(* emacs fontification -*-caml-*- *)

(*
— Introduction —
This file contains a mathematical version of the relaxed memory model of C11 and C++11, written in the specification language of Lem. Lem can compile it to Ocaml, HOL, Isabelle or Latex. The basic model is faithful to the intent of the 2011 standard and included here in full. In addition, there are several simplified models that either remove redundant concepts or provide simplifications for programs that restrict the input language of programs.
There are lots of definitions that make up the models. To help you navigate them, the following table of contents (with unique key phrases) can be used to search the document. Where appropriate, there are comments describing or explaining the definitions. These are especially important for the top-level definitions of the simplified models.
— Contents —

*)

(** *)
(* 1 - Relational definitions *)
(** *)

let $irrefl\ s\ ord = \forall x{\in}s.\ \neg\ ((x,\ x)\ \in\ ord)$

let $trans\ s\ ord = \forall x{\in}s,\ y{\in}s,\ z{\in}s.\ (((x,\ y)\ \in\ ord)\ \wedge\ ((y,\ z)\ \in\ ord))\ \rightarrow\ ((x,\ z)\ \in\ ord)$

let $cross\ S\ T = \{(s,\ t)|\forall s{\in}S,\ t{\in}T\ |\ \mathsf{true}\}$

val $tc : forall' a.('a * 'a)\ \text{SET} \rightarrow ('a * 'a)\ \text{SET}$

let $restrict\_relation\_set\ rel\ s = (rel)\ \cap\ (\text{cross}\ s\ s)$

```
let strict_preorder s ord = irrefl s (ord) ∧ trans s (ord)

let relation_over s rel = ∀(a, b)∈rel. a ∈ s ∧ b ∈ s

let inj_on f A =
  (∀x∈A. (∀y∈A. (f x = f y) → (x = y)))

let total_order_over s ord =
  relation_over s ord ∧ (∀x∈s, y∈s. (x, y) ∈ ord ∨ (y, x) ∈ ord ∨ (x = y))

let strict_total_order_over s ord = strict_preorder s ord ∧ total_order_over s ord

let adjacent_less_than ord s x y =
  (x, y) ∈ ord ∧ ¬ (∃z∈s. (x, z) ∈ ord ∧ (z, y) ∈ ord)

let adjacent_less_than_such_that pred ord s x y =
  pred x ∧ (x, y) ∈ ord ∧ ¬ (∃z∈s. pred z ∧ (x, z) ∈ ord ∧ (z, y) ∈ ord)

(************************************************* *)
(* 2 - Action and location types *)
(************************************************* *)

type ACTION_ID = NUM

type THREAD_ID = NUM

type LOCATION = NUM

type VALUE = NUM

type PROGRAM = NUM

type MEMORY_ORDER =
    MO_SEQ_CST
  | MO_RELAXED
  | MO_RELEASE
  | MO_ACQUIRE
  | MO_CONSUME
  | MO_ACQ_REL

type LOCK_OUTCOME =
    SUCCESS
  | BLOCKED

type ACTION =
  | LOCK of ACTION_ID * THREAD_ID * LOCATION * LOCK_OUTCOME
  | UNLOCK of ACTION_ID * THREAD_ID * LOCATION
  | ATOMIC_LOAD of ACTION_ID * THREAD_ID * MEMORY_ORDER * LOCATION * VALUE
  | ATOMIC_STORE of ACTION_ID * THREAD_ID * MEMORY_ORDER * LOCATION * VALUE
  | ATOMIC_RMW of ACTION_ID * THREAD_ID * MEMORY_ORDER * LOCATION * VALUE * VALUE
  | LOAD of ACTION_ID * THREAD_ID * LOCATION * VALUE
  | STORE of ACTION_ID * THREAD_ID * LOCATION * VALUE
  | FENCE of ACTION_ID * THREAD_ID * MEMORY_ORDER
  | BLOCKED_RMW of ACTION_ID * THREAD_ID * LOCATION

(************************************************* *)
(* - 2.1 - Projection functions *)
(************************************************* *)

let action_id_of a =
  match a with
  | LOCK aid _ _ _ → aid
```

```
    | Unlock aid _ _ → aid
    | Atomic_load aid _ _ _ _ → aid
    | Atomic_store aid _ _ _ _ → aid
    | Atomic_rmw aid _ _ _ _ _ → aid
    | Load aid _ _ _ → aid
    | Store aid _ _ _ → aid
    | Fence aid _ _ → aid
    | Blocked_rmw aid _ _ → aid
    end

let thread_id_of a =
    match a with
      Lock _ tid _ _ → tid
    | Unlock _ tid _ → tid
    | Atomic_load _ tid _ _ _ → tid
    | Atomic_store _ tid _ _ _ → tid
    | Atomic_rmw _ tid _ _ _ _ → tid
    | Load _ tid _ _ → tid
    | Store _ tid _ _ → tid
    | Fence _ tid _ → tid
    | Blocked_rmw _ tid _ → tid
    end

let memory_order_of a =
    match a with
      Atomic_load _ _ mo _ _ → Some mo
    | Atomic_store _ _ mo _ _ → Some mo
    | Atomic_rmw _ _ mo _ _ _ → Some mo
    | Fence _ _ mo → Some mo
    | _ → None
    end

let location_of a =
    match a with
      Lock _ _ l _ → Some l
    | Unlock _ _ l → Some l
    | Atomic_load _ _ _ l _ → Some l
    | Atomic_store _ _ _ l _ → Some l
    | Atomic_rmw _ _ _ l _ _ → Some l
    | Load _ _ l _ → Some l
    | Store _ _ l _ → Some l
    | Fence _ _ _ → None
    | Blocked_rmw _ _ l → Some l
    end

let value_read_by a =
    match a with
      Atomic_load _ _ _ _ v → Some v
    | Atomic_rmw _ _ _ _ v _ → Some v
    | Load _ _ _ v → Some v
    | _ → None
    end

let value_written_by a =
    match a with
      Atomic_store _ _ _ _ v → Some v
    | Atomic_rmw _ _ _ _ _ v → Some v
    | Store _ _ _ v → Some v
    | _ → None
    end
```

```
let is_lock a =
  match a with
    LOCK _ _ _ _ → true
  | _ → false
  end

let is_successful_lock a =
  match a with
    LOCK _ _ _ SUCCESS → true
  | _ → false
  end

let is_blocked_lock a =
  match a with
    LOCK _ _ _ BLOCKED → true
  | _ → false
  end

let is_unlock a =
  match a with
    UNLOCK _ _ _ → true
  | _ → false
  end

let is_atomic_load a =
  match a with
    ATOMIC_LOAD _ _ _ _ _ → true
  | _ → false
  end

let is_atomic_store a =
  match a with
    ATOMIC_STORE _ _ _ _ _ → true
  | _ → false
  end

let is_atomic_rmw a =
  match a with
    ATOMIC_RMW _ _ _ _ _ _ → true
  | _ → false
  end

let is_blocked_rmw a =
  match a with
    BLOCKED_RMW _ _ _ → true
  | _ → false
  end

let is_load a =
  match a with
    LOAD _ _ _ _ → true
  | _ → false
  end

let is_store a =
  match a with
    STORE _ _ _ _ → true
  | _ → false
  end
```

```
let is_fence a =
  match a with
    FENCE _ _ _ → true
  | _ → false
  end

let is_atomic_action a =
  is_atomic_load a ∨ is_atomic_store a ∨ is_atomic_rmw a

let is_read a =
  is_load a ∨ is_atomic_load a ∨ is_atomic_rmw a

let is_write a =
  is_store a ∨ is_atomic_store a ∨ is_atomic_rmw a

(* It is important to note that seq_cst atomics are both acquires and releases *)

let is_acquire a =
  match memory_order_of a with
    SOME MO_ACQUIRE → is_read a ∨ is_fence a
  | SOME MO_ACQ_REL → is_read a ∨ is_fence a
  | SOME MO_SEQ_CST → is_read a ∨ is_fence a
  | SOME MO_CONSUME → is_fence a
  | NONE → is_lock a
  | _ → false
  end

let is_consume a =
  is_read a ∧ (memory_order_of a = SOME MO_CONSUME)

let is_release a =
  match memory_order_of a with
    SOME MO_RELEASE → is_write a ∨ is_fence a
  | SOME MO_ACQ_REL → is_write a ∨ is_fence a
  | SOME MO_SEQ_CST → is_write a ∨ is_fence a
  | NONE → is_unlock a
  | _ → false
  end

let is_seq_cst a = (memory_order_of a = SOME MO_SEQ_CST)

(****************************************** *)
(* - 2.2 - Location kinds *)
(****************************************** *)

type LOCATION_KIND =
    MUTEX
  | NON_ATOMIC
  | ATOMIC

let actions_respect_location_kinds actions lk =
  ∀a∈actions. match location_of a with
      SOME l →
        match lk l with
          MUTEX → is_lock a ∨ is_unlock a
        | NON_ATOMIC → is_load a ∨ is_store a
        | ATOMIC → is_store a ∨ is_atomic_action a ∨ is_blocked_rmw a end
    | NONE → true
    end
```

```
let is_at_location_kind lk a lk0 =
  match location_of a with
    Some l → (lk l = lk0)
  | None → false
  end

let is_at_mutex_location lk a =
  is_at_location_kind lk a Mutex

let is_at_non_atomic_location lk a =
  is_at_location_kind lk a Non_atomic

let is_at_atomic_location lk a =
  is_at_location_kind lk a Atomic

(****************************************** *)
(* 3 - The preferred model *)
(****************************************** *)
```

(* This simplification has ben verified equivalent to the Standard's model (section 4) using the HOL theorem prover. It removes the complicated notion of VSSE's, whose force is covered by the coherence requirements. For those looking to work with C or C++ concurrency, this is the preferred model. Predicates from this model will be used in those that follow. *)

```
(********************************************** *)
(* - 3.1 - Execution records *)
(********************************************** *)

type opsem_execution_part =
  ⟨[actions : action set;
    threads : thread_id set;
    lk : location → location_kind;
    sb : ( action * action) set;
    asw : ( action * action) set;
    dd : ( action * action) set;
    cd : ( action * action) set; ]⟩

type witness_execution_part =
  ⟨[rf : ( action * action) set;
    mo : ( action * action) set;
    sc : ( action * action) set; ]⟩

(****************************************** *)
(* - 3.2 - Well formed action *)
(****************************************** *)

let same_thread a b = (thread_id_of a = thread_id_of b)

let threadwise_relation_over s rel =
  relation_over s rel ∧ (∀x∈rel.
    same_thread (fst x) (snd x))

let same_location a b = (location_of a = location_of b)

let locations_of actions =
{l|∀Some l∈{(location_of a)|∀a∈actions | true} | true}
```

```
let well_formed_action a =
  match a with
      Atomic_load _ _ mem_ord _ _ → mem_ord = Mo_relaxed ∨
                                    mem_ord = Mo_acquire
                                  ∨ mem_ord = Mo_seq_cst
                                  ∨ mem_ord = Mo_consume
    | Atomic_store _ _ mem_ord _ _ → mem_ord = Mo_relaxed
                                  ∨ mem_ord = Mo_release
                                  ∨ mem_ord = Mo_seq_cst
    | Atomic_rmw _ _ mem_ord _ _ _ → mem_ord = Mo_relaxed
                                  ∨ mem_ord = Mo_release
                                  ∨ mem_ord = Mo_acquire
                                  ∨ mem_ord = Mo_acq_rel
                                  ∨ mem_ord = Mo_seq_cst
                                  ∨ mem_ord = Mo_consume
    | _ → true
    end

(********************************************** *)
(* - 3.3 - Well formed threads *)
(********************************************** *)

let well_formed_threads actions threads lk sb asw dd cd =
    inj_on action_id_of (actions) ∧
    (∀a∈actions. well_formed_action a) ∧
    threadwise_relation_over actions sb ∧
    threadwise_relation_over actions dd ∧
    threadwise_relation_over actions cd ∧
    strict_preorder actions sb ∧
    strict_preorder actions dd ∧
    strict_preorder actions cd ∧
    (∀a∈actions. thread_id_of a ∈ threads) ∧
    actions_respect_location_kinds actions lk ∧
    dd subset sb ∧
    relation_over actions asw ∧
    (∀a∈actions.
      (is_blocked_rmw a ∨ is_blocked_lock a)
       →
      ¬ (∃b∈actions. a -sb-> b))

(********************************************** *)
(* - 3.4 - Consistent locks *)
(********************************************** *)

let consistent_locks actions lo hb =
    let mutex_actions = {a|∀a∈actions | (is_lock a ∨ is_unlock a)} in
    let lo_happens_before = restrict_relation_set hb mutex_actions in
    strict_total_order_over mutex_actions lo ∧
    lo_happens_before subset lo ∧
    (∀(a, c)∈lo.
      is_successful_lock a ∧ is_successful_lock c ∧ same_location a c
       →
      (∃b∈actions. same_location a b ∧ is_unlock b ∧ a -lo-> b ∧ b -lo-> c))

(********************************************** *)
(* - 3.5 - Well formed reads from mapping *)
(********************************************** *)

let well_formed_reads_from_mapping actions lk rf =
```

$$
\begin{array}{l}
\quad \text{relation\_over } actions\ rf\ \wedge \\
\quad (\forall a_1{\in}actions,\ a_2{\in}actions,\ b{\in}actions.\ (a_1 \xrightarrow{rf} b\ \wedge\ a_2 \xrightarrow{rf} b)\ \rightarrow\ (a_1 = a_2))\ \wedge \\
\quad (\forall a{\in}actions,\ b{\in}actions.\ a \xrightarrow{rf} b\ \rightarrow \\
\qquad \text{same\_location } a\ b\ \wedge \\
\qquad (\text{value\_read\_by } b = \text{value\_written\_by } a)\ \wedge \\
\qquad \neg\ (a = b)\ \wedge \\
\qquad \neg\ (\text{is\_fence } a)\ \wedge\ \neg\ (\text{is\_fence } b)\ \wedge \\
\qquad (\text{is\_at\_mutex\_location } lk\ a\ \rightarrow\ \mathsf{false})\ \wedge \\
\qquad (\text{is\_at\_non\_atomic\_location } lk\ a\ \rightarrow\ \text{is\_store } a\ \wedge\ \text{is\_load } b)\ \wedge \\
\qquad (\text{is\_at\_atomic\_location } lk\ a\ \rightarrow \\
\qquad\quad (\text{is\_atomic\_store } a\ \vee\ \text{is\_atomic\_rmw } a\ \vee\ \text{is\_store } a)\ \wedge \\
\qquad\quad (\text{is\_atomic\_load } b\ \vee\ \text{is\_atomic\_rmw } b\ \vee\ \text{is\_load } b)))
\end{array}
$$

(** *)
(* - 3.6 - Happens before *)
(** *)

$$
\begin{array}{l}
\mathsf{let}\ rs\_element\ rs\_head\ a = \\
\quad \text{same\_thread } a\ rs\_head\ \vee\ \text{is\_atomic\_rmw } a
\end{array}
$$

$$
\begin{array}{l}
\mathsf{let}\ release\_sequence\ actions\ lk\ mo\ a\_rel\ b = \\
\quad \text{is\_at\_atomic\_location } lk\ b\ \wedge \\
\quad \text{is\_release } a\_rel\ \wedge \\
\quad ((b = a\_rel)\ \vee \\
\qquad (\text{rs\_element } a\_rel\ b\ \wedge\ a\_rel \xrightarrow{mo} b\ \wedge \\
\qquad\quad (\forall c{\in}actions.\ (a\_rel \xrightarrow{mo} c\ \wedge\ c \xrightarrow{mo} b)\ \rightarrow\ \text{rs\_element } a\_rel\ c)))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{let}\ release\_sequence\_set\ actions\ lk\ mo = \\
\quad \{(a,\ b) | \forall a{\in}actions,\ b{\in}actions\ | \\
\qquad \text{release\_sequence } actions\ lk\ mo\ a\ b\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{let}\ hypothetical\_release\_sequence\ actions\ lk\ mo\ a\ b = \\
\quad \text{is\_at\_atomic\_location } lk\ b\ \wedge \\
\quad ((b = a)\ \vee \\
\qquad (\text{rs\_element } a\ b\ \wedge\ a \xrightarrow{mo} b\ \wedge \\
\qquad\quad (\forall c{\in}actions.\ (a \xrightarrow{mo} c\ \wedge\ c \xrightarrow{mo} b)\ \rightarrow\ \text{rs\_element } a\ c)))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{let}\ hypothetical\_release\_sequence\_set\ actions\ lk\ mo = \\
\quad \{(a,\ b) | \forall a{\in}actions,\ b{\in}actions\ | \\
\qquad \text{hypothetical\_release\_sequence } actions\ lk\ mo\ a\ b\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{let}\ synchronizes\_with\ actions\ sb\ asw\ rf\ lo\ rs\ hrs\ a\ b = a \xrightarrow{asw} b\ \vee \\
\quad (\text{same\_location } a\ b\ \wedge\ a\ \in\ actions\ \wedge\ b\ \in\ actions\ \wedge \\
\qquad (\text{* mutex sync *}) \\
\qquad (\text{is\_unlock } a\ \wedge\ \text{is\_successful\_lock } b\ \wedge\ a \xrightarrow{lo} b)\ \vee \\
\qquad (\text{* rel/acq sync *}) \\
\qquad (\text{is\_release } a\ \wedge\ \text{is\_acquire } b\ \wedge\ \neg\ (\text{same\_thread } a\ b)\ \wedge \\
\qquad\quad (\exists c{\in}actions.\ a \xrightarrow{rs} c\ \wedge\ c \xrightarrow{rf} b))\ \vee \\
\qquad (\text{* fence sync *}) \\
\qquad (\neg\ (\text{same\_thread } a\ b)\ \wedge \\
\qquad\quad \text{is\_fence } a\ \wedge\ \text{is\_release } a\ \wedge\ \text{is\_fence } b\ \wedge\ \text{is\_acquire } b\ \wedge \\
\qquad\quad (\exists x{\in}actions,\ y{\in}actions.\ \text{same\_location } x\ y\ \wedge \\
\qquad\qquad \text{is\_atomic\_action } x\ \wedge\ \text{is\_atomic\_action } y\ \wedge\ \text{is\_write } x\ \wedge\ a \xrightarrow{sb} x\ \wedge\ y \xrightarrow{sb} b\ \wedge \\
\qquad\qquad (\exists z{\in}actions.\ x \xrightarrow{hrs} z\ \wedge\ z \xrightarrow{rf} y)))\ \vee \\
\qquad (\neg\ (\text{same\_thread } a\ b)\ \wedge \\
\qquad\quad \text{is\_fence } a\ \wedge\ \text{is\_release } a\ \wedge\ \text{is\_atomic\_action } b\ \wedge\ \text{is\_acquire } b\ \wedge \\
\qquad\quad (\exists x{\in}actions.\ \text{same\_location } x\ b\ \wedge
\end{array}
$$

```
        is_atomic_action x ∧ is_write x ∧ a -sb-> x ∧
        (∃z∈actions. x -hrs-> z ∧ z -rf-> b))) ∨
      (¬ (same_thread a b) ∧
        is_atomic_action a ∧ is_release a ∧
        is_fence b ∧ is_acquire b ∧
        (∃x∈actions. same_location a x ∧ is_atomic_action x ∧ x -sb-> b ∧
          (∃z∈actions. a -rs-> z ∧ z -rf-> x)))))

let synchronizes_with_set actions sb asw rf lo rs hrs =
  {(a, b)|∀a∈actions, b∈actions |
    synchronizes_with actions sb asw rf lo rs hrs a b}

let carries_a_dependency_to_set actions sb dd rf = tc ((rf ∩ sb) ∪ dd)

let dependency_ordered_before actions rf rs cad a d =
   a ∈ actions ∧ d ∈ actions ∧
   (∃b∈actions. is_release a ∧ is_consume b ∧
     (∃e∈actions. a -rs-> e ∧ e -rf-> b) ∧
     (b -cad-> d ∨ (b = d)))

let dependency_ordered_before_set actions rf rs cad =
  {(a, b)|∀a∈actions, b∈actions |
    dependency_ordered_before actions rf rs cad a b}

let compose R1 R2 =
  {(w, z)|∀(w, x)∈R1, (y, z)∈R2 | (x = y)}

let inter_thread_happens_before actions sb sw dob =
  let r = sw ∪ dob ∪ (compose sw sb) in
    tc (r ∪ (compose sb r))

let consistent_inter_thread_happens_before actions ithb =
  irrefl actions ithb

let happens_before actions sb ithb =
  sb ∪ ithb

(********************************************** *)
(* - 3.7 - Consistent SC and modification orders *)
(********************************************** *)

let all_sc_actions actions =
  {a|∀a∈actions | is_seq_cst a ∨ is_lock a ∨ is_unlock a}

let consistent_sc_order actions mo sc hb =
    let sc_happens_before = restrict_relation_set hb (all_sc_actions actions) in
    let sc_mod_order = restrict_relation_set mo (all_sc_actions actions) in
    strict_total_order_over (all_sc_actions actions) sc ∧
    sc_happens_before subset sc ∧
    sc_mod_order subset sc

let consistent_modification_order actions lk sb mo hb =
  (∀a∈actions, b∈actions. a -mo-> b → (same_location a b ∧ is_write a ∧ is_write b)) ∧
  (∀l∈locations_of actions.
    match lk l with
      Atomic →
        (let actions_at_l = {a|∀a∈actions | location_of a = Some l} in
          let writes_at_l = {a|∀a∈actions_at_l | is_write a} in
            strict_total_order_over writes_at_l (restrict_relation_set mo actions_at_l) ∧
```

```
            (* hb is a subset of mo at l *)
            restrict_relation_set hb writes_at_l subset mo)
        | _ ->
            (let actions_at_l = {a|∀a∈actions | location_of a = SOME l} in
            Set.is_empty (restrict_relation_set mo actions_at_l)) end)

(********************************************** *)
(* - 3.8 - Visible side effects *)
(********************************************** *)


let visible_side_effect actions hb a b = a -hb-> b ∧
    is_write a ∧ is_read b ∧ same_location a b ∧
    ¬ (∃c∈actions. ¬ (c = a) ∧ ¬ (c = b) ∧
          is_write c ∧ same_location c b ∧ a -hb-> c ∧ c -hb-> b)

let visible_side_effect_set actions sb hb =
    {(a, b)|∀(a, b)∈hb | visible_side_effect actions hb a b}

(********************************************** *)
(* - 3.9 - Undefined behaviour *)
(********************************************** *)

let indeterminate_reads actions rf =
    {b|∀b∈actions | is_read b ∧ ¬ (∃a∈actions. a -rf-> b)}

let unsequenced_races actions sb =
    {(a, b)|∀a∈actions, b∈actions |
        ¬ (a = b) ∧ same_location a b ∧ (is_write a ∨ is_write b) ∧
        same_thread a b ∧
        ¬ (a -sb-> b ∨ b -sb-> a)}

let data_races actions hb =
    {(a, b)|∀a∈actions, b∈actions |
        ¬ (a = b) ∧ same_location a b ∧ (is_write a ∨ is_write b) ∧
        ¬ (same_thread a b) ∧
        ¬ (is_atomic_action a ∧ is_atomic_action b) ∧
        ¬ (a -hb-> b ∨ b -hb-> a)}

let data_races' Xo Xw lo =
  let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
  let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
  let sw = synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf lo rs hrs in
  let cad = carries_a_dependency_to_set Xo.actions Xo.sb Xo.dd Xw.rf in
  let dob = dependency_ordered_before_set Xo.actions Xw.rf rs cad in
  let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
  let hb = happens_before Xo.actions Xo.sb ithb in
  data_races Xo.actions hb

let good_mutex_use actions lk sb lo a =
    let mutexes_at_l =
      {a'|∀a'∈actions | (is_successful_lock a' ∨ is_unlock a') ∧ (location_of a' = location_of a)}
    in
    let lock_order = restrict_relation_set lo mutexes_at_l in
    (* violated requirement: The calling thread shall own the mutex. *)
    (is_unlock a → (∃al∈actions.
            is_successful_lock al ∧ (location_of al = location_of a) ∧ al -sb-> a ∧
            adjacent_less_than lock_order actions al a)) ∧
```

```
    (* violated requirement: The calling thread does not own the mutex. *)
    (is_lock a  →
      ¬ (∃al∈actions.
          is_successful_lock al  ∧  (location_of a = location_of al)  ∧  same_thread a al ∧
          adjacent_less_than lock_order actions al a))

let bad_mutexes Xo lo =
  {a|∀a∈Xo.actions | ¬ (good_mutex_use Xo.actions Xo.lk Xo.sb lo a)}

let undefined_behaviour Xo Xw =
  ¬ (data_races' Xo Xw Xw.sc = {}) ∨
  ¬ (unsequenced_races Xo.actions Xo.sb = {}) ∨
  ¬ (indeterminate_reads Xo.actions Xw.rf = {}) ∨
  ¬ (bad_mutexes Xo Xw.sc = {})

(********************************************** *)
(* - 3.9 - Consistent reads from mapping *)
(********************************************** *)


let consistent_non_atomic_read_values actions lk rf vse =
    ∀b∈actions.
    (is_read b  ∧  is_at_non_atomic_location lk b)  →
      (if (∃a_vse∈actions. a_vse -vse-> b)
        then (∃a_vse∈actions. a_vse -vse-> b  ∧  a_vse -rf-> b)
        else ¬ (∃a∈actions. a -rf-> b))

let coherent_memory_use actions lk rf mo hb =
    (* CoRR *)
    (∀(x, a)∈rf, (y, b)∈rf.
      (a -hb-> b  ∧  same_location a b  ∧  is_at_atomic_location lk b)  →
      ((x = y)  ∨  x -mo-> y)) ∧
    (* CoWR *)
    (∀(a, b)∈hb, c∈actions.
      (c -rf-> b  ∧  is_write a  ∧  same_location a b  ∧  is_at_atomic_location lk b)  →
      ((c = a)  ∨  a -mo-> c)) ∧
    (* CoRW *)
    (∀(a, b)∈hb, c∈actions.
      (c -rf-> a  ∧  is_write b  ∧  same_location a b  ∧  is_at_atomic_location lk a)  →
      (c -mo-> b))

let rmw_atomicity actions rf mo =
    ∀(a, b)∈rf.
    is_atomic_rmw b  →  adjacent_less_than mo actions a b

let sc_reads_restricted actions rf sc mo hb =
    ∀(a, b)∈rf.
    is_seq_cst b  →
    (adjacent_less_than_such_that (fun c → is_write c  ∧  same_location b c) sc actions a b) ∨
    (¬ (is_seq_cst a) ∧
      (∀x∈actions.
        (adjacent_less_than_such_that (fun c → is_write c  ∧  same_location b c) sc actions x b)  →  ¬ (a -hb-> x)))

let sc_fences_heeded actions sb rf sc mo =
    (* fence restriction N3291 29.3p4 *)
    (∀a∈actions, (x, b)∈sb, y∈actions.
      (is_write a  ∧  is_fence x ∧
        adjacent_less_than sc actions a x ∧
```

```
      is_atomic_action b ∧ same_location a b ∧ y -rf-> b) →
    ((y = a) ∨ a -mo-> y)) ∧
  (* fence restriction N3291 29.3p5 *)
  (∀(a, x)∈sb, (y, b)∈rf.
    ((is_atomic_action a ∧ is_write a ∧
      is_fence x ∧ is_atomic_action b ∧ x -sc-> b ∧
      same_location a b) →
    ((y = a) ∨ a -mo-> y))) ∧
  (* fence restriction N3291 29.3p6 *)
  (∀(a, x)∈sb, (y, b)∈sb, z∈actions.
    (is_atomic_action a ∧ is_write a ∧
      is_fence x ∧ is_fence y ∧ x -sc-> y ∧
      is_atomic_action b ∧ same_location a b ∧ z -rf-> b) →
    ((z = a) ∨ a -mo-> z)) ∧
  (* SC fences impose mo N3291 29.3p7 *)
  (∀(a, x)∈sb, (y, b)∈sb.
    (is_atomic_action a ∧ is_write a ∧
      is_atomic_action b ∧ is_write b ∧
      is_fence x ∧ is_fence y ∧ x -sc-> y ∧
      same_location a b → a -mo-> b)) ∧
  (* SC fences impose mo N3291 29.3p7, w collapsed first write*)
  (∀a∈actions, (y, b)∈sb.
    (is_atomic_action a ∧ is_write a ∧
      is_fence y ∧ a -sc-> y ∧
      is_atomic_action b ∧ is_write b ∧
      same_location a b → a -mo-> b)) ∧
  (* SC fences impose mo N3291 29.3p7, w collapsed second write*)
  (∀(a, x)∈sb, b∈actions.
    (is_atomic_action a ∧ is_write a ∧
      is_fence x ∧ is_atomic_action b ∧ is_write b ∧ x -sc-> b ∧
      same_location a b → a -mo-> b))

let no_vsse_consistent_atomic_read_values actions lk rf hb vse =
  ∀b∈actions.
  (is_read b ∧ is_at_atomic_location lk b) →
    (if (∃a_vse∈actions. a_vse -vse-> b)
      then (∃a∈actions. (a -rf-> b) ∧ ¬ (b -hb-> a))
      else ¬ (∃a∈actions. a -rf-> b))

let no_vsse_consistent_reads_from_mapping actions lk sb rf sc mo hb vse =
  consistent_non_atomic_read_values actions lk rf vse ∧
  no_vsse_consistent_atomic_read_values actions lk rf hb vse ∧
  coherent_memory_use actions lk rf mo hb ∧
  rmw_atomicity actions rf mo ∧
  sc_reads_restricted actions rf sc mo hb ∧
  sc_fences_heeded actions sb rf sc mo

(********************************************** *)
(* 3.11 - Consistent execution *)
(********************************************** *)

(* This simplification has ben verified equivalent to the model in section 3 using the HOL theorem prover. It removes the complicated notion of VSSE's, whose force is covered by the coherence requirements. For those looking to work with C or C++ concurrency, this is the preferred model. *)

let no_vsse_consistent_execution Xo Xw =
```

```
  well_formed_threads Xo.actions Xo.threads Xo.lk Xo.sb Xo.asw Xo.dd Xo.cd ∧
  (let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
   let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
   let sw = synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf Xw.sc rs hrs in
   let cad = carries_a_dependency_to_set Xo.actions Xo.sb Xo.dd Xw.rf in
   let dob = dependency_ordered_before_set Xo.actions Xw.rf rs cad in
   let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
   let hb = happens_before Xo.actions Xo.sb ithb in
   let vse = visible_side_effect_set Xo.actions Xo.sb hb in
   consistent_locks Xo.actions Xw.sc hb ∧
   consistent_inter_thread_happens_before Xo.actions ithb ∧
   consistent_sc_order Xo.actions Xw.mo Xw.sc hb ∧
   consistent_modification_order Xo.actions Xo.lk Xo.sb Xw.mo hb ∧
   well_formed_reads_from_mapping Xo.actions Xo.lk Xw.rf ∧
   no_vsse_consistent_reads_from_mapping Xo.actions Xo.lk Xo.sb Xw.rf Xw.sc Xw.mo hb vse)

(********************************************** *)
(* - 3.12 - Preferred model top level judgement *)
(********************************************** *)


let no_vsse_cmm opsem (p : PROGRAM) =
  let pre_executions =
    {(Xopsem, Xwitness) | opsem p Xopsem ∧ no_vsse_consistent_execution Xopsem Xwitness} in
  if ∃(Xo, Xw)∈pre_executions.
    undefined_behaviour Xo Xw
  then {(Xo, Xw) | true}
  else pre_executions

(********************************************** *)
(* 4 - Standard C/C++ model *)
(********************************************** *)

(* The following definitions make up the memory model described by the 2011 standard. It was constructed in
discussion with the standardisation comittee. *)

let visible_sequence_of_side_effects_tail actions mo hb vsse_head b =
    {c|∀c∈actions | vsse_head -mo-> c ∧ ¬ (b -hb-> c) ∧
      (∀a∈actions.
          (vsse_head -mo-> a ∧ a -mo-> c) → ¬ (b -hb-> a))}

(* visible sequences of side effects have been proven redundant. See the simpler model in section 3. *)

let visible_sequence_of_side_effects actions lk mo hb vsse_head b =
      (b, if is_at_atomic_location lk b then
            {vsse_head} ∪
            visible_sequence_of_side_effects_tail actions mo hb vsse_head b
          else
            {})

let visible_sequences_of_side_effects_set actions lk mo hb vse =
{visible_sequence_of_side_effects actions lk mo hb vsse_head b|
  ∀vsse_head∈actions, b∈actions |
  is_at_atomic_location lk b ∧ is_read b ∧
  (vsse_head -vse-> b)}

let consistent_atomic_read_values actions lk rf vsses =
    ∀b∈actions.
    (is_read b ∧ is_at_atomic_location lk b) →
```

```
    (if (∃(b', v)∈vsses. b = b')
      then (∃(b', v)∈vsses. b = b' ∧
            (∃c∈v. c -rf-> b))
      else ¬ (∃a∈actions. a -rf-> b))

let consistent_reads_from_mapping actions lk sb rf sc mo hb vse vsses =
  consistent_non_atomic_read_values actions lk rf vse ∧
  consistent_atomic_read_values actions lk rf vsses ∧
  coherent_memory_use actions lk rf mo hb ∧
  rmw_atomicity actions rf mo ∧
  sc_reads_restricted actions rf sc mo hb ∧
  sc_fences_heeded actions sb rf sc mo

let consistent_execution Xo Xw =
  well_formed_threads Xo.actions Xo.threads Xo.lk Xo.sb Xo.asw Xo.dd Xo.cd ∧
  (let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
   let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
   let sw = synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf Xw.sc rs hrs in
   let cad = carries_a_dependency_to_set Xo.actions Xo.sb Xo.dd Xw.rf in
   let dob = dependency_ordered_before_set Xo.actions Xw.rf rs cad in
   let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
   let hb = happens_before Xo.actions Xo.sb ithb in
   let vse = visible_side_effect_set Xo.actions Xo.sb hb in
   let vsses = visible_sequences_of_side_effects_set Xo.actions Xo.lk Xw.mo hb vse in
   consistent_locks Xo.actions Xw.sc hb ∧
   consistent_inter_thread_happens_before Xo.actions ithb ∧
   consistent_sc_order Xo.actions Xw.mo Xw.sc hb ∧
   consistent_modification_order Xo.actions Xo.lk Xo.sb Xw.mo hb ∧
   well_formed_reads_from_mapping Xo.actions Xo.lk Xw.rf ∧
   consistent_reads_from_mapping Xo.actions Xo.lk Xo.sb Xw.rf Xw.sc Xw.mo hb vse vsses)

(********************************************** *)
(* - 4.1 - Standard model top level judgement *)
(********************************************** *)


let cmm opsem (p : PROGRAM) =
  let pre_executions = {(Xopsem, Xwitness) | opsem p Xopsem ∧ consistent_execution Xopsem Xwitness} in
  if ∃(Xo, Xw)∈pre_executions.
    undefined_behaviour Xo Xw
  then {(Xo, Xw) | true}
  else pre_executions

(********************************************** *)
(* 5 - Model with seperate lock order *)
(********************************************** *)

(* A version of the no VSSE model with a seperate lock order. *)


let no_vsse_seperate_lo_consistent_execution Xo Xw lo =
  well_formed_threads Xo.actions Xo.threads Xo.lk Xo.sb Xo.asw Xo.dd Xo.cd ∧
  (let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
   let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
   let sw = synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf lo rs hrs in
   let cad = carries_a_dependency_to_set Xo.actions Xo.sb Xo.dd Xw.rf in
   let dob = dependency_ordered_before_set Xo.actions Xw.rf rs cad in
   let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
   let hb = happens_before Xo.actions Xo.sb ithb in
```

```
    let vse = visible_side_effect_set Xo.actions Xo.sb hb in
    consistent_locks Xo.actions lo hb ∧
    consistent_inter_thread_happens_before Xo.actions ithb ∧
    consistent_sc_order Xo.actions Xw.mo Xw.sc hb ∧
    consistent_modification_order Xo.actions Xo.lk Xo.sb Xw.mo hb ∧
    well_formed_reads_from_mapping Xo.actions Xo.lk Xw.rf ∧
    no_vsse_consistent_reads_from_mapping Xo.actions Xo.lk Xo.sb Xw.rf Xw.sc Xw.mo hb vse)

let no_vsse_seperate_lo_undefined_behaviour Xo Xw lo =
  ¬ (data_races' Xo Xw lo = {}) ∨
  ¬ (unsequenced_races Xo.actions Xo.sb = {}) ∨
  ¬ (indeterminate_reads Xo.actions Xw.rf = {}) ∨
  ¬ (bad_mutexes Xo lo = {})

(********************************************** *)
(* - 5.1 - Seperate lock order top level judgement *)
(********************************************** *)


let no_vsse_seperate_lo_cmm opsem (p : PROGRAM) =
  let pre_executions =
    {(Xopsem, (Xwitness, lo)) |
      opsem p Xopsem ∧ no_vsse_seperate_lo_consistent_execution Xopsem Xwitness lo} in
  if ∃(Xo, (Xw, lo))∈pre_executions.
    no_vsse_seperate_lo_undefined_behaviour Xo Xw lo
  then {(Xo, (Xw, lo)) | true}
  else pre_executions

(********************************************** *)
(* 6 - Model with per-location lock orders *)
(********************************************** *)

(* This model uses per location lock orders rather than one shared one. *)


let multi_lo_consistent_locks actions lk lo hb =
    let mutex_actions = {a|∀a∈actions | (is_lock a ∨ is_unlock a)} in
    let lo_happens_before = restrict_relation_set hb mutex_actions in
    lo_happens_before subset lo ∧
    (∀(a, c)∈lo. is_successful_lock a ∧ is_successful_lock c ∧ same_location a c
      →
      (∃b∈actions. same_location a b ∧ is_unlock b ∧ a -lo-> b ∧ b -lo-> c)) ∧
      ∀l∈locations_of actions.
        let actions_at_l = {a|∀a∈actions | location_of a = SOME l} in
        match lk l with
          MUTEX →
            strict_total_order_over actions_at_l (restrict_relation_set lo actions_at_l)
        | _ → Set.is_empty (restrict_relation_set lo actions_at_l) end

let no_vsse_multi_lo_consistent_execution Xo Xw lo =
  well_formed_threads Xo.actions Xo.threads Xo.lk Xo.sb Xo.asw Xo.dd Xo.cd ∧
  (let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
    let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
    let sw = synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf lo rs hrs in
    let cad = carries_a_dependency_to_set Xo.actions Xo.sb Xo.dd Xw.rf in
    let dob = dependency_ordered_before_set Xo.actions Xw.rf rs cad in
    let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
    let hb = happens_before Xo.actions Xo.sb ithb in
    let vse = visible_side_effect_set Xo.actions Xo.sb hb in
```

multi_lo_consistent_locks $Xo.actions\ Xo.lk\ lo\ hb \wedge$
consistent_inter_thread_happens_before $Xo.actions\ ithb \wedge$
consistent_sc_order $Xo.actions\ Xw.mo\ Xw.sc\ hb \wedge$
consistent_modification_order $Xo.actions\ Xo.lk\ Xo.sb\ Xw.mo\ hb \wedge$
well_formed_reads_from_mapping $Xo.actions\ Xo.lk\ Xw.rf \wedge$
no_vsse_consistent_reads_from_mapping $Xo.actions\ Xo.lk\ Xo.sb\ Xw.rf\ Xw.sc\ Xw.mo\ hb\ vse$)

(********** *)
(* - 6.1 - per-location lock order top level judgement *)
(********** *)

let $no\_vsse\_multi\_lo\_cmm\ opsem\ (p:$ PROGRAM$) =$
  let $pre\_executions =$
    $\{(Xopsem, (Xwitness, lo)) \mid$
      $opsem\ p\ Xopsem \wedge$ no_vsse_multi_lo_consistent_execution $Xopsem\ Xwitness\ lo\}$ in
  if $\exists(Xo, (Xw, lo))\in pre\_executions.$
    no_vsse_seperate_lo_undefined_behaviour $Xo\ Xw\ lo$
  then $\{(Xo, (Xw, lo)) \mid$ true$\}$
  else $pre\_executions$

(********** *)
(* 7 - Model with single step mutex synchronisation *)
(********** *)

(* This model creates synchronizes-with edges from each unlock to the next lock at the same location, rather than all subsequent ones. *)

let $lo\_single\_synchronizes\_with\ actions\ sb\ asw\ rf\ lo\ rs\ hrs\ a\ b = a \xrightarrow{asw} b \vee$
  (same_location $a\ b \wedge a \in actions \wedge b \in actions \wedge$
    ((* mutex sync *)
      (is_unlock $a \wedge$ is_successful_lock $b \wedge a \xrightarrow{lo} b \wedge \neg (\exists c\in actions.\ a \xrightarrow{lo} c \wedge c \xrightarrow{lo} b)) \vee$
      (* rel/acq sync *)
      (is_release $a \wedge$ is_acquire $b \wedge \neg$ (same_thread $a\ b) \wedge$
        $(\exists c\in actions.\ a \xrightarrow{rs} c \wedge c \xrightarrow{rf} b)) \vee$
      (* fence sync *)
      ($\neg$ (same_thread $a\ b) \wedge$
        is_fence $a \wedge$ is_release $a \wedge$ is_fence $b \wedge$ is_acquire $b \wedge$
        $(\exists x\in actions, y\in actions.$ same_location $x\ y \wedge$
          is_atomic_action $x \wedge$ is_atomic_action $y \wedge$ is_write $x \wedge a \xrightarrow{sb} x \wedge y \xrightarrow{sb} b \wedge$
          $(\exists z\in actions.\ x \xrightarrow{hrs} z \wedge z \xrightarrow{rf} y))) \vee$
      ($\neg$ (same_thread $a\ b) \wedge$
        is_fence $a \wedge$ is_release $a \wedge$ is_atomic_action $b \wedge$ is_acquire $b \wedge$
        $(\exists x\in actions.$ same_location $x\ b \wedge$
          is_atomic_action $x \wedge$ is_write $x \wedge a \xrightarrow{sb} x \wedge$
          $(\exists z\in actions.\ x \xrightarrow{hrs} z \wedge z \xrightarrow{rf} b))) \vee$
      ($\neg$ (same_thread $a\ b) \wedge$
        is_atomic_action $a \wedge$ is_release $a \wedge$
        is_fence $b \wedge$ is_acquire $b \wedge$
        $(\exists x\in actions.$ same_location $a\ x \wedge$ is_atomic_action $x \wedge x \xrightarrow{sb} b \wedge$
          $(\exists z\in actions.\ a \xrightarrow{rs} z \wedge z \xrightarrow{rf} x)))))$

let $lo\_single\_synchronizes\_with\_set\ actions\ sb\ asw\ rf\ lo\ rs\ hrs =$
  $\{(a, b) \mid \forall a\in actions, b\in actions \mid$
    lo_single_synchronizes_with $actions\ sb\ asw\ rf\ lo\ rs\ hrs\ a\ b\}$

let $no\_vsse\_multi\_lo\_single\_sw\_consistent\_execution\ Xo\ Xw\ lo =$

```
  well_formed_threads Xo.actions Xo.threads Xo.lk Xo.sb Xo.asw Xo.dd Xo.cd ∧
  (let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
   let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
   let sw = lo_single_synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf lo rs hrs in
   let cad = carries_a_dependency_to_set Xo.actions Xo.sb Xo.dd Xw.rf in
   let dob = dependency_ordered_before_set Xo.actions Xw.rf rs cad in
   let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
   let hb = happens_before Xo.actions Xo.sb ithb in
   let vse = visible_side_effect_set Xo.actions Xo.sb hb in
   multi_lo_consistent_locks Xo.actions Xo.lk lo hb ∧
   consistent_inter_thread_happens_before Xo.actions ithb ∧
   consistent_sc_order Xo.actions Xw.mo Xw.sc hb ∧
   consistent_modification_order Xo.actions Xo.lk Xo.sb Xw.mo hb ∧
   well_formed_reads_from_mapping Xo.actions Xo.lk Xw.rf ∧
   no_vsse_consistent_reads_from_mapping Xo.actions Xo.lk Xo.sb Xw.rf Xw.sc Xw.mo hb vse)

let los_single_sw_data_races' Xo Xw lo =
  let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
  let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
  let sw = lo_single_synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf lo rs hrs in
  let cad = carries_a_dependency_to_set Xo.actions Xo.sb Xo.dd Xw.rf in
  let dob = dependency_ordered_before_set Xo.actions Xw.rf rs cad in
  let ithb = inter_thread_happens_before Xo.actions Xo.sb sw dob in
  let hb = happens_before Xo.actions Xo.sb ithb in
  data_races Xo.actions hb

let no_vsse_multi_lo_single_sw_undefined_behaviour Xo Xw lo =
  ¬ (los_single_sw_data_races' Xo Xw lo = {}) ∨
  ¬ (unsequenced_races Xo.actions Xo.sb = {}) ∨
  ¬ (indeterminate_reads Xo.actions Xw.rf = {}) ∨
  ¬ (bad_mutexes Xo lo = {})

(********************************************** *)
(* - 7.1 - single step mutex synchronisation top level judgement *)
(********************************************** *)


let no_vsse_multi_lo_single_sw_cmm opsem (p : PROGRAM) =
  let pre_executions =
    {(Xopsem, (Xwitness, lo)) |
      opsem p Xopsem ∧ no_vsse_multi_lo_single_sw_consistent_execution Xopsem Xwitness lo} in
  if ∃(Xo, (Xw, lo))∈pre_executions.
    no_vsse_multi_lo_single_sw_undefined_behaviour Xo Xw lo
  then {(Xo, (Xw, lo)) | true}
  else pre_executions

(********************************************** *)
(* 8 - Model simplified for programs without consumes *)
(********************************************** *)

(* This model is simplified for use with programs that don't use consume memory orders. Happens-before is transitive.
*)


let no_vsse_consume_happens_before actions sb sw =
    tc (sb ∪ sw)

let no_vsse_consume_consistent_happens_before actions hb =
    irrefl actions hb
```

```
let no_vsse_consume_consistent_execution Xo Xw =
  well_formed_threads Xo.actions Xo.threads Xo.lk Xo.sb Xo.asw Xo.dd Xo.cd ∧
  (let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
    let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
    let sw = synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf Xw.sc rs hrs in
    let hb = no_vsse_consume_happens_before Xo.actions Xo.sb sw in
    let vse = visible_side_effect_set Xo.actions Xo.sb hb in
    consistent_locks Xo.actions Xw.sc hb ∧
    no_vsse_consume_consistent_happens_before Xo.actions hb ∧
    consistent_sc_order Xo.actions Xw.mo Xw.sc hb ∧
    consistent_modification_order Xo.actions Xo.lk Xo.sb Xw.mo hb ∧
    well_formed_reads_from_mapping Xo.actions Xo.lk Xw.rf ∧
    no_vsse_consistent_reads_from_mapping Xo.actions Xo.lk Xo.sb Xw.rf Xw.sc Xw.mo hb vse)

let no_vsse_consume_data_races' Xo Xw lo =
  let rs = release_sequence_set Xo.actions Xo.lk Xw.mo in
  let hrs = hypothetical_release_sequence_set Xo.actions Xo.lk Xw.mo in
  let sw = synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf lo rs hrs in
  let hb = no_vsse_consume_happens_before Xo.actions Xo.sb sw in
  data_races Xo.actions hb

let no_vsse_consume_undefined_behaviour Xo Xw =
  ¬ (no_vsse_consume_data_races' Xo Xw Xw.sc = {}) ∨
  ¬ (unsequenced_races Xo.actions Xo.sb = {}) ∨
  ¬ (indeterminate_reads Xo.actions Xw.rf = {}) ∨
  ¬ (bad_mutexes Xo Xw.sc = {})

(********************************************** *)
(* - 8.1 - No consume top level judgement *)
(********************************************** *)


let no_vsse_consume_cmm opsem (p : PROGRAM) =
  let pre_executions =
    {(Xopsem, Xwitness) |
      opsem p Xopsem ∧ no_vsse_consume_consistent_execution Xopsem Xwitness} in
  if ∃(Xo, Xw)∈pre_executions.
    no_vsse_consume_undefined_behaviour Xo Xw
  then {(Xo, Xw) | true}
  else pre_executions

(********************************************** *)
(* 9 - Model simplified for programs without consumes or relaxed *)
(********************************************** *)

(* Without relaxed, can release sequences go? Unfortunately not. This model is NOT equvalent. *)


let no_vsse_consume_relaxed_synchronizes_with actions sb asw rf lo a b = a -asw-> b ∨
    (same_location a b ∧ a ∈ actions ∧ b ∈ actions ∧
      ((* mutex sync *)
        (is_unlock a ∧ is_lock b ∧ a -lo-> b) ∨
        (* rel/acq sync *)
        (is_release a ∧ is_acquire b ∧ ¬ (same_thread a b) ∧ a -rf-> b)))

let no_vsse_consume_relaxed_synchronizes_with_set actions sb asw rf lo =
    {(a, b)|∀a∈actions, b∈actions |
      no_vsse_consume_relaxed_synchronizes_with actions sb asw rf lo a b}
```

```
let no_vsse_consume_relaxed_consistent_execution Xo Xw =
  well_formed_threads Xo.actions Xo.threads Xo.lk Xo.sb Xo.asw Xo.dd Xo.cd ∧
  (let sw = no_vsse_consume_relaxed_synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf Xw.sc in
    let hb = no_vsse_consume_happens_before Xo.actions Xo.sb sw in
    let vse = visible_side_effect_set Xo.actions Xo.sb hb in
    consistent_locks Xo.actions Xw.sc hb ∧
    no_vsse_consume_consistent_happens_before Xo.actions hb ∧
    consistent_sc_order Xo.actions Xw.mo Xw.sc hb ∧
    consistent_modification_order Xo.actions Xo.lk Xo.sb Xw.mo hb ∧
    well_formed_reads_from_mapping Xo.actions Xo.lk Xw.rf ∧
    no_vsse_consistent_reads_from_mapping Xo.actions Xo.lk Xo.sb Xw.rf Xw.sc Xw.mo hb vse)

let no_vsse_consume_relaxed_data_races' Xo Xw lo =
  let sw = no_vsse_consume_relaxed_synchronizes_with_set Xo.actions Xo.sb Xo.asw Xw.rf lo in
  let hb = no_vsse_consume_happens_before Xo.actions Xo.sb sw in
  data_races Xo.actions hb

let no_vsse_consume_relaxed_undefined_behaviour Xo Xw =
  ¬ (no_vsse_consume_relaxed_data_races' Xo Xw Xw.sc = {}) ∨
  ¬ (unsequenced_races Xo.actions Xo.sb = {}) ∨
  ¬ (indeterminate_reads Xo.actions Xw.rf = {}) ∨
  ¬ (bad_mutexes Xo Xw.sc = {})

(********************************************** *)
(* - 9.1 - No consume or relaxed top level judgement *)
(********************************************** *)


let no_vsse_consume_relaxed_cmm opsem (p : PROGRAM) =
  let pre_executions =
    {(Xopsem, Xwitness) |
      opsem p Xopsem ∧ no_vsse_consume_relaxed_consistent_execution Xopsem Xwitness} in
  if ∃(Xo, Xw)∈pre_executions.
    no_vsse_consume_relaxed_undefined_behaviour Xo Xw
  then {(Xo, Xw) | true}
  else pre_executions

(********************************************** *)
(* 10 - Model simplified for programs without consumes, relaxed, acquires or releases *)
(********************************************** *)

let consistent_total_order actions sb asw tot =
  strict_total_order_over actions tot ∧
  sb subset tot ∧
  asw subset tot

let tot_consistent_reads_from_mapping actions lk rf tot =
  (∀b∈actions.
    (is_read b) →
    (let writes_at_same_location = {a|∀a∈actions | (same_location a b) ∧ is_write a} in
      (if (∃a∈actions.
          adjacent_less_than (restrict_relation_set tot (writes_at_same_location ∪ {b})) actions a b)
        then (∃a∈actions.
          (a -rf-> b) ∧
          adjacent_less_than (restrict_relation_set tot (writes_at_same_location ∪ {b})) actions a b)
        else ¬ (∃a∈actions. a -rf-> b)))))

let tot_consistent_execution Xo rf tot =
  let lo = restrict_relation_set tot {a|∀a∈Xo.actions | is_lock a ∨ is_unlock a} in
```

```
  well_formed_threads Xo.actions Xo.threads Xo.lk Xo.sb Xo.asw Xo.dd Xo.cd ∧
  consistent_total_order Xo.actions Xo.sb Xo.asw tot ∧
  consistent_locks Xo.actions lo tot ∧
  tot_consistent_reads_from_mapping Xo.actions Xo.lk rf tot ∧
  well_formed_reads_from_mapping Xo.actions Xo.lk rf

let tot_hb_data_races Xo rf tot =
  let sc = tot ∩ {(a, b)|∀a∈Xo.actions, b∈Xo.actions | is_seq_cst a ∧ is_seq_cst b} in
  let mo = tot ∩ {(a, b)|∀a∈Xo.actions, b∈Xo.actions | (same_location a b) ∧ is_write a ∧ is_write b} in
  let sw = no_vsse_consume_relaxed_synchronizes_with_set Xo.actions Xo.sb Xo.asw rf tot in
  let hb = no_vsse_consume_happens_before Xo.actions Xo.sb sw in
  data_races Xo.actions hb

let tot_data_races actions tot =
    {(a, b)|∀a∈actions, b∈actions |
       ¬ (a = b) ∧ same_location a b ∧ (is_write a ∨ is_write b) ∧
       ¬ (same_thread a b) ∧
       ¬ (is_atomic_action a ∧ is_atomic_action b) ∧
       (adjacent_less_than tot actions a b ∨ adjacent_less_than tot actions b a)}

let tot_undefined_behaviour Xo rf tot =
  ¬ (tot_hb_data_races Xo rf tot = {}) ∨
  ¬ (unsequenced_races Xo.actions Xo.sb = {}) ∨
  ¬ (indeterminate_reads Xo.actions rf = {}) ∨
  ¬ (bad_mutexes Xo tot = {})

(********************************************** *)
(* - 10.1 - No consume, relaxed, acquire or release top level judgement *)
(********************************************** *)


let tot_cmm opsem (p : PROGRAM) =
  let pre_executions = {(Xopsem, (rf, tot)) | opsem p Xopsem ∧ tot_consistent_execution Xopsem rf tot} in
  if ∃(Xo, (rf, tot))∈pre_executions.
    tot_undefined_behaviour Xo rf tot
  then {(Xo, (rf, tot)) | true}
  else pre_executions
```