}$ denote the vector of derived predicates and $\mathbf{E}$ the associated vector of right-hand sides of defining equations. For example, in the case of

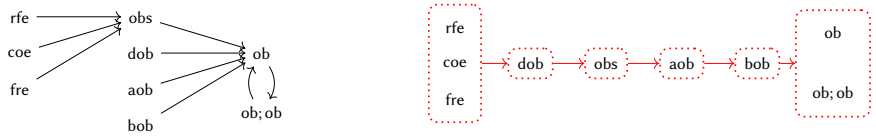

Fig. 4. Snippet of ARMv8's dependency graph (left). A fine-grained stratification (right) computed via Tarjan's SCC algorithm and the first step of our algorithm in Section 4.2.

the ARMv8 memory model of Figure 3, the vector of derived relations contains an entry $\mathbf{d}_i = \mathsf{ob}$ with associated right-hand side $\mathbf{E}_i = \mathsf{obs} \cup \mathsf{dob} \cup \mathsf{aob} \cup \mathsf{bob} \cup (\mathsf{ob};\mathsf{ob})$. We define the interpretation $\mathcal{I}_{\mathbf{d}}$ of $\mathbf{d}$ as a uniquely defined minimal solution of $\mathbf{d} = \mathbf{E}(\mathbf{b}, \mathbf{d})$. Note that the right-hand side of this equation may not be monotonic in $\mathbf{d}$ in the presence of predicate differences like $\mathsf{p}_1 \setminus \mathsf{p}_2$. Memory consistency models like the Linux kernel memory model LKMM and RC11 make use of such differences and thus we need to support them. Despite not being monotonic, the equation still has a unique minimal solution that is defined via the canonical evaluation strategy we explain next.

Consider the equation $\texttt{let}\ \mathsf{d}_2 = (\mathsf{d}_2 \cup (\mathsf{d}_2;\mathsf{d}_2)) \setminus \mathsf{d}_1$ with d_1 a derived relation independent of d_2. In the presence of predicate differences, the order in which the defining equations of d_2 and d_1 get evaluated matters. A simultaneous evaluation, as done by the usual Kleene fixed-point iteration [Nielson et al. 1999], may give a result anywhere between $\mathsf{d}_2 = (\mathsf{d}_2 \setminus \mathsf{d}_1)^+$ and $\mathsf{d}_2 = \mathsf{d}_2^+ \setminus \mathsf{d}_1$, the former when d_1 is fully evaluated first, the latter when d_2 is evaluated until its fixed point, then d_1 gets evaluated, and finally d_2 gets evaluated once more. Seeing that d_2 depends on d_1, the canonical choice is to fully evaluate d_1 and then treat it as a constant when evaluating d_2. We formalize this evaluation strategy.

We build a directed dependency graph that has the predicate symbols of *cm* as nodes. The graph has an edge from p_1 to p_2 if p_1 is part of the definition of p_2, i.e., we have $\texttt{let}\ \mathsf{p}_2 = \ldots\ \mathsf{p}_1\ \ldots$ in the consistency model. The strongly connected components (SCCs) of this dependency graph contain predicates that are defined by mutual recursion. The dependency graph for a snippet of ARMv8 is illustrated in Figure 4 (left). Intuitively, our desired evaluation strategy evaluates the predicates along this dependency graph and only evaluates predicates if all their dependencies have already been fully evaluated.

The evaluation strategy can be formalized via *stratifications*, a concept known from Datalog [Abiteboul et al. 1995]. A stratification of the consistency model *cm* is an ordered partitioning of its predicates $\mathbf{p}$ into disjoint sets $\mathbf{p}^0, \ldots, \mathbf{p}^k$, so-called *strata*, such that the following holds. *(i)* The first stratum consists of all base predicates, $\mathbf{p}^0 = \mathbf{b}$. *(ii)* If a predicate in a stratum $\mathbf{p}^j$ depends on (as defined above) a predicate in $\mathbf{p}^i$, then $i \leq j$. *(iii)* If a predicate in $\mathbf{p}^j$ depends negatively via a predicate difference on a predicate in $\mathbf{p}^i$, then $i < j$. We say that *cm* is *stratifiable* if such a stratification exists. Condition *(iii)* forbids recursively defined predicates to negatively depend on another predicate from their own stratum. A non-stratifiable consistency model would be given, e.g., by $\texttt{let}\ \mathsf{d}_2 = \mathsf{d}_2 \setminus \mathsf{d}_1$ and $\texttt{let}\ \mathsf{d}_1 = \mathsf{d}_2;\mathsf{d}_2$. Here, d_2 negatively depends on itself via d_1. Such equations may not even have a fixed point. From now on, we only consider consistency models that are stratifiable.

Figure 4 (right) shows a possible stratification for (a fragment of) ARMv8, that has been computed from the dependency graph by collapsing the SCCs and determining a topological ordering of the resulting (acyclic) graph. One can show that this always yields a stratification. Moreover, the strata obtained this way are minimally sized, meaning derived and non-recursive predicates form singleton strata, and for recursive predicates the strata match their SCCs.

It remains to define the interpretation $\mathcal{I}_{\mathbf{d}}$ and formulate what it means for a *cm*-structure to be consistent. Let $\mathbf{p}^0, \ldots, \mathbf{p}^k$ be a stratification and $\mathbf{E}^1, \ldots, \mathbf{E}^k$ the right-hand sides of the associated equations ($\mathbf{p}^0 = \mathbf{b}$ has no associated equation). The desired solution $\mathcal{I}_{\mathbf{d}}$ of $\mathbf{d} = \mathbf{E}(\mathbf{b}, \mathbf{d})$ is the

sequence of least solutions given by $\mathbf{p}^{i+1} = \mathbf{E}^{i+1}(\mathbf{p}^0, \ldots, \mathbf{p}^i, \mathbf{p}^{i+1})$. Since $\mathbf{p}^0, \ldots, \mathbf{p}^k$ is a stratification, each right-hand side $\mathbf{E}^{i+1}$ is monotonic in $\mathbf{p}^{i+1}$ and only depends on $\mathbf{p}^{\leq i+1}$. One can show that the solution $\mathcal{I}_\mathbf{d}$ will be the same for every stratification [Abiteboul et al. 1995], which is why we have not chosen a particular one. We extend the *cm*-structure $\mathcal{H}$ to $(D, \mathcal{I})$ with $\mathcal{I} = (\mathcal{I}_\mathbf{b}, \mathcal{I}_\mathbf{d})$. We say that $\mathcal{H}$ is *consistent* (with *cm*) if all axioms of *cm* hold w.r.t. $\mathcal{I}$. In this case, we also call $\mathcal{H}$ a *model* of *cm*. The *semantics* $[\![cm]\!]$ is defined as the set of all models of *cm*.

In the context of program verification (Section 6), *cm*-structures $\mathcal{H}$ represent program executions. The domain consists of the executed memory instructions like read, write, and fence operations. The base predicates contain (at least) the program order po, the read-from relation rf, and the coherence order co. Predicates like *happens-before* (hb) under sequential consistency (SC) and *ordered-before* (ob) under ARMv8 are derived. If $\mathcal{H}$ is consistent as a *cm*-structure, then it represents an execution of the program that *cm*, understood as a memory model, admits. In other words, in this context $[\![cm]\!]$ is the set of all program executions admissible under *cm*.

2.3 From Consistency Models to Theories

Our goal is to associate with each consistency model *cm* a logical theory $\mathcal{T}_{cm}$ [Enderton 1972]. To this end, we assume the consistency model comes with a signature $\Sigma_{cm} = (\Sigma_{const}, \Sigma_{pred})$, where Σ_{const} is an infinite set of constants $v_1, v_2, \ldots$ representing domain elements, and Σ_{pred} is a finite set of predicate symbols corresponding to the base predicates in the consistency model. The arity of predicate symbols is one (for sets) or two (for relations). The symbols $\mathbb{D}$ and id do not belong to Σ_{pred}, because their interpretation will be fixed. We define $\mathcal{L}_{cm}$ as the ground fragment of the first-order language associated with Σ_{cm}. This means Σ_{cm}-atoms are of the form $\mathrm{p}(v_i, v_j)$ and $\mathrm{p}(v_i)$. The sentences (closed formulas) of $\mathcal{L}_{cm}$ are constructed from Σ_{cm}-atoms using the usual logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$. A Σ_{cm}-literal is an atom or a negated atom. To fix the notation, we use ψ for arbitrary sentences, reserve φ for conjunctions of literals, and write Σ_ψ for the set of constants occurring in ψ.

We define the logical theory $\mathcal{T}_{cm}$ in a model-theoretic way [Enderton 1972], as the set of sentences that are satisfied by all models of interest. The models of interest are *Herbrand structures* $\mathcal{H}$, meaning the domain is given by the set of constants and each constant is interpreted by itself. There is no restriction on how to interpret the predicates p. We say that $\mathcal{H}$ satisfies an atom $\mathrm{p}(v_i, v_j)$, if $\mathcal{H}(\mathrm{p})(v_i, v_j)$ is true. This satisfaction relation extends to the Boolean connectives as expected.

A Herbrand structure $\mathcal{H}$ can be seen as a *cm*-structure by adding the predicates $\mathbb{D}$ and id to its signature and interpreting them as intended. Then we say that $\mathcal{H}$ is consistent, if it is consistent as a *cm*-structure. A formula ψ is $\mathcal{T}_{cm}$-satisfiable, if it is satisfied by a consistent $\mathcal{H}$. The theory $\mathcal{T}_{cm}$ is the set of sentences satisfied by all consistent $\mathcal{H}$'s.

To give an example, consider the consistency model $\mathbf{acyclic}(\mathsf{dep} \cup \mathsf{rf})$. Here, dep is a strict variant of the data, address, and control dependency relation dob from above (which was ARMv8 specific). The read-from relation rf matches each read instruction with the write from which it obtains its value. A variant of this axiom is commonly used to forbid *out-of-thin-air* behaviors [Jeffrey and Riely 2016; Lahav et al. 2017]. An example sentence in $\mathcal{T}_{\mathbf{acyclic}(\mathsf{dep} \cup \mathsf{rf})}$ is $\mathrm{dep}(v_1, v_2) \rightarrow \neg\mathrm{rf}(v_2, v_1)$. It says that if the value of v_2 depends on v_1, then v_1 cannot take its value from v_2. Given a formula and a theory, our goal is to decide if the formula is satisfiable in the theory. In $\mathcal{T}_{\mathbf{acyclic}(\mathsf{dep} \cup \mathsf{rf})}$, formula $\mathrm{dep}(v_1, v_2)$ is satisfiable, but $\mathrm{dep}(v_1, v_2) \wedge \mathrm{rf}(v_2, v_1)$ is not.

3 FRAGMENTS OF CONSISTENCY MODELS AND THEIR PROPERTIES

We devise classes of consistency models that are expressive enough to capture many practically relevant consistency models and, at the same time, constrained enough to simplify the satisfiability checking from an algorithmic point of view.

3.1 Domain Independence

One difficulty in checking the satisfiability of a formula ψ is that we have to interpret all constants in the signature, and there will usually be infinitely many of them. To overcome the problem, we introduce the notion of domain independence. It allows us to focus on the constants that occur in ψ in the following sense: if we consider ψ as a formula over this reduced signature and we find a model, we are sure the model can be extended to a model over the original signature. The idea of domain independence stems from database theory [Abiteboul et al. 1995], where it is used in a semantic way. Our notion is syntactic, instead. Before we give the definition, we illustrate the problems that may occur when extending domains.

Consider the (rather questionable) consistency model $\mathbf{empty}(\mathbb{D} \times \mathbb{D} \setminus \mathsf{id})$. The only *cm*-models have a domain of either zero or one element; any structure with a larger cardinality is inconsistent, irrespective of its interpretations of the base predicates. In particular, there are no consistent Herbrand structures and $\mathcal{T}_{cm}$ becomes a degenerate theory without models. Domain independence forbids the free use of the predicates $\mathbb{D}$ and id, but expects them to be guarded by a domain-independent predicate. To give an example, the consistency model $\mathbf{empty}(\mathsf{b} \cap (\mathbb{D} \times \mathbb{D} \setminus \mathsf{id}))$ is similar to the degenerate one above but domain independent. The point is that the interpretation of $\mathbb{D} \times \mathbb{D} \setminus \mathsf{id}$ is effectively restricted to the domain of the interpretation of b.

Definition 3.1. The set of *domain-independent predicates* dip is defined inductively as follows:

$$\begin{aligned}\mathsf{dip} \quad ::= \quad & \mathsf{b} \mid \mathsf{dip} \cap \mathsf{p} \mid \mathsf{p} \cap \mathsf{dip} \mid \mathsf{dip} \setminus \mathsf{p} \mid \mathsf{dip} \cup \mathsf{dip} \mid \mathsf{dip}; \mathsf{dip} \mid \mathsf{dip} \times \mathsf{dip} \\ & \mid \mathsf{dip}^{-1} \mid \mathsf{dip}^{+} \mid \mathsf{dip}^{*} \mid [\mathsf{dip}] \mid \texttt{domain}(\mathsf{dip}) \mid \texttt{range}(\mathsf{dip}).\end{aligned}$$

Here, b is a base predicate different from $\mathbb{D}$ and id and p is any predicate. A normalized consistency model *cm* is domain independent, if for every axiom the predicate is domain-independent.

Proposition 3.2. *Let cm be normalized and domain-independent with signature* $\Sigma_{cm} = (\Sigma_{const}, \Sigma_{pred})$. *If* ψ *is satisfiable as a formula over* $(\Sigma_\psi, \Sigma_{pred})$, *then it is satisfiable as a formula over* Σ_{cm}.

3.2 Semi-Positivity

What if ψ is not satisfiable as a formula over $(\Sigma_\psi, \Sigma_{pred})$? If the proposition does not apply, the formula may still be satisfied by a model that extends the domain beyond the constants in ψ. This is unfortunate from an algorithmic point of view, because there are infinitely many ways of extending the domain. We study this phenomenon and attribute it to a violation of a property called semi-positivity. Similar to domain independence, this notion comes from data base theory [Abiteboul et al. 1995]. For a semi-positive and domain-independent consistency model, we show that a formula ψ is satisfiable if and only if it is satisfiable over $(\Sigma_\psi, \Sigma_{pred})$. We again proceed by an example before turning to the technicalities.

Consider the consistency model $\mathbf{empty}(\mathsf{b}_1 \setminus (\mathsf{b}_2; \mathsf{b}_2))$ with b_1 and b_2 base relations. Assume we want to check the satisfiability of $\psi = \mathsf{b}_1(v_1, v_2) \wedge \neg \mathsf{b}_2(v_1, v_2)$. We can build a Herbrand structure for ψ that evaluates $\mathsf{b}_1(v_1, v_2)$ to true and interprets b_2 as the empty relation. Obviously, this model is inconsistent with the theory. Nevertheless, ψ is satisfiable because there is a different model which is consistent. We pick a new element v_3 and extend the interpretation of b_2 to include $\mathsf{b}_2(v_1, v_3)$ and $\mathsf{b}_2(v_3, v_2)$. The crux in this example is the use of the difference operator over a derived predicate (when normalizing, $\mathsf{b}_2; \mathsf{b}_2$ yields a new derived predicate d).

Definition 3.3. A normalized consistency model *cm* is called *semi-positive* if, in all equations of the form $\texttt{let}\ \mathsf{p}_3 = \mathsf{p}_1 \setminus \mathsf{p}_2$, the predicate p_2 is base.

The problematic consistency model discussed above is not semi-positive but, e.g., $\mathbf{empty}(\mathsf{b}_1 \setminus \mathsf{b}_2)$ is. Semi-positivity makes it impossible to repair an inconsistent Herbrand structure by enlarging

the domain and extending the interpretation of predicates to this larger domain. Phrased positively, a Herbrand structure can only be made consistent by enlarging the interpretation of predicates within the domain. This frees us from the worrisome infinity mentioned in the beginning and leads to the following theorem.

Theorem 3.4. *Let cm be normalized, domain-independent, and semi-positive with signature* $\Sigma_{cm} = (\Sigma_{const}, \Sigma_{pred})$. *Then* ψ *is satisfiable as a formula over* Σ_{cm} *if and only if it is satisfiable over* $(\Sigma_\psi, \Sigma_{pred})$.

We explicitly give the proof as it will be the basis for our decision procedure.

Proof. We first consider the simpler case of a conjunction φ and then lift the result to arbitrary formulas ψ. To check satisfiability, it suffices to find a model of φ. For this, we explore minimal Herbrand structures $\mathcal{H} = (\Sigma_\varphi, \mathcal{I}_\mathbf{b})$. The constants are just the ones occurring in φ. The interpretation of the base predicates is unconstrained and can be chosen arbitrarily.

There are two cases to consider. Assume a minimal structure $\mathcal{H}$ satisfying φ is consistent. Due to domain independence, we can enlarge this model to a model over $(\Sigma_{const}, \Sigma_{pred})$. The second case is that all minimal structures $\mathcal{H}$ satisfying φ are inconsistent. We fix such a $\mathcal{H} = (\Sigma_\varphi, \mathcal{I}_\mathbf{b})$ and show that any Herbrand structure $\mathcal{K} = (D, \mathcal{I}'_\mathbf{b})$ with a larger domain $D \supseteq \Sigma_\varphi$ whose interpretation coincides with $\mathcal{I}_\mathbf{b}$ on the common domain Σ_φ must also be inconsistent. By coincide, we mean that $\mathcal{I}'_\mathbf{b}(\mathrm{b})|_{\Sigma_\varphi} = \mathcal{I}_\mathbf{b}(\mathrm{b})$ holds for all base predicates b. This proves the claim, as it rules out the existence of a model for φ over the larger signature.

Let $\mathcal{I} = (\mathcal{I}_\mathbf{b}, \mathcal{I}_\mathbf{d})$ and $\mathcal{I}' = (\mathcal{I}'_\mathbf{b}, \mathcal{I}'_\mathbf{d})$ be the extended interpretations (as defined in Section 2.2) of $\mathcal{H}$ and $\mathcal{K}$, respectively. We prove by induction along a stratification $\mathbf{p}^0, \dots, \mathbf{p}^k$ that for all strata $\mathbf{p}^i$ we have $\mathcal{I}(\mathbf{p}^i) \subseteq \mathcal{I}'(\mathbf{p}^i)|_{\Sigma_\varphi}$, where $\subseteq$ is to be understood pointwise. This yields the desired consequence: any violation (e.g., a cycle) in $\mathcal{H}$ will also be a violation in $\mathcal{K}$. In the base case $\mathbf{p}^0 = \mathbf{b}$, we have $\mathcal{I}(\mathbf{p}^0) = \mathcal{I}(\mathbf{b}) = \mathcal{I}'(\mathbf{b})|_{\Sigma_\varphi} = \mathcal{I}'(\mathbf{p}^0)|_{\Sigma_\varphi}$ by the assumption. Consider now stratum $i{+}1$ with equation $\mathbf{p}^{i+1} = \mathbf{E}^{i+1}(\mathbf{p}^0, \dots, \mathbf{p}^i, \mathbf{p}^{i+1})$. By definition, we have $\mathcal{I}(\mathbf{p}^{i+1}) = \mu x.\mathbf{E}^{i+1}(\mathcal{I}(\mathbf{p}^0), \dots, \mathcal{I}(\mathbf{p}^i), x)$, where μx denotes the least fixed point operator w.r.t. x. To complete the induction step, we argue that the following inequality chain

$$\begin{aligned}
\mathcal{I}(\mathbf{p}^{i+1}) = \mathcal{I}(\mathbf{p}^{i+1})|_{\Sigma_\varphi} &= (\mu x.\mathbf{E}^{i+1}(\mathcal{I}(\mathbf{p}^0), \dots, \mathcal{I}(\mathbf{p}^i), x))|_{\Sigma_\varphi} \\
&\subseteq (\mu x.\mathbf{E}^{i+1}(\mathcal{I}'(\mathbf{p}^0)|_{\Sigma_\varphi}, \dots, \mathcal{I}'(\mathbf{p}^i)|_{\Sigma_\varphi}, x))|_{\Sigma_\varphi} \\
&\subseteq (\mu x.\mathbf{E}^{i+1}(\mathcal{I}'(\mathbf{p}^0), \dots, \mathcal{I}(\mathbf{p}^i), x))|_{\Sigma_\varphi} = \mathcal{I}'(\mathbf{p}^{i+1})|_{\Sigma_\varphi}
\end{aligned}$$

holds. Towards this, note that $\mathcal{I}'(\mathbf{p}^j)|_{\Sigma_\varphi} \subseteq \mathcal{I}'(\mathbf{p}^j)$ and by the induction hypothesis we also have $\mathcal{I}(\mathbf{p}^j) \subseteq \mathcal{I}'(\mathbf{p}^j)|_{\Sigma_\varphi}$ for all $j < i+1$. If $\mathbf{E}^{i+1}$ is monotonic in all its arguments, then the inequality chain follows from the aforementioned observation that all arguments get larger. Now consider the case that $\mathbf{E}^{i+1}$ is not monotonic. By semi-positivity, it can only be non-monotonic in $\mathbf{b} = \mathbf{p}^0$. This already justifies the first inequality, since the only non-monotonic argument $\mathcal{I}(\mathbf{p}^0) = \mathcal{I}'(\mathbf{p}^0)|_{\Sigma_\varphi}$ does not change by the assumption. We need to justify the second inequality. Notice again that the first argument is the only non-monotonic one, so we need to justify that enlarging this argument outside the domain of φ does not reduce the interpretation $\mathcal{I}'(\mathbf{p}^{i+1})$ inside the domain of φ. By normalization, all non-monotonic equations in stratum $i+1$ are of the form `let` d = p \ b, where d is in $\mathbf{p}^{i+1}$, p is in some $\mathbf{p}^j$ with $j \leq i+1$, and b is a base predicate in $\mathbf{p}^0$. Since $\mathcal{I}'$ is the interpretation of a consistent model, it satisfies all equations and, in particular, we have $\mathcal{I}'(\mathrm{d}) = \mathcal{I}'(\mathrm{p}) \setminus \mathcal{I}'(\mathrm{b})$. Now observe that $\mathcal{I}'(\mathrm{d})|_{\Sigma_\varphi} = (\mathcal{I}'(\mathrm{p}) \setminus \mathcal{I}'(\mathrm{b}))|_{\Sigma_\varphi} = \mathcal{I}'(\mathrm{p})|_{\Sigma_\varphi} \setminus \mathcal{I}'(\mathrm{b})|_{\Sigma_\varphi} = \mathcal{I}'(\mathrm{p})|_{\Sigma_\varphi} \setminus \mathcal{I}'(\mathrm{b})$, where the last equality holds because $\mathcal{I}'(\mathrm{b})|_{\Sigma_\varphi}$ and $\mathcal{I}'(\mathrm{b})$ only differ outside the domain of φ. It follows that enlarging $\mathcal{I}'(\mathrm{b})|_{\Sigma_\varphi}$ to $\mathcal{I}'(\mathrm{b})$ does not reduce the value of $\mathcal{I}'(\mathrm{d})|_{\Sigma_\varphi}$. We conclude that the second inequality $(\mu x.\mathbf{E}^{i+1}(\mathcal{I}'(\mathbf{p}^0)|_{\Sigma_\varphi}, \dots, \mathcal{I}(\mathbf{p}^i)|_{\Sigma_\varphi}, x))|_{\Sigma_\varphi} \subseteq (\mu x.\mathbf{E}^{i+1}(\mathcal{I}'(\mathbf{p}^0), \dots, \mathcal{I}(\mathbf{p}^i), x))|_{\Sigma_\varphi}$ holds.

It remains to show that this reasoning generalizes to arbitrary formulas ψ over Σ_{cm}. Suppose ψ is satisfiable and $\mathcal{H}$ is a model. We look at the finite substructure $\mathcal{H}'$ obtained from $\mathcal{H}$ by restricting it to the domain Σ_ψ. To show that $\mathcal{H}'$ is a model of ψ, we need to show that it satisfies ψ on the Boolean level and that it is consistent in our theory. The Boolean satisfaction is immediate because both $\mathcal{H}$ and $\mathcal{H}'$ satisfy, by construction, the same set of $\mathcal{T}_{cm}$-literals over Σ_ψ. To establish consistency, consider the conjunction of all the finitely many literals that hold in $\mathcal{H}'$. This conjunction allows us to apply the previous reasoning and conclude that $\mathcal{H}'$ must be consistent. If it was not, then by semi-positivity the original and larger model $\mathcal{H}$ would be inconsistent as well. This, however, contradicts the assumption that it is a model of ψ, and hence in particular consistent.

On the other hand, if ψ is unsatisfiable, then it is in particular unsatisfiable by any structure over domain Σ_ψ, since any consistent structure over that domain would be extendable to a model over Σ_{cm} by domain independence. In either case, to show satisfiability over Σ_{cm} it suffices to show satisfiability over $(\Sigma_\psi, \Sigma_{pred})$, which concludes the proof. □

An interesting consequence of the proof is that to find models of a conjunction φ one should analyze the associated *canonical structure* $\mathcal{H}_\varphi$ which evaluates exactly the positive literals in φ to true. This structure is interpretation-minimal in the sense that any model of φ (if one exists) has interpretations at least as large as $\mathcal{H}_\varphi$. This leads us to observe three different cases:

(i) Structure $\mathcal{H}_\varphi$ is consistent and hence φ satisfiable.
(ii) Structure $\mathcal{H}_\varphi$ is inconsistent and all interpretation-larger structures over the same domain are also inconsistent. Therefore, φ is unsatisfiable.
(iii) Structure $\mathcal{H}_\varphi$ is inconsistent but there is a consistent structure $\mathcal{K}$ over the same domain satisfying φ. Then $\mathcal{K}$'s interpretation of the base predicates is larger than $\mathcal{H}_\varphi$'s interpretations.

In the last case, we say $\mathcal{H}_\varphi$ is *repairable* by enlarging its interpretations. We make use of this trichotomy in our decision procedure in Section 4. Unless stated otherwise, we consider a normalized, domain independent, and semi-positive consistency model *cm*.

4 SATISFIABILITY MODULO $\mathcal{T}_{cm}$

We propose a decision procedure for $\mathcal{T}_{cm}$ and explain how to integrate it into the general framework of Satisfiability Modulo Theories (SMT) [Barrett et al. 2009].

4.1 Satisfiability Modulo Theories

SMT problems evaluate the satisfiability of Boolean combinations of literals over logical theories (potentially several). The predominant approach to SMT is a *lazy* integration [Sebastiani 2007] of a SAT solver (usually DPLL [Davis et al. 1962; Davis and Putnam 1960]) and several theory solvers. The SAT solver checks the satisfiability of a Boolean abstraction of the given formula (the Boolean abstraction takes each literal for an atomic proposition). If a satisfying assignment can be found, the theory solvers check the set of literals from their theory for satisfiability (in this theory) [Oppen 1980]. If a theory solver detects unsatisfiability, it returns a *theory explanation*, a set of literals that, together, are unsatisfiable in the theory. The SAT solver adds these literals in negation to the formula of interest and repeats the search. Actually, this is called the *offline* integration. There is also an *online* integration, where the SAT engine can query the theory solver with partial assignments. Since queries from the DPLL-engine tend to be frequent in the online integration, *incrementality* of the theory solver is a must-have. Incrementality means the theory solver shall not redo all reasoning if only part of the input changes.

Our contribution is a solver for the theories $\mathcal{T}_{cm}$ that we integrate into the lazy SMT framework. We argue that incrementality is difficult to achieve in our setting, and therefore can only propose an offline integration. The problem with incrementality is that edge deletions do not propagate

through fixed points. Consider the derived relation `let` d = b ∪ d; d. Let x be the least fixed point of the associated function $f(x) = \mathsf{b} \cup x; x$. Imagine b gets updated to $\mathsf{b}' = \mathsf{b} \cup \Delta$, resulting in the new function $g(x) = \mathsf{b}' \cup x; x$. If you iterate from the previous solution x, you get the new fixed point $x' = g^*(x) = g^*(\bot)$, and the fixed-point computation converges faster than starting from $\bot$. The same incremental computation does not work for $\mathsf{b}' = \mathsf{b} \setminus \Delta$, and we do not know of any way but to restart from $\bot$. We consider incrementality and an online integration an interesting (but difficult) problem for future work. As our experiments show, already an offline integration improves the state-of-the-art by up to two orders of magnitude.

4.2 A Decision Procedure for $\mathcal{T}_{cm}$

We develop a decision procedure for the satisfiability of a conjunction φ of $\mathcal{T}_{cm}$-literals. An overview is given on the left. The procedure has four steps, and we claim novel algorithmic contributions in steps two and four. The first step is an initialization that computes a stratification of the consistency model and sets up auxiliary data structures. The second step constructs the canonical structure $\mathcal{H}_\varphi$ described at the end of Section 3.2. In this process, we compute the derived predicates following the stratification. Here, we contribute an interesting idea: we can compute with differences of Kleene approximants rather than full Kleene approximants, an idea inspired by [Bancilhon 1985]. Step three evaluates the structure $\mathcal{H}_\varphi$ for consistency in $\mathcal{T}_{cm}$. If consistent, the formula is satisfiable (we just found a model) and we are done. Otherwise, $\mathcal{H}_\varphi$ is not consistent and we compute a theory explanation in step four. A typical consistency violation is a cycle in a derived relation that contradicts an acyclicity axiom. The theory explanation determines a reason for this cycle in terms of base predicates. The reason computation thus establishes this link between the derived predicates and the base predicates, and is another algorithmic contribution of ours.

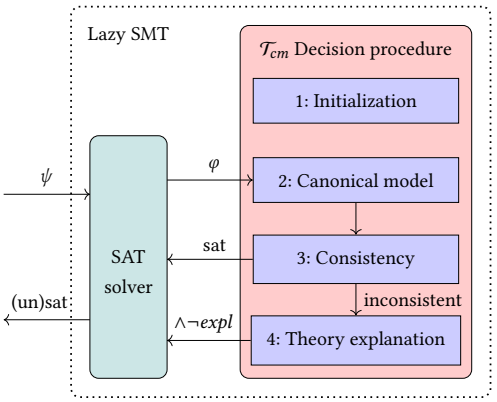

Fig. 5. Lazy SMT

Step 1: Initialization. We determine the dependency relation for the consistency model at hand and represent it as a graph. For this graph, we compute the strongly connected components using Tarjan's algorithm [Tarjan 1971]. Recall that Tarjan produces a topological sort of the SCCs, so this step already gives us a stratification we can work with. We then associate with each unary predicate a *set* and with each binary predicate a directed graph, called its *relation graph*, all initially empty. It is worth noting that the initialization is done only once, even if the SAT and the theory solver evaluate several assignments. The overall initialization takes time linear in the size of the consistency model.

Step 2: Canonical Model. We populate the sets and relation graphs of the base predicates $\mathbf{p}^0$ with the data from $\mathcal{I}_{\mathbf{b}}$, the interpretation given by the SAT solver. We then proceed inductively along the stratification to compute the interpretation $\mathcal{I}_{\mathbf{d}}$ of the derived predicates. When computing the minimal solution of an equation $\mathbf{p}^{i+1} = \mathbf{E}^{i+1}(\mathbf{p}^0, \ldots, \mathbf{p}^{i+1})$, all previous solutions up to $\mathbf{p}^i$ have already been computed and stored in their respective sets and relation graphs. If the current equation system is non-recursive, then $\mathbf{E}^{i+1}$ will be a single operation that matches exactly one rule of the grammar in Figure 2 (due to the normalization of *cm*). Computing the result of such operations is trivial. The (reflexive) transitive closure could be handled by specialized algorithms [Ioannidis

and Ramakrishnan 1988]. We use a recursive formulation, instead, because we need to maintain auxiliary information for the theory explanation.

If the equation is recursive, we solve it using a Kleene fixed-point iteration implemented with a worklist algorithm [Nielson et al. 1999]. What is new in our worklist algorithm is that we work with differences, as first suggested in [Bancilhon 1985]. The items in the worklist are triples (p_1, Δ_1, p_3) with the following meaning. The elements in the set Δ_1 were added to p_1 and need to get propagated to the directly dependent predicate p_3. Initially, the worklist is populated with items (p, Δ, p_1), where p is a predicate from a previous stratum, Δ is the complete set of elements in p, and p_1 is a dependent predicate in the current stratum. We iteratively pick items (p_1, Δ_1, p_3) from the worklist, update p_3, and compute an update Δ_3 that we propagate to all predicates p_4 in the same stratum that depend on p_3 via (p_3, Δ_3, p_4).

The tricky aspect is to devise the update Δ_3. Assume the defining equation is $\texttt{let}\ p_3 = p_1; p_2$. Let $p_1' = p_1 \cup \Delta_1$ denote the new value of p_1. The update (p_1, Δ_1, p_3) leads to the new value

$$p_3' = p_1'; p_2 = (p_1 \cup \Delta_1); p_2 = (p_1; p_2) \cup (\Delta_1; p_2) = p_3 \cup (\Delta_1; p_2) = p_3 \cup ((\Delta_1; p_2) \setminus p_3).$$

Hence, $\Delta_3 := (\Delta_1; p_2) \setminus p_3$ is the update that should be forwarded to the predicates p_4 dependent on p_3. For the remaining operators, the updates Δ_3 are computed as follows from (p_1, Δ_1, p_3):

$\Delta_3 := \Delta_1 \setminus p_2 = \Delta_1 \setminus p_3$	if $p_3 = p_1 \cup p_2$	$\Delta_3 := \mathsf{rng}(\Delta_1) \setminus p_3$	if $p_3 = \texttt{range}(p_1)$
$\Delta_3 := \Delta_1 \cap p_2$	if $p_3 = p_1 \cap p_2$	$\Delta_3 := \Delta_1 \times p_2$	if $p_3 = p_1 \times p_2$
$\Delta_3 := \Delta_1^{-1}$	if $p_3 = p_1^{-1}$	$\Delta_3 := [\Delta_1]$	if $p_3 = [p_1]$
$\Delta_3 := \mathsf{dom}(\Delta_1) \setminus p_3$	if $p_3 = \texttt{domain}(p_1)$.		

The updates based on (p_2, Δ_2, p_3) are similar. The update-based computation still yields the same Kleene approximants for the derived relations, and hence the same fixed point as the standard Kleene iteration. We then proceed with the next stratum in the same way until all strata are covered. The overall statement of soundness, completeness, and complexity is this.

Theorem 4.1. *The update-based worklist algorithm returns $\mathcal{I}_{\boldsymbol{d}}$. The computation of an update for a relational composition and a cartesian product takes time $O(|\Delta| \cdot n)$, where n is the size of the domain. The remaining updates each take time $O(|\Delta|)$. The algorithm is guaranteed not to be less efficient than the standard worklist algorithm, and is considerably more efficient in practice.*

Step 3: Consistency. We need to check the axioms of the consistency model against the predicates we just computed. Emptiness and irreflexivity are straightforward, for acyclicity we again use Tarjan [Tarjan 1971]. If no axiom is violated, then φ is satisfiable because we just found a consistent model $\mathcal{H}_\varphi$. If an axiom is violated, we proceed to step four.

At this point, we have all the ingredients for answering theory queries. Note that for the complexity, *cm* is fixed and φ is the input.

Theorem 4.2. *Given a conjunctive query φ, our theory solver constructs the canonical model $\mathcal{H}_\varphi$ and checks whether it is consistent in time $O(|\varphi|^3)$.*

Proof. Let n be the size of the domain of φ, and thus $n \in O(|\varphi|)$. Let $\Delta_1, \Delta_2, \ldots, \Delta_m$ be the sequence of (not necessarily disjoint) updates that get propagated until a fixed point is reached (we ignore the concrete predicates that get updated). The time to perform these updates is in the worst case $\sum_{i=1}^{m} O(|\Delta_i| \cdot n) = O(n \cdot \sum_{i=1}^{m} |\Delta_i|)$, Theorem 4.1. Now we argue that the sum of all updates $\sum_{i=1}^{m} |\Delta_i|$ is bounded by $O(n^2)$ because *(i)* there are most n^2 many different predicate elements e and *(ii)* every such element appears only $O(1)$ many times over all updates. To see *(ii)*, fix a single element e and observe that every predicate will propagate e at most once (since it can only be added

once) to its dependent predicates. Since the number of predicates is constant (as the consistency model is fixed), this gives $O(1)$ many propagations of e in total. Hence, the total update time is $O(n \cdot \sum_{i=1}^{m} |\Delta_i|) = O(n \cdot n^2) = O(n^3) \subseteq O(|\varphi|^3)$. Checking consistency takes time at most $O(n^2)$ (using Tarjan's algorithm) and is therefore dominated by the update time. If φ describes a set of dense base predicates, that is, when we have $n \in O(\sqrt{|\varphi|})$, then we can even bound the update time by $O(|\varphi|^{\frac{3}{2}})$. □

Step 4: Theory Explanation. If the structure $\mathcal{H}_\varphi$ is inconsistent, the lazy SMT approach expects the theory solver to return a theory explanation to the SAT solver. We elaborate on the shape of this explanation before turning to its computation. A common source of inconsistency is that the graph of a relation r contains a cycle that is forbidden by an axiom **acyclic**(r). We call such a cycle a (consistency) *violation*. To make sure the cycle is not generated again, it may be tempting to return the violation as the theory explanation. This, however, does not work. The shallow reason is that literals over the derived predicates cannot be used in our theory. The deeper reason (and also the reason for this restriction) is that the SAT solver does not understand how the derived predicates are computed from the base predicates. Hence, even if we had a way to directly forbid the violation, the SAT solver could again evaluate the base predicates in a way that leads to the (same) cycle. As a result, the lazy SMT scheme would not make progress towards proving/disproving satisfiability.

To overcome the problem, we introduce the notion of *reason* for a violation. Intuitively, a reason captures how the violation is derived from the base predicates. Technically, a reason is a set of edges (in the case of sets, it is a set of elements) in the base predicates that is guaranteed to lead to a violation (e.g., a cycle) in the derived predicate of interest. Since the SAT solver provides a valuation of the base predicates, reasons are an appropriate way of preventing the violation from being produced again, and thus make progress. In Section 5, we return to the topic and explain how to generalize the set of predicates that the SAT solver understands and that can hence be used for the theory explanation. We restrict the following discussion to the more interesting case of binary predicates, i.e., relations and their relation graphs. We assume for now that there are no unary predicates (sets), but the treatment of those is similar.

Derivation Length. The computation of a reason starts from the derived relation in the violated axiom and proceeds down the dependency graph to the base relations. In the presence of fixed points, one has to be careful to avoid infinite recursion due to cyclic derivations. Furthermore, it is desirable to compute small reasons, because they describe more precisely the root cause of the violation. This rules out more satisfying assignments for the SAT solver, and hence improves the performance of the overall SMT engine.

Fortunately, we can achieve both, avoiding infinite recursion and finding small reasons, with the same concept that we borrow from [Zhao et al. 2019]: we annotate the edges of each relation graph with their *derivation length*. The derivation length is an upper bound on the number of derivation steps needed to derive the edge. Given a relation r and an edge $e \in \mathsf{r}$, we define the derivation length to be $dl(\mathsf{r}, e) := 0$, if r is a base relation, and otherwise

$$dl(\mathsf{r}, e) := \min\{\max\{dl(\mathsf{r}_1, e_1), \ldots, dl(\mathsf{r}_k, e_k)\} + 1 \mid e \text{ can be derived with } e_1, \ldots, e_k \text{ from } \mathsf{r}_1, \ldots, \mathsf{r}_k\}\,.$$

We consider tuples of edges e_1 to e_k that belong to relations r_1 to r_k and that can be used to derive e. The derivation length obtained with such a tuple is $\max\{dl(\mathsf{r}_1, e_1), \ldots, dl(\mathsf{r}_k, e_k)\} + 1$. The derivation length for e is now the minimum over all tuples that can be used to generate e.

The derivation length needs to be determined during the update-based computation of the derived relations, and this computation is both frequent and time consuming. One can show that the precise derivation length (which is defined via a minimum over a set) can be computed in polynomial time: we expand the domain of the relations to also include the derivation length

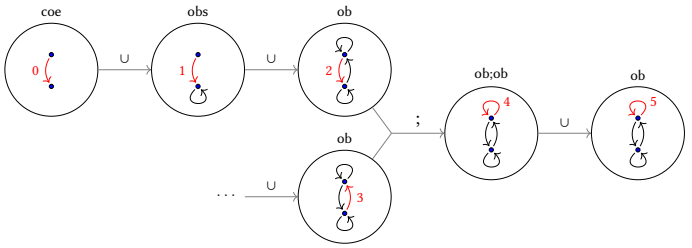

Fig. 6. Example reason computation (right-to-left) under ARMv8.

(initially infinite), and order the same edge with different length as $(e, l_1) \leq (e, l_2)$, provided $l_2 \leq l_1$, meaning a smaller derivation length is better. However, we found this to be both expensive and unnecessary in practice. Instead, we only consider the derivation length as an annotation that does not have an influence on the partial order. When there is more than one way to derive an edge, e.g., when $\mathsf{r}_3 = \mathsf{r}_1 \cup \mathsf{r}_2$, we directly choose the case which leads to a smaller derivation length.

An illustration of the derivation length computation is given in Figure 6. We will give a detailed description of this figure in a moment.

Reasons. With the derivation length at hand, we can give the reason computation. The *reason* for an edge e in a relation r is a conjunction of literals over base relations, denoted by $\mathit{reas}(\mathsf{r}, e)$, that is constructed as follows:

$$\begin{aligned} \mathit{reas}(\mathsf{b}, e) &:= \mathsf{b}(e) && \text{if b is a base relation} \\ \mathit{reas}(\mathsf{r} \setminus \mathsf{b}, e) &:= \mathit{reas}(\mathsf{r}, e) \wedge \neg \mathsf{b}(e) && \text{if b is a base relation} \\ \mathit{reas}(\mathsf{r}, e) &:= \mathit{reas}(\mathsf{r}_1, e_1) \wedge \ldots \wedge \mathit{reas}(\mathsf{r}_k, e_k) && \text{if (*).} \end{aligned}$$

Constraint (*) says that e can be derived from the edges e_1 to e_k in the relations r_1 to r_k, and for all $1 \leq i \leq k$ we have $\mathit{dl}(\mathsf{r}_i, e_i) < \mathit{dl}(\mathsf{r}, e)$. Strictly speaking, the last rule is non-deterministic, and thus *reas* is a relation rather than a function. We elaborate in a moment on how we fix the choice of e_1 to e_k. The definition of reasons is another testament to the importance of semi-positivity. If the consistency model were not semi-positive, the second rule would have to consider $\mathsf{r}_1 \setminus \mathsf{r}_2$ and compute all possible reasons of why $\mathsf{r}_2(e)$ might be absent. We would then lose the ability to express reasons as conjunctions. Even worse, these reasons might need to refer to elements outside the current domain, as we have discussed in Section 3.

Figure 6 shows how to compute the reason of a single violating edge in ARMv8. To keep the example simple, we only use a partially normalized model. The nodes of the relation graphs are v_1 and v_2, where v_1 is denoted by the upper dot. The edges between the nodes denote relationships that hold. There are two consistency violations in the right-most graph, namely the two self-loops $\mathsf{ob}(v_1, v_1)$ and $\mathsf{ob}(v_2, v_2)$ which both violate the axiom **irreflexive**(ob). We focus on the former and highlight its edge red. The red edges of the other graphs explain this violation and, in particular, $\mathsf{coe}(v_1, v_2)$ is (part of) a reason in terms of base predicates. The numbers labeling these red edges show their associated derivation length. The other edges also have a derivation length, which we ignore for this example. It is important to note that the self-loop $(\mathsf{ob}; \mathsf{ob})(v_1, v_1)$ cannot be explained by the composition of $\mathsf{ob}(v_1, v_1)$ with itself, because the latter has a higher derivation length, namely $\mathit{dl}(\mathsf{ob}, (v_1, v_1)) = 5 > 4 = \mathit{dl}(\mathsf{ob}; \mathsf{ob}, (v_1, v_1))$. This makes sure that we do not get stuck in cyclic reasoning.

A single edge may have several derivations, for example, when the relation is defined via union or relational composition. When defining the reason, we focus on a single and short derivation.

For unions, we proceed along the relation graph where the edge has the least derivation length. For relational compositions, we choose any pair of edges that satisfies the condition. Finding the smallest pair is expensive and the bound on the derivation length is sufficient to obtain small reasons in practice. For transitive closures, we use a shortest path algorithm, making sure that only edges with strictly smaller derivation length are used.

The theory explanation of interest is based on these reasons.

Theory Explanation. To return a theory explanation to the SAT solver, the theory solver selects a set E of edges that cause a violation. For an acyclicity axiom, this set will form a cycle; for emptiness and irreflexivity axioms, it will be a singleton edge. We elaborate in a moment on our selection strategy if there are several sets of edges that can be chosen as E. The *explanation for E in r* is the formula $\mathit{expl}(\mathsf{r}, E) := \bigwedge_{e \in E} \mathit{reas}(\mathsf{r}, e)$. It is worth studying this formula more closely.

The theory explanation allows us to distinguish formulas φ that are unsatisfiable from formulas that might become satisfied by repairing their canonical models. If $\mathit{expl}(\mathsf{r}, E)$ is a subformula of φ, then φ is unsatisfiable and the reason is an unsat core.

Lemma 4.3. *If $\mathit{expl}(r, E)$ is not included in φ (i.e., it is not a subset), then $\mathit{expl}(r, E)$ contains at least one literal $\neg b(e)$ where $b(e)$ occurs neither positively nor negatively in φ.*

Proof. To see the lemma, note that all rules in the reason computation except $\mathit{reas}(\mathsf{r}' \setminus \mathsf{b}, e)$ propagate down the existence of edges to the base relations. The edges in the base relation stem from atoms that were chosen from (evaluated to true as proposition in) the Boolean abstraction of φ. Hence, if the inclusion fails, we will have used the rule for $\mathit{reas}(\mathsf{r}' \setminus \mathsf{b}, e)$ at least once. If all negative literals $\neg\mathsf{b}(e)$ were present in φ, the inclusion would have held, so there must exist a negative literal $\neg\mathsf{b}'(e')$ that was obtained when computing $\mathit{reas}(\mathsf{r}' \setminus \mathsf{b}', e')$ for some predicate r', base predicate b', and edge e'. We argue that the positive literal $\mathsf{b}'(e')$ is also not present, because if it were, we would not have encountered edge e' in $\mathsf{r}' \setminus \mathsf{b}'$ and hence never evaluated $\mathit{reas}(\mathsf{r}' \setminus \mathsf{b}', e')$ to begin with. □

Assume the inclusion fails and the SAT solver adds the negation of the explanation to the formula, meaning the next formula to consider is $\varphi \wedge \neg\mathit{expl}(\mathsf{r}, E)$. With the lemma at hand, the explanation will have the shape

$$\bigwedge_i \mathsf{b}_i(e_i) \;\rightarrow\; \bigvee_j \mathsf{b}_j(e_j).$$

Moreover, unless $\mathit{expl}(\mathsf{r}, E) \subseteq \varphi$, some of the $\mathsf{b}_j(e_j)$ do not occur in φ. Any $\mathsf{b}_j(e_j)$ that does not already occur in φ can be understood as a way to repair the inconsistency in the previous satisfying assignment to the Boolean abstraction of φ: enlarge the interpretation of b_j in $\mathcal{H}_\varphi$ to include e_j. It is tempting to implement this modification inside the theory solver. The problem is that the new interpretation may lead to new consistency violations, which calls for backtracking. By adding the formula to the outer SAT solver, we get backtracking for free. Conveniently, we do not need to distinguish the case where φ is unsatisfiable from the case where inconsistencies can be repaired. We just add the negation of the explanation to the formula and the SAT solver will automatically perform the correct task.

When integrating our theory solver into a lazy SMT scheme, we obtain a decision procedure for arbitrary Boolean formulas ψ formed over our theory. Due to semi-positivity, all reasons refer to the domain induced by ψ. Thus, repairing inconsistencies can only be done by changing interpretations but not by changing the domain as we have shown in Theorem 3.4. This bounds the number of models that need to be explored and thus guarantees termination.

Theorem 4.4. *When integrating our theory solver into a lazy SMT scheme, we obtain a decision procedure for satisfiabilty of arbitrary Boolean formulas ψ over $\mathcal{T}_{cm}$.*

We argue that due to the offline integration a failure of $expl(\mathrm{r}, E) \subseteq \varphi$ is rare, meaning the interaction between the SAT solver and the theory solver will terminate quickly. Behind this is the assumption that φ refers to many theory literals. The reason is that in the offline approach the SAT solver will produce a conjunction φ that assigns a truth value to all theory literals appearing in the formula of interest ψ. Therefore, if the inclusion failed to hold due to a literal $\neg\mathrm{b}(e)$, the formula ψ would need to not contain the atom $\mathrm{b}(e)$ at all, meaning it does not constrain the base relation b over the edge e. In our application of the theory solver described in the following Section 6, we always have assignments to all theory literals and no failure of the implication at all.

Theorem 4.5. *Let ψ be a Boolean formula over $\mathcal{T}_{cm}$. If φ is an unsatisfiable conjunction of theory literals from ψ that fully defines all base predicates over the domain Σ_ψ, then $expl(r, E) \subseteq \varphi$ holds.*

Proof. Since φ fully defines all base predicates over Σ_ψ, it must contain either literal $\mathrm{b}(e)$ or literal $\neg\mathrm{b}(e)$ for each base predicate b and edge (or element) e over domain Σ_ψ. So by contraposition of Lemma 4.3, the theorem holds. □

Choosing Violations. A single structure $\mathcal{H}_\varphi$ may contain several consistency violations, from the same or from different axioms. While computing an explanation for a single violation guarantees progress for the overall SMT-engine, this causes many calls to the theory solver. On the other extreme, we could compute all reasons for all violations of all axioms. This yields the strongest explanation possible, but it can take exponential time and hence is infeasible. A practical approach has to balance between explanation strength and computation efficiency.

As mentioned above, we compute small reasons per violation. For emptiness and irreflexivity axioms, we simply collect all violations as their size is polynomially bounded. For acyclicity, a heuristic to obtain small reasons is to choose short cycles. However, the set of shortest cycles in a graph may still be exponential in the size of the graph, and we cannot efficiently explore them all. Furthermore, a long cycle may provide a valuable explanation if it is highly independent of the shorter cycles.

In our solver, we use the following heuristic to choose violations for an acyclicity axiom. We compute a *vertex cycle cover* [West 2000] for each strongly connected component (SCC) in the relation graph. A vertex cycle cover of a graph is a set of cycles such that every node is contained in at least one cycle. Obviously, a node can only ever be covered by a cycle if it is part of some SCC. Since we use Tarjan's algorithm when performing the acyclicity check, we already know all SCCs when computing the theory explanation. For each SCC, we mark all of its nodes as unvisited and iteratively compute a vertex cycle cover as follows: *(i)* choose an unvisited node v, *(ii)* compute a shortest path from v to itself (i.e., a shortest cycle), *(iii)* mark all nodes of the cycle as visited, and *(iv)* repeat until all nodes are visited.

The above algorithm to find violations is polynomial in time. Despite this, the explanation computation may still be exponential because the recursion in the definition of *reas* may descend polynomially many times resulting in a computation tree of polynomial height and potentially exponential size overall. While we did not encounter this case in practice, one may wonder if we can do better in the worst case. The answer is yes. To see this, first observe that if the domain has size n, there are at most $O(n^2)$ many edges in total (we assume cm to be fixed and not part of the input for the complexity analysis). Hence, any exponentially-sized computation tree has to visit the same edges multiple times. If we employ memoization, we can avoid all duplications in this tree and truncate it down to size $O(n^2)$. Next, observe that the explanation of an edge can be of size at most $O(n^2)$ if it refers to all possible literals over base predicates. This means we can compute the

explanation of any node in the computation tree in time $O(n^2)$ given that the nodes below it are already computed. Overall, this gives a worst-case time of $O(n^4)$ to compute the explanation of a violation.

Theorem 4.6. *If $\mathcal{H}_\varphi$ is inconsistent, one can compute theory explanations in time $O(|\varphi|^4)$.*

5 CUTTING

Our theory solver hinges on the assumption of semi-positivity: after normalization all difference constraints $\mathsf{r} \setminus \mathsf{b}$ in the consistency model have to subtract a base relation b. In the context of memory models (cf. Section 7), important representatives are semi-positive, including hardware memory models like TSO, ARMv8, and Power, as well as language memory models like LKMM, RC11, and IMM. The memory models of RISC-V and C11, however, are not semi-positive. The full C11 model is known to be complex and problematic [Batty et al. 2015, 2011; Lahav et al. 2017], and the popular approximations IMM [Podkopaev et al. 2019] and RC11 [Lahav et al. 2017] are semi-positive. Not being able to support RISC-V is a limitation that we address here.

First, it is worth noting that rewriting steps may be able to make a non-semi-positive consistency model semi-positive. We can use De Morgan's laws to push differences in derived predicates down to the base predicates, if possible. Here we use the identities *(i)* $\mathsf{c} \setminus (\mathsf{a} \cup \mathsf{b}) = (\mathsf{c} \setminus \mathsf{a}) \setminus \mathsf{b}$, *(ii)* $\mathsf{c} \setminus (\mathsf{a} \cap \mathsf{b}) = (\mathsf{c} \setminus \mathsf{a}) \cup (\mathsf{c} \setminus \mathsf{b})$, and *(iii)* $\mathsf{c} \setminus (\mathsf{a} \setminus \mathsf{b}) = (\mathsf{c} \setminus \mathsf{a}) \cup (\mathsf{c} \cap \mathsf{b})$. In general, this rewriting may fail to restore semi-positivity, and in particular it fails for C11 and RISC-V because the difference operator is not compatible with relational composition. We need a more general approach.

We introduce *cutting* as a method that makes consistency models semi-positive. Cutting turns derived predicates into base predicates by removing (cutting) their defining equations from the consistency model. This comes at the price of having to eagerly encode the removed equations.

To make the idea of cutting precise, consider the consistency model *cm* and remember that the stratum p_0 contains the base predicates. We show how to turn the predicates in stratum p_1 into base predicates. For simplicity, assume p_1 only consists of the relation d that is defined by the equation $\texttt{let}\ \mathsf{d} = \mathsf{b}_1; \mathsf{b}_2$. The trick is to determine a formula Φ_{p_1} capturing (in the expected way) this defining equation. To construct the formula, we can use all techniques that have been developed in earlier works [Ponce de León et al. 2017; Wickerson et al. 2017]. In particular, Φ_{p_1} may refer to additional theories $\mathcal{T}_1$ to $\mathcal{T}_m$. The defining equation is then removed from the consistency model *cm*, leading to *cm′*. As for the satisfiability of a given formula, the relationship between the consistency models is this.

Theorem 5.1. *$\mathcal{T}_{Data} + \mathcal{T}_{cm} \models \psi$ if and only if $\mathcal{T}_{Data} + \mathcal{T}_{cm'} + \mathcal{T}_1 + \ldots + \mathcal{T}_m \models \psi \wedge \Phi_{p_1}$.*

With cutting, semi-positivity is no longer a limitation: one can always lift the problematic predicates to base predicates.

Corollary 5.2. *Every consistency model can be made semi-positive.*

We illustrate this on the example of RISC-V. The model contains the following defining equations: $\texttt{let}\ \mathsf{r2} = ([R]; \mathsf{po\text{-}loc\text{-}no\text{-}w}; [R]) \setminus \mathsf{rsw}$ and $\texttt{let}\ \mathsf{rsw} = \mathsf{rf}^{-1}; \mathsf{rf}$. We eagerly encode the second equation into a formula Φ_{rsw} and remove it from the consistency model, resulting in RISC-V′. Since rsw no longer has a defining equation, it is a base predicate and RISC-V′ is a semi-positive model. The practicality of cutting heavily depends on the complexity of the derived relations used underneath a difference operator. In the case of C11, for example, the problematic difference operator involves deeply nested derivations, and removing them would mean eagerly encoding half of the consistency model.

Although cutting increases the size of the formula, it may bring advantages for the reasoning engine. Consider the hypothetical model *cm′* from above and assume our theory solver computes a

reason containing the literal $\mathsf{d}(v_1, v_2)$. This literal represents all possible derivations of $\mathsf{d}(v_1, v_2)$, that is, the combination of $\mathsf{b}_1(v_1, v)$ and $\mathsf{b}_2(v, v_2)$ for all v. Hence, a reason containing $\mathsf{d}(v_1, v_2)$ is more powerful in reducing the search space than a reason that splits it into a concrete combination, e.g., $\mathsf{b}_1(v_1, v_3) \wedge \mathsf{b}_2(v_3, v_2)$.

Whether cutting a consistency model is beneficial depends on the use-case. We now propose a static approximation to determine when and where to cut. Consider again the dependency graph among the relations in the consistency model. Understand the base predicates as source nodes and the derived predicates that are used in axioms as sinks. Any *vertex cut* [West 2000] that separates the sources from the sinks (it may contain nodes from both of them) can be used for cutting the consistency model. Let S be such a vertex cut. Then any violation can be reduced to a reason over atoms $\mathsf{r}(x, y)$ with $\mathsf{r} \in S$. Now, we can partially encode the consistency model up to cut S and let our theory solver only generate reasons w.r.t. S.

If most time is spent in the theory solver, cutting can be used to balance the workload and put more effort onto the SMT-engine. This needs a measure for the quality of a vertex cut. Vertex cuts are preferable that involve fewer nodes, as in the example of the relation d assuming b_1 and b_2 are not used elsewhere. More involved measures are possible, like giving a weight to the relations depending on their expected size. We found that setting S to the base relations and moving most reasoning to our theory solver gave the most consistent performance. However, we see room for improvement by optimizing eager encodings with the help of static analyses [Gavrilenko et al. 2019; Nielson et al. 1999].

Another limitation of our theory is that formulas can only talk about base predicates and not about derived predicates. Being able to talk about derived predicates could be useful to formulate constraints that are outside our language like, e.g., functionality or totality constraints. Cutting can also address this limitation by making the derived predicate base and hence directly addressable in the formula. This adds flexibility to our approach.

6 BMC WITH MEMORY MODELS AS INPUTS

We use our new theory solver to tackle the problem stated in Section 1: verifying a given program relative to a given memory model. We haven't explicitly mentioned the syntax of the programs. Since we deal with a variety of hardware and language memory models, the reader might wonder *"can you have one instruction set to rule them all?"*. The answer is yes. The C-code is compiled via LLVM-intermediary to Boogie [Leino 2008] which is then converted to a LISA-like [Alglave and Cousot 2016] internal representation (IR). The reason the IR is so powerful is that instructions can bear tags which can be used to model, e.g., C11 release and acquire annotations, or scopes like in NVIDIA PTX [Alglave et al. 2015]. Concretely, tags are simply base unary predicates in Figure 2.

While there exist theories for concrete models like SC, TSO and PSO (cf. Section 8), this is the first time a lazy SMT encoding is achieved for arbitrary models in the context of memory model-aware verification. This brings several advantages over other approaches. As we will demonstrate (cf. Section 7), the performance of the lazy approach is better than for Dartagnan, and despite its flexibility, in many cases it outperforms stateless model checking techniques. Compared to a tricky eager SMT encoding or an intricate stateless model checking algorithm, the lazy SMT encoding is simpler, and hence easier to implement and to maintain. This simplicity is bought at the price of having to develop the theory solver. This solver, however, achieves the right separation of concerns: it is a stand-alone artifact that interacts through a clearly defined and stable interface with the rest of the SMT machinery.

Consider a program $\mathcal{P}$ annotated by assertions and unrolled in preparation to the bounded model checking. Remember from Section 1 that the correctness of $\mathcal{P}$ under a memory model $\mathcal{M}$ is formulated as $[\![\mathcal{P}_{\text{🐞}}]\!] \cap [\![\mathcal{M}]\!] = \emptyset$. The eager SMT encoding used in Dartagnan would devise

```
1  atomic_bool f, g, x;
2
3  void thread_f() {
4      store(&f, 1, REL);
5      store(&t, 2, REL);
6      if(load(&g, ACQ) != 1 ||
7            load(&t, ACQ) != 2) {
8        /* critical section begins */
9        store(&x, 1, REL);
10       assert(load(&x, ACQ) == 1);
11       /* critical section ends */
12       store(&f, 0, REL);
13     }
14 }
```

```
15 atomic_int t;
16
17 void thread_g() {
18     store(&g, 1, REL);
19     store(&t, 1, REL);
20     if(load(&f, ACQ) != 1 ||
21           load(&t, ACQ) != 1) {
22       /* critical section begins */
23       store(&x, 2, REL);
24       assert(load(&x, ACQ) == 2);
25       /* critical section ends */
26       store(&g, 0, REL);
27     }
28 }
```

Fig. 7. Peterson's mutual exclusion algorithm.

a formula $\Phi_{\mathcal{M}}$ over $\mathcal{T}_{SAT} + \mathcal{T}_{IDL}$ that captures memory model-specific information and check the (un)satisfiability

$$\mathcal{T}_{Data} + \mathcal{T}_{SAT} + \mathcal{T}_{IDL} \models \Phi_{\mathcal{P}} \wedge \Phi_{🐞} \wedge \Phi_{\mathcal{M}}.$$

With our new theory solver, we can skip the formula $\Phi_{\mathcal{M}}$. Instead, the memory model is understood as a consistency model (there is not even a translation, the memory model matches our format) and turned into a theory $\mathcal{T}_{\mathcal{M}}$. We thus check[5]

$$\mathcal{T}_{Data} + \mathcal{T}_{\mathcal{M}} \models \Phi_{\mathcal{P}} \wedge \Phi_{🐞}.$$

By adding the new theory solver, the SMT engine understands the influence of the base relations used in the program encoding on the derived relations defined in the memory model. It is worth noting that the encoding of the program and its specification is the same for the eager and the lazy encoding, so our implementation relies on the Dartagnan infrastructure, which is open source.

We use Peterson's mutual exclusion algorithm from Figure 7 to illustrate how the lazy SMT scheme checks the $\mathcal{T}_{Data} + \mathcal{T}_{\mathcal{M}}$-satisfiability of $\Phi_{\mathcal{P}} \wedge \Phi_{🐞}$. Peterson's algorithm uses the shared variables `f`, `g`, `t`, and `x`. The first two are Boolean flags indicating whether the threads want to enter their critical sections. In case both threads want to enter simultaneously, the variable `t` is used to decide who may enter first. The last variable `x` represents the shared data that must be accessed exclusively. To check mutual exclusion, after writing the data to `x`, the thread reads the value of `x` and makes sure the value has not been modified before leaving the critical section.

For Peterson's algorithm to work as expected, it is crucial that each thread observes when the other one wants to enter its critical section. Under C11, Peterson fails when using the ACQ/REL memory tags in Figure 7. The release-acquire (RA) semantics guarantees that an acquire-load cannot be reordered with instructions following in program order. Similarly, a release-store cannot be reordered with instructions preceding it in program order. However, RA does not prohibit reordering the store from line 4 (resp. 18) with the load from line 6 (resp. 20). As explained above, those two orders are fundamental for the correctness of Peterson's algorithm. Despite being unsafe for C11, we will show that the program is safe under the ARMv8 model from Figure 3.[6] This is because hardware memory models tend to be stricter (admit fewer behaviors) than the memory models of high-level languages like C11.

Figure 8 illustrates the first and the last two calls to our theory solver in the lazy SMT scheme. For each call, we show the program behavior violating some assertion (left) and how this behavior is inconsistent (right) according to ARMv8. For easy understanding, we show ob-cycles whose

[5]To be precise, we still require $\mathcal{T}_{SAT}$ to encode the control flow of the program and $\mathcal{T}_{IDL}$ to encode totality of certain relations. However, the formulas over those theories in the lazy encoding are much smaller than $\Phi_{\mathcal{M}}$ in the eager encoding.

[6]The program from Figure 7 needs to be transformed to ARMv8's low-level assembly. This is done via compiler mappings (https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html). Since ARMv8 has explicit support for ACQ/REL memory accesses, for this example the transformation is just the identity.

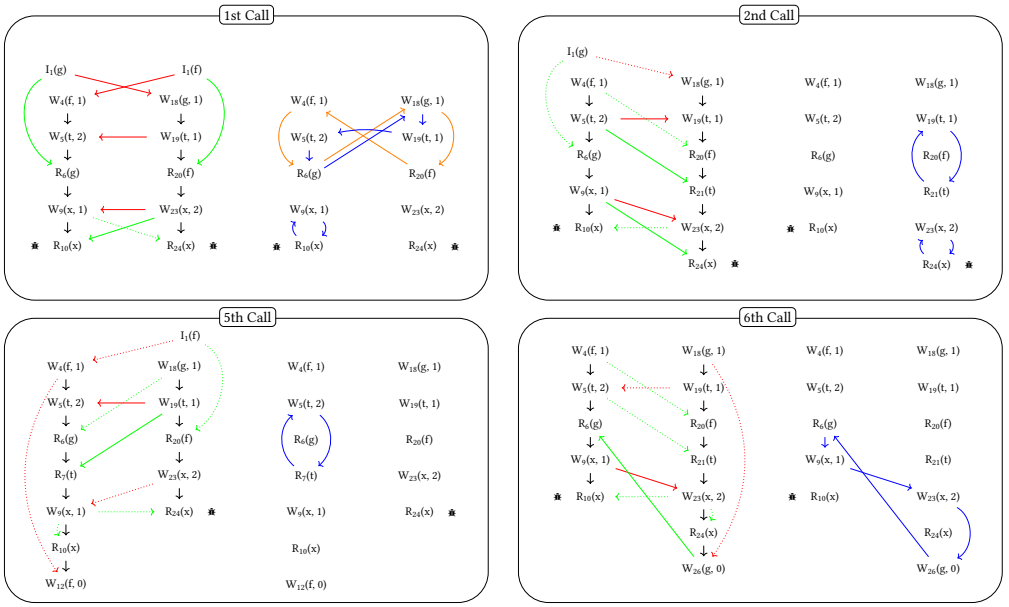

Fig. 8. Analyzing Peterson's algorithm under ARMv8. The left-hand graph in each call shows the program execution with green edges standing for the rf-relation, red edges for the co-relation, and black edges for the po-relation. The full and colored edges in the left-hand graph are those that participate in some of the consistency violations shown in the right-hand graph.

transitive closures cause the self-loops that violate the constraint **irreflexive**(ob). This better illustrates the cyclic dependencies that lead to inconsistencies. Under weak memory models, the program behavior is represented by graphs where nodes model instructions and edges model dependencies. Each node is either a store (W), load (L), or an initialization (I). We use subscripts to match nodes with the corresponding lines of code in Figure 7.

The first violating behavior is a solution to $\Phi_{\mathcal{P}} \wedge \Phi_{\text{🐞}}$ in which the load in each critical section gets its value from (represented by the green edges) the store of the other thread, thus clearly violating mutual exclusion. This execution exhibits three different inconsistencies with ARMv8 illustrated by the three cycles to the right. We cannot tell the SAT solver to forbid those cycles by adding, e.g., the constraint $\neg\mathsf{ob}(W_4, W_4)$. The point is that ob is a derived predicate (defined in Figure 3), and the SAT solver does not understand the interplay between the base predicates and the derived ones. To transform the inconsistency information into something the SAT solver can reason about, our theory solver determines the reason for the induced self-loop $\mathsf{ob}(W_4, W_4)$ (as well as the reasons for the other violations). The reason is the conjunction of base predicate literals

$$\mathsf{rfe}(I_g, R_6) \wedge \mathsf{coe}(I_g, W_{18}) \wedge \mathsf{L}(W_{18}) \wedge \mathsf{po}(W_{18}, R_{20}) \wedge \mathsf{A}(R_{20}) \wedge$$
$$\mathsf{rfe}(I_f, R_{20}) \wedge \mathsf{coe}(I_f, W_4) \wedge \mathsf{L}(W_4) \wedge \mathsf{po}(W_4, R_6) \wedge \mathsf{A}(R_6)\ .$$

In general, our theory solver determines not only one but several reasons for inconsistency and reports them in the form of a theory explanation to the SAT solver. The solver adds the negation of the theory explanation to the formula and repeats the process. In this example, after six iterations, the solver finds that $\Phi_{\mathcal{P}} \wedge \Phi_{\text{🐞}} \wedge \neg expl^1 \wedge \ldots \wedge \neg expl^6$ is unsatisfiable and thus the program safe.

Table 1. Tools used in the evaluation and their memory model support. ✔: fully supported by the theory and implemented, ✘: not supported by the theory or not implemented, !✔: supported by the theory, but not (or only partially) implemented.

	Format		Memory Consistency								
Tool	`litmus`	`C`	Parametric	SC	TSO	Power	ARMv8	RISC-V	IMM	RC11	LKMM
Herd	✔	✘	✔	✔	✔	✔	✔	✔	!✔	✔	✔
Dartagnan	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Nidhugg	✘	✔	✘	✔	✔	✔	✘	✘	✘	✘	✘
GenMC	✘	✔	!✔	✔	!✔	!✔	!✔	!✔	✔	✔	✔
Caat	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

7 EVALUATION

We implemented the approach described in Section 6 as an extension of Dartagnan, a BMC tool which takes memory models as inputs. In the remainder of this section, we use "Dartagnan" to refer to the default version and "Caat" to refer to our extension. Both Dartagnan and Caat are open source and available at https://github.com/hernanponcedeleon/Dat3M.

We compare Caat to several state–of-the-art tools for program verification under memory models. Table 1 summarizes the tools we selected for the evaluation. Section 8 discusses their underlying techniques and limitations. Herd [Alglave et al. 2014] is parametric since it takes the memory model as an input. Since it exhaustively generates *all* violating executions and only then filters the non-consistent ones, it can only verify small programs (i.e., litmus tests). Dartagnan [Ponce de León et al. 2020] also takes memory models as inputs, but it implements the BMC approach from Figure 1 using the formula $\Phi_{\mathcal{P}} \wedge \Phi_{\text{🐞}} \wedge \Phi_{\mathcal{M}}$. This improves scalability w.r.t. the exhaustive approach of Herd, thus it can verify not only litmus tests, but also more complex programs written in C. Nidhugg [Abdulla et al. 2014] and GenMC [Kokologiannakis and Vafeiadis 2021] perform stateless model checking (SMC) of C programs. Nidhugg supports SC, TSO and Power (also a simplified version of ARMv7), but not any of the other models. GenMC builds on top of axiomatic semantics and supports SC, IMM [Podkopaev et al. 2019], RC11 [Lahav et al. 2017] and (a simplified version of) LKMM, but not TSO, Power, or ARMv8.

The remainder of this section provides answers to the following research questions:

(RQ1) Can Caat handle (a large subset of) most well known memory models?
(RQ2) How does the performance of Caat compare against tools supporting memory models?
(RQ3) What is the impact in the performance of Caat when using cutting?

Experimental Setup. The evaluation was done on a MacBook Pro with an Intel Core i7 CPU (4 cores @2.8 GHz) and 16 GB of RAM. We used the following versions of the tools: Herd 7.56, Dartagnan 3.1.0, Nidhugg 0.2, GenMC v0.8.

To answer (RQ1) we run Caat on a large set of litmus tests using five memory models covering a large subset of the language in Figure 2 (including fixed points and predicate differences). The set contains established benchmarks that have been collected during a community effort to formalize and validate memory consistency models [Alglave et al. 2018, 2014; Pulte et al. 2018; Sarkar et al. 2011]. Tests are distributed as follows: 487 for TSO, 2362 for Power, 5141 for ARMv8, 6661 for RISC-V, and 5013 for LKMM. Note that the consistency models of ARMv8 and LKMM are still under constant development. For our evaluation we used the versions from [Pulte et al. 2018] and [Alglave et al. 2018]. We run the same tests with Herd and Dartagnan which also support all five memory models. The results of all three tools match, showing that Caat is precise, its implementation correct, and that it can handle the intrinsics of complex memory models.

On the right we show the accumulated solving times of each tool grouped by the consistency model. Even on these small examples, Caat performs better than the other tools.

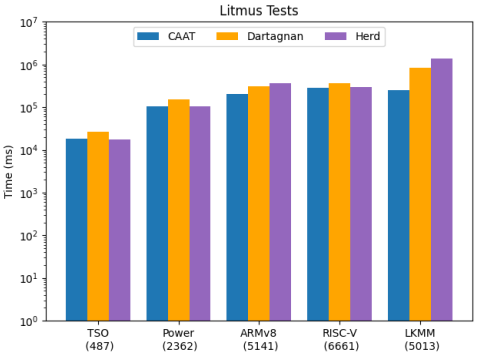

Fig. 9. Performance on litmus tests.

To answer (RQ2), we run Caat, Dartagnan, GenMC and Nidhugg on twelve realistic benchmarks containing different lock implementations and lock-free data structures using C11 atomics. These benchmarks have been previously used to evaluate the performance of tools supporting both C programs and memory models [Kokologiannakis et al. 2019a,b; Kokologiannakis and Vafeiadis 2021; Oberhauser et al. 2021]. The suffix in the name of each benchmark specifies the number of threads. All loops were unrolled once, so we end up with two copies of each loop body. We use the following consistency models: TSO, Power, ARMv8, RISC-V, IMM, and RC11.

Since RISC-V is not semi-positive, running Caat on this model already forces us to use cutting. To answer (RQ3), however, we want to compare the performance of Caat when run on two equivalent models, one using cutting and one not. We thus force the Caat solver to cut each of the models (except RISC-V where cutting is actually required) by eagerly encoding relation `let` $\mathsf{fr} = \mathsf{rf}^{-1}; \mathsf{co}$ which is commonly used by all memory models.

The results of the verification are given in Figure 10. Dartagnan (and thus Caat) uses the JavaSMT library [Baier et al. 2021; Karpenkov et al. 2016] to support different SMT solvers. For this set of benchmarks we report the results obtained by using Z3 [De Moura and Bjørner 2008] which yields the best overall performance. The last set of bars of each figure represents the arithmetic mean $\overline{X}$ of each tool. Nidhugg supports neither ARMv8 nor CAS instructions with acquire/release tags (which all benchmarks use) under Power, thus we omit it in the evaluation of those memory models. GenMC supports IMM and RC11, but none of the hardware models, thus we also omit it in the evaluation of hardware models.

From a verification perspective, the models of TSO, ARMv8 and RISC-V can be considered "simple" since they do not contain fixed points (e.g., transitive closures). Because of this, the improvement of Caat w.r.t. the eager encoding of Dartagnan is only marginal. Since cutting encodes more than the lazy approach, but less than the eager one, its performance sits in between those two approaches. Compared to Nidhugg, Caat is 5x faster on average.

In the presence of transitive closures (IMM and RC11), Caat is 5x faster than Dartagnan on average, and never slower. For some benchmarks, it can be at least one order of magnitude faster (e.g., `dglm`, `ms`, and `treiber`). Despite Caat being more general than GenMC, the performance of both tools is similar on average. While in cases like `mutex` and `mutex-musl` Caat is at least one order of magnitude faster, the opposite can also be true. In particular, in benchmarks with a small search space like `linuxrwlock` (only 24 executions according to GenMC) or using bit-precise reasoning like `harris`, GenMC performs better than Caat.

When going from transitive closures to the more complex fixed points found in Power, the advantage of Caat becomes even more apparent. For this model, Caat is on average one order of magnitude faster than Dartagnan, and never slower. In some cases it can be at least two orders faster (e.g., `mutex` and `lev`). Interestingly, while cutting is not required for models like TSO, Power and ARMv8, in some cases it actually reduces the verification times, see e.g., `ticketlock` on TSO, `spinlock` on Power, or `treiber` on ARMv8.

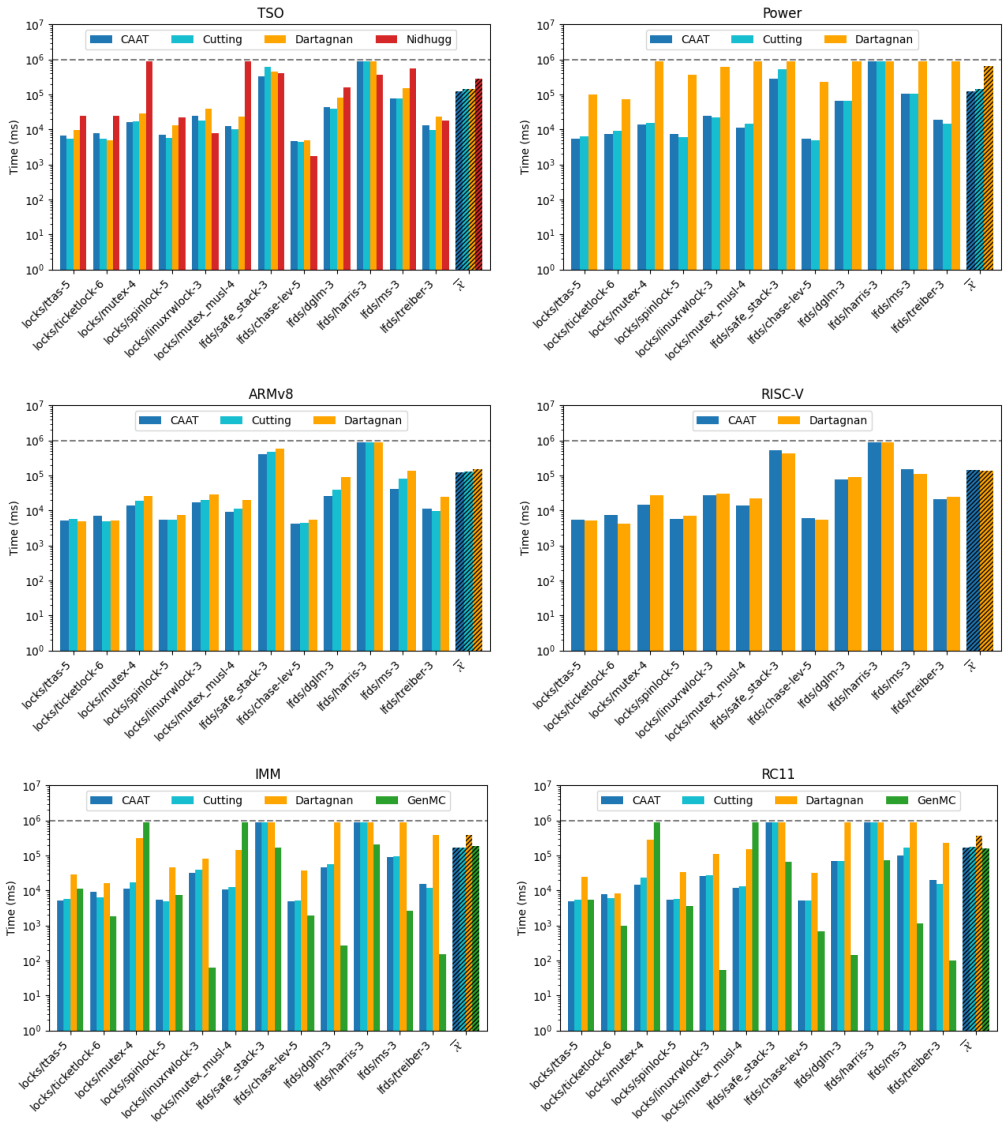

Fig. 10. Performance of tools supporting different memory models.

Bias. In Section 5 we talked about the possibility to eagerly encode parts of the consistency model by cutting derived predicates and turning them into base predicates. This shifts workload from our procedure onto the rest of the SMT-engine. A related, but slightly different approach, is to eagerly encode a model cm_{weak} that is weaker than the intended model cm but simple enough to admit small encodings. By being weaker, we mean that any consistent model of cm_{weak} is also consistent for cm (we assume both models to be over the same set of base predicates). We looked into memory consistency models and identified a few simple constraints most memory models satisfy. These are: (RMW) $\mathbf{empty}(\mathsf{rmw} \cap \mathsf{fre}; \mathsf{coe})$ which enforces atomicity, (Uniproc) $\mathbf{acyclic}(\mathsf{po\text{-}loc} \cup \mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr})$ modeling SC per location, and (Oota) $\mathbf{acyclic}(\mathsf{dep} \cup \mathsf{rf})$ forbidding out-of-thin-air values. When eagerly encoding these constraints we call them *bias constraints*. The first two constraints are found in all memory consistency models, which makes them universally usable as biases. The

Table 2. Performance (given in ms) of Caat using different biases to verify Qspinlock [Paolillo et al. 2022].

LKMM				ARMv8				Power				RISC-V			
Rmw	Oota	Uni	Time	Rmw	Oota	Uni	Time	Rmw	Oota	Uni	Time	Rmw	Oota	Uni	Time
✘	✘	✘	7029	✘	✘	✘	5520	✔	✘	✔	7813	✔	✘	✘	5789
✔	✔	✔	7137	✘	✔	✘	5587	✘	✘	✔	7905	✘	✔	✘	6051
✔	✘	✔	7363	✔	✔	✔	5824	✔	✔	✔	7929	✔	✘	✔	6364
✘	✘	✔	7417	✔	✘	✘	5999	✘	✔	✔	8184	✘	✘	✔	6442
✔	✘	✘	7547	✔	✔	✘	6081	✔	✔	✘	8382	✘	✘	✘	6480
✘	✔	✔	8144	✔	✘	✔	6193	✘	✔	✘	8410	✔	✔	✔	6703
✔	✔	✘	10289	✘	✘	✔	6200	✘	✘	✘	8752	✘	✔	✔	6777
✘	✔	✘	10785	✘	✔	✔	6270	✔	✘	✘	12562	✔	✔	✘	7374
-	-	-	57372	-	-	-	6387	-	-	-	455265	-	-	-	8538

Oota bias, however, is not valid in all memory models. While hardware memory models generally satisfy Oota, language level memory models may not. This is because language level models try to take compiler optimizations which break dependencies into account, so that not *all* syntactic dependencies are preserved. In some cases, this even leads to these models straight-up allowing for out-of-thin-air behavior as is witnessed by the C11 memory model [Batty et al. 2015]. This fact makes the Oota bias actually more interesting from a practical point of view; by strengthening the memory model, it removes even some consistent executions from the search space, reducing it in a way orthogonal to the other biases. In the context of BMC, which already under-approximates the program behavior to begin with, adding this bias just adds another layer to the incompleteness of BMC with the prospect of improving verification times. That being said, in all our test cases (in particular the 5013 LKMM litmus tests), we did not encounter a single case where adding Oota hid any bugs.

Using bias constraints, we observed mixed results in performance. For the benchmarks in Figure 10, we got the best overall results when not encoding any bias constraints at all. Table 2 compares the performance of Caat to verify Qspinlock (the main spinlock in Linux) using all different bias combinations. It also reports the verification times of Dartagnan (rows having entries "-"). Verifying Qspinlock requires bitwise reasoning and we obtained the best results by using the Yices2 SMT solver [Dutertre 2014]. As in Figure 10, Caat clearly outperforms Dartagnan. In models using fixed points like LKMM and Power, Caat is between 8× and 65× faster. While for LKMM and ARMv8 using no bias is the best option (this is aligned with our observation from Figure 10), this is one of the worst combinations for Power (only using Rmw alone is worse). For RISC-V, four out of the eight possible combinations work better than not using any bias.

The impact of bias constraints is, however, more drastic in the verification of the Compact NUMA-aware (CNA) lock [Dice and Kogan 2019]. The verification of the CNA lock [Paolillo et al. 2022] is of special interest because it has been proposed as a new slowpath for Qspinlock. Using the Oota bias alone, we were able to prove that Qspinlock with CNA in its slowpath is correct (up to the unwinding bound) in less than 30 minutes for each of the consistency models in Table 2. Using no bias, however, the verification did not produce any result after 48 hours. It is also worth to emphasize the unique advantage of an offline-integrated theory solver over an online-integrated one: the former can be combined with any SMT backend out-of-the-box. The same verification run that took Yices2 only 30 minutes, took Z3 almost 10 hours.

8 RELATED WORK

Understanding memory models is a problem that has seen considerable interest in the last decade [Alglave et al. 2013, 2014; Atig et al. 2010; Bouajjani et al. 2013; Burckhardt and Musuvathi 2008; Dan et al. 2013, 2015; Turon et al. 2014; Vafeiadis and Narayan 2013]. A considerable achievement in

this line of research is the CAT language [Alglave 2010; Alglave et al. 2016, 2014] in which most memory models of interest can be expressed. CAT is made for rapid prototyping. New models are easy to write so that the developer is able to quickly, yet precisely, assess the behavior of the program under the corresponding semantics. Our consistency language supports arbitrary base predicates while CAT has a set of predefined ones tailored for memory consistency. On the other hand, CAT has powerful primitives, e.g., to compute all possible linearisations of an acyclic relation.

We are not the first to develop theories for memory consistency. An ordering consistency theory can be used to verify concurrent programs [He et al. 2021]. While sharing similarities with $\mathcal{T}_{cm}$, the ordering consistency theory is specialized to SC. The approach has recently been extended to support TSO and PSO [Fan et al. 2022]. However, those models are still very similar to SC and do not deal with the complexities of fixed points and predicate differences. For most memory models, program executions can be simulated using a SAT solver. However, this is not the case for some emerging memory models like the one proposed by Jeffrey-Riely (J+R) [Jeffrey and Riely 2016] which relates to the Java memory model [Manson et al. 2006]. This model makes use of three levels of quantifiers. PrideMM, a model checker built on top of a QBF solver, was used to simulate executions under the J+R memory model [Cooksey et al. 2019]. Unfortunately, it suffers from performance issues even for small litmus tests.

Our consistency language has similarities with Alloy [Jackson 2003, 2019]. Alloy is based on relational logic, a simple but powerful combination of first-order logic, relational algebra, and transitive closure. It has been applied in the context of memory models for verification [Torlak et al. 2010], synthesis [Bornholt and Torlak 2017] and comparing different models [Wickerson et al. 2017]. These approaches use an eager encoding, i.e., relations are translated (using clever encodings) to SAT [Torlak and Jackson 2007]. This is possible because relational logic does not support recursive definitions. Despite handling transitive closures, this is not enough. Transitive closures can only express fixed points over linear functions (w.r.t. relational composition ";"). Models like LKMM, ARMv7, and Power need fixed points over non-linear functions. The difference between the theories was actually stated in [Bornholt and Torlak 2017]: *"MemSynth's relational DSL is similar to the CAT language . . . But CAT includes fixpoint operations, while our DSL does not* . The fact that we can efficiently handle fixed points is the main difference between our work and relational logic.

Instead of encoding the whole program as a single formula, SATCheck [Demsky and Lam 2015] uses concolic testing to efficiently explore parts of the program, and a SAT solver to find new behaviors (new branches, interleavings, input/output library call behaviors). Although it was implemented to support SC and TSO, its techniques should be applicable to more relaxed memory models. We believe their encoding could be solved by the SMT scheme from Section 4.

Nidhugg [Abdulla et al. 2014] is a stateless model checker supporting TSO and Power [Abdulla et al. 2015, 2016], but the algorithm is not parametric in the memory model. GenMC [Kokologiannakis and Vafeiadis 2021] is also a stateless model checker, but built on top of axiomatic semantics. While its main procedure is parametric in the choice of the memory model (subject to a few minimal constraints), the tool currently does not support memory models as input. As stated by its authors, adding precise support to GenMC for other memory models is not trivial [Kokologiannakis and Vafeiadis 2021]: *"LKMM uses complex constraints for checking consistency. . . . , we designed approximations for them"* and *"We plan to implement a DSL for memory models, so as to make it easier to extend GenMC with new models"*.

Cerberus-BMC [Lau et al. 2019] provides reference semantics which simultaneously supports a choice in the concurrency memory model (C11, RC11, LKMM), a memory model object, and well-validated thread-local semantics. However, as stated by the authors, *"it is intended as an executable reference semantics for small test programs, not itself as a verification tool that can be applied to larger bodies of C"*. In fact, the authors mention the recursive definitions involved in LKMM as one of the

main performance bottlenecks. This supports our claim that we need verification technology that can efficiently handle fixed points. Dartagnan [Ponce de León et al. 2018, 2020] was a first step in this direction. However, it achieves poor performance for models like Power or LKMM due to its eager encoding. Contrary to this, Caat efficiently handles recursive definitions.

While "user-mode" concurrency has received considerable attention in the last two decades, only recently *system semantics* emerged. The semantics of instruction fetch and cache maintenance (for ARMv8) were clarified in [Simner et al. 2022], while relaxed virtual memory is explored in [Simner et al. 2020]. There is even an integration of full-scale instruction-set architecture (ISA) semantics with axiomatic concurrency models [Armstrong et al. 2021]. While all such extensions to the semantics come with tool support (e.g., RMEM or Isla), those tools achieve poor performance. Their performance evaluation reports that Herd (which is already slower than Caat) is faster for nearly all tests. However, this is not surprising given the amount of detail in the full-scale instruction semantics. A limitation of Isla is that it cannot handle recursive models. However, the authors stated that they believe that *"relations such as Power's (mutually recursive) preserved program order are nevertheless representable as SMT, so this limitation is mostly in our translation from CAT"*. We believe Isla could benefit from the verification technology proposed by this paper.

9 FUTURE WORK

We plan to extend our work in the following ways. To handle non-semi-positive consistency models, we currently cut derived predicates that appear negatively (as a right-hand side of a difference) from the consistency model in their entirety. We do so because our theory solver cannot compute reasons (in terms of base predicates) for the absence of an edge/element of a derived predicate. However, in practice, it is likely that only a fraction of a predicate is actually relevant for consistency, and hence we could cut only those parts of the predicate that actually appear in theory explanations. We thus plan to encode the problematic predicate incrementally on-demand. This *on-demand cutting* would lead to a hybrid between the eager and the lazy SMT encoding.

Interesting is also an online integration of our theory solver into an SMT engine. While the offline-integration is more flexible in terms of supported backends and shows promising results, we have observed a limitation in the context of BMC. In the offline approach, the consistency reasoning will only be done last, after a full satisfying assignment has been determined. This may be detrimental to performance if the consistency reasoning is the easy bit and finding the satisfying assignment involves invocations to theory solvers that are integrated in an online fashion. For example, if programs have complex data and control flow and hence complex arithmetic constraints, it might be best to defer exactly that arithmetic reasoning until *after* the consistency reasoning.

An online integration calls for incremental theory solving. In the presence of non-monotonic differences, it is not clear how to achieve this. Even when restricting recursion to the simpler case of transitive closures (for which there are dynamic algorithms [Roditty 2008]), we do not know if these algorithms can be made to track derivations lengths. A way out may be to circumvent the non-monotonic reasoning altogether and construct a fully positive model via (on-demand) cutting.

10 DATA AVAILABILITY STATEMENT

The complete benchmark sets of Figure 9 and Figure 10 as well as the code used to generate the figures are provided in the accompanying artifact [Haas et al. 2022].

REFERENCES

Parosh A. Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *TACAS (LNCS, Vol. 9035)*. Springer, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28

Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. In *POPL*. ACM, 373–384. https://doi.org/10.1145/2535838.2535845

Parosh A. Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. Stateless Model Checking for POWER. In *CAV (LNCS, Vol. 9780)*. Springer, 134–156. https://doi.org/10.1007/978-3-319-41540-6_8

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 135:1–135:29. https://doi.org/10.1145/3276505

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc. https://doi.org/10.5555/551350

A. Adir, H. Attiya, and G. Shurek. 2003. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Transactions on Parallel and Distributed Systems* 14, 5 (2003), 502–515. https://doi.org/10.1109/TPDS.2003.1199067

S.V. Adve and K. Gharachorloo. 1996. Shared memory consistency models: a tutorial. *Computer* 29, 12 (1996), 66–76. https://doi.org/10.1109/2.546611

Jade Alglave. 2010. *A Shared Memory Poetics*. Thèse de doctorat. L'université Paris Denis Diderot.

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *ASPLOS*. ACM, 577–591. https://doi.org/10.1145/2786763.2694391

Jade Alglave and Patrick Cousot. 2016. Syntax and analytic semantics of LISA. *CoRR* abs/1608.06583 (2016). https://arxiv.org/abs/1608.06583

Jade Alglave, Patrick Cousot, and Luc Maranget. 2016. Syntax and semantics of the weak consistency model specification language CAT. *CoRR* abs/1608.07531 (2016). https://arxiv.org/abs/1608.07531

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *CAV (LNCS, Vol. 8044)*. Springer, 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. 2018. Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel. In *ASPLOS*. ACM, 405–418. https://doi.org/10.1145/3173162.3177156

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. Fences in weak memory models (extended version). *Formal Methods in System Design* 40, 2 (2012), 170–205. https://doi.org/10.1007/s10703-011-0135-z

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. 2021. Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models. In *CAV (1) (Lecture Notes in Computer Science, Vol. 12759)*. Springer, 303–316. https://doi.org/10.1007/978-3-030-81685-8_14

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *POPL*. ACM, 7–18. https://doi.org/10.1145/1706299.1706303

G. Audemard, A. Cimatti, A. Kornilowicz, and R. Sebastiani. 2002. Bounded Model Checking for Timed Systems. In *FORTE*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-36135-9_16

Daniel Baier, Dirk Beyer, and Karlheinz Friedberger. 2021. JavaSMT 3: Interacting with SMT Solvers in Java. In *CAV (2) (LNCS, Vol. 12760)*. Springer, 195–208. https://doi.org/10.1007/978-3-030-81688-9_9

François Bancilhon. 1985. Naive Evaluation of Recursively Defined Relations. In *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies (Topics in Information Systems)*. Springer, 165–178. https://doi.org/10.1007/978-1-4612-4980-1_17

Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 825–885. https://doi.org/10.3233/FAIA201017

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *POPL*. ACM, 634–648. https://doi.org/10.1145/2837614.2837637

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *ESOP (LNCS, Vol. 9032)*. Springer, 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and compiling C/C++ concurrency: from C++11 to POWER. In *POPL*. ACM, 509–520. https://doi.org/10.1145/2103621.2103717

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL*. ACM, 55–66. https://doi.org/10.1145/1925844.1926394

Dirk Beyer. 2022. Progress on Software Verification: SV-COMP 2022. In *TACAS (2)*. Springer. https://doi.org/10.1007/978-3-030-99527-0_20

Hans-Juergen Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *PLDI*. ACM, 68–78. https://doi.org/10.1145/1379022.1375591

James Bornholt and Emina Torlak. 2017. Synthesizing memory models from framework sketches and Litmus tests. In *PLDI*. ACM, 467–481. https://doi.org/10.1145/3140587.3062353

Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *ESOP (LNCS, Vol. 7792)*. Springer, 533–553. https://doi.org/10.1007/978-3-642-37036-6_29

Sebastian Burckhardt and Madanlal Musuvathi. 2008. Effective Program Verification for Relaxed Memory Models. In *CAV (LNCS, Vol. 5123)*. Springer, 107–120. https://doi.org/10.1007/978-3-540-70545-1_12

Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design* 19, 1 (2001), 7–34. https://doi.org/10.1023/A:1011276507260

William W. Collier. 1992. *Reasoning about parallel architectures*. Prentice Hall.

Simon Cooksey, Sarah Harris, Mark Batty, Radu Grigore, and Mikolás Janota. 2019. PrideMM: Second Order Model Checking for Memory Consistency Models. In *FM Workshops (2) (LNCS, Vol. 12233)*. Springer, 507–525. https://doi.org/10.1007/978-3-030-54997-8_31

Andrei M. Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. 2013. Predicate Abstraction for Relaxed Memory Models. In *SAS (LNCS, Vol. 7935)*. Springer, 84–104. https://doi.org/10.1007/978-3-642-38856-9_7

Andrei M. Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. 2015. Effective Abstractions for Verification under Relaxed Memory Models. In *VMCAI (LNCS, Vol. 8931)*. Springer, 449–466. https://doi.org/10.1007/978-3-662-46081-8_25

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Martin Davis, George Logemann, and Donald W. Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (1962), 394–397. https://doi.org/10.1145/368273.368557

Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (1960), 201–215. https://doi.org/10.1145/321033.321034

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed stateless model checking for SC and TSO. In *OOPSLA*. ACM, 20–36. https://doi.org/10.1145/2814270.2814297

Dave Dice and Alex Kogan. 2019. Compact NUMA-Aware Locks. In *EuroSys*. ACM, 15 pages. https://doi.org/10.1145/3302424.3303984

Bruno Dutertre. 2014. Yices 2.2. In *CAV (Lecture Notes in Computer Science, Vol. 8559)*. Springer, 737–744. https://doi.org/10.1007/978-3-319-08867-9_49

Roman Elizarov, Mikhail A. Belyaev, Marat Akhin, and Ilmir Usmanov. 2021. Kotlin coroutines: design and implementation. In *Onward!* ACM, 68–84. https://doi.org/10.1145/3486607.3486751

Herbert B. Enderton. 1972. *A mathematical introduction to logic*. Academic Press. https://doi.org/10.1016/C2009-0-22107-6

Hongyu Fan, Weiting Liu, and Fei He. 2022. Interference relation-guided SMT solving for multi-threaded program verification. In *PPoPP*. ACM, 163–176. https://doi.org/10.1145/3503221.3508424

Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In *CAV (LNCS, Vol. 11561)*. Springer, 355–365. https://doi.org/10.1007/978-3-030-25540-4_19

Thomas Haas, Roland Meyer, and Hernán Ponce-de León. 2022. CAAT: Consistency as a Theory (Artifact). https://doi.org/10.5281/zenodo.7079674

Fei He, Zhihang Sun, and Hongyu Fan. 2021. Satisfiability modulo ordering consistency theory for multi-threaded program verification. In *PLDI*. ACM, 1264–1279. https://doi.org/10.1145/3453483.3454108

Yannis E. Ioannidis and Raghu Ramakrishnan. 1988. Efficient Transitive Closure Algorithms. In *VLDB*. Morgan Kaufmann, 382–394. https://doi.org/10.5555/645915.671829

Daniel Jackson. 2000. Automating First-Order Relational Logic. *SIGSOFT Softw. Eng. Notes* 25, 6 (2000), 130–139. https://doi.org/10.1145/357474.355063

Daniel Jackson. 2003. Alloy: A Logical Modelling Language. In *ZB (Lecture Notes in Computer Science, Vol. 2651)*. Springer, 1. https://doi.org/10.1007/3-540-44880-2_1

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

Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *LICS*. ACM, 759–767. https://doi.org/10.1145/2933575.2934536

Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. 2016. JavaSMT: A Unified Interface for SMT Solvers in Java. In *VSTTE (LNCS, Vol. 9971)*. Springer, 139–148. https://doi.org/10.1007/978-3-319-48869-1_11

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019a. Model checking for weakly consistent libraries. In *PLDI*. ACM, 96–110. https://doi.org/10.1145/3314221.3314609

Michalis Kokologiannakis, Xiaowei Ren, and Viktor Vafeiadis. 2019b. Dynamic Partial Order Reductions for Spinloops. In *FMCAD*. TU Wien Academic Press, 163–172. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_25

Michalis Kokologiannakis and Viktor Vafeiadis. 2021. GenMC: A Model Checker for Weak Memory Models. In *CAV (LNCS, Vol. 12759)*. Springer, 427–440. https://doi.org/10.1007/978-3-030-81685-8_20

Nikita Koval, Dmitry Khalanskiy, and Dan Alistarh. 2021. A Formally-Verified Framework for Fair Synchronization in Kotlin Coroutines. *CoRR* abs/2111.12682 (2021). https://arxiv.org/abs/2111.12682

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *POPL*. ACM, 649–662. https://doi.org/10.1145/2837614.2837643

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*. ACM, 618–632. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Stella Lau, Victor B. F. Gomes, Kayvan Memarian, Jean Pichon-Pharabod, and Peter Sewell. 2019. Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C. In *CAV (LNCS, Vol. 11561)*. Springer, 387–397. https://doi.org/10.1007/978-3-030-25540-4_22

K. Rustan M. Leino. 2008. This is Boogie 2. (2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. 2012. An Axiomatic Memory Model for POWER Multiprocessors. In *CAV (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 495–512. https://doi.org/10.1007/978-3-642-31424-7_36

Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2016. Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings. *CoRR* abs/1611.01507 (2016). https://arxiv.org/abs/1611.01507

Jeremy Manson, William Pugh, and Sarita V. Adve. 2006. The Java memory model. In *POPL*. ACM, 378–391. https://doi.org/10.1145/1040305.1040336

Roland Meyer and Sebastian Wolff. 2019. Decoupling lock-free data structures from memory reclamation for static analysis. *PACMPL* 3, POPL (2019), 58:1–58:31. https://doi.org/10.1145/3290371

Roland Meyer and Sebastian Wolff. 2020. Pointer life cycle types for lock-free data structures with memory reclamation. *PACMPL* 4, POPL (2020), 68:1–68:36. https://doi.org/10.1145/3371136

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer. https://doi.org/10.1007/978-3-662-03811-6

Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: push-button verification and optimization for synchronization primitives on weak memory models. In *ASPLOS*. ACM, 530–545. https://doi.org/10.1145/3445814.3446748

Derek C Oppen. 1980. Complexity, convexity and combinations of theories. *Theoretical computer science* 12, 3 (1980), 291–302. https://doi.org/10.1016/0304-3975(80)90059-6

Antonio Paolillo, Hernán Ponce de León, Diogo Behrens Thomas Haas, Rafael Lourenco de Lima Chehab, Ming Fu, and Roland Meyer. 2022. Verifying and Optimizing Compact NUMA-Aware Locks on Weak Memory Models. *CoRR* abs/2111.15240 (2022). https://arxiv.org/abs/2111.15240

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *PACMPL* 3, POPL (2019), 69:1–69:31. https://doi.org/10.1145/3290382

Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2017. Portability Analysis for Weak Memory Models. PORTHOS: One Tool for all Models. In *SAS (LNCS, Vol. 10422)*. Springer, 299–320. https://doi.org/10.1007/978-3-319-66706-5_15

Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2018. BMC with Memory Models as Modules. In *FMCAD*. IEEE, 1–9. https://doi.org/10.23919/FMCAD.2018.8603021

Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2020. Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution). In *TACAS (2) (LNCS, Vol. 12079)*. Springer, 378–382. https://doi.org/10.1007/978-3-030-45237-7_24

Pablo Ponzio, Ariel Godio, Nicolás Rosner, Marcelo Arroyo, Nazareno Aguirre, and Marcelo F. Frias. 2021. Efficient Bounded Model Checking of Heap-Manipulating Programs using Tight Field Bounds. In *FASE*. Springer International Publishing, 218–239. https://doi.org/10.1007/978-3-030-71500-7_11

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL (2018), 19:1–19:29. https://doi.org/10.

1145/3158107

Liam Roditty. 2008. A Faster and Simpler Fully Dynamic Transitive Closure. 4, 1 (2008). https://doi.org/10.1145/1328911.1328917

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *PLDI*. ACM, 175–186. https://doi.org/10.1145/1993316.1993520

Roberto Sebastiani. 2007. Lazy Satisability Modulo Theories. *J. Satisf. Boolean Model. Comput.* 3, 3-4 (2007), 141–224. https://doi.org/10.3233/SAT190034

Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (apr 1988), 282–312. https://doi.org/10.1145/42190.42277

Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. 2022. Relaxed virtual memory in Armv8-A. In *ESOP (Lecture Notes in Computer Science, Vol. 13240)*. Springer, 143–173. https://doi.org/10.1007/978-3-030-99336-8_6

Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. 2020. ARMv8-A System Semantics: Instruction Fetch in Relaxed Architectures. In *ESOP (Lecture Notes in Computer Science, Vol. 12075)*. Springer, 626–655. https://doi.org/10.1007/978-3-030-44914-8_23

Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. 1992. *Formal Specification of Memory Models*. Springer US, Boston, MA, 25–41. https://doi.org/10.1007/978-1-4615-3604-8_2

Robert Tarjan. 1971. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 114–121. https://doi.org/10.1109/SWAT.1971.10

Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS (Lecture Notes in Computer Science, Vol. 4424)*. Springer, 632–647. https://doi.org/10.1007/978-3-540-71209-1_49

Emina Torlak, Mandana Vaziri, and Julian Dolby. 2010. MemSAT: Checking axiomatic specifications of memory models. In *PLDI*. ACM, 341–350. https://doi.org/10.1145/1809028.1806635

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*. ACM, 691–707. https://doi.org/10.1145/2660193.2660243

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *POPL*. ACM, 209–220. https://doi.org/10.1145/2676726.2676995

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*. ACM, 867–884. https://doi.org/10.1145/2544173.2509532

Douglas B. West. 2000. *Introduction to Graph Theory*. Prentice Hall.

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically Comparing Memory Consistency Models. In *POPL*. ACM, 190–204. https://doi.org/10.1145/3093333.3009838

David Zhao, Pavle Subotic, and Bernhard Scholz. 2019. Provenance for Large-scale Datalog. *CoRR* abs/1907.05045 (2019). https://arxiv.org/abs/1907.05045