Kater: Automating Weak Memory Model Metatheory and Consistency Checking

Contents

1 Introduction

Kater is a tool that can automatically prove metatheoretic queries about (weak) memory consistency models (e.g., compilation correctness, program transformations), and also produce consistency checking routines for stateless model checkers (currently only GenMC [1] is supported). The underlying theory of Kater is described in a POPL 2023 paper [2].

2 Usage

An invocation of Kater resembles the following:

```
kater [OPTIONS] file
```

where `OPTIONS` include options that can be passed to Kater (use `--help` for a full list), and `file` is a file in the KAT language (see 2.1 for more details).

Kater can operate in two modes: *proof mode*, where it will try to prove all metatheoretic queries of the input, and *export mode* (invoked with `-e`), which will export consistency checking code.

2.1 Syntax

A KAT file is a series of `relation`, `predicate`, `disjoint`, `let`, `assert` and `assume` statements (see 2.2 for examples). Kater also provides syntax relevant to code exporting, but this code is unstable and subject to change, so we do not provide more details here (see 2.3 for a glimpse).

`relation r` declares a new relation `r` and its per-location counterpart `r-loc`, `predicate p` declares a new predicate `p`, `disjoint L` declares that the `#`-separated predicates in list `L` are pairwise disjoint, `let` is used to define a derived relation, `assert` to declare a metatheoretic constraint to be checked by Kater, and `assume` to declare an assumption to be taken into account when proving the file's `assert`s.

To facilitate proofs, Kater has a builtin theory with relations and predicates commonly used in weak memory literature: `po, po-loc, mo, rf, fr, detour, ctrl, addr, data` (as well as their `-imm` counterparts, where applicable), and `R, W, F, UR, UW, NA, RLX, ACQ, REL, SC, True, False`. Users are encouraged to use the builtin theory if possible, and not re-declare such primitive relations.

Kater also knows the following properties:

- Disjoint tests: e.g., `R & W = 0`
- Domains and ranges: e.g., `rf <= [W]; rf; [R]`
- Transitivity: e.g., `po ; po <= po`
- From-read properties: `rf ; fr <= mo` and `fr ; mo+ <= fr`

Users can define additional properties using `assume` statements.

2.2 Proof Mode

2.2.1 Complete Fragment

We begin with some examples that fall within Kater's complete fragment to demonstrate matching endpoints and rotations.

The code below proves equivalence between two versions of the Release-Acquire (RA) model.

```
let hb = ([R|W];po;[R|W] | rf)+
let ra = hb;(mo|fr)?

let hb2 = ([R|W];po;[W] | [R];po;[W|R] | rfe)+
let ra2 = hb2;(mo|fr)? | po;fr

assert ra & id <= [R|W];ra2;[R|W] & id
```

The construct `id & r` is used to enforce that the endpoints are the same in the examined paths of `r`. `ra2` is restricted to `[R|W]` because Kater cannot otherwise know that there are no other types of events (in which case the stated equivalence does not hold if we do not appropriately change the definitions of `hb`,`hb2`).

Similarly, the code below proves equivalence between two other versions of RA.

```
let hb = (po | rf)+
let ra1 = (hb;fr | hb;mo | hb)

let ra3 = (mo | fr)?;hb

assert [R|W];ra1;[R|W] & id <= rot ra3
assert ra3 & id <= rot ra1
```

`rot` is used to consider the rotations of the RHS.

In both examples above, no assumption was necessary to prove the desired result, as Kater has all the necessary information in its builtin theory.

2.2.2 Incomplete Fragment

Kater treats generic non-emptiness assumptions like `a <= b` in a heuristic fashion[1]. Such assumptions are particularly useful in compilation schemes, like the one below from RC11 to TSO.

```
// file: tso.kat
let ppo = ([R]; po | po ; [F] | [UW|F] ; po | po ; [W])+
let tso = (ppo | rfe | mo | fr)

// file: rc11.kat
let eco = (rf | mo | fr)+

let sw = [REL] ; ([F] ; po)? ; (rf ; rmw)* ; rf ; (po ; [F])? ; [ACQ]
let hb = (po | sw)+

let SC' = ([F] | [R] | [W]) ; SC
let FSC = [F] ; [SC]
let fhb = [F] ; hb
let hbf = hb ; [F]
let scb = po | rf | mo | fr
let psc = [SC'] ; po ; hb ; po ; [SC']
        | [SC'] ; fhb? ; scb ; hbf? ; [SC']
        | FSC ; hb ; FSC
        | FSC ; hb ; eco ; hb ; FSC
        | SC' ; po ; SC'

let psc_strong = ([SC'] | [F];[SC]; hb?); (hb | eco); ([SC'] | hb?; [F];[SC])

let ar = psc+ |
         hb;eco |
        (po | rf)+

// file: compilation.kat
include "./tso.kat"
include "./rc11.kat"

assume [W];po;[R] <= [W];po;[F];po;[R]
assert rc11::psc+ <= tso::tso+
```

The compilation scheme is provided as an assumption, and Kater produces a counterexample if it is omitted.

2.2.3 Custom Relations and Primitives

Users can declare additional relations using the `relation r` command. This will declare `r` as well `r-loc`, and equips Kater with the knowledge that `r-loc <= r`.

An example where `rf` is re-defined as a user-declared relation can be seen below.

```
relation rf
relation rf-1
relation co

let nfr = rf-1; co

let eco1 = (rf | co | nfr)+
let eco2 = rf | (co|nfr);rf?

assume co;co <= co
assume co;rf-1 = 0
assume rf;rf = 0
assume rf;co = 0
assume rf;rf-1 <= id

assert eco1 <= eco2
```

Kater required quite a few assumptions in order to prove the desired result. Using the builtin theory in this example eliminates the need for such assumptions.

In a similar manner, users can declare additional predicates using the `predicate p` command. The `disjoint` command can be used to declare that some #-separated predicates are pairwise disjoint, as in the example below:

```
// Declare some predicates
predicate M
predicate N
predicate O
predicate X
predicate Y
predicate Z

// M,N,O are pairwise disjoint
```

[1]Excluding assumptions of the form `r;r <= r` or `r <= id` which maintain completeness.

```
disjoint M # O # N

// X and Y do not compose with Z
disjoint X # Z
disjoint Y # Z

// Check some invalid combinations
assert [M;N] = 0
assert [O;N] = 0
assert [O;M] = 0
assert [O;M;N] = 0
assert [X;Y;Z] = 0
```

2.3 Export Mode

The code below produces GenMC consistency checking code for RC11 [3].

```
// Extended coherence order
let eco = (rf | mo | fr)+

// Export mode needs to define a PPO relation
let ppo = po

// Extend SW with thread creation
let relseq = [REL] ; ([F|TC|TE] ; po)? ; (rf ; rmw)*
let sw_to_r = relseq ; rf ; [ACQ]
let sw_to_f = relseq ; rf ; po ; [F|TJ|TB] ; [ACQ]
let sw = sw_to_r | sw_to_f
let asw = tc | tj

assert sw = [REL] ; ([F|TC|TE] ; po)? ; (rf ; rmw)* ; rf ; (po ; [F|TJ|TB])? ; [ACQ]

// An hb_stable view needs to be provided for coherence
view hb_stable = (po-imm | (sw_to_r | asw) ; po-imm | sw_to_f)+
let hb = (hb_stable | hb_stable? ; (sw_to_r | asw))

assert hb = (po | sw | asw)+

// Coherence : Optimize the checking of irreflexive (hb ; eco)
coherence (hb_stable)

// No OOTA + RMW atomicity are taken care of by GenMC
// acyclic (po | rf)

// Sequential consistency order
let FSC = [F] ; [SC]
let fhb = [F] ; hb
let hbf = hb ; [F]
let scb = po | rf | mo | fr
let psc = [SC] ; po ; hb ; po ; [SC]
        | [SC] ; fhb? ; scb ; hbf? ; [SC]
        | FSC ; hb ; FSC
        | FSC ; hb ; eco ; hb ; FSC
        | SC ; po ; SC
acyclic psc

// ----------------------------------------------------------------------
// RC11 error detection example
// ----------------------------------------------------------------------

let ww_conflict = [W] ; loc-overlap ; [W]
let wr_conflict = [W] ; loc-overlap ; [R] | [R] ; loc-overlap ; [W]
let conflicting = ww_conflict | wr_conflict
let na_conflict = [NA] ; conflicting | conflicting ; [NA]

error VE_AccessNonMalloc unless  alloc <= hb_stable
error VE_DoubleFree      unless  [Free] ; loc-overlap ; [Free] = 0
error VE_AccessFreed     unless  loc-overlap ; [Free] <= hb_stable
error VE_AccessFreed     unless  [Free] ; loc-overlap = 0

error VE_RaceNotAtomic     unless  na_conflict <= hb_stable
```

3 Contact

For feedback, questions and bug reports please send an e-mail to michalis@mpi-sws.org.

References

[1] Michalis Kokologiannakis and Viktor Vafeiadis. GenMC: A model checker for weak memory models. In Alexandra Silva and K. Rustan M. Leino, editors, *CAV 2021*, volume 12759 of *LNCS*, pages 427–440. Springer, 2021.

[2] Michalis Kokologiannakis, Ori Lahav, and Viktor Vafeiadis. Kater: Automating weak memory model metatheory and consistency checking. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[3] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI 2017*, pages 618–632, New York, NY, USA, 2017. ACM.