Static Analysis of Memory Models for SMT Encodings

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

The goal of this work is to improve the efficiency of bounded model checkers that are modular in the memory model. Our first contribution is a static analysis for the given memory model that is performed as a preprocessing step and helps us significantly reduce the encoding size. Memory model make use of relations to judge whether an execution is consistent. The analysis computes bounds on these relations: which pairs of events may or must be related. What is new is that the bounds are relativized to the execution of events. This makes it possible to derive, for the first time, not only upper but also meaningful lower bounds. Another important feature is that the analysis can import information about the verification instance from external sources to improve its precision. Our second contribution are new optimizations for the SMT encoding. Notably, the lower bounds allow us to simplify the encoding of acyclicity constraints. We implemented our analysis and optimizations within a bounded model checker and evaluated it on challenging benchmarks. The evaluation shows up-to 40% reduction in verification time (including the analysis) over previous encodings. Our optimizations allow us to efficiently check safety, liveness, and data race freedom in Linux kernel code.

CCS Concepts: • **Theory of computation** → **Axiomatic semantics**; **Abstraction**; *Program verification*; *Concurrency*.

Additional Key Words and Phrases: Abstract interpretation, weak memory models, bounded model checking

ACM Reference Format:
Thomas Haas, René Maseli, Roland Meyer, and Hernán Ponce de León. 2023. Static Analysis of Memory Models for SMT Encodings. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 279 (October 2023), 30 pages.
https://doi.org/10.1145/3622855

1 INTRODUCTION

In the verification of fine-grained concurrent programs, it has long been recognized [Burckhardt et al. 2006, 2007; Burckhardt and Musuvathi 2008] that the memory model of the underlying platform cannot be ignored. Fine-grained concurrent programs do not preserve the data race freedom guarantee [Adve and Hill 1990], and therefore the assumption of a sequentially consistent memory [Lamport 1979] is not justified. The verification community has taken up the challenge and, over the past decade, developed practical algorithms for a range of platforms [Abdulla et al. 2015, 2016; Dan et al. 2015; Demsky and Lam 2015; Kuperstein et al. 2011; Norris and Demsky 2013, 2016] underpinned by theoretical results about the decidability and complexity status of the corresponding verification problems [Abdulla et al. 2020, 2021; Atig et al. 2010, 2012; Krishna et al. 2022; Lahav and Boker 2020, 2022]. After an initial enthusiasm about the outburst of results, however, the massive need for new technology felt daunting.

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

2475-1421/2023/10-ART279
https://doi.org/10.1145/3622855

Developing a verification tool is a major undertaking of several years over which the range of relevant memory models grows, not only due to new developments but also due to corrections of existing models. To give a non-exhaustive list of memory models that have been studied: TSO [Sarkar et al. 2009], Power [Mador-Haim et al. 2012; Sarkar et al. 2011], ARM in versions 7 [Alglave et al. 2014b] and 8 [Alglave et al. 2021; Pulte et al. 2018], Java [Bender and Palsberg 2019; Manson et al. 2006], C11 [Vafeiadis et al. 2015] in a number of revisions [Batty et al. 2016; Lahav et al. 2017], and recently the Linux kernel memory model (LKMM) [Alglave et al. 2018]. In a sense, this is a race between the programming languages and systems communities coming up with new memory models, and the verification community trying to keep up. A feeling in the verification community emerged that, with the classical technology, this race could not be won. The way out was to lift the level of abstraction at which verification tools would work.

There is a trend towards verification technology that is *modular* in the memory model: that can easily exchange the memory model or even accept a description of it as input. This modularity was made possible by the insight that, despite all semantic differences, most memory models can be formalized in the same specification language: CAT [Alglave 2010; Alglave et al. 2014b]. Interestingly, the first modular tool, MemSat [Torlak et al. 2010], was proposed around the same time, but targeted a different formalism and weaker class of models. With Herd7, Alglave et al. [2014b] developed the first tool for the CAT language. Since then, a landscape of techniques evolved. For proving program correctness, modular approaches have remained largely unexplored, with the program logics due to Alglave and Cousot [2017]; Doherty et al. [2022] being the only representatives. For bug hunting, there is more. Modular bug hunting tools fall into two categories that have orthogonal strengths: stateless model checkers and bounded model checkers.

Stateless model checkers [Godefroid 1996] execute the program in a systematic way in order to explore the state space. The approach is very efficient under a moderate number of executions, and the notion of execution can be chosen to incorporate partial order and symmetry information. Nidhugg [Abdulla et al. 2015] is a stateless model checker which reasons about the TSO memory model and has some support for Power and ARMv7 [Abdulla et al. 2016]. Its exploration algorithm is parametric in the definition of traces. Adding support for a model requires coming up with a new notion of trace that captures the intended semantics. Kokologiannakis and Vafeiadis [2020, 2021] developed GenMC, a modular stateless model checker. GenMC has been successfully applied to the verification of synchronization primitives [Oberhauser et al. 2021] and complex data structures [Beck et al. 2023; Wang et al. 2022]. The implementation supports RC11 [Lahav et al. 2017] and IMM [Lahav et al. 2017], but has not yet been lifted to industrial models like ARMv8, RISC-V, or LKMM.

The alternative technology are bounded model checkers [Clarke et al. 2001]. They statically encode the verification task into a logical formula that is satisfiable if and only if the program has a bug. Well-maintained SMT solvers developed in long-term efforts like Mathsat5 [Cimatti et al. 2013], Yices2 [Dutertre 2014], or Z3 [De Moura and Bjørner 2008] are then used to discharge the satisfiability problem. Modular bounded model checkers (that take into account a memory model) can be implemented in eager and lazy style. The *eager* method is to encode the entire verification problem into the logical theories supported by the SMT backend. This was the approach followed in the early versions of Dartagnan [Gavrilenko et al. 2019; Ponce de León et al. 2017, 2018]. While flexible, the prohibitively large encoding of the memory model hindered scalability.

He et al. [2021] observed that the memory model significantly increases the encoding size, although only few clauses are needed to determine the satisfiability status. They developed dedicated logical theories that capture the consistency requirements of store atomic memory models and integrated them into the SMT solver [Sun et al. 2022]. Inspired by this, the Dartagnan team showed that nearly arbitrary CAT models can be turned into logical theories [Haas et al. 2022]. With logical theories for consistency, bounded model checkers can be implemented in *lazy* style, meaning the consistency requirement is understood natively by the solver and does not have to be encoded. This approach is taken by DEAGLE [He et al. 2021] and the recent version of DARTAGNAN.

Motivation. This paper starts from the observation that modular bounded model checkers which are implemented in lazy style cannot handle certain correctness conditions. To understand the problem, note that a memory model distinguishes between so-called base relations and derived relations. The base relations define the execution: the program order, the address and data dependencies, the read-from relation, and memory coherence. The derived relations have a different role, they are used to judge whether the execution should be considered consistent. This means the derived relations do not introduce new information to the execution but are computed from the base relations. The problem with lazy bounded model checkers is that their SMT encoding can only access the base relations. The derived relations are not mentioned in the encoding, but only computed inside the consistency theory solver. Indeed, removing the derived relations from the eager SMT encoding was precisely the goal of the lazy approach.

Lazy bounded model checkers cannot handle correctness conditions that are formulated in terms of derived relations of the memory model.

To develop an intuition to the correctness conditions that the lazy approach cannot handle, consider the task of *detecting data races in the Linux kernel.* The definition of a data race is almost the standard one: there is an execution with two unordered instructions that access a common location and at least one is a write. The point is that the notion of order refers to the Linux happens-before relation, a derived relation in LKMM [Alglave et al. 2018]. Since a lazy bounded model checker does not encode this derived relation, it cannot encode the correctness condition either. It would be attractive to extend the consistency theory solvers [Haas et al. 2022; Sun et al. 2022] that are behind the lazy approach to support DRF natively, but this seems to be very hard to do. The problem is that DRF is formulated negatively, yet the theory solver has to give an explanation of inconsistency to the SMT solver's SAT engine. How can one explain to the SAT engine that there is no race and then guide it to find a race? The absence of a race is a universal statement, and to rule out the given execution one would have to formulate that for every pair of instructions there is no race (without new ideas, the resulting explanation would be a large disjunction of conjunctions). The problem did not go unnoticed. Haas et al. [2022] proposed the technique of cutting to add derived relations to the lazy encoding. For relations like happens-before (that incorporate a large number of derived relations), however, cutting corresponds to encoding a large portion of the memory model, and thus means switching from lazy to eager bounded model checking.

Correctness of concurrent Linux kernel code is a problem that has recently attracted attention. The reason is a series of issues with `qspinlock` [Corbet 2014], one of the main synchronization mechanisms in the kernel. The lock provides good performance, fairness guarantees [Corbet 2008], and overcomes what is known as the cache-line bouncing problem [Corbet 2013b]. These benefits are the result of a complex algorithm with a number of low-level optimizations. Unfortunately, the optimizations turned out complicated enough to cause troubles. Under LKMMv6.2, it was shown that `qspinlock` does not guarantee mutual exclusion, it can deadlock, and has a data race [Paolillo et al. 2021]. This lead to a revision in LKMMv6.3 [Stern 2023] that addressed the former issues. The data race, however, was not removed but classified as benign. Such a classification is a conscious decision that requires developers to know about the data race in the first place.

The kernel data race detection problem is still considered unsolved.

As argued above, the recent lazy bounded model checkers cannot formulate the data race. Eager bounded model checkers would be applicable in principle, but have scalability issues. Stateless model checkers do not support LKMM. Even specialized race detectors like the kernel sanitizer

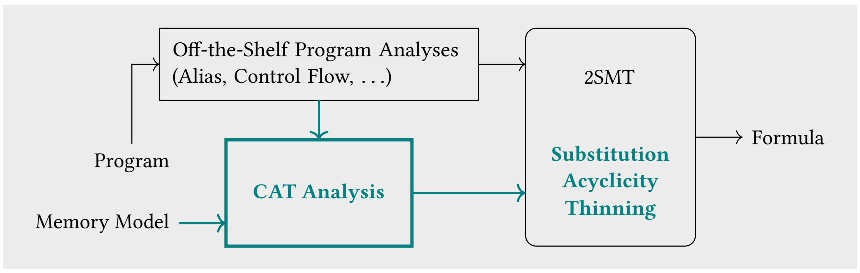

Fig. 1. Workflow of the improved eager encoding with contributions highlighted.

KCSAN cannot handle the entire model: *"Note, KCSAN will not report all data races due to missing memory ordering[...]"* [kcs 2023]. Interestingly, solving the data race detection problem under the C/C++ memory model has seen more success [Luo and Demsky 2021].

There are further properties beyond DRF that require constraints on derived relations. As a rule of thumb, derived relations come into play whenever the correctness statement depends on the synchronization among threads. This is true for locking [C11 2023; lin 2023a], RCU [lin 2023b] (RCU locks and unlocks, misuse of SRCU), and interrupts, to give examples. Constraints on derived relations are also used to sanitize the memory model rather than the program [lin 2023c]. All these applications require reasoning about derived relations, usually involving a negative formulation, that cannot be handled by the existing consistency theory solvers [Haas et al. 2022; Sun et al. 2022].

In the light of the above, we believe there is a need to improve the scalability of modular bounded model checking tools that rely on eager SMT encodings. A particular use case is data race detection in the Linux kernel. A classical strategy to improve the scalability of verification tools is to perform an upfront static analysis of the program to obtain information, like moverness or alias information, that reduces the search space. For the new input of modular verification tools, namely the memory model, corresponding analyses are almost non-existent, with the rare exceptions being the early [Torlak et al. 2010] and [Gavrilenko et al. 2019]. The works [Kokologiannakis et al. 2023; Mador-Haim et al. 2010; Wickerson et al. 2017] compare memory models, which is different.

Contribution. We propose the first static analysis for memory models written in CAT that can give not only upper but also lower bounds for relations. The analysis alleviates the scalability issues of the eager encoding by introducing the workflow depicted in Figure 1. Before the SMT encoding, we analyze the memory model. The resulting information enables new optimizations of the SMT encoding that are also contributions of this paper. An interesting aspect of our static analysis for memory models is that its precision can be improved by providing information about the program at hand. This information is expected in a standardized format and can be obtained with off-the-shelf program analysis tools.

The challenge in analyzing memory models is that the CAT language is very different from the imperative, functional, or declarative languages typically analyzed. There are three things that make CAT stand out: it works over relational algebras, has recursive definitions, and there is a program and a correctness condition behind the memory model. If anything, CAT can be seen as a blend of Alloy [Jackson 2003, 2019] and datalog [Ceri et al. 1989]. A consequence of this unique combination is that CAT requires unconventional constructs for its static analysis.

Our first contribution is a reformulation of the CAT semantics that is better suited for static analysis. By a reformulation, we mean that we define a new semantics and prove it to coincide with the standard semantics. The unconventional construct behind our new semantics is that of a filter. Filters can be understood as a mechanism to remove, from a given set of executions, executions that should be considered inconsistent, say because they violate an equation of the memory model. We have not invented filters from scratch. Axiomatic semantics always view the memory model as a filter [Alglave 2010; Alglave et al. 2014b; Jackson 2003]. In abstract interpretation, filters are strengthened to lower closure operators [Cousot 2021, Chapter IV, Section 11.7] and used to approximate the conditions in `if`- and `while`-constructs [Cousot 2021, Chapter IX, Section 29]. What we observe is that filters are useful for the static analysis of memory models.

The first benefit of the filter semantics is that it is flexible enough to incorporate information about the verification instance (the program and the correctness condition). The added information will reduce the set of executions denoted by the memory model and hence improve the precision of the static analysis. We achieve this flexibility by defining the new semantics to be parametric in a set of filters. This is made possible by the fact that the filter semantics is compositional. Our insight is that not only the memory model itself acts as a filter, but it also decomposes into smaller filters. For example, to incorporate information about the program p at hand, we define a program filter f_p and obtain the instantiation of our new semantics $[\![mm]\!]^{\{f_p\}} = [\![mm]\!] \cap [\![p]\!] = [\![p]\!]_{mm}$. Given the program filter, the new semantics coincides with the semantics of the program under the memory model. When performing a static analysis of the new semantics, we will approximate f_p by static information about the program. In this sense, filters serve as an interface to external analyses. In our implementation, we rely on control-flow and alias analyses.

The second benefit of the new semantics is that it admits precise approximations. The observation is that memory models propagate information in two directions. They compute, bottom-up, from the base relations the derived relations that are evaluated in axioms. But the axioms also constrain, top-down, the derived and therefore the base relations that can occur in consistent executions. Filters make this bidirectionality explicit. We approximate them in a precise way by not only defining forward but also backward approximations of the operations supported by CAT. Forward and backward approximations have been prominently used in the abstract interpretation of logic programs [Cousot and Cousot 1992], but our abstract domain is entirely different.

The third benefit of the filter semantics is that it is easy to generalize. To incorporate a new base relation, say for mixed-size accesses, the only thing to do is devise an appropriate filter. The semantics (and so the analysis that builds on it) will then (automatically) take the filter into account, and thus understand how to deal with the new base relation. Again, we rely here on the compositionality of the new semantics.

Our second contribution is the actual static analysis for the new semantics. To be able to run it as a preprocessing step to bounded model checking, the analysis has to be efficient. At the same time, it has to be precise to make sure the analysis information significantly simplifies the SMT encoding. We meet this trade-off with the design of the abstract domain. We approximate a relation n by a triple $(\mathsf{acf}, \mathsf{must}(\mathsf{n}), \mathsf{may}(\mathsf{n}))$. The so-called must-set $\mathsf{must}(\mathsf{n})$ will form a lower bound and the may-set $\mathsf{may}(\mathsf{n})$ an upper bound for the relation. Importantly, the meaning of these sets is relativized to the occurrence of events. Membership $(\mathsf{x}_1, \mathsf{x}_2) \in \mathsf{must}(\mathsf{n})$ does not mean that the events x_1 and x_2 will definitely be related by n in every consistent execution, but it means they will definitely be related *whenever they occur*. Similarly, $(\mathsf{x}_1, \mathsf{x}_2) \in \mathsf{may}(\mathsf{n})$ says that there is a consistent execution in which the two events occur and are related by n. Without this relativization, it is impossible to compute lower bounds: if events are missing in an execution, should they be considered related? Upper bounds for relations have been proposed before in DARTAGNAN [Gavrilenko et al. 2019]. Also MUSKETEER [Alglave et al. 2014a] computes upper bounds on the inter-thread communication relation to find critical cycles for fence insertion. KODKOD [Torlak 2009] has upper and lower bounds, but the relations are absolute in the sense that they are not tied to an execution. MEMSAT computes bounds in a setting similar to ours [Torlak et al. 2010], but can give lower bounds only under absolute knowledge: there has to be a guarantee that the events occur in all executions. Neither Kodkod nor MemSat handle the non-linear recursion needed for models like LKMM and Power. Unrolling fixed points does not scale as discussed in [Haas et al. 2022; Wickerson et al. 2017]. In the setting of finite model finding for first-order theories with recursion, [Wittocx et al. 2013] computes atoms that must be true resp. false in all models, but again lacks a notion of relativization. The definition of relativized sets for the computation of lower bounds is a main contribution of ours.

May- and must-sets are, in the static analysis sense, non-relational [Cousot 2021, Chapter IX]: membership of one pair in a set does not allow us to draw conclusions about membership of another. This loss of information makes it difficult to fill must-sets. To overcome the problem, our abstract domain has an abstract control-flow component acf that maintains relational information about the execution of events. It contains implications of the form $x_1 \longrightarrow x_2$ and $x_1 \longrightarrow \neg x_2$ saying that if event x_1 occurs in an execution, then so does x_2 (resp. x_2 has to be missing). While the memory model cannot provide abstract control-flow information, a program filter that is given to the concrete semantics as a parameter will do so. We thus enrich the abstract domain (by acf) to be able to exploit information from external analyses. To see how the abstract control flow helps us fill must-sets, let $(x, y), (y, z) \in \mathsf{must}(n)$, and $x \longrightarrow y \in \mathsf{acf}$. We claim that also $(x, z) \in \mathsf{must}(n; n)$, where ; is relation composition. To see this, consider an execution in which x and z occur. By $x \longrightarrow y$, also y occurs. Now the must-set says that (x, y) and (y, z) are related by n. Hence, (x, z) is related by $n; n$.

Our third contribution are two novel optimizations for the eager SMT encoding of CAT models. The first optimization simplifies the encoding of acyclicity axioms. The problem is that every edge which may be part of a cycle adds a constraint in integer difference logic (IDL) [Gebser et al. 2014], and these constraints slow-down the solver. Our optimization removes the IDL constraints for edges that are shortcuts of longer paths. The notion of shortcut is again defined in a relative way: if the edge occurs in an execution, so does the path. This is precisely the information our must-sets give, and the optimization cannot be implemented without them. Satune [Gorjiara et al. 2020] has a transitivity analysis that is related in that it applies to sets of graphs and relies on must-information about edges. The difference to our work is that the must-information is absolute: the corresponding edges have to exist in every graph in the set. Our optimization works with relativized must-sets (nodes may be missing in an execution), and our reasoning is all about this relativization.

Our second optimization reduces the size of the SMT encoding for (recursive) equations. The problem is that one equation like `let` $d := r$ translates into a large number of equivalences $d(x, y) \leftrightarrow r(x, y)$, namely one equivalence for each pair of events (x, y) in the program. The optimization removes equivalences for which it can statically determine that they have no influence on the axioms, and hence the satisfiability of the encoding. Imagine the only axiom is $\mathbf{empty}(n \cap d)$ and $(x, y) \notin \mathsf{may}(n)$. Then we can safely skip the above equivalence. This may, in turn, render other equivalences irrelevant. The optimization is inspired by the magic sets transformation for improved datalog query evaluation [Bancilhon et al. 1986]. It also has a precursor in [Gavrilenko et al. 2019], but the lower bounds we compute allow us to strengthen that work.

Our last contribution is an implementation of the new static analysis and optimization algorithms inside a bounded model checker [Clarke et al. 2001]. We evaluated the implementation on a benchmark suite consisting of (i) ten challenging fine-grained concurrent programs, (ii) safety, liveness, and data race freedom as the correctness criteria, and (iii) memory models that no other tool supports over such complex benchmarks, namely ARMv8, RISC-V, and LKMM. The results are favorable, we experience a reduction in verification time of almost 40% over the optimization from [Gavrilenko et al. 2019]. Coming back to the correctness of `qspinlock` under LKMM: our optimized eager encoding finds the data race in minutes (from 2 to 10), and proves a fixed version of the program correct in 1.5h, including an analysis of safety and liveness requirements. As always

```
#define N ... // Parameter of the Program
int i = 1, j = 1;

void thread_i() {
  for (int p = 0; p < N; p++) {
    i = i + j;
  }}

void thread_j() {
  for (int q = 0; q < N; q++) {
    j = j + i;
  }}

void main() {
  // The (2N+2)-th Fibonacci Number
  int correct = fib(2*N + 2);
  assert (i <= correct && j <= correct);
}
```

```
/* 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]
  ∪ addr; po; [W]
  ∪ (ctrl ∪ data); coi
  ∪ (addr ∪ data); rfi
/* Ordered-before */
let ob = obs ∪ dob
/* External visibility requirement */
acyclic ob
```

Fig. 2. Fibonacci program (left) and snippet of memory model ARMv8 [Pulte et al. 2018] (right).

in bounded model checking, our programs are unrolled and hence loop-free versions of the original code, and correctness is only guaranteed up to the unrolling bound.

To summarize, we make the following contributions.

- We give a new semantics for the CAT language that is designed with static analysis in mind: it is easy to approximate in a precise way, and it can integrate information about the verification instance that has been obtained with external analyses.
- We give a precise and efficient static analysis for CAT. It approximates relations by lower and upper bounds. Importantly, the bounds are defined relative to the execution of events. We obtain this execution information by linking the analysis to the external information in the new semantics.
- We propose two new optimizations for the eager SMT encoding of CAT models. The first reduces the number of IDL constraints. The optimization is only made possible by the lower bound information we compute. The second optimization improves the encoding of (recursive) equations.
- We have implemented the techniques inside a modular bounded model checker and evaluated the resulting tool on a comprehensive benchmark suite.

2 OVERVIEW

The semantics of a concurrent program p under a memory model mm is the set of so-called executions $[\![p]\!]_{mm} = [\![p]\!] \cap [\![mm]\!]$. We are interested in analyzing this semantics with the help of bounded model checking. To this end, we assume that the program is acyclic and encode the above set into a formula of the form $\varphi = \varphi_p \wedge \varphi_{mm} \wedge \varphi_{key}$. The satisfying assignments are precisely the executions that are possible in the program and also consistent according to the memory model. We will introduce φ_{key} in a moment.

Our goal is to improve the SMT encoding, to find a formula that is equisatisfiable to φ but simpler to solve. If the program comes with assertions, then equisatisfiability is enough to find bugs. Our approach is to perform a static analysis of $[\![p]\!]_{mm}$ determining information that is valid over all executions in the program semantics and hence need not be encoded. It is well-known how to statically analyze the program semantics $[\![p]\!]$. We propose a new analysis for the semantics of the memory model $[\![mm]\!]$. To understand which static information will be helpful to simplify the SMT encoding, we explain this semantics and its encoding on a running example.

2.1 Running Example

Consider the concurrent implementation of the Fibonacci sequence executed on ARMv8 [Pulte et al. 2018] given in Figure 2. The program has two threads that repeatedly add up numbers i and j. The number of additions is given as a parameter N so that the program is essentially acyclic. The correctness condition says that $fib(2N + 2)$ will never be exceeded.

Whether or not the assertion is met depends on the memory model of the underlying platform. Interestingly, the program is correct under very weak consistency requirements: it should not be possible to invent values out-of-thin-air. ARMv8 guarantees this. It models a processor with a store-atomic shared memory, meaning a write, once it arrives in memory, becomes visible to all threads. Writes to different memory locations may arrive in memory in an order that deviates from the program text, as long as it respects address and data dependencies. All this is captured in the memory model through the axiom `acyclic` ob, saying that a relation named ob should not contain contradictions. The relation is defined as a union of a relation named obs that tracks the observations threads make about the shared memory, and another relation called dob tracking the thread-local dependencies.

2.2 Memory Models

We consider memory models defined in the CAT language [Alglave 2010; Alglave et al. 2014b] as given in Figure 3. A memory model consists of binary relations and constraints, so-called axioms, over those binary relations. The axioms are emptiness, irreflexivity, and acyclicity. Relations may be derived from other relations using the following operations: union ($\cup$), intersection ($\cap$), difference ($\setminus$), composition ($;$), and inverse ($\bullet^{-1}$). Derived relations may be named `let` $d := r$, drawing their names from an unspecified set of relation names $\mathbf{D}$. In this case, we call `let` $d := r$ a *defining equation* for d. Derived relations may also be defined by mutual recursion, `let rec` $d_1 := r_1$ `and` $\dots$ `and` $d_k := r_k$ or `let rec` $\vec{d} := \vec{r}$ to shorten the syntax. There is a predefined set $\mathbf{B}$, disjoint from $\mathbf{D}$, of relation names to which we refer to as *base relations*. We denote by $\mathbf{R} = \mathbf{B} \cup \mathbf{D}$ the set of all relation names. We expect well-formedness: (i) each relation name is defined at most once, (ii) recursively dependent relations are defined within the same `let rec` construct, and (iii) recursion is monotonic.

$$
\begin{array}{rcl}
mm & ::= & def \mid const \mid mm \wedge mm \\
const & ::= & \mathbf{acyclic}(r) \mid \mathbf{irreflexive}(r) \mid \mathbf{empty}(r) \\
def & ::= & \texttt{let}\ d := r \mid \texttt{let rec}\ d := r\ (\texttt{and}\ d := r)^* \\
r & ::= & b \mid d \mid r^{-1} \mid r;r \mid r \cap r \mid r \cup r \mid r \setminus r \\
b & ::= & \mathsf{po} \mid \mathsf{rf} \mid \mathsf{co} \mid \mathsf{loc} \mid [t] \mid t \times t \mid \dots \\
t & ::= & \mathbb{W} \mid \mathbb{R} \mid \mathbb{F} \mid \dots
\end{array}
$$

Fig. 3. Grammar for memory models.

We explain these concepts on ARMv8 of Figure 2 (right). The relations ca, obs, dob, and ob are all derived relations that have been named. They are derived from other relations, many of which are base relations like po, co, and rf that do not have a defining equation. From the perspective of the memory model, the base relations can be arbitrary. It is the program that will give them proper semantics. For example, po will model the order in which program instructions are issued, co will model the order in which store operations are committed to memory, and rf will model the interaction between a load operation and the store operation it takes its value from. The base relation `[W]` is the identity on all store operations, i.e., relates each store to itself.

The semantics $[\![fib(N)]\!]_{\mathrm{ARMv8}}$ or more generally $[\![p]\!]_{mm}$ is defined in terms of executions, directed and edge-labeled graphs (X, I) as the one in Figure 4 (ignore all highlighting for the moment). The

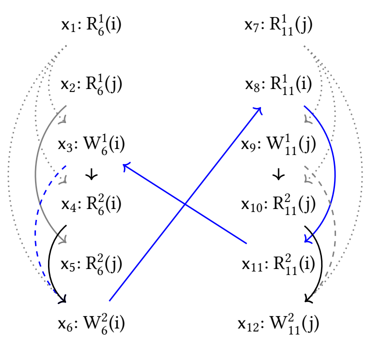

Fig. 4. Execution of Fibonacci, $N = 2$.

finite set of so-called events X represents the program commands that have been executed. In the figure, the six events to the left represent two executions of the command `i = i+j`. For each event, we add a subscript to denote the line of code and a superscript to denote the loop iteration that produced the event. The first two events read the current values of i and j, the next write event installs the new value of i, and then the process repeats. We use $\mathbb{X}_p$ for the set of all events that are possible in the program. The labeled edges are given by an interpretation $\mathsf{I} : \mathbf{R} \longrightarrow \mathsf{X} \times \mathsf{X} \longrightarrow \{0, 1\}$ that assigns to each relation name from the finite set $\mathbf{R}$ an actual relation over X. In Figure 4, the interpretation of the program order I(po) consists of all edges between events x_i and x_j from the same thread where $i < j$. The figure only shows I(po) ∩ I(loc), the restriction of the program order to events accessing the same location. The interpretation of the coherence relation I(co) is given by the dashed edges, so the edge from x_3 to x_6 is labeled by po, loc, and co. The edge from x_6 to x_8 is read-from (I(rf)), all other read events access an initial write that is not depicted. The edge from x_{11} to x_3 is from-read I(fr), with fr defined as rf^{-1}; co. Indeed, x_{11} reads the initial value of i which gets overwritten by x_3. The example does not belong to ⟦ARMv8⟧ because the cycle in blue contradicts `acyclic` (po ∩ loc) ∪ ca ∪ rf.

2.3 SMT Encoding

We briefly explain the SMT encoding φ_{mm} of the memory model semantics ⟦mm⟧ that we are about to optimize. Two sets of variables are of particular interest. Boolean-valued so-called execution variables exec_x indicate whether the event $\mathsf{x} \in \mathbb{X}_p$ is present in the execution (belongs to the set X). So-called relation variables $\mathsf{n}_{\mathsf{x},\mathsf{y}}$ express the value of relation I(n) at entry (x, y). Note that the value of $\mathsf{n}_{\mathsf{x},\mathsf{y}}$ is only meaningful if both x and y are also executed. So $\mathsf{I(n)(x, y)} = 1$ is not witnessed by the truth of just $\mathsf{n}_{\mathsf{x},\mathsf{y}}$ but needs $\mathsf{exec}_\mathsf{x} \land \mathsf{exec}_\mathsf{y} \land \mathsf{n}_{\mathsf{x},\mathsf{y}}$. Similarly, $\mathsf{I(n)(x, y)} = 0$ is represented by $\mathsf{exec}_\mathsf{x} \land \mathsf{exec}_\mathsf{y} \land \neg\mathsf{n}_{\mathsf{x},\mathsf{y}}$. There is a degree of freedom of how to evaluate $\mathsf{n}_{\mathsf{x},\mathsf{y}}$ when at least one of the events is missing in the execution, i.e., we have $\neg\mathsf{exec}_\mathsf{x} \lor \neg\mathsf{exec}_\mathsf{y}$. We use this freedom to additionally require the implication

$$\mathsf{n}_{\mathsf{x},\mathsf{y}} \quad \longrightarrow \quad \mathsf{exec}_\mathsf{x} \land \mathsf{exec}_\mathsf{y}. \tag{1}$$

We refer to Implication (1) as the *key implication*, because it leads to a particularly simple encoding of the memory model as $\varphi_{mm} = \varphi_{def} \land \varphi_{axiom}$. The defining equations of derived relations are encoded into φ_{def} as defined in Figure 5 (left). We emphasize that this straightforward translation is only sound thanks to the key implication, which we add to the overall encoding as φ_{key}. The translation is not sound for all CAT-definable memory models, because it does not respect the least fixed point semantics of recursive definitions. Interestingly, however, it is sound for all practically known memory models. The reason is that (i) CAT axioms $const(r)$ are antitonic in their parameter r (antitonic means they become harder to satisfy as the relation grows), a fact that was observed in [Ponce de León et al. 2018], and (ii) in all memory models we know of r depends monotonically on all recursively defined relations. The axioms are encoded as defined in **??** (right). Emptiness and irreflexivity simply negate the relation variables. The encoding of the acyclicity axiom turns an edge into an ordering requirement in integer difference logic [Gebser et al. 2014]. All quantifiers in the encodings range over finite sets and will be compiled down into disjunctions and conjunctions. The base relations are *not* encoded into φ_{mm} and instead are part of the program encoding φ_p. For an overview of the program encoding, we refer to the appendix.

`let` $c = a \cup b$:	$c_{x,y} \leftrightarrow a_{x,y} \vee b_{x,y}$		
`let` $c = a \cap b$:	$c_{x,y} \leftrightarrow a_{x,y} \wedge b_{x,y}$	**empty**(n) :	$\forall x, y \in \mathbb{X}_p : \neg n_{x,y}$
`let` $c = a \setminus b$:	$c_{x,y} \leftrightarrow a_{x,y} \wedge \neg b_{x,y}$	**irreflexive**(n) :	$\forall x \in \mathbb{X}_p : \neg n_{x,x}$
`let` $c = a; b$:	$c_{x,y} \leftrightarrow \exists z \in \mathbb{X}_p : a_{x,z} \wedge b_{z,y}$	**acyclic**(n) :	$\forall x, y \in \mathbb{X}_p : n_{x,y} \rightarrow clk_{n,x} < clk_{n,y}$
`let` $c = a^{-1}$:	$c_{x,y} \leftrightarrow a_{y,x}$		

Fig. 5. Encodings of derived relations and axioms as formulas φ_{def} resp. φ_{axiom}.

2.4 SMT Optimizations

We observe that a certain form of static information is particularly helpful to optimize the above encoding. This information are upper and lower bounds on the relations in the memory model. The *may-set* $may(n)$ associated to relation name n is an upper bound on the interpretation $I(n)$ that holds over all executions consistent with the memory model. Whenever events x and y are related by n in a consistent execution, meaning $I(n)(x, y) = 1$, then we have $(x, y) \in may(n)$. Dually, the *must-set* $must(n)$ gives a lower bound on the interpretation. If $(x, y) \in must(n)$ and the two events occur in a consistent execution, then we know they are related, $I(n)(x, y) = 1$. Importantly, the must-set is defined relative to the execution of events, otherwise it would be constantly empty.

Assume the may and must-sets have already been computed. We explain how we use them to optimize the SMT encoding.

Step 1: Substitution. The first step is to simplify the encoding by replacing relation variables with known values. Suppose $(x, y) \notin may(n)$, meaning (x, y) are never related by n in any consistent executions. Then all satisfying assignments of our encoding φ must also satisfy $n_{x,y} \leftrightarrow$ *false*. This allows us to simply replace $n_{x,y}$ by *false* in the encoding. Similarly, if we have $(x, y) \in must(n)$, then all satisfying assignments of φ will also satisfy $exec_x \wedge exec_y \rightarrow n_{x,y}$. Combined with the key implication, we get $exec_x \wedge exec_y \leftrightarrow n_{x,y}$, which allows us to replace $n_{x,y}$ by $exec_x \wedge exec_y$ in the encoding. The substitutions may already make clauses true, which we then eliminate.

Step 2: Acyclicity. The second step is to optimize the encoding of acyclicity axioms. Compared to emptiness and irreflexivity, acyclicity is a complex condition expressed best using integer difference logic.[1] The encoding of **acyclic**(n) introduces a clock variable $clk_{n,x}$ for every event x and adds to the SMT encoding for every pair of events (x, y) the implication $n_{x,y} \rightarrow clk_{n,x} < clk_{n,y}$.

We propose an optimization that reduces the number of such implications in two ways. The substitution performed in Step 1. already removed implications related to variables $n_{x,y}$ that are known to be always false. We further exploit the may-set information to also remove implications related to edges $n_{x,y}$ that can never occur in a cycle. Then we perform an optimization inspired by transitive reductions: we do not add an implication, if the edge $n_{x,y}$ forms a shortcut for a full path in relation n leading from x to y. However, edges may be missing in an execution. To be sound, the other path has to exist whenever the shortcut $n_{x,y}$ exists. Fortunately, combining must-set information with a standard control-flow analysis of the program gives us precisely the information we need. We expect the control-flow information to be given as a set of implications of the form $x \rightarrow y$ or $x \rightarrow \neg y$, stating that whenever x is executed then so is y, resp. that x and y are mutually exclusive. These implications will guarantee us the existence of the intermediary events while the must-set will guarantee us that those events are connected to form the desired path.

[1]Also when expressed in pure SAT, acyclicity is a complex condition to solve [Gebser et al. 2014].

We illustrate the optimization on the execution depicted in Figure 4, but emphasize that the static information has to be valid for all executions in $[\![p]\!]_{mm}$. The first phase removes the dotted edges: the may-set $\mathsf{may}(\mathsf{n})$ tells us that x_1 resp. x_7 will not have incoming edges, and therefore cannot be part of a cycle. The second phase removes the dashed edges with the following argument. The edge $\mathsf{n}_{\mathsf{x}_3,\mathsf{x}_6}$ forms a shortcut for $\mathsf{n}_{\mathsf{x}_3,\mathsf{x}_4}$ and $\mathsf{n}_{\mathsf{x}_4,\mathsf{x}_6}$, drawn in black, with $\mathsf{n} = (\mathsf{po} \cap \mathsf{loc}) \cup \mathsf{ca} \cup \mathsf{rf}$. The must-set tells us that $\mathsf{n}_{\mathsf{x}_3,\mathsf{x}_4} \leftrightarrow \mathsf{exec}_{\mathsf{x}_3} \wedge \mathsf{exec}_{\mathsf{x}_4}$ and $\mathsf{n}_{\mathsf{x}_4,\mathsf{x}_6} \leftrightarrow \mathsf{exec}_{\mathsf{x}_4} \wedge \mathsf{exec}_{\mathsf{x}_6}$, meaning the edges are present if the events get executed. Moreover, since the events are executed unconditionally ($\mathsf{exec}_{\mathsf{x}_i} \leftrightarrow \mathit{true}$), the edges are always present. We can thus drop the implication $\mathsf{n}_{\mathsf{x}_3,\mathsf{x}_6} \rightarrow \mathsf{clk}_{\mathsf{n},\mathsf{x}_3} < \mathsf{clk}_{\mathsf{n},\mathsf{x}_6}$, because the other two edges already guarantee $\mathsf{clk}_{\mathsf{n},\mathsf{x}_3} < \mathsf{clk}_{\mathsf{n},\mathsf{x}_4} < \mathsf{clk}_{\mathsf{n},\mathsf{x}_6}$.

Step 3: Thinning. The optimizations given so far modify the encoding of the memory model to a formula $\psi_{mm} = \psi_{def} \wedge \psi_{axiom}$. They guarantee that ψ_{def} is a conjunction of equivalences

$$\mathsf{n}_{\mathsf{x},\mathsf{y}} \leftrightarrow \mathsf{def}(\mathsf{n}, \mathsf{x}, \mathsf{y}) \qquad \mathit{false} \leftrightarrow \mathsf{def}(\mathsf{n}, \mathsf{x}, \mathsf{y}) \qquad \mathsf{exec}_\mathsf{x} \wedge \mathsf{exec}_\mathsf{y} \leftrightarrow \mathsf{def}(\mathsf{n}, \mathsf{x}, \mathsf{y})$$

so that each $\mathsf{n}_{\mathsf{x},\mathsf{y}}$ appears at most once on a left-hand side. We refer to these equivalences as defining constraints and denote the set of all *defining constraints* by $\mathbb{C}(\psi_{def})$.

Our third optimization is to thin out the set of defining constraints. The idea is to identify the defining constraints that are *active* in the sense that they actually relate memory model axioms to base relations. The remaining constraints are inactive and can be removed from $\mathbb{C}(\psi_{def})$ without changing the satisfiability status of the encoding. Consider Figure 6. The edges show dependencies between variables. The source of a black edge is contained in a must-set, while the source of a blue edge has unknown value. The must-set tells us that the value of $\mathsf{n}_{\mathsf{x}_3,\mathsf{x}_6}$ is always equal to $\mathsf{exec}_{\mathsf{x}_3} \wedge \mathsf{exec}_{\mathsf{x}_6}$. This equality holds irrespective of the values of its dependencies $\mathsf{rf}_{\mathsf{x}_3,\mathsf{x}_6}$ and $\mathsf{ca}_{\mathsf{x}_3,\mathsf{x}_6}$. Since $\mathsf{ca}_{\mathsf{x}_3,\mathsf{x}_6}$ no longer influences the evaluation of the acyclicity axiom, we can drop its defining constraint $\mathsf{ca}_{\mathsf{x}_3,\mathsf{x}_6} \leftrightarrow \mathsf{fr}_{\mathsf{x}_3,\mathsf{x}_6} \vee \mathsf{co}_{\mathsf{x}_3,\mathsf{x}_6}$ from the encoding. But now the relation variable $\mathsf{fr}_{\mathsf{x}_3,\mathsf{x}_6}$ (which is derived as $\mathsf{fr} = \mathsf{rf}^{-1};\mathsf{co}$) becomes irrelevant and we can drop its defining constraint as well. Although we had no static information about $\mathsf{ca}_{\mathsf{x}_3,\mathsf{x}_6}$ and $\mathsf{fr}_{\mathsf{x}_3,\mathsf{x}_6}$, we could eliminate both of them from our encoding.

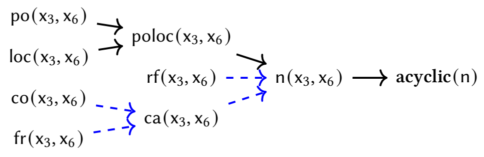

Fig. 6. Illustration of thinning on the running example, $\mathsf{poloc} = \mathsf{po} \cap \mathsf{loc}$ and $\mathsf{n} = \mathsf{poloc} \cup \mathsf{ca} \cup \mathsf{rf}$.

We propose a static analysis of the active constraints that processes ψ_{mm} top down, starting from the axioms. Say we have an axiom **irreflexive**(n). Then we only need to observe the diagonal entries of n, and can drop all defining constraints $\mathsf{n}_{\mathsf{x},\mathsf{y}} \leftrightarrow \mathsf{def}(\mathsf{n}, \mathsf{x}, \mathsf{y})$ where $\mathsf{x} \neq \mathsf{y}$.

3 A NEW FORMULATION OF $[\![mm]\!]$

We give the semantics of memory models a new formulation that is better suited for static analysis. In the standard formulation from [Alglave 2010; Alglave et al. 2014b], the semantics of memory models is *update-based*: it computes an interpretation of the derived relations from an interpretation of the base relations. We propose instead a *filter-based* semantics [Cousot 2021]: it starts from an interpretation of all relations and filters out undesirable interpretations. We show that the original update semantics and the new filter semantics coincide, and so the filter semantics is indeed a reformulation of the original definition.

Before we turn to the details, we explain the benefits of the filter semantics over the update semantics for static analysis. Memory models have a bidirectional information flow: the base relations propagate information to the axioms, and the axioms constrain the base relations. The update semantics only captures the first flow. Being undirected in nature, filters immediately reflect the bidirectional information flow. This leads to more precise abstractions.

Filters will also act as a generic interface to incorporate external information, in particular program information, into our semantics. Indeed, we will define the semantics parametric in a set of additional filters $\mathcal{F}$. This allows us to capture not only the update semantics $[\![mm]\!]_{\mathsf{up}}$ but also the program semantics under the memory model $[\![p]\!]_{mm}$. We denote the parameterized filter semantics by $[\![mm]\!]^{\mathcal{F}}$ and write $[\![mm]\!]$ for $[\![mm]\!]^{\emptyset}$ resp. $[\![mm]\!]^{f}$ for $[\![mm]\!]^{\{f\}}$. The following theorem will justify the definition of the filter semantics given below.

Theorem 3.1 (Soundness and completeness of the filter semantics). *$[\![mm]\!]_{\mathsf{up}} = [\![mm]\!]$ and $[\![p]\!]_{mm} = [\![mm]\!]^{f_p}$, where $f_p(\mathcal{E}) = \mathcal{E} \cap [\![p]\!]$.*

3.1 Update Semantics

The update semantics of a memory model mm is defined as the set of *consistent* executions. An execution $\mathsf{E}_{\mathbf{B}} = (\mathsf{X}, \mathsf{I}_{\mathbf{B}})$ has a set X of events and an interpretation $\mathsf{I}_{\mathbf{B}} : \mathbf{B} \to \mathsf{X} \times \mathsf{X} \to \{0, 1\}$ of all base relation names in terms of actual binary relations over the events X. The memory model determines consistency of the execution in two steps. First, it extends the interpretation $\mathsf{I}_{\mathbf{B}}$ to a complete interpretation $\mathsf{I} : \mathbf{B} \cup \mathbf{D} \to \mathsf{X} \times \mathsf{X} \to \{0, 1\}$ over also the derived relation names. The idea is to evaluate the defining equations, for recursive definitions this involves fixed points. The details can be found in the appendix. As the extension from $\mathsf{E}_{\mathbf{B}}$ to E is unique, we use both notions interchangeably. Then the memory model checks whether all axioms $const(r)$ are satisfied by E. If so, the execution is consistent. The update semantics is $[\![mm]\!]_{\mathsf{up}} = \{\mathsf{E} \mid \mathsf{E} \text{ is consistent}\}$.

We introduce helpful notation for sets of executions and relations. Let $\mathbb{X}$ be the set of all events that may occur in any program. We denote by $\mathsf{CExec}_{\mathsf{X}}$ the set of all executions over domain $\mathsf{X} \subseteq \mathbb{X}$ and by $\mathsf{CExec} = \bigcup_{\mathsf{X} \subseteq \mathbb{X}} \mathsf{CExec}_{\mathsf{X}}$ the set of all executions. We also use $\mathsf{CRel}_{\mathsf{X}}$ to refer to the set of binary relations $\mathsf{X} \times \mathsf{X} \to \{0, 1\}$ over domain X and $\mathsf{CRel} = \bigcup_{\mathsf{X} \subseteq \mathbb{X}} \mathsf{CRel}_{\mathsf{X}}$ to refer to binary relations over all possible domains.

3.2 Filter Semantics

The idea behind our new semantics is that axioms and defining equations in a memory model can be seen as semantic objects of the same type: *filters*. A filter on $\mathbb{P}(\mathsf{CExec})$ is a function that removes undesirable executions from a given set of executions. While it should be clear that axioms yield filters, this may need a word for defining equations `let` $d := r$. The update semantics takes the defining equation as a description of how to compute d from r. The filter semantics takes the defining equation for what it really is, namely a requirement of equality. Given a set of executions $\mathcal{E}$, the filter $[\![$`let` $d := r]\!]$ removes the executions where the interpretation of relation d is different from the interpretation of the relation expression r. To be able to reuse it in the abstract setting, we give the definition of filters for general lattices.

Definition 3.2. A *filter* on a lattice $(L, \sqsubseteq)$ is a function $f : L \to L$ that is reductive, $f(l) \sqsubseteq l$, and monotonic, $l \sqsubseteq m$ implies $f(l) \sqsubseteq f(m)$, for all $l, m \in L$.

A filter is thus a lower closure operator [Cousot 2021, Chapter IV, Section 11.7] without the requirement of idempotence. An important consequence of the definition for our new semantics is that we can characterize greatest common fixed points.

Lemma 3.3 (Compositionality). *Let f and g be filters on a complete lattice $(L, \sqsubseteq)$. Then gfp.$(f \sqcap g)$ is the largest element that is both a fixed point of f and of g.*

The lemma is key to the compositionality of the filter semantics mentioned in the introduction. It allows us to define the semantics parametric in a set of filters and generalize it to new base relations by simply adding the corresponding filters.

$$
\begin{array}{rcll}
\llbracket mm \rrbracket^{\mathcal{F}} & = & gfp.(\dot{\bigcap}_{f\in\mathcal{F}} f \,\dot{\cap}\, \dot{\bigcap}_{\pi\in mm} \llbracket \pi \rrbracket) & : \mathbb{P}(\mathsf{CExec}) \\
\llbracket const(r) \rrbracket(\mathcal{E}) & = & \{\mathsf{E} \in \mathcal{E} \mid \llbracket const \rrbracket(\llbracket r \rrbracket(\mathsf{E}))\} & : \mathbb{P}(\mathsf{CExec}) \\
\llbracket \texttt{let}\ d := r \rrbracket(\mathcal{E}) & = & \{\mathsf{E} \in \mathcal{E} \mid \llbracket d \rrbracket(\mathsf{E}) = \llbracket r \rrbracket(\mathsf{E})\} & : \mathbb{P}(\mathsf{CExec}) \\
\llbracket \texttt{let rec}\ \vec{d} := \vec{r} \rrbracket(\mathcal{E}) & = & \{\mathsf{E} \in \mathcal{E} \mid \llbracket \vec{d} \rrbracket(\mathsf{E}) = lfp.\llbracket \Theta(\vec{d},\vec{r}) \rrbracket(\mathsf{E})\} & : \mathbb{P}(\mathsf{CExec}) \\
\llbracket \Theta(\vec{d},\vec{r}) \rrbracket(\mathsf{E}) & = & \lambda\vec{u}.\llbracket \vec{r} \rrbracket(\mathsf{E}[\vec{d} \leftarrow \vec{u}]) & : \mathsf{CRel}^k \to \mathsf{CRel}^k \\
\llbracket r_1 \oplus r_2 \rrbracket(\mathsf{E}) & = & \llbracket \oplus \rrbracket(\llbracket r_1 \rrbracket(\mathsf{E}), \llbracket r_2 \rrbracket(\mathsf{E})) & : \mathsf{CRel} \\
\llbracket \mathsf{n} \rrbracket(\mathsf{E}) & = & \mathsf{E}(\mathsf{n}) & : \mathsf{CRel}
\end{array}
$$

Fig. 7. Concrete filter semantics.

The filter semantics is given in Figure 7. It is the largest set of executions that satisfies the filters $\llbracket \pi \rrbracket$ for all axioms, equations, and systems of equations in the memory model, $\pi \in mm$. By satisfying a filter, we mean the filter no longer reduces the set of executions: the set is a fixed point of the filter. Using Lemma 3.3, and disregarding the parameter $\mathcal{F}$, the semantics is the greatest common fixed point of the meet over the filters $\llbracket \pi \rrbracket$. We already discussed the filters $\llbracket const(r) \rrbracket$ and $\llbracket \texttt{let}\ d := r \rrbracket$ for axioms resp. defining equations. Consider a system of equations $\texttt{let rec}\ d_1 := r_1\ \texttt{and}\ \ldots\ \texttt{and}\ d_k := r_k$. Each relation expression r_1 to r_k may make use of all the derived relations d_1 to d_k, meaning the system is recursive. The interpretation of the derived relations sought is the least solution to the system of equations. This least solution can be expressed as a least fixed point. To this end, we understand each relation expression r from r_1 to r_k as a function of type $\mathsf{CRel}^k \to \mathsf{CRel}$. The input value of type CRel^k is the interpretation of the relation names d_1 to d_k. Inserting these relations into the expression r yields a return value of type CRel. We adapt this view for all expressions r_1 to r_k and obtain a function of type $\mathsf{CRel}^k \to \mathsf{CRel}^k$. In the definition of the semantics in Figure 7, this function is written as $\llbracket \Theta(\vec{d},\vec{r}) \rrbracket(\mathsf{E})$. The purpose of the execution E is to interpret the remaining relations. It can be shown that the least fixed point $lfp.\llbracket \Theta(\vec{d},\vec{r}) \rrbracket(\mathsf{E})$ is the least solution to the system of equations $\texttt{let rec}\ \vec{d} := \vec{r}$. The filter $\llbracket \texttt{let rec}\ \vec{d} := \vec{r} \rrbracket$ then checks that the interpretation of the derived relations d_1 to d_k in E corresponds to the interpretation computed with the least fixed point. We remark that the fixed point computation is crucial to guarantee the equivalence of the filter with the update semantics. The semantics of relation expressions $\llbracket r \rrbracket(\mathsf{E})$ is as expected. We have omitted unary operations, which are defined like their binary counterparts.

Lemma 3.4. *The functions $\llbracket \pi \rrbracket$ defined in Figure 7 are filters.*

We elaborate on the purpose of parameter $\mathcal{F}$ highlighted in Figure 7. In the context of bounded model checking, the semantics of interest is $\llbracket p \rrbracket_{mm} = \llbracket p \rrbracket \cap \llbracket mm \rrbracket$, i.e., the subset of the memory model semantics that also adheres to the program semantics. We can view the program p as a filter $f_p(\mathcal{E}) = \mathcal{E} \cap \llbracket p \rrbracket$ that removes from a set of executions the ones that do not belong to the program. By adding this filter to our semantics, we obtain the semantics of interest $\llbracket mm \rrbracket^{f_p} = \llbracket p \rrbracket_{mm}$. There are further applications, notably in conditional model checking [Beyer et al. 2012].

4 A STATIC ANALYSIS FOR $\llbracket mm \rrbracket^{\mathcal{F}}$

We devise a static analysis to compute the previously mentioned may and must-sets. Our approach is to construct an abstract filter semantics $\llbracket mm \rrbracket^{\mathcal{F}\#}$ from $\llbracket mm \rrbracket^{\mathcal{F}}$. The abstract semantics will take into account $\mathcal{F}$ by assuming to be given an abstract filter $f^{\#}$ for each $f \in \mathcal{F}$. Abstract filters are thus the standardized format in which we import information from external program analyses. The

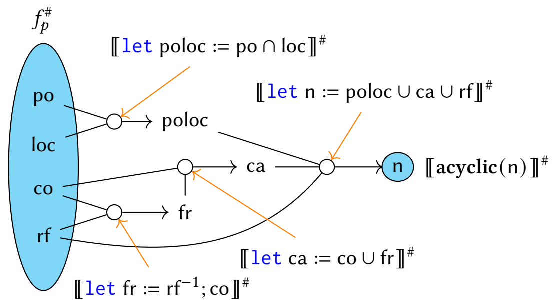
$f_p^\#$
po
loc
co
rf
poloc
ca
fr
n
$[\![\texttt{let}\ \mathsf{poloc} := \mathsf{po} \cap \mathsf{loc}]\!]^\#$
$[\![\texttt{let}\ \mathsf{n} := \mathsf{poloc} \cup \mathsf{ca} \cup \mathsf{rf}]\!]^\#$
$[\![\mathbf{acyclic}(\mathsf{n})]\!]^\#$
$[\![\texttt{let}\ \mathsf{ca} := \mathsf{co} \cup \mathsf{fr}]\!]^\#$
$[\![\texttt{let}\ \mathsf{fr} := \mathsf{rf}^{-1}; \mathsf{co}]\!]^\#$

Fig. 8. Propagation of analysis information through filters for relation n and its dependencies.

sets of control-flow implications $x \longrightarrow y$ and $x \longrightarrow \neg y$ mentioned earlier are particularly simple (but useful) abstract filters. The given abstract filters may also constrain the may- and must-sets. In particular, having at least an abstract program filter $f_p^\#$ is important, because it is the program that gives semantics to the base relations: read-from (rf) has to be functional, the program order (po) relates same-thread events, the coherence order (co) only relates same-address writes, and no base relation ever relates events of mutually exclusive program branches. We give a brief idea of how to construct an abstract program filter from static analysis information about the program. Whenever we can deduce that (x, y) do not alias, then we know that these events can never be related by read-from or coherence, $(x, y) \notin \mathsf{may}(\mathsf{rf})$ and $(x, y) \notin \mathsf{may}(\mathsf{co})$. Similarly, if a control-flow analysis tells us that (x, y) are mutually exclusive, then $(x, y) \notin \mathsf{may}(\mathsf{n})$ for all relations $\mathsf{n} \in \mathbf{R}$.

Figure 8 gives an overview of how the (abstract) program filter, the equation filters, and the axiom filters interact on an example fragment of the ARMv8 memory model. First, the program filter $f_p^\#$ infers information about the base relations. Then the equation filters propagate this information upwards towards relation n. Now the axiom filter $[\![\mathbf{acyclic}(\mathsf{n})]\!]^\#$ uses it to derive more information about n that is then propagated downwards again using the equation filters. If during the downward propagation new information reaches the base relations, the program filter $f_p^\#$ may be applied again and the propagation process repeats until a fixed point is reached. Note that while we explained an alternation of upward and downward propagation, filters are bidirectional and any iteration strategy would reach the same fixed point. Before we proceed to the details, we illustrate a computation of the abstract semantics on the Fibonacci program.

We start from an abstract execution $\mathsf{A} = \top$ (the exact shape of A does not matter for now) saying that all must-sets are empty (trivial lower bounds) and all may-sets are full (trivial upper bounds). First, we apply the abstract program filter $f_p^\#$. The resulting abstract execution $f_p^\#(\mathsf{A})$ approximates all base relations as depicted in the first graph (top-left) of Figure 9. For rf and co, we get the following: the may-sets contain only write-read resp. write-write event pairs that access the same location (the alias information for Fibonacci is simple). The must-sets of both relations are empty. The other two base relations po and loc are approximated precisely (the may and must-sets coincide). While this is always the case for the program order (because it is fixed by the syntax), the reason this also holds for the same-location relation loc is because we have precise alias information for Fibonacci (there are no dynamic memory accesses). On our example graphs, the may and must-set of the loc relation is the complete graph, so we omit the relation to simplify the figure. We propagate the above information upwards by applying the abstract equation filters $[\![\texttt{let}\ \mathsf{poloc} := \mathsf{po} \cap \mathsf{loc}]\!]^\#$, $[\![\texttt{let}\ \mathsf{fr} := \mathsf{rf}^{-1}; \mathsf{co}]\!]^\#$, $[\![\texttt{let}\ \mathsf{ca} := \mathsf{co} \cup \mathsf{fr}]\!]^\#$, and $[\![\texttt{let}\ \mathsf{n} := \mathsf{poloc} \cup \mathsf{ca} \cup \mathsf{rf}]\!]^\#$. This results in the abstract information depicted in the top-right graph of Figure 9. The backward may-edges in the bottom half of the graph stem from the from-read relation fr (propagated over ca), while the

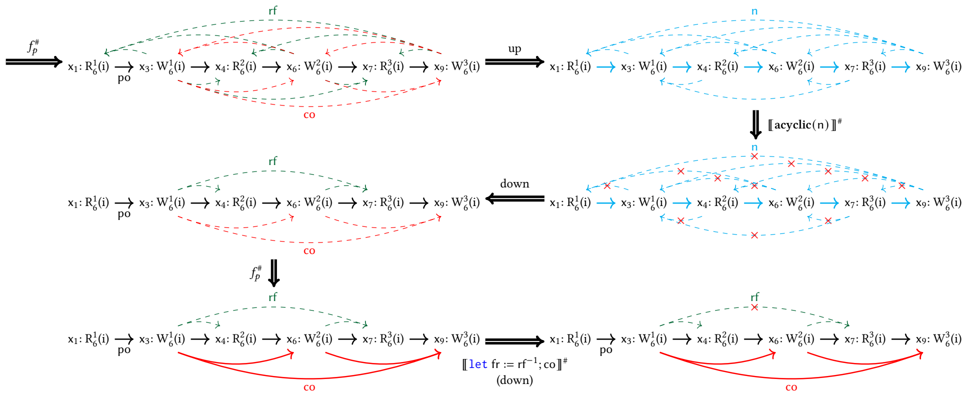

Fig. 9. Static analysis on `thread_i` of Fibonacci with N=3. Events on variable j are omitted. Dashed edges represent may-sets and thick edges represent must-sets. Transitive must-edges in po and n are omitted.

remaining edges stem directly from the base relations depicted in the first graph. The reader may wonder why the forward edges of co and rf are absent in this graph. The reason is that those edges are already transitively implied by the must-edges contributed by po (via poloc) and so we omit them. Now the abstract acyclicity filter $[\![\textbf{acyclic}(\mathsf{n})]\!]^\#$ will deduce that the backward may-edges opposing the forward must-edges cannot exist, for otherwise there would be a cycle: $(\mathsf{y}, \mathsf{x}) \notin \mathsf{may}(\mathsf{n})$ for all backward may-edges. This results in the third graph (middle-right) of Figure 9. By applying the four equation filters again, we propagate the reduced may-sets downwards to the base relations, resulting in the fourth graph (middle-left) where only forward edges remain. Interestingly, the program filter $f_p^\#$ can use this to deduce further information: co is a total order over store events to the same address, so if the backward co edges are impossible, then we must have the forward co edges as depicted in the fifth graph (bottom-left). With this, we can deduce from the abstract filter $[\![\texttt{let}\ \mathsf{fr} := \mathsf{rf}^{-1};\mathsf{co}]\!]^\#$ that $\mathsf{rf}(\mathsf{x}_3, \mathsf{x}_7)$ is impossible; if that edge was ever present, we could compose its inverse with the must-edge $\mathsf{co}(\mathsf{x}_3, \mathsf{x}_6)$ to obtain the disallowed from-read edge $\mathsf{fr}(\mathsf{x}_7, \mathsf{x}_6)$ (this edge was removed from the may-set of fr in the first downward propagation). Actually, we need control-flow information to justify this reasoning: it would not hold if it was possible to execute x_3 and x_7 without x_6. However, a simple control-flow analysis will show that whenever x_7 gets executed then so does x_6.

On a high-level, we (automatically) computed the following information from the memory model: (i) a load cannot read from a store that is later in the same thread, (ii) stores to the same location are committed into memory in the order they appear in the program syntax, and (iii) a load cannot observe a same-thread store that was already overwritten within the thread. Notice that none of this information is derivable by analyzing $[\![mm]\!]$ or $[\![p]\!]$ alone.

4.1 Domains

Recall that the concrete semantics of memory models is defined over concrete executions (CExec) that define concrete relations (CRel). These concrete relations consist of both a domain ($\mathsf{X} \subseteq \mathbb{X}$) and a valuation ($\mathsf{X} \times \mathsf{X} \to \{0, 1\}$). This strict distinction between the domain and the valuation will be helpful for defining our static analysis. Our analysis is based on a set of abstract domains that mimic the structure of the concrete domains. These domains are the *abstract executions* AExec, *abstract relations* ARel, *abstract control-flow* ACF, and *abstract valuations* AVal. They are as follows:

$$\begin{array}{rclrcl} \mathsf{AExec} & = & \mathbf{R} \to \mathsf{ARel} & \mathsf{ACF} & \subseteq & \mathbb{P}(\{\mathsf{x} \to \mathsf{y}, \mathsf{x} \to \neg\mathsf{y} \mid \mathsf{x}, \mathsf{y} \in \mathbb{X}\}) \\ \mathsf{ARel} & = & \mathsf{ACF} \times \mathsf{AVal} & \mathsf{AVal} & \subseteq & \mathbb{X} \times \mathbb{X} \to \mathbb{P}(\{0,1\})\ . \end{array}$$

We explain the abstract domains and their connection to the concrete ones via a set of Galois connections [Cousot 2021, Chapter IV, Section 11]:

$$\begin{array}{l} (\mathbb{P}(\mathsf{CExec}), \subseteq) \ \underset{\alpha_\times}{\overset{\gamma_\times}{\leftrightarrows}} \ (\mathbf{R} \to \mathbb{P}(\mathsf{CRel}), \dot{\subseteq}) \ \underset{\dot{\alpha}_{rel}}{\overset{\dot{\gamma}_{rel}}{\leftrightarrows}} \ (\mathbf{R} \to \mathsf{ARel}, \dot{\sqsubseteq}) = (\mathsf{AExec}, \dot{\sqsubseteq}) \\ (\mathbb{P}(\mathsf{CRel}), \subseteq) \ \underset{\alpha_{rel}}{\overset{\gamma_{rel}}{\leftrightarrows}} \ (\mathsf{ARel}, \sqsubseteq) = (\mathsf{ACF} \times \mathsf{AVal}, \sqsubseteq_{cf} \times \sqsubseteq_{val}) \\ (\mathbb{P}(\mathsf{CRel}), \subseteq) \ \underset{\alpha_{cf}}{\overset{\gamma_{cf}}{\leftrightarrows}} \ (\mathsf{ACF}, \sqsubseteq_{cf}) \qquad (\mathbb{P}(\mathsf{CRel}), \subseteq) \ \underset{\alpha_{val}}{\overset{\gamma_{val}}{\leftrightarrows}} \ (\mathsf{AVal}, \sqsubseteq_{val})\ . \end{array}$$

An abstract execution takes the shape $\mathsf{A} : \mathbf{R} \to \mathsf{ARel}$. Each relation name $\mathsf{n} \in \mathbf{R}$ is mapped to an abstract relation $\mathsf{A(n)} : \mathsf{ARel} = \mathsf{ACF} \times \mathsf{AVal}$. The abstract relation represents a set of concrete relations $\mathsf{R} \in \mathbb{P}(\mathsf{CRel})$ by capturing their domains in an abstract domain $\mathsf{D} \in \mathsf{ACF}$ and their valuations in an abstract valuation $a \in \mathsf{AVal} \subseteq \mathbb{X} \times \mathbb{X} \to \mathbb{P}(\{0,1\})$. We understand D as control-flow information and represent it as a set of implications $\mathsf{D} \in \mathsf{ACF} \subseteq \mathbb{P}(\{\mathsf{x} \to (\neg)\mathsf{y} \mid \mathsf{x}, \mathsf{y} \in \mathbb{X}\})$. Implications $\mathsf{x} \to \mathsf{y}$ mean that whenever x is in the domain of a concrete relation $\mathsf{r} \in \mathsf{R}$, then so is y. Implications $\mathsf{x} \to \neg\mathsf{y}$ mean that the events are mutually exclusive: there is no relation $\mathsf{r} \in \mathsf{R}$ that has both events in its domain. We assume that all sets of control-flow implications in ACF are closed in the following sense: (i) if $\mathsf{x} \to \mathsf{y}$ and $\mathsf{y} \to (\neg)\mathsf{z}$ are contained, then so is $\mathsf{x} \to (\neg)\mathsf{z}$, and (ii) if $\mathsf{x} \to \neg\mathsf{y}$ is contained, then so is its contrapositive $\mathsf{y} \to \neg\mathsf{x}$. We equip ACF with the superset ordering $\sqsubseteq_{cf} = \supseteq$ so that smaller (more precise) elements contain more implications. The join $\sqcup_{cf}$ is intersection, the meet $\sqcap_{cf}$ is union followed by closing the resulting set of implications as just defined. The abstraction function $\alpha_{cf}(\mathsf{R}) = \{\mathsf{x} \to (\neg)\mathsf{y} \mid \forall \mathsf{r} \in \mathsf{R} : \mathsf{x} \in \mathsf{dom(r)} \Rightarrow (\neg)\mathsf{y} \in \mathsf{dom(r)}\}$ collects the implications that hold over all relations in the given set.

For a set of concrete relations R, the abstract valuation $a = \alpha_{val}(\mathsf{R})$ just collects all concrete valuations into one: $a(\mathsf{x},\mathsf{y}) = \{\mathsf{r}(\mathsf{x},\mathsf{y}) \mid \mathsf{r} \in \mathsf{R} \land \mathsf{x}, \mathsf{y} \in \mathsf{dom(r)}\}$. For example, we have $a(\mathsf{x},\mathsf{y}) = \{1\}$ if all concrete relations $\mathsf{r} \in \mathsf{R}$ that have x, y in their domain satisfy $\mathsf{r}(\mathsf{x},\mathsf{y}) = 1$, and there exists at least one relation with both events in its domain. The value $a(\mathsf{x},\mathsf{y}) = \emptyset$ means that there is no concrete relation defined over (x,y). Note that in this case we can derive the abstract control-flow information $\mathsf{x} \to \neg\mathsf{y}$ from the abstract valuation. We will later define a reduction operator on $\mathsf{ARel} = \mathsf{ACF} \times \mathsf{AVal}$ to exchange this type of information between the abstract domains. The abstract valuations obtained by the abstraction α_{val} inherit domain-consistency properties from the concrete relations. For example, if $a(\mathsf{x},\mathsf{y}) = \emptyset$ then we must also have $a(\mathsf{y},\mathsf{x}) = \emptyset$, because every concrete relation that is undefined at (x,y) is also undefined at (y,x). We restrict our domain of abstract valuations and define $\mathsf{AVal} = \alpha_{val}(\mathbb{P}(\mathsf{CRel})) \subseteq \mathbb{X} \times \mathbb{X} \to \mathbb{P}(\{0,1\})$. We equip AVal with the pointwise lifting $\dot{\subseteq}$ of the subset ordering on $\mathbb{P}(\{0,1\})$. For the join and meet, we also use the pointwise liftings $\sqcup_{val} = \dot{\cup}$ and $\sqcap_{val} = \dot{\cap}$.

We combine the control-flow abstraction α_{cf} and the valuation abstraction α_{val} into a relation abstraction α_{rel} with $\alpha_{rel}(\mathsf{R}) = (\alpha_{cf}(\mathsf{R}), \alpha_{val}(\mathsf{R}))$. The concretization is $\gamma_{rel}(\mathsf{a}) = \gamma_{cf}(\mathsf{a}) \cap \gamma_{val}(\mathsf{a})$. The ordering on ARel is defined componentwise, $\sqsubseteq = \sqsubseteq_{cf} \times \sqsubseteq_{val}$. For an abstract relation $\mathsf{a} = (\mathsf{D}, a)$, we often write $\mathsf{a}(\mathsf{x},\mathsf{y})$ to mean $a(\mathsf{x},\mathsf{y})$.

It remains to explain the abstraction α_{exec} from sets of concrete executions $\mathbb{P}(\mathsf{CExec})$ to abstract executions AExec. The abstraction $\alpha_{exec} = \dot{\alpha}_{rel} \circ \alpha_\times$ is performed in two steps. First, a cartesian abstraction [Cousot 2021, Chapter IX] pushes the powerset over executions to the concrete relations. It maps a set of concrete executions $\mathcal{E}$ to a single function $\alpha_\times(\mathcal{E})$ that, given a relation name, collects the interpretation over all executions in the set: $\alpha_\times(\mathcal{E}) = \lambda \mathsf{r}.\{\mathsf{I(r)} \mid \mathsf{E} = (\mathsf{X}, \mathsf{I}) \in \mathcal{E}\}$. On

$\mathbf{R} \to \mathbb{P}(\mathsf{CRel})$, we use the pointwise lifting $\dot{\subseteq}$ of the subset ordering on $\mathbb{P}(\mathsf{CRel})$. The abstraction drops the information which concrete relations occur together in an execution. Assume we have the relation names $\{\mathsf{n}, \mathsf{m}\}$ and executions $\mathcal{E} = \{\mathsf{E}_1, \mathsf{E}_2\}$ with $\mathsf{E}_i = (\mathsf{X}_i, \mathsf{I}_i)$. The abstraction is the function $\alpha_\times(\mathcal{E}) = \lambda \mathsf{r} \in \{\mathsf{n}, \mathsf{m}\}.\{\mathsf{I}_1(\mathsf{r}), \mathsf{I}_2(\mathsf{r})\}$. The concretization of this function is the set of executions that, besides $\mathcal{E}$, also contains $\mathsf{E}_3 = (\mathsf{X}_3, \mathsf{I}_3)$ with $\mathsf{I}_3(\mathsf{n}) = \mathsf{I}_1(\mathsf{n})$ and $\mathsf{I}_3(\mathsf{m}) = \mathsf{I}_2(\mathsf{m})$, n is interpreted as in E_1 and m as in E_2. The second abstraction $\dot{\alpha}_{rel}$ is the pointwise lifting of α_{rel} as we defined before.

The reader may be wondering how the abstract executions are connected to the may- and must-sets we have talked about initially. The may- and must-sets are extracted as follows:

$$\mathsf{may}(\mathsf{n}) = \{(\mathsf{x}, \mathsf{y}) \mid 1 \in \mathsf{A}(\mathsf{n})(\mathsf{x}, \mathsf{y})\} \qquad \mathsf{must}(\mathsf{n}) = \{(\mathsf{x}, \mathsf{y}) \mid \mathsf{A}(\mathsf{n})(\mathsf{x}, \mathsf{y}) = \{1\}\}\,.$$

We remark that, besides the may- and must-sets, our abstract domain explicitly contains control-flow information. This is crucial to compute precise must-sets, as we will see later.

Reduced domains. The domains ARel and AExec contain redundant representations, that is, abstract elements that represent the same set of concrete elements. In both cases, the redundancy is due to inconsistent domain information. Consider an abstract relation $\mathsf{a} = (\mathsf{D}, a)$ with $\mathsf{x} \to \neg\mathsf{y} \in \mathsf{D}$ and $a(\mathsf{x}, \mathsf{y}) = \{0, 1\}$. The abstract valuation says that the represented set of concrete relations contains some relations that evaluate to 0 and some that evaluate to 1 at (x, y). However, the control-flow implication $\mathsf{x} \to \neg\mathsf{y}$ says that no relation is defined at (x, y). As a consequence, $a[(\mathsf{x}, \mathsf{y}) \leftarrow \emptyset]$ is a more precise representation of the same set of concrete relations. We recover this information with a reduction operator $\rho_{rel} : \mathsf{ARel} \to \mathsf{ARel}$. It will be helpful to define this reduction in terms of the two functions $\beta_{val} = (\alpha_{val} \circ \gamma_{cf}) : \mathsf{ACF} \to \mathsf{AVal}$ and $\beta_{cf} = (\alpha_{cf} \circ \gamma_{val}) : \mathsf{AVal} \to \mathsf{ACF}$. They transform an element of one abstract domain to an element of the other as follows:

$$\beta_{val}(\mathsf{D})(\mathsf{x}, \mathsf{y}) \;=\; \begin{cases} \emptyset, & \text{if } \mathsf{x} \to \neg\mathsf{y} \in \mathsf{D} \\ \{0, 1\}, & \text{else} \end{cases} \qquad\qquad \beta_{cf}(a) \;=\; \{\mathsf{x} \to \neg\mathsf{y} \mid a(\mathsf{x}, \mathsf{y}) = \emptyset\}.$$

We now set $\rho_{rel}(\mathsf{D}, a) = (\mathsf{D} \sqcap_{cf} \beta_{cf}(a), a \sqcap_{val} \beta_{val}(\mathsf{D}))$. The function ρ_{rel} is monotonic and reductive by construction, and it can be shown to be idempotent. Importantly, it is sound.

Lemma 4.1 (Soundness of the reduction on abstract relations). *$\gamma_{rel} = \gamma_{rel} \circ \rho_{rel}$.*

Now we address the redundancies in AExec. Concrete executions have the shape $\mathsf{E} = (\mathsf{X}, \mathsf{I})$ where $\mathsf{I} : \mathbf{R} \to \mathsf{X} \times \mathsf{X} \to \{0, 1\}$, meaning all relation names are interpreted over the same domain. In abstract executions, the domains of abstract relations may be different. We define a reduction $\rho_{exec} : \mathsf{AExec} \to \mathsf{AExec}$ that collects the domains of all abstract relations, intersects them ($\sqcap_{cf}$), and then reduces all abstract relations to that common domain. Formally, we have the following definition: $\rho_{exec}(\mathsf{A})(\mathsf{n}) = \mathsf{let}\ \mathsf{D}_\mathsf{A} = \bigsqcap_{\mathsf{m} \in \mathbf{R}} \pi_1(\mathsf{A}(\mathsf{m}))\ \mathsf{in}\ \rho_{rel}(\mathsf{A}(\mathsf{n}) \sqcap (\mathsf{D}_\mathsf{A}, \top_{val}))$.

Lemma 4.2 (Soundness of the reduction on abstract executions). *$\gamma_{exec} = \gamma_{exec} \circ \rho_{exec}$.*

To simplify the presentation of our static analysis, we will assume that the reduction operators are applied throughout the computation, so that we only work over reduced elements.

4.2 Static Analysis

Our goal is to compute static information about the filter semantics $[\![mm]\!]^{\mathcal{F}}$. To this end, we define an abstract filter semantics $[\![mm]\!]^{\mathcal{F}\#}$ that soundly approximates the concrete semantics, $\alpha_{exec}([\![mm]\!]^{\mathcal{F}}) \subseteq [\![mm]\!]^{\mathcal{F}\#}$, and can be computed efficiently. The definition of the abstract semantics mimics the definition of the concrete semantics: it has abstract filters, abstract relation expressions, and abstract operators/predicates, corresponding to $[\![\pi]\!]$, $[\![r]\!]$, and $[\![\oplus]\!]/[\![const]\!]$, respectively. In the concrete semantics, filters are built from relation expressions, and relation expressions are built from operators. We retain this compositionality in the abstract semantics, but the abstract operators come in two flavors: a forward version and a backward version. These versions can be understood as propagating information from the inputs of the operator to its output (forwards), and from its output to its inputs (backwards). We require both versions to construct our abstract filters which simultaneously propagate information from base relations to derived relations (forwards) and from derived relations to base relations (backwards).

Abstract Filter Semantics. We explain the approximation of filters. The idea common to all definitions is to first evaluate the forward approximations of some relation expressions, restrict the results according to the filter, and then use the restricted results in the backward approximations of those expressions. Consider the approximation $[\![const(r)]\!]^{\#}(\mathsf{A})$ of the axiom filter $[\![const(r)]\!](\mathcal{E})$. We first compute the forward semantics $[\![r]\!]^{\#}_{F}(\mathsf{A})$ of the relation expression and apply the abstract constraint $[\![const]\!]^{\#}([\![r]\!]^{\#}_{F}(\mathsf{A}))$ on the result. This computation is very similar to the concrete filtering semantics, except that we evaluate abstract forward operators $[\![\oplus]\!]^{\#}_{F}$ (as part of $[\![r]\!]^{\#}_{F}$) and an abstract version $[\![const]\!]^{\#}$ of the constraint. We defer the definition of both for the moment. The result c of the abstract constraint is then given as an additional parameter to the backward computation $[\![r]\!]^{\#}_{B}(\mathsf{A}, \mathsf{c})$. The backward computation is best explained in the concrete. Let $\mathcal{E}$ be the concretization of A and R the concretization of c. The backward computation determines the largest set of executions $\mathcal{E}' \subseteq \mathcal{E}$ so that when evaluating expression r we obtain a relation in R. This has the effect of determining from the result R of the computation information about the input $\mathcal{E}'$. With this in mind, the backward semantics of a relation expression yields an abstract execution, not an abstract relation.

For the filters resulting from defining equations, we approximate the desired equality by a meet. For recursive systems of defining equations, we approximate the least fixed point in two steps. The reason is that a least fixed point comes with two requirements: the value should be a fixed point and it should be the least one. For the first requirement, we use the meet $\dot{\bigsqcap} [\![\texttt{let}\ d_i := r_i]\!]^{\#}$ over the approximations of the single equations. For the second requirement, we iterate the forward semantics $[\![\Theta(\vec{d}, \vec{r})]\!]^{\#}_{F}$ of the recursive system, starting from a suitable element in the lattice. We do not know of a backward computation that would guarantee minimality of the fixed point. Also note that, forwards, we do not simply compute the least fixed point. The reason is that the fixed point computations in the concrete semantics happen over relations $\mathsf{X} \times \mathsf{X} \to \{0, 1\}$ where the set $\{0, 1\}$ is ordered logically: $0 < 1$ (*false* $<$ *true*). The ordering $\sqsubseteq$ on abstract relations does not reflect this logical ordering, but instead the subset ordering of $\mathbb{P}(\mathsf{CRel})$. The operator $lfp^{\#}$ solves the problem by forming a Kleene chain with the forward abstract union $[\![\cup]\!]^{\#}_{F}$ instead of the abstract join. The chain starts from an abstract execution in which all relation names map to $\{0\}$ (reflecting the least element in logical order) rather than $\emptyset$ (reflecting the least element in subset order). The construction of the abstract filter semantics is given in Figure 10. We define the missing operators below. Under the standard soundness assumption for the abstract filters that are given as parameters, $\alpha_{exec} \circ f \circ \gamma_{exec} \mathrel{\dot{\sqsubseteq}} f^{\#}$ for all $f \in \mathcal{F}$, our abstract semantics is sound.

Theorem 4.3 (Soundness). $\alpha_{exec}([\![mm]\!]^{\mathcal{F}}) \subseteq [\![mm]\!]^{\mathcal{F}\#}$.

4.3 Abstract Constraints and Operators

We define the forward and backward semantics of constraints and operators. Recall that we always apply the reductions ρ_{rel} and ρ_{exec} before evaluating filters and operators. This allows us to assume that whenever we compute $[\![\oplus]\!]^{\#}(\mathsf{a}, \mathsf{b})$, both abstract relations have a common domain $\mathsf{D} \in \mathsf{ACF}$ and the result will also have that domain. It thus remaims to define the abstract valuation of the result. We define abstract valuations using sets of implications of the form $\mathsf{premise}(\vec{x}) \Rightarrow \mathsf{conclusion}(\vec{x})$.

$$
\begin{array}{rcll}
[\![mm]\!]^{\mathcal{F}\#} & = & gfp.(\dot{\bigsqcap}_{f\in\mathcal{F}} f^{\#} \mathbin{\dot\sqcap} \dot{\bigsqcap}_{\pi\in mm} [\![\pi]\!]^{\#}) & : \mathsf{AExec} \\
[\![const(r)]\!]^{\#}(\mathsf{A}) & = & \mathtt{let}\ c = [\![const]\!]^{\#}([\![r]\!]^{\#}_{F}(\mathsf{A}))\ \mathtt{in}\ [\![r]\!]^{\#}_{B}(\mathsf{A}, c) & : \mathsf{AExec} \\
[\![\mathtt{let}\ d := r]\!]^{\#}(\mathsf{A}) & = & \mathtt{let}\ c = \mathsf{A}(d) \sqcap [\![r]\!]^{\#}_{F}(\mathsf{A})\ \mathtt{in}\ \mathsf{A}[d \leftarrow c] \mathbin{\dot\sqcap} [\![r]\!]^{\#}_{B}(\mathsf{A}, c) & : \mathsf{AExec} \\
[\![\mathtt{let\ rec}\ \vec{d} := \vec{r}]\!]^{\#}(\mathsf{A}) & = & \mathtt{let}\ \vec{c} = \mathsf{A}(\vec{d}) \mathbin{\dot\sqcap} lfp^{\#}.[\![\Theta(\vec{d},\vec{r})]\!]^{\#}_{F}(\mathsf{A}) & \\
 & & \quad \mathtt{in}\ \mathsf{A}[\vec{d} \leftarrow \vec{c}] \mathbin{\dot\sqcap} \dot{\bigsqcap} [\![\mathtt{let}\ d_i := r_i]\!]^{\#}(\mathsf{A}) & : \mathsf{AExec} \\
 & & & \\
[\![\Theta(\vec{d},\vec{r})]\!]^{\#}_{F}(\mathsf{A}) & = & \lambda \vec{a}.[\![\vec{r}]\!]^{\#}_{F}(\mathsf{A}[\vec{d} \leftarrow \vec{a}]) & : \mathsf{ARel}^{k} \longrightarrow \mathsf{ARel}^{k} \\
[\![r_1 \oplus r_2]\!]^{\#}_{F}(\mathsf{A}) & = & [\![\oplus]\!]^{\#}_{F}([\![r_1]\!]^{\#}_{F}(\mathsf{A}), [\![r_2]\!]^{\#}_{F}(\mathsf{A})) & : \mathsf{ARel} \\
[\![\mathsf{n}]\!]^{\#}_{F}(\mathsf{A}) & = & \mathsf{A}(\mathsf{n}) & : \mathsf{ARel} \\
 & & & \\
[\![r_1 \oplus r_2]\!]^{\#}_{B}(\mathsf{A}, a) & = & \mathtt{let}\ \langle b, c\rangle = [\![\oplus]\!]^{\#}_{B}([\![r_1]\!]^{\#}_{F}(\mathsf{A}), [\![r_2]\!]^{\#}_{F}(\mathsf{A}), a) & \\
 & & \quad \mathtt{in}\ [\![r_1]\!]^{\#}_{B}(\mathsf{A}, b) \mathbin{\dot\sqcap} [\![r_2]\!]^{\#}_{B}(\mathsf{A}, c) & : \mathsf{AExec} \\
[\![\mathsf{n}]\!]^{\#}_{B}(\mathsf{A}, a) & = & \mathsf{A}[\mathsf{n} \leftarrow \mathsf{A}(\mathsf{n}) \sqcap a] & : \mathsf{AExec}
\end{array}
$$

Fig. 10. Abstract filter semantics.

These implications refer to valuations defined in the context of the implication plus a valuation $\bullet$ which is meant to be defined. If the premise is simply true, we omit it. A valuation r satisfies such an implication if the implication holds with $\bullet$ replaced by r for all assignments $\vec{x} \mapsto \vec{\mathsf{x}}$ of the variables to events from $\mathbb{X}$. For a set of implications I, we write $\lceil I \rceil$ for the greatest relation with domain D that satisfies all implications in I. To further ease the notation, we write $\mathsf{x} \longrightarrow \mathsf{y}$ for $\mathsf{x} \longrightarrow \mathsf{y} \in \mathsf{D}$. The abstract emptiness and irreflexivity constraints are defined by

$$
[\![\mathbf{empty}]\!]^{\#}(\mathsf{a}) \quad = \quad \mathsf{a} \sqcap \lceil 1 \notin \bullet(\mathsf{x},\mathsf{y}) \rceil \qquad\qquad [\![\mathbf{irreflexive}]\!]^{\#}(\mathsf{a}) \quad = \quad \mathsf{a} \sqcap \lceil 1 \notin \bullet(\mathsf{x},\mathsf{x}) \rceil \,.
$$

The abstract acyclicity is $[\![\mathbf{acyclic}]\!]^{\#}(\mathsf{a}) = \mathsf{a} \sqcap \lceil i_0, i_1, \ldots, i_{|\mathbb{X}|} \rceil$ where implication i_k is given by

$$
0 \notin \mathsf{a}(\mathsf{x}_0, \mathsf{x}_1) \wedge \cdots \wedge 0 \notin \mathsf{a}(\mathsf{x}_{k-1}, \mathsf{x}_k) \wedge (\forall 0 < i < k : \mathsf{x}_0 \longrightarrow \mathsf{x}_i \vee \mathsf{x}_k \longrightarrow \mathsf{x}_i) \Rightarrow 1 \notin \bullet(\mathsf{x}_k, \mathsf{x}_0) \,.
$$

Note that we require control-flow information to evaluate the abstract acyclicity constraint. The reason is that to certainly exclude an edge, we need to argue that it is part of a cycle C whenever it occurs. To guarantee the existence of C, we need to argue about both the existence of its nodes (using control-flow information) and its edges (using must-sets). To understand the implication i_k, assume that the premise was true but the conclusion was not for a concrete relation $\mathsf{r} \in \gamma_{rel}(\mathsf{a})$. Then there are executed events x_0 and x_k with $\mathsf{r}(\mathsf{x}_k, \mathsf{x}_0) = 1$. By the control-flow implications in the premise, all intermediate events x_i are also executed, so relation r must be defined on all edges $(\mathsf{x}_i, \mathsf{x}_{i+1})$, taking either value 0 or 1. However, the premise excludes value 0, so we must have $\mathsf{r}(\mathsf{x}_i, \mathsf{x}_{i+1}) = 1$ for all $0 \le i < k$. Together with $\mathsf{r}(\mathsf{x}_k, \mathsf{x}_0) = 1$ this would mean that r is cyclic, contradicting the acyclicity constraint. The abstract composition operations are defined similarly:

$$
\begin{array}{rcl}
[\![;]\!]^{\#}_{F}(\mathsf{a}, \mathsf{b}) & = & \lceil i_1, i_2 \rceil \\
i_1 & = & (\forall \mathsf{z} \in \mathbb{X} : 1 \notin \mathsf{a}(\mathsf{x},\mathsf{z}) \vee 1 \notin \mathsf{b}(\mathsf{z},\mathsf{y})) \Rightarrow 1 \notin \bullet(\mathsf{x},\mathsf{y}) \\
i_2 & = & (\exists \mathsf{z} \in \mathbb{X} : 0 \notin \mathsf{a}(\mathsf{x},\mathsf{z}) \wedge 0 \notin \mathsf{b}(\mathsf{z},\mathsf{y}) \wedge (\mathsf{x} \longrightarrow \mathsf{z} \vee \mathsf{y} \longrightarrow \mathsf{z})) \Rightarrow 0 \notin \bullet(\mathsf{x},\mathsf{y}) \\
 & & \\
[\![;]\!]^{\#}_{B}(\mathsf{a}, \mathsf{b}, \mathsf{c}) & = & \langle \mathsf{a} \sqcap \lceil j_4, j_3 \rceil, \mathsf{b} \sqcap \lceil j_2, j_1 \rceil \rangle \\
j_1 & = & 0 \notin \mathsf{c}(\mathsf{x},\mathsf{y}) \wedge (\mathsf{z} \longrightarrow \mathsf{x}) \wedge (\forall \mathsf{z}' \neq \mathsf{z} : 1 \notin \mathsf{a}(\mathsf{x},\mathsf{z}') \vee 1 \notin \mathsf{b}(\mathsf{z}',\mathsf{y})) \Rightarrow 0 \notin \bullet(\mathsf{z},\mathsf{y}) \\
j_2 & = & 1 \notin \mathsf{c}(\mathsf{x},\mathsf{y}) \wedge (\mathsf{z} \longrightarrow \mathsf{x}) \wedge 0 \notin \mathsf{a}(\mathsf{x},\mathsf{z}) \Rightarrow 1 \notin \bullet(\mathsf{z},\mathsf{y}) \\
j_3 & = & 0 \notin \mathsf{c}(\mathsf{x},\mathsf{y}) \wedge (\mathsf{z} \longrightarrow \mathsf{y}) \wedge (\forall \mathsf{z}' \neq \mathsf{z} : 1 \notin \mathsf{a}(\mathsf{x},\mathsf{z}') \vee 1 \notin \mathsf{b}(\mathsf{z}',\mathsf{y})) \Rightarrow 0 \notin \bullet(\mathsf{x},\mathsf{z}) \\
j_4 & = & 1 \notin \mathsf{c}(\mathsf{x},\mathsf{y}) \wedge (\mathsf{z} \longrightarrow \mathsf{y}) \wedge 0 \notin \mathsf{b}(\mathsf{x},\mathsf{z}) \Rightarrow 1 \notin \bullet(\mathsf{x},\mathsf{z}).
\end{array}
$$

Notice that the composition operators, like the acyclicity operator, rely on control-flow information to guarantee the existence of an intermediary event z for the relation composition. The definitions of the remaining operators can be found in the appendix.

5 OPTIMIZING THE SMT ENCODING WITH STATIC INFORMATION

We optimize the SMT encoding φ of the program semantics $[\![p]\!]_{mm}$ based on the ideas explained in Section 2. We assume that the static analysis from Section 4 has been performed and that we now have an abstract domain element D as well as may and must-sets for all relations of the memory model. We refer to this information simply as *static information* and denote it by S. The static information can be translated into a logical formula φ_S by conjoining the following set of formulas: $\mathsf{n}_{\mathsf{x},\mathsf{y}} \leftrightarrow \mathit{false}$ if $(\mathsf{x},\mathsf{y}) \notin \mathsf{may}(\mathsf{n})$, $\mathsf{n}_{\mathsf{x},\mathsf{y}} \leftrightarrow \mathsf{exec}_\mathsf{x} \wedge \mathsf{exec}_\mathsf{y}$ if $(\mathsf{x},\mathsf{y}) \in \mathsf{must}(\mathsf{n})$, and $\mathsf{exec}_\mathsf{x} \longrightarrow (\neg)\mathsf{exec}_\mathsf{y}$ if $\mathsf{x} \longrightarrow (\neg)\mathsf{y} \in \mathsf{D}$. The soundness of our analysis implies that $\varphi = \varphi_p \wedge \varphi_{mm} \wedge \varphi_{key}$ is logically equivalent to $\varphi \wedge \varphi_S$, denoted by $\varphi \equiv \varphi \wedge \varphi_S$. This equivalence allows us to perform a direct simplification by substitution: if we have $\mathsf{n}_{\mathsf{x},\mathsf{y}} \leftrightarrow \mathit{false}\,(\mathsf{exec}_\mathsf{x} \wedge \mathsf{exec}_\mathsf{y})$, then we can replace all occurrences of the left-hand side by the right-hand side. These substitution may make some clauses trivially true, for example, consider `let` c = a ∩ b and assume that $(\mathsf{x},\mathsf{y}) \notin \mathsf{may}(\mathsf{a})$ and $(\mathsf{x},\mathsf{y}) \notin \mathsf{may}(\mathsf{c})$. Then our encoding contains a subformula $\mathsf{c}_{\mathsf{x},\mathsf{y}} \leftrightarrow \mathsf{a}_{\mathsf{x},\mathsf{y}} \wedge \mathsf{b}_{\mathsf{x},\mathsf{y}}$ which, after substitution, becomes $\mathit{false} \leftrightarrow \mathit{false} \wedge \mathsf{b}_{\mathsf{x},\mathsf{y}} \equiv \mathit{false} \leftrightarrow \mathit{false} \equiv \mathit{true}$. We remark that in practice, we do not perform the substitutions explicitly and instead generate the formula directly in simplified form.

Before addressing the more involved optimizations presented in Section 2, we define the helpful notion of relativized logical equivalence. We write $\varphi \equiv_S \psi$ to mean $\varphi \wedge \varphi_S \equiv \psi \wedge \varphi_S$, that is, both formulas are logically equivalent modulo the given static information. Similarly, we write $\varphi \equiv_{S,key} \psi$ when we also want to take the key implication φ_{key} into account. We also use this relativized notation for logical implications $\varphi \Rightarrow_S \psi$, meaning that $\varphi \wedge \varphi_S \Rightarrow \psi$ holds.

5.1 Optimizing the Acyclicity Encoding

Among the three types of axioms in a memory model, acyclicity is the most expensive to handle for the SMT solver. We now optimize the encoding of an **acyclic**(n) axiom based on the given static information. We make precise what it means to optimize the acyclicity encoding. We associate with relation n the edge-labeled, complete graph $G = (\mathbb{X}, \mathbb{X} \times \mathbb{X}, \lambda(\mathsf{x},\mathsf{y}).\mathsf{n}_{\mathsf{x},\mathsf{y}})$. The nodes are all events, we have all possible edges, and each edge (x,y) is labeled by the Boolean variable $\mathsf{n}_{\mathsf{x},\mathsf{y}}$. When given an assignment σ of the variables $\mathsf{n}_{\mathsf{x},\mathsf{y}}$, we obtain the subgraph $\sigma(G)$ consisting of only those edges where $\sigma(\mathsf{n}_{\mathsf{x},\mathsf{y}}) = \mathit{true}$. Encoding the acyclicity of n then amounts to finding a formula φ_G so that for all assignments σ we have

$$\sigma \models \varphi_G \text{ iff } \sigma(G) \text{ is acyclic.}$$

The satisfying assignments of the formula are precisely the ones that make the graph acyclic.

There are different ways to construct φ_G. We implemented two options. The first introduces integer-valued clock variables $\mathsf{clk}_{\mathsf{n},\mathsf{x}}$ for each event x and requires a strict ordering whenever two events are linked by an edge: $\forall \mathsf{x},\mathsf{y} \in \mathbb{X} : \mathsf{n}_{\mathsf{x},\mathsf{y}} \longrightarrow \mathsf{clk}_{\mathsf{n},\mathsf{x}} < \mathsf{clk}_{\mathsf{n},\mathsf{y}}$. We also have a more subtle Boolean encoding based on vertex elimination [Rankooh and Rintanen 2022]. Importantly, the optimization we give in the following is based on the graph representation and works for any encoding.

Our optimization determines a smaller graph H that is equivalent to G as far as acyclicity is concerned and when taking into account both the static information S and the key implication (1). We will guarantee that $\varphi_H \equiv_{S,key} \varphi_G$, the acyclicity formulas of both graphs are logically equivalent when conjoined with the static information φ_S and the key implication φ_{key}. We construct H as the result of a sequence G, G_1, G_2, H of graphs. While both G_1 and G_2 will be decreasing subgraphs of G, the final graph H may in general not be.

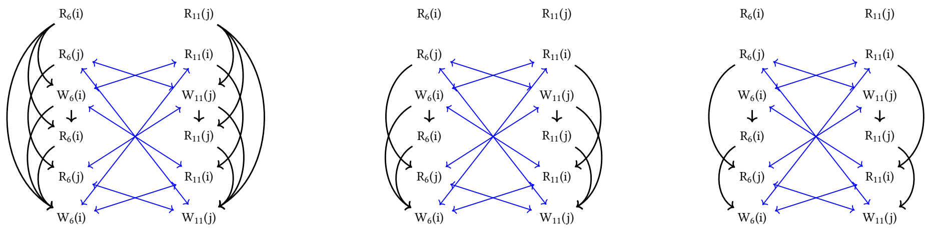

Fig. 11. Graphs G_1 (left), G_2 (middle), and H (right) for relation $\mathsf{n} = (\mathsf{po} \cap \mathsf{loc}) \cup \mathsf{ca} \cup \mathsf{rf}$ and axiom $\mathbf{acyclic}(\mathsf{n})$. Static edges are depicted in black. The graph G is the complete graph and is omitted.

If we apply this three-step construction to the axiom $\mathbf{acyclic}((\mathsf{po} \cap \mathsf{loc}) \cup \mathsf{ca} \cup \mathsf{rf})$ of our running example, we get the three graphs G_1, G_2, and H as depicted in Figure 11.

Step 1: Restriction by static information. We remove from G the edges that are statically known to be absent. Let G_1 be the restriction of G to the edges $(\mathsf{x}, \mathsf{y}) \in \mathsf{may}(\mathsf{n})$.

Step 2: Restriction to strongly connected components (SCCs). An edge can only be part of a cycle if it belongs to an SCC. We compute all SCCs in G_1 using Tarjan's algorithm [Tarjan 1971] and remove all edges that are not part of an SCC. Let the result be G_2.

Step 3: Transitive reduction. In the last step, the idea is to remove from G_2 edges (x, y) that are shortcuts in the following sense: whenever $\sigma(G_2)$ contains the edge, then it also contains a path from x to y. Removing shortcuts is sound indeed: whenever $\sigma(G_2)$ has a cycle containing the edge (x, y), then it also has a cycle that replaces the edge by the path.

To compute the shortcut edges (x, y), note that the requirement of having a path from x to y resembles the condition that the edge is transitively implied, meaning it is part of a transitive closure formed from other edges. We thus intend to retain what resembles a transitive reduction, a set of edges the transitive closure of which contains all edges. However, the paths we are looking for have to satisfy the extra condition that they exist whenever the edge exists. Consider a graph over three events $\mathsf{x}, \mathsf{y}, \mathsf{z}$ with edges (x, y), (y, z), and (x, z). Naively, one would say that $(\mathsf{x}, \mathsf{y}); (\mathsf{y}, \mathsf{z})$ is a long path for (x, z), however, for this to hold true we also have to have the following implication between their edge-labels: $\mathsf{n}_{\mathsf{x},\mathsf{z}} \Rightarrow \mathsf{n}_{\mathsf{x},\mathsf{y}} \wedge \mathsf{n}_{\mathsf{y},\mathsf{z}}$. In general, we cannot establish this implication between arbitrary edges because we lack the relational information needed between the different relation variables. However, we can use must-sets as well as our key implication to relate the relation variables to execution variables. Then we exploit the relational control-flow implications to show the necessary implications between those. In the above example, assume that we know $(\mathsf{x}, \mathsf{y}), (\mathsf{y}, \mathsf{z}) \in \mathsf{must}(\mathsf{n})$ and $\mathsf{x} \longrightarrow \mathsf{y}$. Then we can establish the necessary implication as follows:

$$\mathsf{n}_{\mathsf{x},\mathsf{z}} \Rightarrow_{key} \mathsf{exec}_{\mathsf{x}} \wedge \mathsf{exec}_{\mathsf{z}} \Rightarrow_{S} \mathsf{exec}_{\mathsf{x}} \wedge \mathsf{exec}_{\mathsf{y}} \wedge \mathsf{exec}_{\mathsf{z}} \Rightarrow_{S} \mathsf{n}_{\mathsf{x},\mathsf{y}} \wedge \mathsf{n}_{\mathsf{y},\mathsf{z}}.$$

We can incorporate this extra requirement into a stronger definition of edge composition $;_S$ so that we indeed have $(\mathsf{x}, \mathsf{y}) ;_S (\mathsf{y}, \mathsf{z}) = (\mathsf{x}, \mathsf{z})$ only if the necessary implication holds. With respect to this new composition, the notions of transitive closure and transitive reduction apply without modification. In order to define these notions, we need to allow for a slightly more generalized edge-labeling λ that labels an edge with either a relation variable $\mathsf{n}_{\mathsf{x},\mathsf{y}}$ or with $\mathsf{exec}_{\mathsf{x}} \wedge \mathsf{exec}_{\mathsf{y}}$. Call an edge (x, z) *S-static* if it is labeled by $\mathsf{exec}_{\mathsf{x}} \wedge \mathsf{exec}_{\mathsf{z}}$ or if this labeling can be obtained from S, i.e.,

$(\mathsf{x}, \mathsf{z}) \in \mathsf{must}(\mathsf{n})$. We define the *$S$-static composition* $;_S$ of S-static edges as the partial operation

$$(\mathsf{x}, \mathsf{z}) \,;_S (\mathsf{z}, \mathsf{y}) \;=\; \begin{cases} (\mathsf{x}, \mathsf{y}) & \text{if } \mathsf{x} \rightarrow \mathsf{z} \in S \text{ or } \mathsf{y} \rightarrow \mathsf{z} \in S \\ \mathit{undef} & \text{otherwise.} \end{cases}$$

We can only compose S-static edges (x, z) and (z, y), if the result (x, y) can be guaranteed to be again S-static. As it turns out. this definition closely resembles the abstract forward semantics $[\![;]\!]^{\#}_F$ we have defined in Section 4.3. Given a set E of S-static edges, we use $E^{+s} = \bigcup_{i \geq 1} E^i$ with $E^1 = E$ and $E^{i+1} = E \cup E^i \,;_S E^i$ for $i \geq 1$ to denote the transitive closure w.r.t. S-static composition. We extend this notion to graphs $K = (\mathbb{X}, E, \lambda)$ with extended labeling in the following way: if $E_S \subseteq E$ denotes the subset of S-static edges, then we define the S-static transitive closure of K to be $K^{+s} = (\mathbb{X}, (E \setminus E_S) \cup E_S^{+s}, \lambda^{+s})$ where λ^{+s} labels $(\mathsf{x}, \mathsf{y}) \in E_S^{+s}$ with $\mathsf{exec}_\mathsf{x} \wedge \mathsf{exec}_\mathsf{y}$ and matches λ on all other edges. The graph K and its transitive closure K^{+s} are related by the following lemma.

Lemma 5.1. *For any given S, we have $\varphi_K \equiv_{S,key} \varphi_{K^{+s}}$.*

The lemma allows us to replace G_2 by a different graph H as long as both graphs have the same transitive closure, i.e., we have $G_2^{+s} = H^{+s}$. Notice that H does not have to be a subgraph of G_2 for the lemma to apply, so we are not only allowed to delete edges from G_2 but we can also introduce new ones. Ideally, we want H to be of minimal size, in which case it is also known as a transitive reduction of G_2. Before we address the computation of H, we state our main theorem.

Theorem 5.2. *For any given S, we have $\varphi_G \equiv_{S,key} \varphi_H$.*

In order to formulate how H is constructed from G_2, we describe the computation of the transitive reduction on a general graph K. We first consider the transitive reduction of a set E of S-static edges and then lift this definition to the level of graphs. For our computation, we additionally require E to be acyclic. While this is a theoretical restriction, in practice we do not encounter the cyclic case: a static cycle involving events $\mathsf{X} \subseteq \mathbb{X}$ would imply that all program executions that include X are always deemed inconsistent by the memory consistency model. The computation of the transitive reduction is given by the following lemma.

Lemma 5.3. *Let E be an acyclic set of static edges, then $E^{-s} = E^{+s} \setminus (E^{+s} \,;_S E^{+s})$ is a transitive reduction of E.*

We lift this transitive reduction to the level of graphs. Let K be an edge-labeled graph and let $K^{+s} = (\mathbb{X}, E, \lambda)$ be its transitive closure. Let $E_S \subseteq E$ be the subset of S-static edges which we assume to be acyclic. We then define the transitive reduction of K to be $K^{-s} = (\mathbb{X}, (E \setminus E_S) \cup E_S^{-s}, \lambda)$. With this definition at hand, we can now set $H = G_2^{-s}$ to complete Step 3. of our construction

5.2 Thinning

The idea of the thinning optimization is to remove from the SMT encoding constraints that define a relation variable which does not influence (directly or indirectly) non-defining constraints of the encoding. By non-defining constraints, we mean constraints imposed by the axioms of the memory model as well as constraints imposed by the program (e.g., constraints on the base relations). The computation is driven by these non-defining constraints and the static information S will play an important role. Say we have the axiom **empty**$(\mathsf{n} \cap \mathsf{m})$ and S tells us that some entries of m are false, i.e., are outside of the may-set. Then we do not need to know the corresponding entries of n to decide emptiness. Their defining constraints can be dropped from φ_{def}.

Thinning has to work on the full SMT encoding, including the program encoding, that has already been optimized. The optimized version will still have the shape $\varphi_{S,p} \wedge \varphi_{S,mm} \wedge \varphi_{S,key}$ with

$\varphi_{S,mm} = \varphi_{S,def} \land \varphi_{S,axiom}$. Moreover, the optimizations guarantee that $\varphi_{S,def}$ is a conjunction of equivalences of one of the following forms:

$$\mathsf{n}_{\mathsf{x,y}} \leftrightarrow \mathsf{def}_S(\mathsf{n},\mathsf{x},\mathsf{y}) \qquad \mathit{false} \leftrightarrow \mathsf{def}_S(\mathsf{n},\mathsf{x},\mathsf{y}) \qquad \mathsf{exec}_\mathsf{x} \land \mathsf{exec}_\mathsf{y} \leftrightarrow \mathsf{def}_S(\mathsf{n},\mathsf{x},\mathsf{y}).$$

Here, $\mathsf{def}_S(\mathsf{n},\mathsf{x},\mathsf{y})$ is a formula (resulting from substitution) that we do not need to specify further. Importantly, $\mathsf{n}_{\mathsf{x,y}}$ appears at most once on a left-hand side of an equivalence in $\varphi_{S,def}$. We can therefore understand the equivalences as a simplified subset of the defining constraints $\mathbb{C}$ in φ_{def}.

The goal of thinning is to identify a subset of the defining constraints that can be removed from the encoding without affecting satisfiability. We call those constraints *inactive*. Algorithmically, we proceed the other way around and compute a set $\mathsf{active} \subseteq \mathbb{C}$ of constraints that cannot be removed. The complement $\mathsf{inactive} = \mathbb{C} \setminus \mathsf{active}$ are then the inactive constraints. Actually, we do not remove just the inactive defining constraint of a relation variable $\mathsf{n}_{\mathsf{x,y}}$ but also its associated key implication $\mathsf{n}_{\mathsf{x,y}} \to \mathsf{exec}_\mathsf{x} \land \mathsf{exec}_\mathsf{y}$ from $\varphi_{S,key}$.

The computation of the active constraints is by saturation. Given a set of constraints that have already been identified as active, we determine the relation variables contained in them and declare the corresponding defining constraints also active. We thus make use of a helper set $\mathsf{relevant} \subseteq \mathsf{RelationVars}(\varphi_{S,mm})$ of so-called *relevant variables*. Then, the sets of relevant variables and active constraints are the least solution to the following equation system

$$\begin{aligned} \mathsf{relevant} &= \mathsf{relevant}_0 \cup \{\mathsf{n}_{\mathsf{x,y}} \mid \exists c \in \mathsf{active} : \mathsf{n}_{\mathsf{x,y}} \in \mathsf{RelationVars}(c)\} \\ \mathsf{active} &= \mathsf{active}_0 \cup \{\mathsf{n}_{\mathsf{x,y}} \leftrightarrow \mathsf{def}_S(\mathsf{n},\mathsf{x},\mathsf{y}) \mid \mathsf{n}_{\mathsf{x,y}} \in \mathsf{relevant}\}. \end{aligned}$$

The saturation is initialized with

$$\begin{aligned} \mathsf{relevant}_0 &= \mathsf{RelationVars}(\varphi_{S,axiom}) \cup \mathsf{RelationVars}(\varphi_{S,p}) \\ \mathsf{active}_0 &= \{c \in \mathbb{C} \mid c \text{ is } \mathit{false} \leftrightarrow \mathsf{def}_S(\mathsf{n},\mathsf{x},\mathsf{y}) \text{ or } \mathsf{exec}_\mathsf{x} \land \mathsf{exec}_\mathsf{y} \leftrightarrow \mathsf{def}_S(\mathsf{n},\mathsf{x},\mathsf{y})\}. \end{aligned}$$

Initially, all relation variables that appear in either the axioms or the program are relevant, corresponding to the idea that those variables are important to decide whether an execution is consistent with the memory model and whether it belongs to the program. The program formula $\varphi_{S,p}$ usually refers to precisely the base relations, meaning that initially all base relations as well as the relations of axioms are relevant, but any derived relation in between is irrelevant. Moreover, we declare active all equivalences where the left-hand side is not a relation variable, i.e., all equivalences of the form $\mathit{false} \leftrightarrow \mathsf{def}_S(\mathsf{n},\mathsf{x},\mathsf{y})$ and $\mathsf{exec}_\mathsf{x} \land \mathsf{exec}_\mathsf{y} \leftrightarrow \mathsf{def}_S(\mathsf{n},\mathsf{x},\mathsf{y})$. These constraints were obtained by substituting in known values without eliminating the defining constraint. The fact that the constraints were not eliminated means that the static information we have computed top-down from the axioms was enough to establish the left-hand side, while the information computed bottom-up from the base relations was not precise enough to resolve the right-hand side $\mathsf{def}_S(\mathsf{n},\mathsf{x},\mathsf{y})$. The constraints thus reflect a gap in the static information coming from above and from below. This gap has an influence on the semantics of the memory model and thus has to be encoded.

Let active denote the set of active constraints in the least solution to the equation system, and let $\mathsf{inactive} = \mathbb{C} \setminus \mathsf{active}$ be the set of inactive constraints. We denote by $\varphi^{red}_{S,mm}$ the formula that is obtained by dropping all inactive constraints from the subformula $\varphi_{S,def}$. Similarly, we let $\varphi^{red}_{S,key}$ be the formula obtained by dropping all implications whose relation variable is defined by an inactive constraint. Replacing $\varphi_{S,mm}$ and $\varphi_{\mathsf{stat},key}$ by their reduced forms gives us an equisatisfiable encoding.

Theorem 5.4. $\varphi_{S,p} \land \varphi_{S,mm} \land \varphi_{S,key} \approx \varphi_{S,p} \land \varphi^{red}_{S,mm} \land \varphi^{red}_{S,key}$.

6 EVALUATION

We evaluate how the static analysis introduced in Section 4 simplifies the SMT encoding for memory models as described in Section 5. We address the following research questions:

(RQ1) What is the reduction in the SMT encoding size by using static information and how does this compare with the previous static analysis for memory models [Gavrilenko et al. 2019]?
(RQ2) What is the impact of using smaller encodings on verification time?
(RQ3) Does static information also improve the verification time of SMT theories tailored to memory models [Haas et al. 2022]?

It is not a-priori clear that smaller encodings are better. We designed the evaluation to assess exactly this. We implemented our static analysis on top of Dartagnan which already implemented the encoding reduction described in [Gavrilenko et al. 2019]. It also supports a recently developed SMT theory solver for memory consistency [Haas et al. 2022]. Below, we briefly describe these techniques and how they relate to and can be combined with our work. Note that, while a comparison to dynamic approaches like stateless model checking is interesting (and done in [Haas et al. 2022]), it cannot be used to judge the presented techniques (which target bounded model checkers).

Relation analysis [Gavrilenko et al. 2019] is a static analysis for determining the pairs of events that may influence axioms of the memory model. Any remaining pair can be dropped from the encoding. It can thus be understood as a form of thinning that is weaker than the present development as we will explain. Relation analysis makes use of may-sets (but no must-sets) and a simplified notion of active sets. The resulting information is used to remove a defining constraint if both sides are known and coincide. Substitution, as we propose it, is not performed. Our notion is more general and allows us to replace the left-hand side in isolation (if the value is known) by making use of top-down propagation. *Consistency as a Theory* [Haas et al. 2022] proposes a family of logical theories for capturing the consistency requirements of memory models. The theories admit a generic and efficient solver that works for TSO, Power, ARMv8, RISC-V, RC11, IMM, and LKMM. For the latter, however, it cannot handle the definition of data races. This verification approach encodes $[\![p]\!]$ using traditional theories and uses the specific theory solver to reason about $[\![mm]\!]$. This approach can be combined with the static information we compute about base relations to reduce the size of φ_p.

Our benchmark set contains programs that have been previously used to compare the performance of modular verification tools [Haas et al. 2022; Kokologiannakis et al. 2019; Oberhauser et al. 2021]. It includes non-trivial (thousands of executions) implementations of lock primitives (`CLH, CNA, Mutex, Musl mutex, RWLock, Spinlock, Ticketlock, Ticketlock with array waiting nodes, TTAS`) and lock-free data structures (`Chase-Lev, DGLM, Michael-Scott, Treiber`) using C11 atomics. We use common compiler mappings [Sevcik and Sewell 2011] to convert those atomics into the assembly instructions that memory models understand. Due to the incompatibility between C11 atomics and LKMM [Corbet 2013a], we only analyze these benchmarks w.r.t. ARMv8 and RISC-V. To include LKMM in the evaluation, we further use the version of `qspinlock` from [Paolillo et al. 2021] which has Linux kernel atomics. Dartagnan compiles those to ARMv8 and RISC-V using the inline assembly kernel implementation of atomics[2]. Each program contains 3 to 6 threads (excluding the main thread). All loops were unrolled[3] twice, except spin loops which need to be unrolled only once (Dartagnan automatically detects this). After unrolling, the resulting programs contain between 80 and 250 memory accesses (not counting memory initialization) and fences.

[2]Located in the kernel source tree in `arch/<target-arch>/include/asm/atomic.h`
[3]Unrolled N times means that the body of the loop appears N times in the unrolled program.

Table 1. Impact of the optimizations on different BMC approaches.

BMC APPROACH	MAY	MUST	UNKN.	ACTIVE	ACYC. CON.	VERIF. TIME	SAF.	LIV.	D.R.F.
				ARMv8					
CAV19	2843320	0	2843320	397067	93214	00hs 30min 28s	✔	✔	*N/A*
THINN. + SUBST.	2802649	2539540	263109	195227	83598	00hs 26min 40s	✔	✔	*N/A*
THINN. + SUBST. + ACYC.	2802649	2539540	263109	194837	68817	00hs 19min 22s	✔	✔	*N/A*
OOPSLA22 + CAV19	26594	0	26594	26594	0	00hs 12min 23s	✔	✔	*N/A*
OOPSLA22 + FLATT.	23778	1528	22250	22250	0	00hs 08min 27s	✔	✔	*N/A*
				RISC-V					
CAV19	3218051	0	3218051	386904	91614	00hs 15min 50s	✔	✔	*N/A*
THINN. + SUBST.	3196006	2921072	274934	150427	84474	00hs 19min 42s	✔	✔	*N/A*
THINN. + SUBST. + ACYC.	3196006	2921072	274934	150427	70504	00hs 13min 16s	✔	✔	*N/A*
OOPSLA22 + CAV19	26594	0	26594	26594	0	00hs 10min 21s	✔	✔	*N/A*
OOPSLA22 + FLATT.	23794	1528	22266	22266	0	00hs 07min 05s	✔	✔	*N/A*
				LKMMv6.3					
CAV19	2980569	0	2980569	733090	64242	02hs 46min 10s	✔	✔	✔
THINN. + SUBST.	2924423	910682	2013741	563966	58448	01hs 31min 59s	✔	✔	✔
THINN. + SUBST. + ACYC.	2924423	910682	2013741	563966	55472	01hs 37min 14s	✔	✔	✔
OOPSLA22 + CAV19	8094	0	8094	8094	0	00hs 03min 10s	✔	✔	**?**
OOPSLA22 + FLATT.	7318	632	6686	6686	0	00hs 02min 27s	✔	✔	**?**

The results of our evaluation are given in Table 1. We refer to the verification approach from [Gavrilenko et al. 2019] as CAV19, the second uses THINNING, and the 3rd additionally the ACYCLICITY optimization. We emphasize that without any static information (e.g., at least the one computed by CAV19), the verification time to solve the eager encoding is intractable for all the given benchmarks. Since the approach described in [Haas et al. 2022] used relation analysis, we refer to it as OOPSLA22 + CAV19. The approach encodes only the base relations and uses a dedicated theory solver to reason about the derived relations and the axioms. Therefore, neither THINNING nor our improved ACYCLICITY encoding apply. However, we can still substitute known values for the base relations. Columns MAY and MUST show the number of pairs in the may and must-sets (summed over all relations and all benchmarks). UNKNOWN is the difference between the previous two columns. The number of ACTIVE constraints (the subset of UNKNOWN for which we introduce Boolean variables) is given in the 4th column. The 5th column reports the number of implications $n_{x,y} \rightarrow clk_{n,x} < clk_{n,y}$ encoding acyclicity axioms. The overall VERIFICATION TIME reports the sum of the static analysis, encoding, and SMT solving times. DARTAGNAN uses JAVASMT [Baier et al. 2021; Karpenkov et al. 2016], a library that supports several SMT solvers, to discharge the verification problem. The table reports the times obtained with YICES2. We have tried other solvers (Z3 and MATHSAT5) and derived similar results. The right-most part of the table shows the correctness conditions that have been checked. We checked SAFETY in the form of assertions (e.g., guaranteeing mutual exclusion), LIVENESS as described in [Lahav et al. 2021], and DATA RACE FREEDOM. The notion of data race only makes sense for programming languages and thus does not apply (N/A) to hardware memory models. For LKMM we used the notion from [Alglave et al. 2018]. As previously stated, the lazy encoding from [Haas et al. 2022] does not encode the derived happens-before relation, and thus does not handle data races (**?** in the table).

To answer (RQ1), we focus on the columns MAY, MUST, ACTIVE, ACYCLICITY CONSTRAINTS, and the first three rows of each memory model. For the rest of the paragraph, we take CAV19 as the baseline. Our static analysis provides more precise may-sets. While CAV19 only has forward propagation, we also propagate information from relations back to their definitions, thanks to filters. The effect is amplified by must-sets (which Cav19 lacks). Both, may and must-sets, justify more aggressive reductions of the active constraints, which directly influence the number of Boolean variables in the encoding. Our analysis reduces the number of Boolean variables between 23% (LKMM) and 62% (RISC-V). While Cav19 already removes acyclicity constraints $\mathit{false} \rightarrow \mathsf{clk}_{\mathsf{n,x}} < \mathsf{clk}_{\mathsf{n,y}}$, our stronger static analysis leads to more simplifications of the same form. The further improvements in Thinn. + Subst. + Acyc. are due to the improved acyclicity encoding. Overall, our analysis removes 26%, 23%, and 14% of the acyclicity constraints in ARMv8, RISC-V, and LKMM, respectively.

The column Verification Time answers (RQ2). The encoding optimizations reduce the verification times of the eager approach w.r.t. Cav19 by 36%, 16%, and 41% for ARMv8, RISC-V, and LKMM, respectively. Interestingly without using the improved acyclicity encoding, our analysis performs worse than Cav19 for RISC-V. On the other hand, for LKMM we get the best results by just applying Thinning without the acyclicity optimization. In either case, applying all proposed encoding optimizations gives consistently better results than the Cav19 encoding.

To answer (RQ3), we focus on the last two rows of each memory model. The SMT encoding of Oopsla22 + Cav19 only refers to base relations. The interpretation of other relations is derived from the solution of the SMT solver for φ_p. Because of this, the encoding is much simpler than Cav19 and there is less room for optimizations compared to the eager encoding. Nevertheless, the results show a benefit from the static analysis comparable to the eager encoding.

In general, we observe that the optimizations are weaker the more the derived relations depend on dynamic base relations (like rf and co, as opposed to static base relations like po). Dynamic information makes it harder for our analysis to compute information, a fact that is amplified by complex dependencies (long chains) in a memory model. As a concrete example, the static po relation plays an important role in the happens-before relation of sequential consistency (defined as `let` hb = po ∪ rf ∪ co ∪ fr). Because of this, our analysis propagates a lot of information from po to hb. Weaker models typically define the preserved program order relation ppo ⊆ po which is then used to define hb-like relations. Since the analysis can derive less information from ppo than from po (from the simple fact that typically it contains fewer pairs of events), there is less information to propagate from ppo to the derived relations depending on it. Besides a weaker notion of program order to formulate happens-before, models like Power, LKMM, and ARMv7 also make use of an additional propagation relation to express their non-multicopy atomicity [Mador-Haim et al. 2012; Pulte et al. 2018]. This propagation relation tends to be both larger in size and less static than hb which makes it harder to analyze statically. This explains why we observe for LKMM a smaller reduction in both the Active constraints and the Acyclicity constraints compared to the multicopy-atomic hardware memory models.

7 FUTURE WORK

We have given a new filter-based semantics for memory models defined in the CAT language. An important feature of this semantics is its flexibility to incorporate information from external sources (via filters). So far, we have used this opportunity only to incorporate program information. However, we believe there are further interesting applications.

Exploiting symmetries in the verification task is known to be important for scalability [Emerson and Sistla 1993]. Consider identical threads T_1 and T_2 that try to enter their critical sections CS_i. For verification, it may be sufficient to assume that T_1 will go first. A *symmetry filter* could filter out all executions where T_1 is not first. Its abstract version would give us the must information that all events in CS_1 are guaranteed to be happens-before related to all events in CS_2.

We see this forced happens-before ordering as a *scheduling constraint* [Metzler et al. 2019]. We plan to use scheduling constraints to parallelize the verification [Nguyen et al. 2017]. We split the task into *scheduled* subtasks, analyze and simplify these subtasks using a *scheduling filter*, and then solve the simpler subtasks in parallel.

Another use case is to incorporate information about the correctness condition. In bounded model checking, we are only interested in executions that violate correctness. Suppose correctness is formulated as ASSERT (x != 42). An *assertion filter* could filter out executions where the assertion holds. Using this information in the abstract is not immediate, though: the assertion restricts the value of x, but abstract filters restrict relations. A data-flow analysis is needed to bridge the gap.

8 DATA AVAILABILITY STATEMENT

The complete benchmark set as well as the code to generate Table 1 of Section 6 are provided in the accompanying artifact [Haas et al. 2023]. The DARTAGNAN tool used in the evaluation can be found at https://github.com/hernanponcedeleon/Dat3M.

REFERENCES

2023. C11: Bad Locking and Races. https://github.com/herd/herdtools7/blob/master/herd/libdir/c11_orig.cat#L17-L20. Accessed: 08/12/2023.

2023. KCSAN. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

2023a. Linux Memory Model: Locks. https://github.com/torvalds/linux/blob/master/tools/memory-model/lock.cat#L39-L58. Accessed: 08/12/2023.

2023b. Linux Memory Model: RCU. https://github.com/torvalds/linux/blob/master/tools/memory-model/linux-kernel.bell#L56-L73. Accessed: 08/12/2023.

2023c. Linux Memory Model: Sanitization. https://lkml.org/lkml/2023/1/18/575. Accessed: 08/12/2023.

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, Mohamed Faouzi Atig, Ahmed Bouajjani, Egor Derevenetc, Carl Leonardsson, and Roland Meyer. 2020. On the State Reachability Problem for Concurrent Programs Under Power. In *NETYS (Lecture Notes in Computer Science, Vol. 12129)*. Springer, 47–59. https://doi.org/10.1007/978-3-030-67087-0_4

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Adwait Godbole, Shankara Narayanan Krishna, and Viktor Vafeiadis. 2021. The Decidability of Verification under PS 2.0. In *ESOP (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 1–29. https://doi.org/10.1007/978-3-030-72019-3_1

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

S.V. Adve and M.D. Hill. 1990. Weak ordering-a new definition. In *ISCA*. 2–14. https://doi.org/10.1109/ISCA.1990.134502

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

Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: an invariance proof method for weak consistency models. In *POPL*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 3–18. https://doi.org/10.1145/3009837.3009883

Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 8:1–8:54. https://doi.org/10.1145/3458926

Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2014a. Don't Sit on the Fence - A Static Analysis Approach to Automatic Fence Insertion. In *CAV (LNCS, Vol. 8559)*. Springer, 508–524. https://doi.org/10.1145/2994593

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, and Michael Tautschnig. 2014b. 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

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

Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2012. What's Decidable about Weak Memory Models?. In *ESOP (LNCS, Vol. 7211)*. Springer, 26–46. https://doi.org/10.1007/978-3-642-28869-2_2

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, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,* *March 24-26, 1986, Cambridge, Massachusetts, USA*, Avi Silberschatz (Ed.). ACM, 1–15. https://doi.org/10.1145/6012.15399

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

Martin Beck, Koustubha Bhat, Lazar Stricevic, Geng Chen, Diogo Behrens, Ming Fu, Viktor Vafeiadis, Haibo Chen, and Hermann Härtig. 2023. AtoMig: Automatically Migrating Millions Lines of Code from TSO to WMM. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 61–73. https://doi.org/10.1145/3575693.3579849

John Bender and Jens Palsberg. 2019. A formalization of Java's concurrent access modes. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 142:1–142:28. https://doi.org/10.1145/3360568

Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. 2012. Conditional model checking: a technique to pass information between verifiers. In *FSE*, Will Tracz, Martin P. Robillard, and Tevfik Bultan (Eds.). ACM, 57. https://doi.org/10.1145/2393596.2393664

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2006. Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models: A Case Study. In *CAV (Lecture Notes in Computer Science, Vol. 4144)*, Thomas Ball and Robert B. Jones (Eds.). Springer, 489–502. https://doi.org/10.1007/11817963_45

Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. 2007. CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In *PLDI*. ACM, 12–21. https://doi.org/10.1145/1250734.1250737

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

S. Ceri, G. Gottlob, and L. Tanca. 1989. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering* 1, 1 (1989), 146–166. https://doi.org/10.1109/69.43410

Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 93–107. https://doi.org/10.1007/978-3-642-36742-7_7

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

Jonathan Corbet. 2008. Ticket spinlocks. https://lwn.net/Articles/267968/.

Jonathan Corbet. 2013a. C11 atomic variables and the kernel. https://lwn.net/Articles/586838/.

Jonathan Corbet. 2013b. Improving ticket spinlocks. https://lwn.net/Articles/531254/.

Jonathan Corbet. 2014. MCS locks and qspinlocks. https://lwn.net/Articles/590243/.

P. Cousot. 2021. *Principles of abstract interpretation.* MIT Press.

Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation and Application to Logic Programs. *J. Log. Program.* 13, 2&3 (1992), 103–179. https://doi.org/10.1016/0743-1066(92)90030-7

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

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS, Vol. 4963)*. Springer, 337–340.

Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed Stateless Model Checking for SC and TSO. In *OOPSLA*. 20–36. https://doi.org/10.1145/2858965.2814297

Simon Doherty, Sadegh Dalvandi, Brijesh Dongol, and Heike Wehrheim. 2022. Unifying Operational Weak Memory Verification: An Axiomatic Approach. *ACM Trans. Comput. Log.* 23, 4 (2022), 27:1–27:39. https://doi.org/10.1145/3545117

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

E. Allen Emerson and A. Prasad Sistla. 1993. Symmetry and Model Checking. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV '93)*. Springer-Verlag, Berlin, Heidelberg, 463–478. https://doi.org/10.1007/BF00625970

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

Martin Gebser, Tomi Janhunen, and Jussi Rintanen. 2014. SAT Modulo Graphs: Acyclicity. In *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8761)*, Eduardo Fermé and João Leite (Eds.). Springer, 137–151. https://doi.org/10.1007/978-3-319-11558-0_10

Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* Lecture Notes in Computer Science, Vol. 1032. Springer. https://doi.org/10.1007/3-540-60761-7

Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2020. Satune: Synthesizing Efficient SAT Encoders. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 146 (nov 2020), 32 pages. https://doi.org/10.1145/3428214

Thomas Haas, René Maseli, Roland Meyer, and Hernan Ponce de Leon. 2023. *Static Analysis of Memory Models for SMT Encodings (Artifact).* https://doi.org/10.5281/zenodo.8313104

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

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

Daniel Jackson. 2003. Alloy: A Logical Modelling Language. In *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2651)*, Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén (Eds.). 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

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, Ori Lahav, and Viktor Vafeiadis. 2023. Kater: Automating Weak Memory Model Metatheory and Consistency Checking. *Proc. ACM Program. Lang.* 7, POPL (2023), 544–572. https://doi.org/10.1145/3571212

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

Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model Checking for Hardware Memory Models. In *ASPLOS 2020*. ACM, 1157–1171. https://doi.org/10.1145/3373376.3378480

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

Shankara Narayanan Krishna, Adwait Godbole, Roland Meyer, and Soham Chakraborty. 2022. Parameterized Verification under Release Acquire is PSPACE-complete. In *PODC*, Alessia Milani and Philipp Woelfel (Eds.). ACM. https://doi.org/10.1145/3519270.3538445

Michael Kuperstein, Martin T. Vechev, and Eran Yahav. 2011. Partial-coherence abstractions for relaxed memory models. In *PLDI*, Mary W. Hall and David A. Padua (Eds.). ACM, 187–198. https://doi.org/10.1145/1993498.1993521

Ori Lahav and Udi Boker. 2020. Decidable verification under a causally consistent shared memory. In *PLDI*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 211–226. https://doi.org/10.1145/3385412.3385966

Ori Lahav and Udi Boker. 2022. What's Decidable About Causally Consistent Shared Memory? *ACM Trans. Program. Lang. Syst.* 44, 2 (2022), 8:1–8:55. https://doi.org/10.1145/3505273

Ori Lahav, Egor Namakonov, Jonas Oberhauser, Anton Podkopaev, and Viktor Vafeiadis. 2021. Making weak memory models fair. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485475

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

Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 630–646. https://doi.org/10.1145/3445814.3446711

Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. 2010. Generating Litmus Tests for Contrasting Memory Consistency Models. In *CAV (LNCS, Vol. 6174)*. Springer, 273–287. https://doi.org/10.1007/978-3-642-14295-6_26

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 (LNCS, Vol. 7358)*. Springer, 495–512. https://doi.org/10.1007/978-3-642-31424-7_36

Jeremy Manson, William Pugh, and Sarita V. Adve. 2006. The Java memory model. In *POPL*. ACM, 378–391. https://doi.org/10.1145/1047659.1040336

Patrick Metzler, Neeraj Suri, and Georg Weissenbacher. 2019. Extracting Safe Thread Schedules from Incomplete Model Checking Results. In *SPIN (Lecture Notes in Computer Science, Vol. 11636)*, Fabrizio Biondi, Thomas Given-Wilson, and Axel Legay (Eds.). Springer, 153–171. https://doi.org/10.1007/978-3-030-30923-7_9

Truc L. Nguyen, Peter Schrammel, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2017. Parallel Bug-Finding in Concurrent Programs via Reduced Interleaving Instances. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE '17)*. IEEE Press, 753–764. https://doi.org/10.1109/ASE.2017.8115686

Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *OOPSLA*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 131–150. https://doi.org/10.1145/

2509136.2509514

Brian Norris and Brian Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.* 38, 3 (2016), 10:1–10:51. https://doi.org/10.1145/2806886

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

Antonio Paolillo, Hernán Ponce de León, Thomas Haas, Diogo Behrens, Rafael Lourenco de Lima Chehab, Ming Fu, and Roland Meyer. 2021. Verifying and Optimizing Compact NUMA-Aware Locks on Weak Memory Models. *CoRR* abs/2111.15240 (2021). arXiv:2111.15240 https://arxiv.org/abs/2111.15240

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.

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

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

Masood Feyzbakhsh Rankooh and Jussi Rintanen. 2022. Propositional Encodings of Acyclicity and Reachability by Using Vertex Elimination. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022.* AAAI Press, 5861–5868. https://doi.org/10.1609/aaai.v36i5.20530

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *PLDI*. ACM, 175–186. https://doi.org/10.1145/1993498.1993520

Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. 2009. The semantics of x86-CC multiprocessor machine code. In *POPL*. ACM, 379–391. https://doi.org/10.1145/1480881.1480929

Jaroslav Sevcik and Peter Sewell. 2011. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. Accessed: 04/07/2023.

Alan Stern. 2023. tools: memory-model: Add rmw-sequences to the LKMM. https://lkml.org/lkml/2022/11/16/1555.

Zhihang Sun, Hongyu Fan, and Fei He. 2022. Consistency-Preserving Propagation for SMT Solving of Concurrent Program Verification. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022). https://doi.org/10.1145/3563321

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

Emina Torlak. 2009. *A constraint solver for software engineering: finding models and cores of large relational specifications.* Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. https://hdl.handle.net/1721.1/46789

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

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

Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. 2022. BBQ: A Block-based Bounded Queue for Exchanging Data and Profiling. In *ATC*, Jiri Schindler and Noa Zilberman (Eds.). USENIX Association, 249–262. https://www.usenix.org/conference/atc22/presentation/wang-jiawei

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

Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. 2013. Constraint Propagation for First-Order Logic and Inductive Definitions. *ACM Trans. Comput. Logic* 14, 3, Article 17 (aug 2013), 45 pages. https://doi.org/10.1145/2499937.2499938

Received 2023-04-14; accepted 2023-08-27