Kater: Automating Weak Memory Model Metatheory and Consistency Checking

MICHALIS KOKOLOGIANNAKIS, MPI-SWS, Germany
ORI LAHAV, Tel Aviv University, Israel
VIKTOR VAFEIADIS, MPI-SWS, Germany

The metatheory of axiomatic weak memory models covers questions like the correctness of compilation mappings from one model to another and the correctness of local program transformations according to a given model—topics usually requiring lengthy human investigation. We show that these questions can be solved by answering a more basic question: "Given two memory models, is one weaker than the other?" Moreover, for a wide class of axiomatic memory models, we show that this basic question can be reduced to a language inclusion problem between regular languages, which is decidable.

Similarly, implementing an efficient check for whether an execution graph is consistent according to a given memory model has required non-trivial manual effort. Again, we show that such efficient checks can be derived automatically for a wide class of axiomatic memory models, and that incremental consistency checks can be incorporated in GenMC, a state-of-the-art model checker for concurrent programs. As a result, we get the first time- and space-efficient bounded verifier taking the axiomatic memory model as an input parameter.

CCS Concepts: • **Theory of computation** → **Concurrency**; **Verification by model checking**; **Regular languages**.

Additional Key Words and Phrases: Model Checking, Weak Memory Models, Kleene Algebra with Tests

ACM Reference Format:
Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. 2023. Kater: Automating Weak Memory Model Metatheory and Consistency Checking. *Proc. ACM Program. Lang.* 7, POPL, Article 19 (January 2023), 29 pages. https://doi.org/10.1145/3571212

1 INTRODUCTION

Axiomatic memory consistency models define the semantics of concurrent programs by means of labeled directed graphs called *execution graphs*, which satisfy certain consistency constraints dictated by the memory model. Execution graphs are generalizations of execution traces. Their nodes, called *events*, record the individual memory accesses (e.g., reads and writes) performed by the program, while their edges record the various (partial) ordering constraints implied by the structure of the program, such as the program order (po), which orders events according to their control-flow order, and the memory consistency model, such as the reads-from relation (rf), which associates each read event with the write event from which it got its value.

As examples, consider the execution graphs corresponding to the interesting behaviors of the "store buffering" and "load buffering" programs below. In our programs, we use x, y, z for shared variables and $a, b, c, \ldots$ for thread-local variables (registers), and assume that all variables are

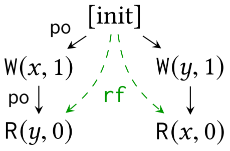
Authors' addresses: Michalis Kokologiannakis, MPI-SWS, Saarland Informatics Campus, Germany, michalis@mpi-sws.org; Ori Lahav, Tel Aviv University, Israel, orilahav@tau.ac.il; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany, viktor@mpi-sws.org.

2475-1421/2023/1-ART19
https://doi.org/10.1145/3571212

initialized to 0. As it can be seen, the graphs contain one event for each memory access, po edges between events of the same thread, and an incoming rf-edge for every read event.

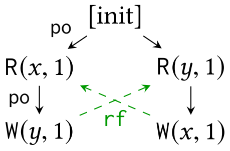
Store buffering (SB)

$x := 1 \parallel y := 1$
$a := y \parallel b := x$
Outcome: $a = b = 0$



Load buffering (LB)

$a := x \parallel b := y$
$y := 1 \parallel x := 1$
Outcome: $a = b = 1$



Now, depending on the consistency constraints of the memory model, these outcomes may be allowed or not. For instance, *sequential consistency* (SC) [Lamport 1979] deems both execution graphs inconsistent (and thereby forbids the SB and LB behaviors), *total store ordering* (TSO) [Owens et al. 2009] admits the SB behavior and forbids the LB one (because it requires $\mathsf{po} \cup \mathtt{rf}$ to be acyclic), while the Arm8 memory model [Pulte et al. 2018] allows both SB and LB.

The consistency constraints are typically expressed in relational algebra, requiring certain relations to be empty or irreflexive. This notation became popular with the herd7 tool [Alglave et al. 2014], which takes as input a definition of an axiomatic memory model written in the "cat" language (which, for the purposes of this paper, is just relational algebra) and a litmus test (i.e., a small concurrent program annotated with a behavior of interest), and checks whether the program can exhibit that behavior according to the memory model.

Along with formal definitions of memory models, there is also a long line of work trying to establish basic meta-theoretic properties of these definitions, answering questions such as:

- Is a given memory model *monotone* with respect to various natural strengthenings, such as inserting a memory fence, merging two threads into a single thread, or, if applicable, strengthening the type of a memory access (e.g., from release to sequentially consistent)?
- Does a given model admit *local program transformations*, such as reordering of independent memory accesses?
- Given two memory models A and B, is A *weaker* than B? More generally, is a given *compilation scheme* from A to B (e.g., by inserting certain fences) sound?
- Does a given model rule out "out-of-thin-air" (OOTA) outcomes? That is, does it rule out dependency cycles?
- Is a given model suitable for stateless model checking (i.e., enumerating all consistent executions of a given bounded program)? In particular, does it satisfy the prefix-closedness and extensibility conditions that are required by the state-of-the-art GenMC [Kokologiannakis et al. 2019] and TruSt [Kokologiannakis et al. 2022] algorithms?

Along with the last question come two more practical questions:

- For a given memory model, how can we check consistency of an execution graph efficiently (e.g., in time proportional to the size of the execution graph) and incrementally (i.e., given a consistent graph extended with a single event, check for consistency of the resulting graph)?
- How can we best integrate such a consistency check in a state-of-the-art model checker?

Prior work has answered some of the first set of questions for specific (pairs of) memory models, like TSO [Owens et al. 2009], Arm8 [Pulte et al. 2018], Power [Alglave et al. 2014], IMM [Podkopaev et al. 2019], C11 [Batty et al. 2011], RC11 [Lahav et al. 2017]. There has also been some work that has tried to answer the first three questions automatically by exhaustively searching for counterexamples of a certain (small) size. Similarly, for the second set of questions, existing work has only considered specific memory models in an ad hoc fashion.

In this paper, we present a sound and automated approach for answering both sets of questions for axiomatic memory models that are expressed as a set of emptiness, irreflexivity, and acyclicity constraints of relational algebra terms.

First, in §3, we show that checking whether one model is weaker than another can naturally be expressed as a language inclusion problem that can be decided using finite-state automata. The key observation is that the fragment of relational algebra used in most definitions of memory models (e.g., SC, TSO, PSO, Power, Arm8, RC11, IMM) corresponds closely to *Kleene Algebra with Tests* (KAT) [Kozen 1997], an extension of regular expressions with a Boolean algebra over a collection of predicates describing a state. While the constraints themselves are not directly encodable in vanilla KAT, memory model inclusion can be reduced to proving entailments between KAT formulae, which is decidable for simple classes of entailments. Using this result, we also show how to check monotonicity, correctness of program transformations, correctness of compilation mappings, lack of OOTA behaviors, prefix-closedness, and extensibility.

We have implemented these procedures in a tool, called KATER, using which we managed to prove automatically a number of positive results from the literature, such as the equivalence between different axiomatizations of release-acquire consistency (§3.2), of the Coherence axiom (§3.5) and of TSO (§3.6), the correctness of compilation from RC11 to TSO, Arm8, and Power (§3.7) and the soundness of local reorderings in release/acquire (§3.2) and RC11 (§3.7). We were similarly able to reproduce some negative results from the literature, such as to show that the proposed compilation mapping from the original version of the C11 model to Power is unsound (§3.7).

Second, in §4, we show that for a slightly restricted class of memory models, whose consistency checks are expressed in terms of acyclicity constraints (rather than general irreflexivity constraints), we can automatically synthesize more efficient code for checking the consistency of an execution graph. Consistency checking can be reduced to performing a specialized depth-first-search traversal over the given execution graph, which has linear worst-case time and space complexity in the size of the execution graph for models that do not record dependencies between instructions (e.g., SC, TSO, RC11) and quadratic for models that do record dependencies (e.g., IMM, Arm8, Power).

Finally, we have integrated these checks into the GENMC model checker [Kokologiannakis et al. 2021] thereby obtaining the first efficient stateless model checker that takes as a parameter a declarative definition of the axiomatic memory model (see §5). Our experimental evaluation (§7) demonstrates that the performance of the automatically generated checks is similar to that of the much more complex and error-prone manual consistency checks that GENMC provides for its built-in RC11 model. In more detail, using KATER's automatically generated checks on average is twice as fast as running GENMC with the full consistency checks enabled. Running GENMC in its default mode (that employs approximate consistency checks) on average is twice as fast as using KATER, but is orders of magnitude slower on benchmarks with many SC accesses and/or fences, leading to timeouts in larger cases.

2 PRELIMINARIES

We review the (standard) relational notation that we will use in the paper, as well as the definitions of finite-state automata, Kleene algebra with tests (KAT), and a declarative framework for memory models based on execution graph consistency. Readers familiar with these concepts can skip this section.

2.1 Relational Notation

We write $\emptyset$, $\mathtt{univ}$, and $\mathtt{id}$ for the empty, the full, and the identity relation, respectively. Given a relation R, we write $R^?$, R^+ and R^* for its reflexive, transitive and reflexive-transitive closures, respectively, and R^{-1} for its inverse, i.e., $\{\langle a, b\rangle \mid \langle b, a\rangle \in R\}$. We write $dom(R)$ and $rng(R)$ for the domain and range of R, respectively. Given two relations R_1 and R_2, we write $R_1 ; R_2$ for their relational composition, i.e., $\{\langle a, b\rangle \mid \exists c.\ \langle a, c\rangle \in R_1 \wedge \langle c, b\rangle \in R_2\}$. Given a set A, we write $[A]$ for the identity relation on A: $\{\langle a, a\rangle \mid a \in A\}$.

We say that a relation R is *irreflexive* if $\nexists a.\ \langle a, a\rangle \in R$ and *acyclic* if R^+ is irreflexive. A relation is a *strict partial order* if it is irreflexive and transitive. A relation R is *total* on a set A if $\langle a, b\rangle \in R \cup R^{-1} \cup [A]$ for all $a, b \in A$. A relation is a *strict total order* on a set A if it is a strict partial order that is total on A. The following lemma (used in §3.6) connects these concepts.

Lemma 2.1. *A relation R is acyclic if and only if R is irreflexive and there exists a strict total order T on $dom(R) \cup rng(R)$ such that $T ; R$ is irreflexive.*

Proof. In the forward direction, take T to be any total order extending R^+. In the backward direction, by means of contradiction, consider a cycle in R. Since R is irreflexive, the cycle will contain at least two distinct nodes. Because T is total, all pairs of adjacent nodes will be ordered by $T \cup T^{-1} \cup \mathtt{id}$. However, it cannot be the case that all pairs of adjacent nodes will be ordered by $T \cup \mathtt{id}$, or else we would get a cycle in T. So, there has to be a pair of R-adjacent nodes ordered by T^{-1}, contradicting the assumption that $T ; R$ is irreflexive. □

2.2 Regular Languages and Finite State Automata

We fix an *alphabet* (i.e., a finite non-empty set) Σ. A *language* L is a set of words in Σ^*. We use $a, b, \ldots$ to range over Σ, and $u, v, w, \ldots$ to range over Σ^*.

A *non-deterministic finite automaton* (NFA) over Σ is a tuple $\langle Q, \delta, S, F\rangle$ where Q is a finite set of states, $S \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function which, given a state $q \in Q$ and a letter $a \in \Sigma$, returns the set of possible next states $\delta(q, a)$. By abuse of notation, we extend the domain of the transition function to take as parameters a set of states and a word as follows: $\delta(S, a) \triangleq \bigcup_{q \in S} \delta(q, a)$, $\delta(S, \epsilon) \triangleq S$, and $\delta(S, aw) \triangleq \delta(\delta(S, a), w)$.

The *language accepted* by an NFA contains all words for which there is a path from an initial state of the NFA to a final state: $\mathsf{L}(\langle Q, \delta, S, F\rangle) \triangleq \left\{w \in \Sigma^* \;\middle|\; \delta(S, w) \cap F \neq \emptyset\right\}$. Two NFAs are *language-equivalent* iff they accept the same language.

A *deterministic finite automaton* (DFA) is an NFA that has exactly one initial state and where for every $q \in Q$ and $a \in \Sigma$, the set $\delta(q, a)$ contains at most one element. The *powerset construction* transforms an NFA $\langle Q, \delta, S, F\rangle$ over Σ into a language-equivalent DFA $\langle \mathcal{P}(Q), \delta_p, \{s_0\}, F'\rangle$ where $s_0 \triangleq S$, $F' \triangleq \left\{s \subseteq Q \;\middle|\; s \cap F \neq \emptyset\right\}$, and $\delta_p(s, a) \triangleq \{\delta(s, a)\}$.

A *regular language* is one described by a regular expression or equivalently one accepted by an NFA. There are standard conversions from regular expressions to NFAs and vice versa. Regular languages are closed under:

- *union* ($L_1 \cup L_2$);
- *concatenation* ($L_1 ; L_2$);
- *repetition* (L^*);
- *intersection* ($L_1 \cap L_2$), by the product construction on NFAs: $\langle Q_1 \times Q_2, \delta_P, S_1 \times S_2, F_1 \times F_2\rangle$ where $\delta_P(\langle q_1, q_2\rangle) \triangleq \delta_1(q_1) \times \delta_2(q_2)$;
- *complementation* ($\overline{L}$), by conversion to DFA and complementing the set of final states;
- *reversal* (L^{-1}), by swapping initial and final states in NFA, and reversing the transitions;
- *substitution* ($L[L_1/a_1, \ldots, L_n/a_n]$), by replacing all a_i transitions of an NFA with automata accepting L_i;
- *rotational closure* ($\mathsf{ROT}(L) \triangleq \{uv \mid vu \in L\}$), which can be computed on an NFA N as $\bigcup_{q \in N.Q} \mathsf{After}_q ; \mathsf{Before}_q$ where After_q is the NFA obtained from N by making q be its only initial state and Before_q is the NFA obtained from N by making q be the only final state; and

- *deduplication closure* ($\mathsf{DEDUP}(L) \triangleq \{w \in \Sigma^* \mid \exists n.\, w^n \in L\}$), which can be computed on an NFA (see, e.g., [(https://cs.stackexchange.com/users/683/yuval-filmus) 2022]).

Finally, inclusion and equivalence of regular languages are decidable (PSPACE-complete) by noting that $L_1 \subseteq L_2 \Leftrightarrow L_1 \cap \overline{L_2} = \emptyset$. Given that the expensive part of this inclusion checking is the DFA conversion as part of the complementation of L_2, there are algorithms that avoid performing the DFA conversion upfront and perform it "on demand" while traversing the NFA of L_1 (e.g., [Bonchi et al. 2013]).

2.3 Kleene Algebra with Tests

Kleene algebra with tests (KAT) [Kozen 1997] extends regular languages with a set of *tests*, over which there is a Boolean algebra.

Let Predicate be a finite set of primitive predicate symbols and Relation be a finite set of primitive relation symbols. KAT tests (t) and expressions (e) are given by the following grammar:

$$t ::= p \mid \mathsf{true} \mid \mathsf{false} \mid t_1 \cup t_2 \mid t_1 \cap t_2 \mid \overline{t}$$
$$e ::= [t] \mid r \mid e_1 \cup e_2 \mid e_1 \mathbin{;} e_2 \mid e^*$$

where $p \in \mathsf{Predicate}$ ranges over primitive predicates and $r \in \mathsf{Relation}$ over primitive relations. KAT tests contain the usual Boolean operators, while KAT expressions contain tests, relations, union, sequencing, and iteration. Tests allow us to express the empty relation $\emptyset \triangleq [\mathsf{false}]$ and the identity relation $\mathtt{id} \triangleq [\mathsf{true}]$. Moreover, as usual, reflexive closure is expressed as $e^? \triangleq e \cup \mathtt{id}$ and transitive closure as $e^+ \triangleq e \mathbin{;} e^*$.

KAT expressions are standardly interpreted as languages of *guarded* words, that is, alternating sequences of satisfiable tests and relations starting and ending with a test, $t_1 r_1 t_2 r_2 \ldots t_n r_n t_{n+1}$ for some $n \geq 0$. We write $\mathsf{L}(e)$ for the language induced by a KAT expression e.

KAT expressions can equivalently be interpreted as binary relations over a certain universe. In our context, we call these models *execution graphs*. Each execution graph G consists of a set E of nodes, called *events*, and interpretations of primitive tests as subsets of events and of primitive relations as binary relations on events:

$$[\![.]\!]_G : \mathsf{Predicate} \rightarrow \mathcal{P}(\mathsf{E}) \qquad [\![.]\!]_G : \mathsf{Relation} \rightarrow \mathcal{P}(\mathsf{E} \times \mathsf{E})$$

This interpretations are extended to KAT tests and expressions in the obvious way:

$$\begin{array}{ccc} [\![\mathsf{true}]\!]_G \triangleq \mathsf{E} & [\![\mathsf{false}]\!]_G \triangleq \emptyset & [\![\overline{t}]\!]_G \triangleq \mathsf{E} \setminus [\![t]\!]_G \\ [\![t_1 \cup t_2]\!]_G \triangleq [\![t_1]\!]_G \cup [\![t_2]\!]_G & [\![t_1 \cap t_2]\!]_G \triangleq [\![t_1]\!]_G \cap [\![t_2]\!]_G & [\![[t]]\!]_G \triangleq [[\![t]\!]_G] \\ [\![e_1 \cup e_2]\!]_G \triangleq [\![e_1]\!]_G \cup [\![e_2]\!]_G & [\![e_1 \mathbin{;} e_2]\!]_G \triangleq [\![e_1]\!]_G \mathbin{;} [\![e_2]\!]_G & [\![e^*]\!]_G \triangleq [\![e]\!]_G^* \end{array}$$

On top of KAT expressions, KAT *formulas* are defined by the following grammar:

$$\varphi ::= e_1 \subseteq e_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2$$

KAT formulas are interpreted as sets of execution graphs in the standard way: e.g., $[\![e_1 \subseteq e_2]\!] \triangleq \{G \mid [\![e_1]\!]_G \subseteq [\![e_2]\!]_G\}$ and $[\![\varphi_1 \Rightarrow \varphi_2]\!] \triangleq \{G \mid G \in [\![\varphi_1]\!] \Rightarrow G \in [\![\varphi_2]\!]\}$. The interpretation is extended to sets of KAT formulas in the obvious way: $[\![\Phi]\!] \triangleq \bigcap_{\varphi \in \Phi} [\![\varphi]\!]$. We say that a KAT formula φ *holds*, denoted by $\vdash \varphi$, if $[\![\varphi]\!]$ is equal to the set of all graphs. We write $\Phi \vdash \varphi$ if $[\![\Phi]\!] \subseteq [\![\varphi]\!]$.

Inclusion between KAT expressions (i.e., $\vdash e_1 \subseteq e_2$) is PSPACE-complete, and remains so even under basic assumptions like emptiness of a KAT expression ($e = \emptyset$) or transitivity of a primitive relation ($r \mathbin{;} r \subseteq r$) [Kozen et al. 1996]. Inclusion and equivalence can be decided either by algebraic techniques or by reduction to finite state automata. In the latter case, it is convenient to first convert the automata into a normal form that accepts only guarded words, and then apply standard ways of checking language inclusion/equivalence between automata.

Conversion into the normal form has to ensure: (1) that each automaton state has incoming edges being predicates and outgoing edges being relations (or the other way round), (2) that all outgoing edges from initial states are predicate edges, and (3) that all incoming edges to accepting states are predicate edges. To do so, any states with both kinds of incoming and outgoing transitions have to be duplicated and suitably restricted: adjacent predicate transitions of the form $[p_1]\,;[p_2]$ are replaced with single composite transitions of the form $[p_1 \cap p_2]$, while adjacent transitions with relations are moved apart by adding a dummy [true] transition between them. Similarly, outgoing relation edges from initial states have to be prefixed with a dummy [true] transition, and conversely incoming relation edges to accepting states have to be postfixed with a [true] transition.

2.4 Memory Models as Emptiness and Irreflexivity Constraints over KAT

Axiomatic memory models can be formulated as a single emptiness constraint and a single irreflexivity constraint over KAT. For this purpose, we extend KAT formulas with a new construct $\mathsf{irreflexive}(e)$ with semantics $[\![\mathsf{irreflexive}(e)]\!] \triangleq \{G \mid \nexists a.\ \langle a, a\rangle \in [\![e]\!]_G\}$. Models with multiple such constraints can be encoded because of the following basic relational algebra properties:

$$e_1 = \emptyset \land e_2 = \emptyset \Leftrightarrow e_1 \cup e_2 = \emptyset$$
$$\mathsf{irreflexive}(e_1) \land \mathsf{irreflexive}(e_2) \Leftrightarrow \mathsf{irreflexive}(e_1 \cup e_2)$$

Similarly, acyclicity constraints can be encoded as $\mathsf{acyclic}(e) \triangleq \mathsf{irreflexive}(e^+)$.

Formally, a *memory model* M is a pair of KAT expressions $\langle e_\emptyset, e_{\mathrm{irr}}\rangle$, interpreted as a collection of execution graphs by $[\![\langle e_\emptyset, e_{\mathrm{irr}}\rangle]\!] \triangleq [\![e_\emptyset = \emptyset \land \mathsf{irreflexive}(e_{\mathrm{irr}})]\!]$. We say that a memory model M_1 is *stronger* than another model M_2 (and M_2 is *weaker* than M_1) if $[\![M_1]\!] \subseteq [\![M_2]\!]$. Two models are equivalent if they are both stronger and weaker than each other.

3 KATER: AUTOMATING THE METATHEORY OF (WEAK) MEMORY MODELS

In this section, we demonstrate the kinds of results that KATER manages to prove automatically. We will go through a series of examples and explain the key features of KATER along the way.

3.1 Extended Coherence Order

Let us start with a rather simple example. Execution graphs typically contain two basic relations: the *reads-from* relation ($\mathtt{rf}$), which connects writes to reads, and the *coherence* order, $\mathtt{co}$, which is a strict partial order on writes, or, more precisely, a disjoint union of strict total orders, each of which orders all writes to a given location.

From these two relations, we can define a couple more relations. First, we can define the *from-read* relation, $\mathtt{fr}$ (a.k.a. reads-before), to relate a read and a write if the read reads from a coherence-earlier write; i.e., $\mathtt{fr} \triangleq \mathtt{rf}^{-1}\,;\mathtt{co}$. Moreover, Lahav et al. [2017, §3.2] define the *extended coherence order*, $\mathtt{eco} \triangleq (\mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr})^+$, as the transitive closure of these three relations. Observe that $\mathtt{eco}$ can equivalently be expressed without the transitive closure as $\mathtt{rf} \cup (\mathtt{co} \cup \mathtt{fr})\,;\mathtt{rf}^?$.

Suppose that we want to automatically verify the latter claim. The idea is to think of the two different formulations of $\mathtt{eco}$ as regular expressions over the alphabet $\{\mathtt{rf}, \mathtt{rf}^{-1}, \mathtt{co}\}$, and then check for equivalence between them. In KATER, we would write the following:[1]

[1] This a pretty-printed version of the actual input syntax, which uses ASCII (e.g., | for union and <= for inclusion).

```
declare rf rf^-1 co
let fr = rf^-1 ; co
let eco1 = (rf ∪ co ∪ fr)^+
let eco2 = rf ∪ (co ∪ fr) ; rf^?
assert eco1 = eco2
```

With this input, KATER immediately returns a counterexample saying that eco_1 accepts the string rf ; rf but eco_2 does not.

We clearly want to dismiss this counterexample because rf takes us from a write to a read, and we assume that an event cannot be both a read and a write. (For the paper presentation, we assume there are no read-modify-write (RMW) operations; our KATER implementation handles RMW operations as two separate events.) One way to do so is to tell KATER that the rf does not compose with itself:

```
assume rf ; rf = 0
```

Adding assumptions makes the language inclusion/equivalence problem more challenging. For some very simple kinds of assumptions, such as ones of the form $e = \emptyset$ (where e is a KAT expression), language inclusion remains decidable.

Proposition 3.1 ([Kozen et al. 1996, Theorems 6 and 9]). *Let e, e_1, and e_2 be KAT expressions. Then, $e = \emptyset \vdash e_1 \subseteq e_2$ if and only if $\vdash e_1 \subseteq e_2 \cup \mathsf{Predicate}^* ; e ; \mathsf{Predicate}^*$.*

This time KATER returns rf ; co as a counterexample, which we dismiss for the same reason. And since we are at it, let's also state that $\mathsf{co} ; \mathsf{rf}^{-1} = \emptyset$.

```
assume rf ; co = 0
assume co ; rf^-1 = 0
```

Next comes a more interesting counterexample: co ; co. Here, the equivalence proof relies upon co being transitive, but KATER has not way of knowing that. So, let's add the assumption:

```
assume co ; co ⊆ co
```

Such transitivity assumptions can also be eliminated completely: to check that $\Phi \vdash \varphi$ under the additional assumption that a primitive relation r is transitive, we can replace all uses of r in Φ and φ with r^+.

Proposition 3.2. *Let φ be a KAT formula, Φ be a set of KAT formulas, and r be a primitive relation symbol. Then, $\Phi, r ; r \subseteq r \vdash \varphi$ if and only if $\Phi[r^+/r] \vdash \varphi[r^+/r]$.*

Running KATER now reveals another interesting counterexample: $\mathsf{rf} ; \mathsf{rf}^{-1} ; \mathsf{co}$. What is missing is the knowledge that rf^{-1} is functional: every read reads from exactly one write. Adding the missing assumption

```
assume rf ; rf^-1 ⊆ id
```

allows KATER to complete the equivalence proof and report success.

Assumptions of the form $e \subseteq \mathsf{id}$ for an inclusion query $e_1 \subseteq e_2$ can be eliminated by saturating the right-hand-side. In terms of KAT expressions, we let $\mathsf{satid}(e, e_2) \triangleq e^* ; e_2'$, where e_2' is obtained from e_2 by replacing every $r \in \mathsf{Relation}$ with $r ; e^*$. This transformation can also be defined in terms of NFAs. For each state of the automaton, we can add a self loop accepting the language described by e. If e is a primitive relation r', then this construction immediately reaches a fixpoint: running the construction on a saturated automaton will not introduce any new edges. If e is a composite expression, however, the construction does not reach a fixpoint. Since it introduces new states in the automaton, for completeness, the construction needs to be repeated again (and again). In principle, this repetition can be stopped after exceeding the number of states of e_1, but we stop it after a single iteration.

Proposition 3.3. *Let Φ be a set of KAT formulas, and e, e_1 and e_2 be KAT expressions. If $\Phi \vdash e_1 \subseteq \mathsf{satid}(e, e_2)$, then $\Phi, e \subseteq \mathtt{id} \vdash e_1 \subseteq e_2$.*

We note that instead of assuming that $\mathtt{rf} ; \mathtt{rf}^{-1} \subseteq \mathtt{id}$, the proof can also be completed with the assumption $\mathtt{rf} ; \mathtt{fr} \subseteq \mathtt{co}$. This assumption can be used by saturating the right-hand-side of the inclusion query in a similar way: wherever there is a $\mathtt{co}$ transition from state a to b in its NFA, construct new states m and n, and add an $\mathtt{rf}$ transition from a to m, an $\mathtt{rf}^{-1}$ transition from m to n, and a $\mathtt{co}$ transition from n to b. Although this construction adds more new states for each substitution, it has the benefit that it applies only to states with $\mathtt{co}$ transitions as opposed to all states of the NFA of the right-hand-side.

Proposition 3.4. *Let Φ be a set of KAT formulas, φ be a KAT formula, e be a KAT expression, and r be a primitive relation. If $\Phi[(r \cup e)/r] \vdash \varphi[(r \cup e)/r]$, then $\Phi, e \subseteq r \vdash \varphi$. Furthermore, in the case that $\vdash e[(r \cup e)/r] \subseteq r \cup e$, the converse holds as well.*

To avoid users having to explicitly define assumptions as those above, we equip kater with built-in theory Φ_{base} with primitive relations $\mathtt{rf}$, $\mathtt{co}$, and $\mathtt{fr}$ and predicates R and W. It consists of the following assumptions encoding the basic properties:

- Disjoint tests: $\mathsf{R} \cap \mathsf{W} = \emptyset$.
- Domain and range restrictions: $\mathtt{rf} = [\mathsf{W}] ; \mathtt{rf} ; [\mathsf{R}]$, $\mathtt{co} = [\mathsf{W}] ; \mathtt{co} ; [\mathsf{W}]$, and $\mathtt{fr} = [\mathsf{R}] ; \mathtt{fr} ; [\mathsf{W}]$.
- Transitivity: $\mathtt{co} ; \mathtt{co} \subseteq \mathtt{co}$.
- From-read properties: $\mathtt{rf} ; \mathtt{fr} \subseteq \mathtt{co}$ and $\mathtt{fr} ; \mathtt{co}^{+} \subseteq \mathtt{fr}$.

The disjointness assumption is used to remove edges from the (normal-form) NFAs corresponding to KAT expressions: kater removes any transitions labeled with tests containing (i.e., stronger than) $\mathsf{R} \cap \mathsf{W}$. More generally, disjointness assumptions can be eliminated using the following claim.

Proposition 3.5. *Let Φ be a set of KAT formulas, φ be a KAT formula, p be a primitive predicate, and t be a test. Then, $\Phi[p \cap \overline{t}/p] \vdash \varphi[p \cap \overline{t}/p]$ if and only if $\Phi, p \cap t = \emptyset \vdash \varphi$.*

In turn, the domain and range restrictions can easily be eliminated by replacing the left-hand-sides of the inclusions with their right-hand-sides throughout. This transformation is formally justified by the following proposition.

Proposition 3.6. *Let Φ be a set of KAT formulas, φ be a KAT formula, e be a KAT expression, and r be a primitive relation. If $\Phi[e/r] \vdash \varphi[e/r]$, then $\Phi, r = e \vdash \varphi$. Furthermore, in the case that $\vdash e[e/r] = e$, the converse holds as well.*

Finally, the transitivity assumption is eliminated using Prop. 3.2, and the from-read properties are eliminated using Prop. 3.4.

Using the converse directions of the propositions above, we also obtain completeness of this process, which entails decidability as we state next.

Proposition 3.7. *The question whether $\Phi_{\text{base}} \vdash e_1 \subseteq e_2$ given two KAT expressions e_1 and e_2 is decidable.*

Proof (sketch). First, the assumption $\mathtt{fr} ; \mathtt{co}^+ \subseteq \mathtt{fr}$ can be eliminated using Prop. 3.4, and completeness follows since $\vdash (\mathtt{fr} ; \mathtt{co}^+)[(\mathtt{fr} \cup \mathtt{fr} ; \mathtt{co}^+)/\mathtt{fr}] \subseteq \mathtt{fr} \cup \mathtt{fr} ; \mathtt{co}^+$. After applying this elimination, we obtain a theory that can be shown to be equivalent to Φ_1 that consists of the disjointness assumption, the domain range restrictions, and the assumption $(\mathtt{rf} ; \mathtt{fr} \cup \mathtt{co})^+ \subseteq \mathtt{co}$. Then, again, $(\mathtt{rf} ; \mathtt{fr} \cup \mathtt{co})^+ \subseteq \mathtt{co}$ can be eliminated using Prop. 3.4, and completeness follows since $\vdash (\mathtt{rf} ; \mathtt{fr} \cup \mathtt{co})^+[(\mathtt{co} \cup (\mathtt{rf} ; \mathtt{fr} \cup \mathtt{co})^+)/\mathtt{co}] \subseteq \mathtt{co} \cup (\mathtt{rf} ; \mathtt{fr} \cup \mathtt{co})^+$. Finally the domain assumptions can be eliminated using Prop. 3.6 and the disjointness assumption is eliminated using Prop. 3.5. All in all, we obtained a sequence of substitutions S to be performed on $e_1 \subseteq e_2$, such that $\Phi_{\text{base}} \vdash e_1 \subseteq e_2$ iff $\vdash e_1[S] \subseteq e_2[S]$. Decidability then follows from decidability of inclusion in KAT (without assumptions). □

3.2 Release-Acquire Consistency

In our next example, we will show equivalence between two different definitions of the release/acquire consistency model [Lahav et al. 2016a]. This example is, in fact, motivated by wanting to show the correctness of a program optimization, namely store-load de-ordering (e.g., $x := 1 ; a := y \rightsquigarrow x := 1 \parallel a := y$ under any program context). The effect of this transformation on execution graphs is to remove certain po edges from write events to read events. A simple way to show that a memory model allows this transformation is if its consistency condition does not depend at all on $[\mathtt{W}] ; \mathsf{po} ; [\mathtt{R}]$ edges. In other words, the model should not be affected if we substitute all instances of po by $\mathsf{po} \setminus [\mathtt{W}] ; \mathsf{po} ; [\mathtt{R}]$ in its definition.

The first model is the usual definition of release-acquire consistency. An execution graph is RA-consistent if $\mathtt{hb} ; (\mathtt{co} \cup \mathtt{fr})^?$ is irreflexive, where $\mathtt{hb}$ is the *happens-before* order, that relates two events a and b if there is a path composed of po and $\mathtt{rf}$ edges from a to b. In terms of relational algebra: $\mathtt{hb} \triangleq (\mathsf{po} \cup \mathtt{rf})^+$.

According to the second definition, an execution is release/acquire-consistent if $\mathtt{hb}_2 ; (\mathtt{co} \cup \mathtt{fr})^?$ is irreflexive and $\mathtt{fr}$ does not contradict po (i.e., $\mathsf{po} ; \mathtt{fr}$ is irreflexive). In this definition,

$\mathtt{hb}_2$ is a subset of happens-before which avoids using any $[\mathtt{W}] ; \mathsf{po} ; [\mathtt{R}]$ edges in its definition: $\mathtt{hb}_2 \triangleq ([\mathtt{R}] ; \mathsf{po} \cup \mathsf{po} ; [\mathtt{W}] \cup \mathtt{rfe})^+$, where $\mathtt{rfe}$ denotes all external $\mathtt{rf}$ edges (where the write and the read are not po-related).

So, now let's try to prove equivalence between the two versions. First, we need to equip kater with some additional built-in knowledge: (1) that $\mathtt{rf}$-edges are either internal (inside po) or external; (2) that internal $\mathtt{rf}$-edges are included in the program order; and (3) that the program order is transitive.

$$\mathtt{rf} = \mathtt{rfi} \cup \mathtt{rfe} \qquad \mathtt{rfi} \subseteq \mathsf{po} \qquad \mathsf{po} ; \mathsf{po} \subseteq \mathsf{po} \tag{po-properties}$$

Elimination of these assumptions can be done using Propositions 3.2, 3.4 and 3.6.

Then, we can simply formulate the following kater query:

```
let hb = ([R∪W]; po; [R∪W] ∪ rf)^+
let ra = hb;(co∪fr)^?
let hb_2 = ([R∪W]; po;[W] ∪ [R];po;[W∪R] ∪ rfe)^+
let ra_2 = hb_2;(co∪fr)^? ∪ po;fr
assert ra = ra_2
```

Running kater yields the counterexample $[\mathtt{W}] ; \mathsf{po} ; [\mathtt{R}]$. The point is that while $\mathtt{ra} = \mathtt{ra}_2$ is a sufficient condition for $\text{irreflexive}(\mathtt{ra}) \Leftrightarrow \text{irreflexive}(\mathtt{ra}_2)$, it is *not* a necessary one. (For example, $\vdash \text{irreflexive}(r_1 ; r_2) \Leftrightarrow \text{irreflexive}(r_2 ; r_1)$ but $\nvdash r_1 ; r_2 = r_2 ; r_1$.) Here, as a standard assumption, we know that the program order is always irreflexive. So, to check for equivalence between the two models, it suffices to check the following equality:

```
assert (ra ∪ po) = (ra_2 ∪ po)
```

which KATER can easily prove.

3.3 Irreflexivity Implications: Matching Endpoints and Rotations

There is, in fact, another way to prove the equivalence between the two release-acquire models without assuming that po is irreflexive. Under our assumption that writes and reads are disjoint, there can never be a cycle of form [W] ; … ; [R].

In reality, what we want to show is that $\mathtt{ra} \cap \mathtt{id} = \mathtt{ra}_2 \cap \mathtt{id}$ but this falls outside of the known decidable fragments. (Although regular languages are closed under intersection, they do not support a concept like the identity relation. We cannot simply treat id as an uninterpreted symbol because we need it to denote the identity relation.) We can, however, express a somewhat weaker constraint in KAT, which KATER can easily prove.

```
assert sameEnds(ra) = sameEnds(ra_2)
```

where $\mathsf{sameEnds}(e)$ restricts e to enforce that its endpoints are compatible. In the fragment we have seen so far, that would be that $\mathsf{sameEnds}(e)$ returns $[\mathtt{R}] ; e ; [\mathtt{R}] \cup [\mathtt{W}] ; e ; [\mathtt{W}] \cup [\mathtt{F}] ; e ; [\mathtt{F}]$. Soundness easily follows from the following proposition.

Proposition 3.8. *For every KAT expression e,* $\vdash \mathsf{irreflexive}(e) \Leftrightarrow \mathsf{irreflexive}(\mathsf{sameEnds}(e))$.

Consider now a third version of release-acquire consistency defined as $\mathsf{irreflexive}(\mathtt{ra}_3)$ where $\mathtt{ra}_3 \triangleq (\mathtt{co} \cup \mathtt{fr})^? ; \mathtt{hb}$, which we would like to show equivalent to the first version. If we just ask KATER to show $\mathtt{ra} = \mathtt{ra}_3$, we will get counterexamples such as $\mathtt{po} ; \mathtt{co}$ and $\mathtt{co} ; \mathtt{po}$. The issue is that $\mathtt{ra}_3$ is not equal to ra, but to a *rotation* of it.

Therefore, to prove the equivalence between the two models, we employ the *rotational closure* operator $\mathsf{ROT}(L) \triangleq \{uv \mid vu \in L\}$. Recall from § 2.2 that regular languages are closed under rotational closure. By extension, KAT expressions are closed under rotational closure as well (so we can freely use $\mathsf{ROT}(e)$ for a KAT expression e). We now show that employing rotational closure is sound for proving implications between irreflexivity constraints.

Proposition 3.9. *For every KAT expression e,* $\vdash \mathsf{irreflexive}(e) \Leftrightarrow \mathsf{irreflexive}(\mathsf{ROT}(e))$.

Proof. For the right-to-left direction, it suffices to note that $[\![e]\!]_G \subseteq [\![\mathsf{ROT}(e)]\!]_G$. For the converse, consider a loop in $[\![\mathsf{ROT}(e)]\!]_G$, i.e., there exists a such that $\langle a, a\rangle \in [\![\mathsf{ROT}(e)]\!]_G$. From the definition of ROT(.), we get $\langle a, a\rangle \in [\![uv]\!]_G$ for some $vu \in \mathsf{L}(e)$. The definition of $[\![.]\!]_G$ ensures that there exists b such that $\langle a, b\rangle \in [\![u]\!]_G$ and $\langle b, a\rangle \in [\![v]\!]_G$, from which we can obtain that $\langle b, b\rangle \in [\![e]\!]_G$, which means that $[\![e]\!]_G$ is not irreflexive. □

Putting the two together, to prove an implication of the form $\mathsf{irreflexive}(e_1) \Rightarrow \mathsf{irreflexive}(e_2)$, we will ask KATER to prove $\mathsf{sameEnds}(e_1) \subseteq \mathsf{ROT}(e_2)$.

3.4 Completeness and Decidability

The above method for checking implications between irreflexivity constraints is sound but incomplete. Indeed, we have $\mathsf{irreflexive}(r) \Rightarrow \mathsf{irreflexive}(r ; r)$, but $\mathsf{sameEnds}(r) \nsubseteq \mathsf{ROT}(r ; r)$. To recover completeness, we need the *deduplication closure* operator $\mathsf{DEDUP}(L) \triangleq \{w \mid \exists n.\ w^n \in L\}$. Recall from §2.2 that regular languages are closed under deduplication closure, and by extension, so are KAT expressions.

Proposition 3.10. *For every KAT expression e,* $\vdash \mathsf{irreflexive}(e) \Leftrightarrow \mathsf{irreflexive}(\mathsf{DEDUP}(e))$.

Proof. For the right-to-left direction, it suffices to note that $[\![e]\!]_G \subseteq [\![\mathsf{DEDUP}(e)]\!]_G$. For the converse, consider a loop in $[\![\mathsf{DEDUP}(e)]\!]_G$, i.e., there exists a such that $\langle a, a\rangle \in [\![\mathsf{DEDUP}(e)]\!]_G$. From the definition of $\mathsf{DEDUP}(.)$, there is some n and w such that $\langle a, a\rangle \in [\![w]\!]_G$ and $w^n \in \mathsf{L}(e)$. Since $\langle a, a\rangle \in [\![w]\!]_G$, we also have $\langle a, a\rangle \in [\![w^n]\!]_G$, and so $\langle a, a\rangle \in [\![e]\!]_G$, which means that $[\![e]\!]_G$ is not irreflexive. □

With same-ends, rotation, and deduplication together, we can rephrase irreflexivity entailment queries as inclusion queries in a sound and complete way:

Proposition 3.11. *For every two KAT expressions e_1 and e_2,* $\vdash \mathsf{sameEnds}(e_1) \subseteq \mathsf{DEDUP}(\mathsf{ROT}(e_2))$ *if and only if* $\vdash \mathsf{irreflexive}(e_2) \Rightarrow \mathsf{irreflexive}(e_1)$.

Proof. For the left-to-right direction, suppose that $\vdash \mathsf{sameEnds}(e_1) \subseteq \mathsf{DEDUP}(\mathsf{ROT}(e_2))$. It easily follows that $\vdash \mathsf{irreflexive}(\mathsf{DEDUP}(\mathsf{ROT}(e_2))) \Rightarrow \mathsf{irreflexive}(\mathsf{sameEnds}(e_1))$. By Propositions 3.9 and 3.10, we have $\vdash \mathsf{irreflexive}(e_2) \Leftrightarrow \mathsf{irreflexive}(\mathsf{DEDUP}(\mathsf{ROT}(e_2)))$. By Prop. 3.8, we have $\vdash \mathsf{irreflexive}(e_1) \Leftrightarrow \mathsf{sameEnds}(e_1)$. Hence, it follows that $\vdash \mathsf{irreflexive}(e_2) \subseteq \mathsf{irreflexive}(e_1)$.

For the converse, suppose that $\vdash \mathsf{irreflexive}(e_2) \Rightarrow \mathsf{irreflexive}(e_1)$. We show that $\mathsf{L}(\mathsf{sameEnds}(e_1)) \subseteq \mathsf{L}(\mathsf{DEDUP}(\mathsf{ROT}(e_2)))$. Let $t_1 r_1 t_2 r_2 \ldots t_n r_n t_{n+1} \in \mathsf{L}(\mathsf{sameEnds}(e_1))$. Let G be an execution graph with: (1) n events, $a_1, \ldots, a_n$, such that a_i satisfies t_i for every $1 \leq i \leq n$ and a_n satisfies t_1 (this is possible due to $\mathsf{sameEnds}(.)$ closure); (2) the relations of G are constructed such that $\langle a_i, a_{i+1}\rangle \in r_i$ for every $1 \leq i \leq n-1$ and $\langle a_n, a_1\rangle \in r_n$. This construction ensures that $\langle a_1, a_1\rangle \in [\![e_1]\!]_G$. Then, the assumption that $\vdash \mathsf{irreflexive}(e_2) \Rightarrow \mathsf{irreflexive}(e_1)$ entails that $\langle a_i, a_i\rangle \in [\![e_2]\!]_G$ for some $1 \leq i \leq n$. By the construction of G, it follows that there exists $m \geq 0$ such that $t_i r_i \ldots r_n t_{n+1} (t_1 r_1 \ldots r_n t_{n+1})^m t_1 r_1 \ldots t_{i-1} r_{i-1} \in \mathsf{L}(e_2)$. Hence, $(t_1 r_1 \ldots r_n t_{n+1})^{m+1} \in \mathsf{L}(\mathsf{ROT}(e_2))$, which means that $t_1 r_1 \ldots r_n t_{n+1} \in \mathsf{L}(\mathsf{DEDUP}(\mathsf{ROT}(e_2)))$. □

This leads to a decision procedure for queries of the form $\Phi_{\mathrm{base}} \vdash \mathsf{irreflexive}(e_1) \Rightarrow \mathsf{irreflexive}(e_2)$ (Φ_{base} can be extended with the additional po-properties assumptions mentioned in §3.2). Indeed, one can apply the elimination of assumptions as in the proof of Prop. 3.7, and finally apply Prop. 3.11.

3.5 Coherence

We move on to another topic concerning implications between irreflexivity constraints. Memory models often contain the following axiom, which is known as "Coherence" or "SC-per-location".

$$\mathtt{poloc} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr} \text{ is acyclic, where } \mathtt{poloc} \triangleq \mathsf{po} \cap \mathtt{sameloc}$$

We would like to show that this axiom is equivalent to $\mathsf{po} ; \mathtt{eco}$ being irreflexive.

A first obvious problem is that KATER cannot support the term "$\mathsf{po} \cap \mathtt{sameloc}$" for the same reason it could not support the term "$\mathtt{ra} \cap \mathtt{id}$". We can work around this problem by making KATER treat $\mathtt{poloc}$ as an uninterpreted relation, and adding two basic assumptions about $\mathtt{poloc}$: that it is transitive and that it is included in po.

Simply doing so, however, is not sufficient. KATER will return us a counterexample: $\mathsf{po} ; \mathtt{rf}$ is included in $\mathsf{po} ; \mathtt{eco}$ but not in any rotation of $(\mathtt{poloc} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr})^+$. The problem lies in the initial po edge. KATER should not really be considering arbitrary paths of $\mathsf{po} ; \mathtt{eco}$, but only ones that start and end with the *same* event. Following this principle, we have so far ruled out paths starting with a read event and ending with a write event. Now, we additionally want to rule out paths that start and end with events of different locations. Specifically, we can extend KATER's built-in knowledge with the $\mathtt{sameloc}$ relation and its basic properties:

$$\mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr} \cup \mathtt{id} \cup (\mathtt{sameloc} ; \mathtt{sameloc}) \subseteq \mathtt{sameloc}$$

Thus, as part of sameEnds(e), we will intersect e with $\mathtt{sameloc}$ and try to distribute the intersection to the primitive relations with rules such as $(r_1 \mathbin{;} r_2) \cap \mathtt{sameloc} = (r_1 \cap \mathtt{sameloc}) \mathbin{;} r_2$ provided $r_2 \subseteq \mathtt{sameloc}$. While this procedure is generally incomplete (it will not always succeed in pushing the $\_ \cap \mathtt{sameloc}$ to primitive relations), when applied to $\mathsf{po} \mathbin{;} \mathtt{eco}$, it will yield the term $\mathtt{poloc} \mathbin{;} \mathtt{eco}$, and so will rule out the counterexample.

Still, however, this is not enough. KATER will now return us another counterexample: $[\mathtt{W}] \mathbin{;} \mathtt{poloc} \mathbin{;} [\mathtt{W}] \mathbin{;} \mathtt{co} \mathbin{;} [\mathtt{W}] \mathbin{;} \mathtt{poloc} \mathbin{;} [\mathtt{W}] \mathbin{;} \mathtt{co}$, which is clearly not included in any rotation of $\mathtt{poloc} \mathbin{;} \mathtt{eco}$.

The problem is that KATER does not (yet) know that $\mathtt{co}$ is total over all writes to the same location. From totality and $\mathtt{poloc} \mathbin{;} \mathtt{co}$ irreflexivity, it follows that $[\mathtt{W}] \mathbin{;} \mathtt{poloc} \mathbin{;} [\mathtt{W}] \subseteq \mathtt{co}^?$. Adding this inclusion as an assumption, lets KATER proceed further and generate another counterexample, which can be resolved by adding the assumption $[\mathtt{W}] \mathbin{;} \mathtt{poloc} \mathbin{;} \mathtt{fr} \mathbin{;} [\mathtt{W}] \subseteq \mathtt{co}^?$. This assumption, however, is still not enough. With a few more iterations, we can arrive at the constraint: $[\mathtt{W}] \mathbin{;} \mathtt{rf}^? \mathbin{;} \mathtt{poloc} \mathbin{;} \mathtt{fr}^? \mathbin{;} [\mathtt{W}] \subseteq \mathtt{co}^?$, which lets KATER complete the proof.

The question is how can we arrive at such constraints without the manual trial-and-error loop. The solution is again by a saturation procedure on the right-hand-side of an inclusion query.

$$\mathsf{TOT}(r, L) \triangleq L \cup (r \cup \{w \mid rw \in L, w \neq \epsilon\})^+$$

PROPOSITION 3.12. *Let Φ be a set of KAT formulas, e_1 and e_2 be KAT expressions, and r be a primitive relation. If $\Phi \vdash e_1 \subseteq \mathsf{TOT}(r, e_2)$, then Φ, r is a strict total order,* $\mathsf{irreflexive}(e_2) \vdash \mathsf{irreflexive}(e_1)$.

PROOF. Let $e' \triangleq \{w \mid rw \in \mathsf{L}(e_2), w \neq \epsilon\}$. By means of contradiction, consider an execution graph $G \in [\![\Phi, r \text{ is a strict total order}, \mathsf{irreflexive}(e_2)]\!]$ and a loop in $[\![e_1]\!]_G$, i.e., a such that $\langle a, a\rangle \in [\![e_1]\!]_G$. From our assumptions, we have $\langle a, a\rangle \in [\![\mathsf{TOT}(r, e_2)]\!]_G$. From the definition of $\mathsf{TOT}(.)$, we either get a loop in $[\![e_2]\!]_G$, which contradicts our hypothesis, or a cyclic path $\langle a, a\rangle \in [\![r \cup e']\!]^+_G$. Let $n \geq 1$ and $a_1, \dots, a_n$ such that $\langle a_i, a_{i+1}\rangle \in [\![r \cup e']\!]_G$ for every $1 \leq i \leq n$ (to simplify the notation here we work modulo n, so $n + 1 = 1$). We claim that for every $1 \leq i \leq n$, we must have $\langle a_i, a_{i+1}\rangle \in [\![r]\!]_G$. From this claim we obtain $\langle a_1, a_1\rangle \in [\![r^+]\!]_G$, which contradicts our hypothesis that $[\![r]\!]_G$ is a strict order. To prove this claim, let $1 \leq i \leq n$. The totality of $[\![r]\!]_G$ ensures that either $\langle a_i, a_{i+1}\rangle \in [\![r]\!]_G$ or $\langle a_{i+1}, a_i\rangle \in [\![r]\!]_G$. By means of contradiction, suppose that $\langle a_{i+1}, a_i\rangle \in [\![r]\!]_G$. Since $\langle a_i, a_{i+1}\rangle \in [\![r \cup e']\!]_G$ and r is a strict order, we must have $\langle a_i, a_{i+1}\rangle \in [\![e']\!]_G$, and so we have $\langle a_i, a_{i+1}\rangle \in [\![w]\!]_G$ for some w such that $rw \in \mathsf{L}(e_2)$. Then, we obtain $\langle a_{i+1}, a_{i+1}\rangle \in [\![r \mathbin{;} w]\!]_G \subseteq [\![e_2]\!]_G$, which is a loop in $[\![e_2]\!]_G$, and contradicts our hypothesis. □

3.6 Total Store Ordering (TSO)

Our next example concerns the SPARC/x86 TSO memory model [Owens et al. 2009; SPARC International Inc. 1994]. As with the previous example, the goal is to prove equivalence between two different definitions of TSO.

The first model is the standard one. It defines the *preserved program order*, $\mathsf{ppo} \triangleq [\mathsf{R} \cup \mathsf{F}] \mathbin{;} \mathsf{po} \cup \mathsf{po} \mathbin{;} [\mathtt{W} \cup \mathsf{F}]$, to include all program order edges except for edges from writes to reads, and requires that $\mathtt{tso} \triangleq \mathsf{ppo} \cup \mathtt{rfe} \cup \mathtt{co} \cup \mathtt{fr}$ be acyclic and the coherence property hold.

The second model, due to Lahav et al. [2016b], requires that $\mathtt{hb} \mathbin{;} \mathtt{fr}^?$ be irreflexive and that there be a strict total order *mo* over *all* write and fence events such that $mo ; \mathtt{tso}_2$ is irreflexive where

$$\mathtt{hb} \triangleq (\mathsf{po} \cup \mathtt{rf})^+ \quad \text{and} \quad \mathtt{tso}_2 \triangleq (\mathtt{co} \cup [\mathsf{F} \cup \mathtt{W}] \mathbin{;} \mathtt{hb} \mathbin{;} [\mathsf{F} \cup \mathtt{W}] \cup ([\mathsf{F}] \cup \mathtt{rfe}) \mathbin{;} \mathsf{po} \mathbin{;} \mathtt{fr})\,.$$

First, note that irreflexivity of $\mathtt{tso}_2$ holds from irreflexivity of $\mathtt{hb} \mathbin{;} \mathtt{fr}^?$ and $\mathtt{co}$.

```
assert sameEnds(tso2) ⊆ hb;fr? ∪ co
```

Thus, by applying Lemma 2.1, the two models are equivalent provided that $\mathtt{tso}^+ \cup \mathsf{po}; \mathtt{eco}$ is irreflexive iff $\mathsf{hb} ; \mathtt{fr}^? \cup \mathtt{tso}_2^+$ is irreflexive, which KATER proves with the following queries:

```
assert sameEnds(tso_2^+ ∪ hb;fr^?) ⊆ rot (tso^+)
assert sameEnds(tso^+) ⊆ rot (tso_2^+ ∪ hb;fr^?)
```

3.7 C11 Compilation Results

Let us now see how KATER can establish some more substantial results about the revised C11 memory model of Lahav et al. [2017] without their ($\mathsf{po} \cup \mathtt{rf}$)-acyclicity constraint.

First, correctness of local transformations can be achieved in a similar way as in §3.2 concerning the release/acquire memory model. The idea is to prove equivalence with respect to a variant of the C11 definition obtained by replacing all instances of po with the following subset of po

$$\mathsf{ppo}_{C11} \triangleq [\mathsf{ACQ}] ; \mathsf{po} \cup \mathsf{po} ; [\mathsf{REL}] \cup [\mathsf{SC}] ; \mathsf{po} ; [\mathsf{SC}]$$

that is guaranteed to be preserved by the transformations, and adding the usual coherence axiom asserting acyclicity of ($\mathtt{poloc} \cup \mathtt{eco}$). KATER easily proves the equivalence.

Next, we examine the correctness of C11's default compilation mappings to the various hardware architecture models.

C11 to Arm8. We start with compilation to the Arm8 model. Although the Arm model is more complicated than some other hardware memory models, compilation from C11 to Arm8 is actually easier to establish than to some other memory models because the compilation mapping is the identity. That is, every primitive C11 access or fence maps to exactly one access or fence at the architecture level.

Therefore, to prove compilation correctness, we have to show that C11 is weaker than Arm8. C11 consistency checks three properties:

- Coherence with respect to happens-before (i.e., irreflexive($\mathsf{hb} ; \mathtt{eco}$));
- RMW-atomicity; and
- psc acyclicity.

Arm8 consistency has three other properties:

- Coherence with respect to the program order (i.e., acyclicity of $\mathtt{poloc} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr}$, which, as we have seen, is equivalent to irreflexive($\mathsf{po} ; \mathtt{eco}$));
- RMW atomicity; and
- Acyclicity of its ob relation.

Therefore, to prove correctness of compilation, it suffices to call KATER with the following input:

```
include "C11.kat"
include "Arm8.kat"
assert C11::hb ; eco ⊆ po ; eco ∪ Arm8::ob^+
assert C11::psc ⊆ Arm8::ob^+
```

This code snippet demonstrates two small features of KATER: (1) it allows one to include files containing additional definitions, and (2) it provides a simple name resolution mechanism to refer to definitions from other files. KATER easily proves these assertions.

C11 to x86-TSO. Our next compilation result concerns the mapping from C11 to the x86 model. There are actually two mappings of interest: one which inserts TSO fences right after SC-atomic stores, and one which inserts TSO fences right before SC-atomic loads, In both cases, all remaining accesses are mapped to plain TSO accesses, C11's SC fences are mapped to TSO fences and all remaining fences to NOPs.

Our general approach for handling such mappings is to define the architecture model in terms of the C11 access modes (e.g., only treat $\mathsf{F} \cap \mathsf{sc}$ as a TSO fence) and add additional assertions about the presence of additional fences induced by the mapping. In particular, we let KATER prove the following:

```
assume [W;SC];po;[R;SC] ⊆ po;[F;SC];po
assert C11::hb ; eco ⊆ po ; eco ∪ TSO::tso+
assert C11::psc+ ⊆ TSO::tso+
```

The assumption states that the mapping always introduces an SC fence between an SC write and a subsequent SC read from the same thread, and allows KATER to complete the proof, establishing the correctness of both mappings at once.

KATER uses such assumptions in a heuristic fashion whenever it is asked to prove an inclusion assertion. Given an assumption $A \subseteq B$, it searches for pairs of states $\langle x, y \rangle$ in the NFA representing the right-hand-side of the inclusion such that there is a B path from x to y. Whenever this is the case, it adds an A path from x to y, which may introduce further states if A is a composite expression.

C11 to Power. Next, we consider the compilation to Power, which is substantially more complex than the compilations to TSO and Arm8, and has led to incorrect claims about the compilation of the original C11 model to it. Here, we will follow the axiomatic Power model of Alglave et al. [2014], which consists of the following axioms:

- Coherence: $(\mathtt{poloc} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr})$ is acyclic.
- No-thin-air: A certain hb relation containing preserved program order edges (due to dependencies or fences) and rfe edges is acyclic.
- Propagation: $\mathtt{co} \cup \mathtt{prop}$ is acyclic, where prop is Power's propagation order.
- Observation: $\mathtt{obs} \triangleq \mathtt{fr} \mathbin{;} \mathtt{prop} \mathbin{;} \mathtt{hb}^*$ is irreflexive.

There exist multiple correct compilation mapping schemes from RC11 to Power. For concreteness, we will present the "leading-sync with lwsyncs" scheme. This scheme maps C11's SC fences to Power's global synchronization fence (sync), C11's other fences to Power's lightweight synchronization fence (lwsync), introduces an lwsync fence before every release write and after every acquire read, and a sync fence before every SC access (read or write). We therefore model Power's sync as C11's SC-fence, lwsync as any C11's fence, and formulate the following assumptions about the presence of additional fences.

```
assume [R;ACQ];po ∪ po;[W;REL] ⊆ po;[F];po
assume po;[R;SC]  ∪ po;[W;SC]  ⊆ po;[F;SC];po
```

To establish the correctness of C11's coherence axiom, we ask KATER the following:

```
let r = eco;po?;eco* ∪ Pow::hb+ ∪ (co ∪ prop)+ ∪ obs
assert sameEnds(eco ; C11::hb) ⊆ r
```

which it proves in less than a minute. Note that to assist KATER's inclusion check, we have incorporated a small 'optimization' in this query. We have used a reformulation of the coherence axiom that is equivalent to Power's acyclic$(\mathtt{poloc} \cup \mathtt{rf} \cup \mathtt{co} \cup \mathtt{fr})$ axiom, as already shown in §3.1, and already incorporates a rotation.

Next, to show that C11's psc relation is acyclic, ideally one would ask KATER the following query.

```
assert sameEnds(C₁₁::psc⁺) ⊆ rot(r)
```

While KATER can in principle prove this inclusion, in practice it takes forever for KATER to return. The issue is that applying the rotational closure and the implication assumptions to r generates a huge automaton, and so the various simplification passes and the inclusion checking takes too long.

By performing these transformations manually to the Power model, KATER is able to establish the inclusion (without the rotation). Specifically, to avoid the explicit assumptions, we adapt the definition of the Power relations to include two additional disjuncts, which are shown in comments below.

```
let sync = po;[F;SC];po //∪ po;[R;SC] ∪ po;[W;SC]
let lwsync = po;[F];po //∪ [R;ACQ];po ∪ po;[W;REL]
```

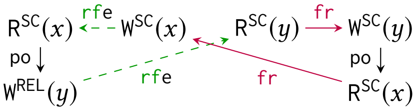
Let us now consider a simplified version of the original C11 model, whose compilation to Power turned out to be incorrect. The goal is to (dis)prove the following inclusion:

```
assert sameEnds(([SC];(hb∪co∪fr);[SC])⁺) ⊆ rot(r)
```

Again, rewriting the Power model to avoid the scalability issues, we get the counterexample:

```
[SC∩R];po;[REL∩W];rfe;[SC∩R];fr;[SC∩W];po;[SC∩R];fr;[SC∩W];rfe
```

which is allowed by Power but not by C11. We depict it also as an execution graph and a litmus test below:



$$\begin{array}{l||l||l||l} a := x^{\mathsf{SC}} & x^{\mathsf{SC}} := 1 & b := y^{\mathsf{SC}} & y^{\mathsf{SC}} := 2 \\ y^{\mathsf{REL}} := 1 & & & c := x^{\mathsf{SC}} \end{array}$$

Outcome: $x = a = b = 1 \wedge y = 2 \wedge c = 0$

3.8 Other Meta-theoretic Properties as Constraints over KAT

In addition to comparing memory models, as we discussed so far, we can use KAT queries to check for prefix-closedness, extensibility, and monotonicity. Key to establishing these properties is the observation that all primitive relations are used only positively in KAT expressions, while KAT expressions are used negatively in the model definitions (since $x = \emptyset \Leftrightarrow x \subseteq \emptyset$).

- *Prefix-closedness* [Kokologiannakis et al. 2019] holds by construction for every expressible memory model: removing edges from an execution graph cannot create any additional paths that cannot exist or cannot be cyclic.
- *Extensibility* [Kokologiannakis et al. 2019] holds as long as the model is defined purely in terms of the built-in relations (po, rf, rfe, rfi, co, and fr) and does not contain emptiness checks. Adding an event e maximally to a consistent execution graph does not create any outgoing edges from e, and so it cannot create any new cycles.
- For *monotonicity* with respect to the merging of the threads, it suffices for the memory model to be defined purely in terms of the relations like po, rf, co, and fr and not in terms of relations like rfi and rfe (internal and external reads-from, respectively), which distinguish between events originating from the same thread or not. This holds, for example, for the SC and RC11 models, but not for TSO and Arm8.

```
let sc = (po ∪ rf ∪ co ∪ fr)+
acyclic sc
```

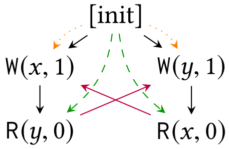


Fig. 1. Sequential consistency written in kat (left) and an inconsistent execution of SB (right)

- For *monotonicity* with respect to access mode strengthenings, e.g., from acquire to SC, it suffices for the "acquire" predicate of the memory model to also include SC accesses ($[\![\mathsf{SC}]\!]_G \subseteq [\![\mathsf{ACQ}]\!]_G$ for all G), and to never use predicates in a negative context, i.e., never take the complement of a predicate or the set difference between two predicates.

Moreover, given a way to prove that a model is weaker than another, we can leverage it to answer the remaining two meta-theoretical questions from the introduction.

- For *local program transformations*, it suffices to prove equivalence with a model where the correctness of the transformation is evident (see example in §3.2).
- For the absence of *out-of-thin-air behaviors*, it suffices to show that the model is stronger than $\langle \emptyset, (\mathsf{deps} \cup \mathsf{rf})^+ \rangle$, where deps represents the set of program-induced dependencies between po-ordered events, i.e., the union of the address, data, and control dependencies.

4 KATER: AUTOMATICALLY CHECKING GRAPH CONSISTENCY

Let us now move on to the second mode of operation of KATER: synthesizing code that checks efficiently whether a given execution graph is consistent according to a fixed memory model. As we show in §5, the generated consistency checkers can be integrated into existing execution-graph-based model checking tools like GenMC [Kokologiannakis et al. 2022; 2019].

But how do we check consistency of a graph given an arbitrary memory model M to begin with? Since M is expressed as emptiness and irreflexivity constraints over some relations, a simple solution is to calculate a fixpoint of the corresponding relations, and then check for emptiness/irreflexivity.

As an example, consider the annotated execution of the SB program from §1 under the SC memory model (cf. Fig. 1). Naively checking consistency for this execution graph boils down to calculating the transitive closure of the sc relation, yielding a complexity of $O(n^3)$, where n is the number of graph nodes.

That said, for SC in particular, we can do much better than $O(n^3)$. To check whether a graph is cyclic, one does not need to compute any transitive closures; one can simply perform a plain depth-first search through the graph, recording at each node whether it has been visited and whether the recursive visits of its children have been completed. The depth-first search has complexity $O(n + m)$ where n is the number of graph nodes and m the number of sc edges. Since a graph with n nodes can have $O(n^2)$ sc edges, the overall complexity is quadratic.

We can, however, do even better and bring down the DFS complexity to $O(n)$ by making the graph sparse. The idea is to observe that:

$$\mathsf{acyclic}(\mathsf{sc}) \Leftrightarrow \mathsf{acyclic}(\mathsf{po}|_{\mathrm{imm}} \cup \mathsf{rf} \cup \mathsf{co}|_{\mathrm{imm}} \cup \mathsf{rf}^{-1};\mathsf{co}|_{\mathrm{imm}})$$

and to use the immediate counterparts of po, co, and fr to make the graph sparse, thus bringing down the DFS complexity to $O(n)$.

The fact that SC admits such fast consistency checks begs the question of whether such efficient consistency checks can be generalized for an arbitrary memory model. At a first glance, this does not seem obvious. Even though SC is expressed as a single transitive closure of some primitive

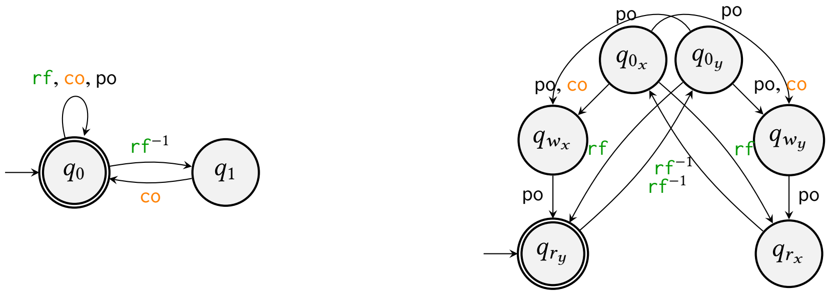


Fig. 2. Consistency checks with kater (non-immediate relations are drawn to not clutter the presentation)

relations, this is not the case for memory models in general: a model may require the acyclicity of relations defined in terms of other (potentially complex) relations, and thus merely performing a depth-first search is insufficient.

4.1 Checking Consistency for Arbitrary Acyclicity Constraints

For general acyclicity constraints, we solve the above issue by treating the execution graph itself as another automaton. Given an automaton Nfa$_M$ corresponding to a memory model M, and an automaton Nfa$_G$ corresponding to a graph G, take the intersection of Nfa$_G$ and Nfa$_M$ utilizing the product construction, and then "run" the two automata in parallel to detect whether any (non-empty) cycle in G violates M's acyclicity constraint. As Nfa$_G$ does not have any obvious initial states, and the cycle in G has to start and end at the same node, we in fact consider multiple intersections (one for every node of G), each of which has a single node as an initial/final state.

For illustration purposes, consider the SC model again where fr is decomposed into its constituent parts $\mathtt{fr} \triangleq \mathtt{rf}^{-1};\mathtt{co}|_{\mathrm{imm}}*$ so that Nfa$_{SC}$ is not completely trivial. An example of how this procedure rules out the SB behavior of Fig. 1 can be seen in Fig. 2. There are two things to notice in Fig. 2. First, Nfa$_G$ has two states corresponding to the graph's initial node. That is because the initial node corresponds to the initializing writes to all memory locations and therefore needs to be decomposed. (Besides, it would be wrong to add a co transition from the same state to both states q_{w_x} and q_{w_y}.) Second, we have picked q_{r_y} as the initial state for Nfa$_G$, although kater will examine all graph states as initial.

Let us now see how we detect the violation of Fig. 1. Starting from Nfa$_G$'s initial state q_{r_y}, perform a depth-first search on Nfa$_G$ while at the same time maintaining Nfa$_{SC}$'s state. Whenever a cycle on Nfa$_G$ is detected, check whether Nfa$_M$ is in a final state. If so, a violation is detected; otherwise, the exploration proceeds normally. In the case of Fig. 2, we explore the following pairs of states before detecting a violation (we use overline notation to denote the product's final state):

$$\overline{\langle q_0, q_{r_y}\rangle} \xrightarrow{\mathtt{rf}^{-1}} \langle q_1, q_{0_y}\rangle \xrightarrow{\mathtt{co}} \langle q_0, q_{w_y}\rangle \xrightarrow{\mathtt{po}} \langle q_0, q_{r_x}\rangle \xrightarrow{\mathtt{rf}^{-1}} \langle q_1, q_{0_x}\rangle \xrightarrow{\mathtt{co}} \langle q_0, q_{w_x}\rangle \xrightarrow{\mathtt{po}} \overline{\langle q_0, q_{r_y}\rangle}$$

On the other hand, when we take q_{0_y} as the initial state of Nfa$_G$, the violation is not detected. One cycle starting and ending at q_{0_y} is the following.

$$\overline{\langle q_0, q_{0_y}\rangle} \xrightarrow{\mathtt{po}} \langle q_0, q_{w_y}\rangle \xrightarrow{\mathtt{po}} \langle q_0, q_{r_x}\rangle \xrightarrow{\mathtt{rf}^{-1}} \langle q_1, q_{0_x}\rangle \xrightarrow{\mathtt{co}} \langle q_0, q_{w_x}\rangle \xrightarrow{\mathtt{po}} \langle q_0, q_{r_y}\rangle \xrightarrow{\mathtt{rf}^{-1}} \langle q_1, q_{0_y}\rangle$$

In this case, even though a cycle from/to q_{0_y} was detected in Nfa$_G$, no violation is reported as Nfa$_{SC}$ is not in a final state at that point. In fact, this is the case for all cycles starting and ending at q_{0_y}: since they will have to end with an $\mathtt{rf}^{-1}$ edge (the only incoming edge to q_{0_y}), Nfa$_{SC}$ will

Algorithm 1 The general form of an optimal DPOR based on TruSt [Kokologiannakis et al. 2022]

```
1:  procedure Verify($P$)
2:      Visit($P, G_\emptyset$)
3:  procedure Visit($P, G$)
4:      if $\mathsf{consistent}_\mathsf{m}(G) \land a \leftarrow \mathsf{next}_P(G)$ then
5:          $G \leftarrow \mathsf{add}(G, a)$
6:          if IsErroneous($G$) then exit(“error”)
7:          if $a \in \mathtt{R}$ then
8:              for $w \in G.\mathtt{W}_{\mathtt{loc}(a)}$ do Visit($P$, SetRF($G, a, w$))
9:          else if $a \in \mathtt{W}$ then
10:             CalcRevisits($P, G, a$)
11:         Visit($P, G$)
```

be in a non-final state. This also explains why one has to try *all* states of Nfa$_G$ as its initial states: we have to find a proper starting point in a cycle of G so that it also becomes a word accepted by Nfa$_{SC}$.

4.2 Checking Consistency Incrementally

The consistency checking procedure above works reasonably well, however, since we are interested in using that procedure in the context of dynamic partial order reduction (DPOR), we can adjust it to our specific setting and obtain an even more efficient algorithm.

To see how, let us briefly recall how state-of-the-art DPOR algorithms work. Such algorithms verify a concurrent program by enumerating all of its execution graphs, and checking that none of them contains an `error`. The general form of an optimal DPOR based on TruSt [Kokologiannakis et al. 2022] can be seen in Algorithm 1.

The Verify procedure verifies a concurrent program P by starting from the graph $G_\emptyset$ containing only the initialization events, and recursively explores the executions of P by calling Visit (Line 2).

At each step, as long as G remains consistent according to the memory model (Line 4), Visit extends the current execution G by one event a (Line 5) obtained via $\mathsf{next}_P(G)$. If there are no more events to add, then $\mathsf{next}_P(G)$ returns $\bot$, and Visit returns (G is complete). If a denotes an error (e.g., an assertion violation), it is reported to the user and verification terminates (Line 6).

The next action taken depends on the type of a.

If a is a read, then it must read from some write in G. To this end, for each write w in a's location, Visit sets w as the rf option for a, and recursively calls itself (Line 8).

If a is a write, it needs to revisit existing reads of the same location in G, because a was not present in the graph when Visit was considering possible rf options for these reads. To that end, Visit calls CalcRevisits (Line 10), which will take care of placing a in co, appropriately restricting G, and recursively calling Visit. As the explanation of how these recursive explorations are performed is not relevant to this paper, we do not present it here; we instead refer interested readers to Kokologiannakis et al. [2022], where the explanation of DPOR is given in full.

For all other types of events, Visit simply recursively calls itself (Line 11).

What is important for our purposes is that DPOR does not check for consistency of an arbitrary graph. Rather, given an execution graph G for which $\mathsf{consistent}(G)$ holds, it *incrementally* checks whether $\mathsf{consistent}(\mathsf{add}(G, e))$ holds.

Exploiting this very fact, we can make our consistency checking algorithm more efficient with a simple trick. The key insight here is that instead of trying all of Nfa$_G$'s states as initial states, we now know which state of $\textsc{Nfa}_G$ we have to use as an initial state: the state q_e, where e is a newly added event. Since G is consistent, any cycle that exists in $\mathrm{add}(G, e)$ must involve e. Therefore, we do not have to perform a separate DFS from all of $\textsc{Nfa}_G$'s states (as the cycles detected in these explorations might not involve e), but only DFSs that start from q_e.

Then, assuming that $\textsc{Nfa}_M$ has a single initial state that is also final, we can run $\textsc{Nfa}_G$ and $\textsc{Nfa}_M$ in parallel, and check whether we can detect a cycle in $\textsc{Nfa}_G$ while passing through a final state in $\textsc{Nfa}_M$. If that is the case, and because $\textsc{Nfa}_M$ has a single initial/final state, then the cycle detected is an actual violation, as some rotation of its constituent relations is accepted by $\textsc{Nfa}_M$. Since $\textsc{Nfa}_M$ is typically small and (in contrast to $\textsc{Nfa}_G$) does not depend on the size of the input program, this procedure is much more efficient than the previous one: its time complexity is $O(n \times m)$, where n, m are the number of states of $\textsc{Nfa}_G$ and $\textsc{Nfa}_M$, respectively.

Finally, note that we can enforce that $\textsc{Nfa}_M$ has a single initial/final state merely by taking its reflexive-transitive closure. Since the generated DFS code discards empty paths anyway (any cycle in G comprises at least two events), taking the reflexive-transitive closure of $\textsc{Nfa}_M$ is safe, and also leads to an automaton with a single initial state that is also final.

5 KATER: INTEGRATION WITH GENMC

We now present how we integrate the consistency checks of § 4.2 into GenMC [Kokologiannakis et al. 2021], an open-source, state-of-the-art stateless model checking tool that implements the TruSt algorithm. First, we show how we can use kater to generate consistency checking routines that can be plugged into GenMC (§ 5.1) as well as how these routines can be optimized (§ 5.2), and then we show how we can use kater to validate some memory-model properties required by GenMC in order for it to operate (§ 5.2.1).

5.1 Integrating kater with GenMC

Integrating the consistency checking procedure of § 4.2 into GenMC consists of merely generating C++ code that GenMC can use in order to check graph consistency.

An example consistency checker for SC (see Fig. 1) is shown in Fig. 3. In order to check consistency upon the addition of a new event e in a graph G, `isConsistent(G,e)` initiates a single DFS exploration by calling `visit0` and `visit1`, essentially modeling that $\textsc{Nfa}_M$ can be in any state (i.e., q_0 or q_1) when a G-cycle accepted by $\textsc{Nfa}_M$ passes through e. Whenever the DFS algorithm detects a cycle (i.e., whenever it encounters a back edge; e.g., in line 5), it checks whether $\textsc{Nfa}_M$ passed through an accepting state[2], and if so, returns `false` to denote a consistency violation.

5.2 Optimizing Consistency Checking for GenMC

Even though the above procedure is linear in the size of the product of $\textsc{Nfa}_M$ and $\textsc{Nfa}_G$, there are still a couple of ways we can improve it in the context of GenMC.

First, we can make $\textsc{Nfa}_M$ even smaller by merging its transitions. Take $\textsc{Nfa}_{SC}$ (cf. Fig. 1), for instance. If we merge $\mathtt{rf}^{-1}; \mathtt{co}|_{\mathrm{imm}}{}^*$ into a single $\mathtt{fr}|_{\mathrm{imm}}$ transition, we can get rid of `visit1` and end up with a single-state automaton for SC, yielding a twofold complexity improvement. The only difference in the generated DFS code is that we will have to iterate over the $\mathtt{fr}|_{\mathrm{imm}}$ successor instead of the $\mathtt{rf}^{-1}$ successor in line 16.

Deciding whether to merge two transitions or not is largely a matter of engineering and tuning (automaton size vs transition complexity). In our experience, however, it is almost always worth merging predicate (guard) transitions with their successors. Such transitions boil down to if-statements in the generated DFS code and can thus be very efficiently merged with their successors.

[2] The bookkeeping code for checking whether a cycle passed through an accepting state is omitted here for brevity.

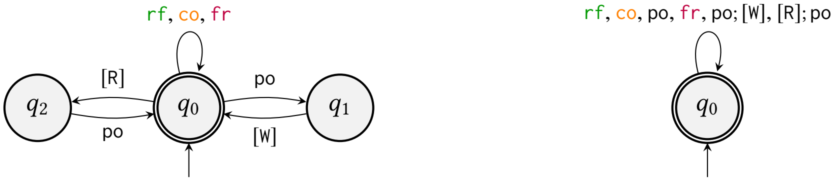
```
1   bool visit0(const ExecutionGraph &g, Event e)
2   {
3     setStatusAt0(e, ENTERED);
4     for (auto &p : po_imm_succs(g, e)) {
5       if (getStatusAt0(p) == UNSEEN && !visit0(p)) return false;
6       else if (getStatusAt0(p) == ENTERED && cycleHasAccepting()) return false;
7     }
8     for (auto &p : rf_succs(g, e)) {
9       if (getStatusAt0(p) == UNSEEN && !visit0(p)) return false;
10      else if (getStatusAt0(p) == ENTERED && cycleHasAccepting()) return false;
11    }
12    for (auto &p : co_imm_succs(g, e)) {
13      if (getStatusAt0(p) == UNSEEN && !visit0(p)) return false;
14      else if (getStatusAt0(p) == ENTERED && cycleHasAccepting()) return false;
15    }
16    for (auto &p : rf_inv_succs(g, e)) {
17      if (getStatusAt1(p) == UNSEEN && !visit1(p)) return false;
18      else if (getStatusAt0(p) == ENTERED && cycleHasAccepting()) return false;
19    }
20    setStatusAt0(e, LEFT);
21    return true;
22  }
23
24  bool visit1(const ExecutionGraph &g, Event e)
25  {
26    setStatusAt1(e, ENTERED);
27    for (auto &p : co_imm_succs(g, e)) {
28      if (getStatusAt0(p) == UNSEEN && !visit0(p)) return false;
29      else if (getStatusAt0(p) == ENTERED && cycleHasAccepting()) return false;
30    }
31    setStatusAt1(e, LEFT);
32    return true;
33  }
34
35  bool isConsistent(const ExecutionGraph &G, Event e)
36  {
37    return visit0(G,e) && visit1(G,e);
38  }
```

Fig. 3. C++ code generated by kater for consistency checking under SC

To see this, consider the automaton corresponding to the TSO memory model [Owens et al. 2009] before and after merging predicate transitions with their successors Fig. 4. If no transitions are merged (cf. Fig. 4, left), Nfa$_{TSO}$ has three states, and each predicate transition will lead to a separate state, thereby unnecessarily enlarging the state space. If we do merge predicate transitions and their successors, on the other hand, then Nfa$_{TSO}$ has a single-state (cf. Fig. 4, right), and the merged transitions will generate the same code as before, with the only difference being that the if-statements will be used as guards before/after iterating the successors of an event (e.g., line 6).

The second way we can optimize our consistency checking routine is inspired by GenMC's existing infrastructure, and consists of saving and reusing (parts of) relations. In many memory models, various intermediate relations are defined and then used multiple times in subsequent relation definitions. Take RC11's psc, for example (cf. Fig. 5): eco and hb are used multiple times in psc's definition. Recalculating these relations every time they are used is quite costly, as it may redo the same computation for a given graph event, for different states of Nfa$_M$.

To alleviate this problem, kater provides a save keyword that can be used to store the respective relation's predecessors for a given event, so that they do not have to ever be recalculated. If a user declares a given relation r as "saved", kater will generate code that calculates an event's r-predecessors when the event is first added, and then store these predecessors in the graph so that they do not have to be recalculated. In addition, for relations that are transitive, it is sufficient to only calculate the immediate predecessors of the event, so as to reduce the memory and time complexity of the calculation.



```
bool visit0(const ExecutionGraph &g, Event e)
{
  setStatus0(e, ENTERED);
  [...]
  for (auto &p : po_imm_succs(g, e)) {
    if (getStatus0(p) == UNSEEN && !visit1(p))
      return false;
    else if (getStatus0(p) == ENTERED &&
      cycleHasAccepting()) return false;
  }
  if (isRead(g, e)) {
    if (getStatus0(e) == UNSEEN && !visit2(e))
      return false;
    else if (getStatus0(e) == ENTERED &&
      cycleHasAccepting()) return false;
  }
  setStatus0(e, LEFT);
  return true;
}

bool visit1(const ExecutionGraph &g, Event e)
{
  setStatus1(e, ENTERED);
  if (isWrite(g, e)) {
    if (getStatus0(e) == UNSEEN && !visit0(p))
      return false;
    else if (getStatus0(p) == ENTERED &&
      cycleHasAccepting()) return false;
  }
  setStatus1(e, LEFT);
  return true;
}

bool visit2(const ExecutionGraph &g, Event e)
{
  setStatus2(e, ENTERED);
  for (auto &p : po_imm_succs(g, e)) {
    if (getStatus0(p) == UNSEEN && !visit0(p))
      return false;
    else if (getStatus0(p) == ENTERED &&
      cycleHasAccepting()) return false;
  }
  setStatus2(e, LEFT);
  return true;
}
```

```
bool visit0(const ExecutionGraph &g, Event e)
{
  setStatus0(e, ENTERED);
  [...]
  for (auto &p : po_imm_succs(g, e))
    if (isWrite(p)) {
      if (getStatus0(p) == UNSEEN && !visit1(p))
     return false;
      else if (getStatus0(p) == ENTERED &&
     cycleHasAccepting()) return false;
    }
  if (isRead(g, e))
    for (auto &p : po_imm_succs(g, e)) {
      if (getStatus0(e) == UNSEEN && !visit2(e))
     return false;
      else if (getStatus0(e) == ENTERED &&
     cycleHasAccepting()) return false;
    }
  setStatus0(e, LEFT);
  return true;
}
```

Fig. 4. NFA$_{TSO}$ and generated code before (left) and after (right) merging predicate transitions

```
let eco = (rf ∪ mo ∪ fr)⁺
let sw = [REL] ; ([F] ; po)? ; (rf ; rmw)* ; rf ; (po ; [F])? ; [ACQ]
save hb = (po ∪ sw)⁺
let psc = [SC]; po ; hb ; po ; [SC]
        ∪ [SC]; ([F]; hb)? ; (po ∪ rf ∪ mo ∪ fr) scb ; (hb; [F])? ; [SC]
        ∪ [F]; [SC] ; hb ; [F]; [SC]
        ∪ [F]; [SC] ; hb ; eco ; hb ; [F]; [SC]
acyclic psc
```

Fig. 5. The psc acyclicity axiom of RC11 [Lahav et al. 2017] written in kat

5.2.1 Checking GenMC's Requirements for Memory Models. Allowing users to specify memory models and to optimize their consistency checks (e.g., by saving relations) makes porting new models to GenMC much easier.

However, there are still some subtleties one has to take care of before adding support for a new model. First, GenMC requires memory models to provide a $\mathsf{ppo} \subseteq \mathsf{po}$ relation such that $\mathsf{pporf} \triangleq (\mathsf{ppo} \cup \mathsf{rf})^+$ is irreflexive in consistent executions because by design it generates only pporf-acyclic graphs. Second, GenMC cannot usefully save arbitrary relations. Since GenMC continuously modifies the current execution graph (e.g., when trying a different rf edge for a read), we can only save information about predecessors of an event e that will never be removed from the graph for as long as e remains in the graph, namely its $\mathsf{pporf};\mathsf{ppo}$ predecessors.

To preclude nonsensical GenMC behaviors when such requirements are violated, kater checks that GenMC's memory model requirements are satisfied before generating consistency checks. Concretely, kater will statically ensure that (1) the memory model acyclicity constraints imply irreflexivity of pporf, and (2) for each saved relation r, we have $r \subseteq \mathsf{pporf};\mathsf{ppo}$, and if r has, moreover, been declared as transitive, then $r;r \subseteq r$.

6 IMPLEMENTATION

kater can be used both as a prover for metatheoretic properties of memory models and as a consistency check generator for stateless model checkers. In this section, we review some of the design decisions we took while doing so, as well as some optimizations we performed in our implementation.

kater as a Proof Framework. The most important design decision we had to take as far as language inclusion is concerned is the inclusion algorithm itself. We opted for a breadth-first version of the Hopcroft-Karp algorithm that constructs DFAs on the fly, instead of constructing them a priori. Even though we could have used a more sophisticated algorithm for inclusion checking (e.g., the ones described in [Bonchi et al. 2013]), the Hopcroft-Karp algorithm seems to perform well enough for the tests we have so far (see §7.1). Breadth-first traversal naturally leads to minimal counterexamples, which are easier to understand by humans.

To reduce the size of the automata used in the inclusions, we perform some of the saturations described in §3 implicitly. Instead of replacing rf with $\mathsf{rfe} \cup \mathsf{rfi}$ on the right-hand side of an inclusion, we simply modified our inclusion algorithm to allow the right-hand side to take an rfe/rfi step whenever the left-hand side takes an rf step. More generally, given an inclusion $a \subseteq b$, we do allow b to take a transition t_b when a takes a transition t_a, as long as $t_a \subseteq t_b$. Finally, in order for us to avoid empty assumptions like $\mathsf{rf} ; \mathsf{co} = \emptyset$ we have equipped kater with some domain knowledge so that it can automatically understand when two transitions do not compose.

kater as a Consistency Check Generator. kater currently generates consistency checks only for acyclicity constraints, which we have integrated into GenMC. To ensure that the generated consistency checks for acyclicity constraints involving only the built-in relations run in linear time with respect to the size of the given execution graph, for the purpose of generating consistency checks, we take as primitives $\mathsf{po}|_{\mathrm{imm}}$, rf, $\mathsf{co}|_{\mathrm{imm}}$, and $\mathsf{fr}|_{\mathrm{imm}}$, which are all linear in the size of the execution graph, and set $\mathsf{po} \triangleq \mathsf{po}|_{\mathrm{imm}}{}^+$, $\mathsf{co} \triangleq \mathsf{co}|_{\mathrm{imm}}{}^+$, and $\mathsf{fr} \triangleq \mathsf{fr}|_{\mathrm{imm}} ; \mathsf{co}|_{\mathrm{imm}}{}^*$.

In principle, all of GenMC's code that performs calculations on execution graphs could have been replaced by kater-generated code. For simplicity, however, we decided to keep the code that checks for coherence (i.e., whether $\mathsf{hb};\mathsf{eco}$ is irreflexive) and RMW atomicity violations. Similarly, we also avoid checking for porf acyclicity, since GenMC's model checking algorithm enforces that property anyway.

Table 1. KATER queries to prove correctness of compilations (left) and of transformations/equivalences (right).

	Time	Result
$\mathrm{C11}_{\mathrm{Lahav}} \rightarrow \mathrm{IMM}$	0.05	✓
$\mathrm{IMM} \rightarrow \mathrm{TSO}$	0.01	✓
$\mathrm{C11}_{\mathrm{Lahav}} \rightarrow \mathrm{TSO}$	0.04	✓
$\mathrm{C11}_{\mathrm{strong}} \rightarrow \mathrm{TSO}$	0.04	✓
$\mathrm{IMM} \rightarrow \mathrm{Arm8}$	3.36	✓
$\mathrm{C11}_{\mathrm{Lahav}} \rightarrow \mathrm{Arm8}$	3.48	✓
$\mathrm{C11}_{\mathrm{strong}} \rightarrow \mathrm{Arm8}$	1.93	✓
$\mathrm{C11}_{\mathrm{Lahav}}/\mathrm{RA} \rightarrow \mathrm{POWER}$	3.11	✓
$\mathrm{C11}_{\mathrm{Lahav}}/\mathrm{SC} \rightarrow \mathrm{POWER}_{\mathrm{weak}}$	14.32	✓
$\mathrm{C11}_{\mathrm{Lahav}}/\mathrm{SC}_{\mathrm{RW}} \rightarrow \mathrm{POWER}$	138.35	✓
$\mathrm{C11}_{\mathrm{Lahav}}/\mathrm{SC}_{\mathrm{F}} \rightarrow \mathrm{POWER}$	56.67	✓
$\mathrm{C11}_{\mathrm{orig}} \rightarrow \mathrm{POWER}_{\mathrm{weak}}$	4.09	✗
$\mathrm{C11}_{\mathrm{orig}} \rightarrow \mathrm{POWER}$	16.00	✗
$\mathrm{POWER} \rightarrow \mathrm{POWER}_{\mathrm{simpl}}$	2.48	✓

	Time	Result
$\mathrm{RA} \leftrightarrow \mathrm{RA}_2$	0.01	✓
$\mathrm{RA} \leftrightarrow \mathrm{RA}_3$	0.01	✓
$\mathrm{C11}_{\mathrm{Lahav}} \leftrightarrow \mathrm{RC11}_{\mathrm{alt}}$	0.39	✓
$\mathrm{TSO} \leftrightarrow \mathrm{TSO}_{\mathrm{FM}}$	0.15	✓
$\mathrm{SC} \leftrightarrow \mathrm{SC}_{\mathrm{FM}}$	0.06	✓
$\mathrm{eco} \leftrightarrow \mathrm{eco}_2$	0.01	✓
$\mathrm{Coh} \leftrightarrow \mathrm{Coh}_2$	8.82	✓

To increase the efficiency of the generated code, KATER performs a number of optimizations in the memory model's NFA. First, it takes its reflexive-transitive closure, which typically helps in simplifying the NFA (e.g., when merging initial and final states). This optimization is safe to do as the generated DFS code discards empty paths anyway (see § 5.2). Second, it simplifies the NFA by (1) merging similar states (and transitions to such states), and (2) constructing its state composition matrix [Kameda et al. 1970], which helps to further simplify the automaton.

7 EVALUATION

Our evaluation of KATER comprises two parts. In the first part (§ 7.1), we summarize different metatheoretic properties we were able to prove with KATER. In the second part (§ 7.2), we evaluate the performance of our adaptation of GENMC equipped with KATER-generated consistency checks.

Experimental Setup. We conducted all experiments on a Dell PowerEdge M620 blade system, running a custom Debian-based distribution, with two Intel Xeon E5-2667 v2 CPU (8 cores @ 3.3 GHz), and 256GB of RAM. We used LLVM 11.0.1 for GENMC. Unless explicitly noted otherwise, all reported times are in seconds. We set a timeout limit of 30 minutes.

7.1 Metatheoretic Results

An overview with some mapping-correctness and equivalence results we were able to prove with KATER can be seen in Table 1. A ✓ entry denotes a successful proof, while an ✗ entry denotes that KATER (correctly) identified a counterexample while trying to complete a proof. Besides of the hard-coded assumptions on the primitive relations (see § 3.1), each of the mapping tests requires to encode the compilation scheme as KATER assumptions (see § 3.7). The tests are available in the artifact accompanying this paper.

As is evident from the table, the time required to complete a proof is proportional to the complexity of the memory models involved, and ranges from a few seconds to a few minutes. As expected, KATER requires less time to produce a counterexample than to prove compilation for models of similar complexity.

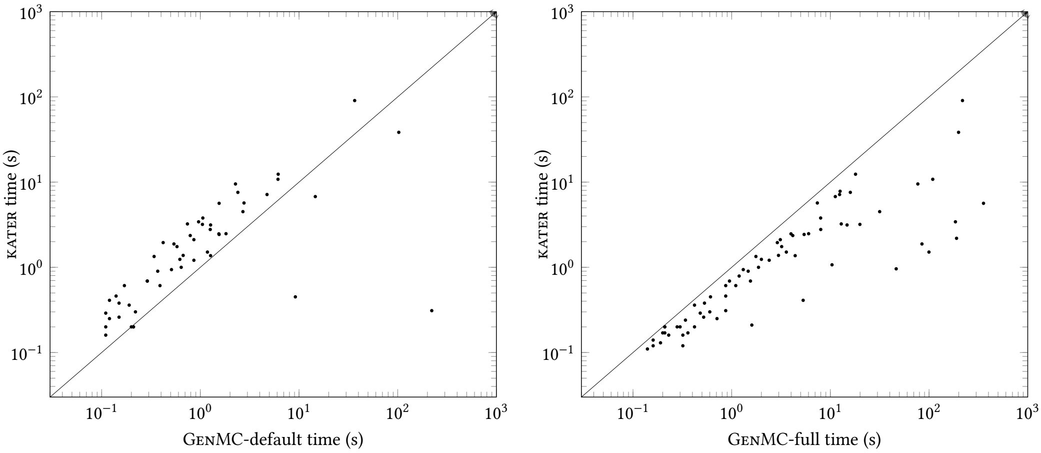


Fig. 6. Comparison between GenMC–default and kater (left) and GenMC–full and kater (right)

7.2 kater-generated Consistency Checks

In order to evaluate the performance of our kater-generated consistency checks, we subsequently answer the following questions:

- How well do the kater-generated consistency checks perform against the baseline GenMC implementation?
- How do the kater-generated checks scale as the memory model becomes more complex?

In all our tests, we ran GenMC under its (default) RC11 memory model. We henceforth use the term kater to refer to our version of GenMC employing kater-generated consistency checks.

7.2.1 GenMC vs kater. To answer our first question we first ran kater against GenMC's default test suite, excluding tests for which either tool finished in less than 0.10 seconds. The results can be seen in Fig. 6 (left). kater is on average two times slower than GenMC, though in certain cases it outperforms GenMC by a large factor. While this may come off as a surprise at first, the reason behind it is simple: GenMC does not check full consistency at each step, and is therefore generally faster. As part of an optimization (and precisely because checking consistency can be expensive), GenMC only checks for full consistency when an error is detected. Not checking for full consistency at every step means that in certain cases GenMC can explore orders of magnitude more executions than necessary and therefore run slower than kater.

Now, if we force GenMC to check full consistency at each program step, the results change dramatically (cf. Fig. 6, right). kater is never slower than GenMC (on average it is two times faster), while in many cases it is an order of magnitude faster, thereby demonstrating the efficiency of our generated consistency checks.

Still, one may wonder why kater is not always orders of magnitude faster than GenMC given that kater's consistency checks are linear in the size of the product of the memory model and the graph. The answer is twofold. First, most of GenMC's tests solely utilize weakly ordered accesses, and thus do not even require checking full RC11 consistency (the latter is mostly concerned with SC accesses). GenMC's handwritten consistency checking mechanism is able to leverage the non-existence of SC accesses and optimize away the checks, while kater always performs certain calculations. Second, many of these tests have small graphs and so GenMC's worse complexity does not show.

Table 2. SC benchmarks

	GenMC–default			GenMC–full	kater–SC	kater–TSO	kater–RC11
	Executions	*Time*	*Executions*	*Time*	*Time*	*Time*	*Time*
`szymanski(1)`	384	0.05	6	0.02	0.01	0.01	0.02
`szymanski(2)`	1 115 118	221.30	78	0.87	0.12	0.14	0.31
`szymanski(3)`	🕒	🕒	1068	34.95	2.33	2.90	6.90
`peterson(2)`	1848	0.07	48	0.03	0.02	0.02	0.03
`peterson(3)`	222 956	9.18	588	0.61	0.14	0.16	0.45
`peterson(4)`	32 468 072	1636.76	7360	12.55	2.11	2.51	7.81
`parker(1)`	232	0.03	54	0.04	0.02	0.02	0.04
`parker(2)`	139 425	14.62	6701	11.22	1.99	2.51	6.76
`dekker_f(2)`	302	0.03	71	0.11	0.05	0.05	0.09
`dekker_f(3)`	21 259	1.27	1344	4.87	0.51	0.62	2.53
`dekker_f(4)`	1 681 140	102.53	26 797	199.10	10.39	15.25	38.39
`fib_bench(4)`	34 205	0.17	19 605	1.10	0.19	0.21	0.61
`fib_bench(5)`	525 630	2.40	218 243	15.93	2.14	2.37	7.48
`fib_bench(6)`	8 149 079	36.63	2 363 803	218.16	23.54	26.31	90.57
`lamport(2)`	28	0.01	16	0.02	0.01	0.01	0.01
`lamport(3)`	54 851	4.74	9216	12.40	2.23	2.67	7.15

To better evaluate how efficient the kater-generated checks are, we conducted another case study in benchmarks containing many SC accesses. Such benchmarks do require extensive consistency checks under RC11, and give us a clearer picture of how kater's checks compare against the built-in ones.

The results are summarized in Table 2 (columns GenMC-default, GenMC-full, and kater-RC11). In the first four benchmarks, not checking for full consistency leads GenMC-default to perform poorly compared to GenMC-full and kater-rc11, as it explores orders of magnitude more executions than necessary. In the last two benchmarks, on the other hand, where GenMC-default does not explore a lot of redundant executions, GenMC-full and kater-RC11 are slower due to the complexity induced by the consistency checks. In all cases, however, kater-RC11 outperforms GenMC-full, and is also competitive against GenMC-default, even when the latter is faster than GenMC-full.

We end this part of our evaluation with two observations. First, in `dekker_f`, GenMC-full has comparable performance to GenMC-default, even though it explores two orders of magnitude fewer executions. kater-RC11, on the other hand, outperforms GenMC-default by a much larger margin, thereby allowing us to observe first-hand the difference in the computational complexity between the checks of the two tools. Second, in `lamport`, something similar happens for kater-RC11, which has comparable performance to GenMC-default even though it explores fewer executions. In this case, however, kater-RC11 does not explore exponentially fewer executions than GenMC-default. In addition, when the cost per execution is small (which is the case for `lamport`), it is expected that GenMC-default outperforms kater-RC11, though not by a large factor.

7.2.2 kater and Different Memory Models. To evaluate how well kater scales when the memory model becomes more complex, we added support for two models that GenMC did not previously support (SC and TSO), and compared kater-RC11 against kater-SC and kater-TSO in the computationally expensive benchmarks of Table 2 (columns kater-SC, kater-TSO and kater-RC11).

As expected, as the memory model becomes more complex, kater becomes slower. Both kater-SC and kater-TSO are much faster than kater-RC11, since the generated automata for these models comprise just one state, in contrast to the one for RC11, which comprises twelve states. However, even though the automata for SC and TSO have the same number of states, checking for SC is faster than TSO since the transitions in the TSO automaton are composite (i.e., they contain both predicates and relations; see Fig. 4 and §5.2).

In terms of performance against GenMC-default, notice that kater-SC and kater-TSO outperform GenMC-default even in cases where GenMC-default outperforms kater-RC11 (e.g., `fib_bench`, `lamport`), as their consistency checks are effectively linear in the size of the graph.

8 RELATED WORK

There has been a large body of work both in establishing metatheoretic properties of weak memory models (WMMs), as well as in developing effective model-checking algorithms for them.

WMM Metatheory. As far as metatheoretic properties are concerned, most existing works proved such properties for specific (pairs of) memory models with manual proof efforts (e.g., [Alglave et al. 2018; Batty et al. 2012; Dolan et al. 2018; Flur et al. 2016; 2017; Lahav et al. 2016a; b; 2017; Lamport 1979; Owens et al. 2009; Podkopaev et al. 2019; Pulte et al. 2018; 2019; Sarkar et al. 2012; 2011; SPARC International Inc. 1992; Vafeiadis et al. 2015]). Many of these results were not even mechanized, which led to the publication of some incorrect results (e.g., [Sarkar et al. 2012]).

There do exist a handful of approaches for automatically checking metatheoretic properties of weak memory models. To the best of our knowledge, Mador-Haim et al. [2010] first considered the problem of comparing memory models, but used the rather naive technique of exhaustively generating all litmus tests up to a bounded size. Mador-Haim et al. [2011] later showed that a fairly restricted class of memory models enjoyed a small model property and thus checking for whether a memory model is weaker than another is decidable if both models belong to that very restricted class, which is sufficient for expressing SC and TSO, but not Power, Arm or C11.

More recently, Wickerson et al. [2017] developed MemAlloy, a tool that performs an incomplete bounded search through possible litmus test skeletons to distinguish between memory models and to validate correctness of compiler mappings and optimizations. MemSynth [Bornholt et al. 2017] is a synthesis-based tool that uses SMT-solvers in its backend to answer similar queries about memory models as MemAlloy does, and additionally can generate memory model definitions that match a given set of litmus test outcomes and a sketch of the model.

With the exception of Mador-Haim et al. [2011], which works only for a very small class of models, all other approaches are *not* sound. When, for example, checking for inclusion between weak memory model definitions, they search for counterexamples up to a given bounded size, and can thus provide no formal guarantees about whether the property holds.

WMM Model Checking. On the model checking side, again there exists a lot of work targeting specific memory models (e.g., [Abdulla et al. 2015; Kokologiannakis et al. 2017; Norris et al. 2013]) or that is parametric in the choice of the memory model (e.g., [Kokologiannakis et al. 2022; 2019; 2020]), but which requires a substantial amount of work in order to add support for another memory model.

As far as model checking is concerned, the only tools providing roughly similar functionality to kater are MemSAT [Torlak et al. 2010], `herd7` [Alglave et al. 2014], and Dartagnan [Gavrilenko et al. 2019], although none of them is a stateless model checker. Specifically, given a small bounded program and a memory model, MemSAT and Dartagnan construct a formula representing the possible executions of the program according to the model and query a SAT/SMT solver to see whether a given program outcome is possible. In contrast, `herd7` follows the more naive strategy of explicitly generating all possible executions matching the program, and filtering out the inconsistent ones by checking the constraints specified by the memory model. In terms of scalability, the first two tools—MemSAT and herd7—were only meant to be used for small litmus tests, and so do not scale to larger examples like the ones used in §7.2. Dartagnan, on the other hand, uses cleverer encodings into SAT and various optimizations and is thus able to scale reasonably well.

Whether a SAT-based or stateless model checking approach works best depends largely on the program to be verified. SAT-based tools tend to scale better for programs with a large state space and no local computation, while stateless model checkers work best for programs with a relatively small number of distinct program executions, but which may include a lot of arithmetic computations. These two techniques have already been compared in [Abdulla et al. 2015; Kokologiannakis et al. 2020].

9 CONCLUSION & FUTURE WORK

We presented kater, a framework for proving metatheoretic properties of axiomatic memory models and generating efficient consistency checking routines that can be plugged into state-of-the-art stateless model checking tools. The key insight behind kater is that the memory models are commonly expressed as irreflexivity and emptiness constraints about regular expressions, and so checking for particular properties can be reduced to decidable language inclusion problems.

In the future, we plan to improve kater's performance by avoiding some expensive computations in repeatedly normalizing and simplifying the NFAs, which we expect may resolve the performance bottlenecks we get in verifying the correctness of the compilation from C11 and Power, which currently requires a manual selective application of rotations. Similarly, we plan to optimize the generated consistency checks to reduce their memory footprint and investigate whether it is possible to use the same infrastructure for generating decision procedures for weak memory that can be used by SMT solvers, following the recent work of He et al. [2021].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported by a European Research Council (ERC) Consolidator Grant for the project "PERSIST" and Starting Grant for the project "VAPLCS" under the European Union's Horizon 2020 research and innovation programme (grant agreement numbers 101003349 and 851811, respectively), and by the Israel Science Foundation (grant numbers 1566/18 and 814/22).

REFERENCES

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

Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan Stern (2018). "Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel." In: *ASPLOS 2018*. Williamsburg, VA, USA: ACM, pp. 405–418. doi: https://doi.org/10.1145/3173162.3177156.

Jade Alglave, Luc Maranget, and Michael Tautschnig (July 2014). "Herding cats: Modelling, simulation, testing, and data mining for weak memory." In: *ACM Trans. Program. Lang. Syst.* 36.2, 7:1–7:74. doi: https://doi.org/10.1145/2627752.

Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell (2012). "Clarifying and compiling C/C++ concurrency: From C++11 to POWER." In: *POPL 2012*. Philadelphia, PA, USA: ACM, pp. 509–520. doi: https://doi.org/10.1145/2103656.2103717.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber (2011). "Mathematizing C++ concurrency." In: *POPL 2011*. Austin, Texas, USA: ACM, pp. 55–66. doi: https://doi.org/10.1145/1926385.1926394.

Filippo Bonchi and Damien Pous (2013). "Checking NFA equivalence with bisimulations up to congruence." In: *POPL 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, pp. 457–468. doi: https://doi.org/10.1145/2429069.2429124.

James Bornholt and Emina Torlak (2017). "Synthesizing memory models from framework sketches and Litmus tests." In: *PLDI 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, pp. 467–481. doi: https://doi.org/10.1145/3062341.3062353.

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy (2018). "Bounding Data Races in Space and Time." In: *PLDI 2018*. Philadelphia, PA, USA: ACM, pp. 242–255. doi: https://doi.org/10.1145/3192366.3192421.

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell (2016). "Modelling the ARMv8 architecture, operationally: Concurrency and ISA." In: *POPL 2016*. St. Petersburg, FL, USA: ACM, pp. 608–621. DOI: https://doi.org/10.1145/2837614.2837615.

Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell (2017). "Mixed-size concurrency: ARM, POWER, C/C++11, and SC." In: *POPL 2017*. Paris, France: ACM, pp. 429–442. DOI: https://doi.org/10.1145/3009837.3009839.

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 2019*. Ed. by Isil Dillig and Serdar Tasiran. Cham: Springer International Publishing, pp. 355–365. DOI: https://doi.org/10.1007/978-3-030-25540-4_19.

Fei He, Zhihang Sun, and Hongyu Fan (2021). "Satisfiability modulo Ordering Consistency Theory for Multi-Threaded Program Verification." In: *PLDI 2021*. Virtual, Canada: ACM, pp. 1264–1279. DOI: https://doi.org/10.1145/3453483.3454108.

Tsunehiko Kameda and Peter Weiner (1970). "On the State Minimization of Nondeterministic Finite Automata." In: *IEEE Trans. Computers* 19.7, pp. 617–627. DOI: https://doi.org/10.1109/T-C.1970.222994.

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis (Dec. 2017). "Effective stateless model checking for C/C++ concurrency." In: *Proc. ACM Program. Lang.* 2.POPL, 17:1–17:32. DOI: https://doi.org/10.1145/3158105.

Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis (Jan. 2022). "Truly stateless, optimal dynamic partial order reduction." In: *Proc. ACM Program. Lang.* 6.POPL. DOI: https://doi.org/10.1145/3498711.

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis (2019). "Model checking for weakly consistent libraries." In: *PLDI 2019*. New York, NY, USA: ACM. DOI: https://doi.org/10.1145/3314221.3314609.

Michalis Kokologiannakis and Viktor Vafeiadis (2020). "HMC: Model checking for hardware memory models." In: *ASPLOS 2020*. ASPLOS '20. Lausanne, Switzerland: ACM, pp. 1157–1171. DOI: https://doi.org/10.1145/3373376.3378480.

Michalis Kokologiannakis and Viktor Vafeiadis (2021). "GenMC: A model checker for weak memory models." In: *CAV 2021*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. LNCS. Springer, pp. 427–440. DOI: https://doi.org/10.1007/978-3-030-81685-8_20.

Dexter Kozen (1997). "Kleene Algebra with Tests." In: *ACM Trans. Program. Lang. Syst.* 19.3. DOI: https://doi.org/10.1145/256167.256195.

Dexter Kozen and Frederick Smith (1996). "Kleene Algebra with Tests: Completeness and Decidability." In: *CSL 1996*. Ed. by Dirk van Dalen and Marc Bezem. Vol. 1258. LNCS. Springer, pp. 244–259. DOI: https://doi.org/10.1007/3-540-63172-0_43.

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis (2016a). "Taming Release-acquire Consistency." In: *POPL 2016*. St. Petersburg, FL, USA: ACM, pp. 649–662. DOI: https://doi.org/10.1145/2837614.2837643.

Ori Lahav and Viktor Vafeiadis (2016b). "Explaining Relaxed Memory Models with Program Transformations." In: *FM 2016*. Springer, pp. 479–495. DOI: https://doi.org/10.1007/978-3-319-48989-6_29.

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

Leslie Lamport (Sept. 1979). "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs." In: *IEEE Trans. Computers* 28.9, pp. 690–691. DOI: https://doi.org/10.1109/TC.1979.1675439.

Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin (2010). "Generating Litmus Tests for Contrasting Memory Consistency Models." In: *CAV 2010*. Ed. by Tayssir Touili, Byron Cook, and Paul B. Jackson. Springer. DOI: https://doi.org/10.1007/978-3-642-14295-6_26.

Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin (2011). "Litmus tests for comparing memory consistency models: how long do they need to be?" In: *DAC 2011*. Ed. by Leon Stok, Nikil D. Dutt, and Soha Hassoun. ACM. DOI: https://doi.org/10.1145/2024724.2024842.

Brian Norris and Brian Demsky (2013). "CDSChecker: Checking concurrent data structures written with C/C++ atomics." In: *OOPSLA 2013*. ACM, pp. 131–150. DOI: https://doi.org/10.1145/2509136.2509514.

Scott Owens, Susmit Sarkar, and Peter Sewell (2009). "A better x86 memory model: x86-TSO." In: *TPHOLs 2009*. Munich, Germany: Springer, pp. 391–407. DOI: https://doi.org/10.1007/978-3-642-03359-9_27.

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

Yuval Filmus (https://cs.stackexchange.com/users/683/yuval-filmus) (2022). *Is the power of a regular language regular? Is the root of a regular language regular?* Computer Science Stack Exchange. URL:https://cs.stackexchange.com/q/99371 (version: 2018-10-31). eprint: https://cs.stackexchange.com/q/99371. URL: https://cs.stackexchange.com/q/99371 (visited on Oct. 20, 2022).

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." In: *Proc. ACM Program. Lang.* 2.POPL, 19:1–19:29. DOI: https://doi.org/10.1145/3158107.

Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur (2019). "Promising-ARM/RISC-V: A simpler and faster operational concurrency model." In: *PLDI 2019*. Phoenix, AZ, USA: ACM, pp. 1–15. doi: https://doi.org/10.1145/3314221.3314624.

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams (2012). "Synchronising C/C++ and POWER." In: *PLDI 2012*. ACM, pp. 311–322. doi: https://doi.org/10.1145/2254064.2254102.

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

SPARC International Inc. (1992). *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc.

SPARC International Inc. (1994). *The SPARC architecture manual (version 9)*. Prentice-Hall.

Emina Torlak, Mandana Vaziri, and Julian Dolby (2010). "MemSAT: checking axiomatic specifications of memory models." In: *PLDI 2010*. Ed. by Benjamin G. Zorn and Alexander Aiken. ACM, pp. 341–350. doi: https://doi.org/10.1145/1806596.1806635.

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 2015*. Mumbai, India: ACM, pp. 209–220. doi: https://doi.org/10.1145/2676726.2676995.

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides (2017). "Automatically Comparing Memory Consistency Models." In: *POPL 2017*. ACM, pp. 190–204. doi: https://doi.org/10.1145/3009837.3009838.

Received 2022-07-07; accepted 2022-11-07