A Generic Formalised Framework for Reasoning About Weak Memory Models

Jade Alglave, Assia Mahboubi

▶ **To cite this version:**

Jade Alglave, Assia Mahboubi. A Generic Formalised Framework for Reasoning About Weak Memory Models. 2011. inria-00604656

HAL Id: inria-00604656
https://inria.hal.science/inria-00604656

Preprint submitted on 29 Jun 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Generic Formalised Framework for Reasoning About Weak Memory Models

Jade Alglave[1,2] and Assia Mahboubi[1]

[1] INRIA [2] Oxford University

Abstract. This paper describes Coq libraries devoted to the semantic of relaxed memory models. These libraries formalise a framework which covers a large class of industrial models. Implementing this framework inside a proof assistant has significantly helped improving its design and crafting the most concise and relevant specifications. Similarly the use of a proof assistant has been instrumental in the study of the semantic of synchronisation primitives, which we illustrate by the formal proof of a barrier placement theorem. We explain the choices we made to re-design our Coq libraries, and in particular what we gained from adopting a small-scale reflection methodology.

1 Introduction

Although multiprocessors are an industry standard nowadays, programming and reasoning about such systems remains a challenge. Concurrent programs running on modern multiprocessors exhibit subtle behaviours, making them hard to understand and to debug. Indeed, the order in which the read instructions (*e.g.* loads) and write instructions (*e.g.* stores) of a given program are actually executed may no longer be intuitive, *i.e.* may no longer follow the Sequential Consistency (SC) model [16]. The SC model ensures that a program behaves as an interleaving of the different instructions involved by the different threads or processors. However, modern multiprocessors (*e.g.* x86 or Power) allow optimisations such as *instruction reordering*, *store buffering* or *write atomicity relaxation* [2], leading to *weak memory models*, which are no more SC.

To understand the executions a concurrent program leads to, we need to state precisely what are the guarantees provided by a given memory model. In fact, a memory model should define what value a read instruction to a shared-memory location can read. For some memory models, such as x86 [15] and Power [1], the vendor documentation is in informal prose, thus ambiguous. Hence we need to formalise these models in precise and rigorous mathematics, as already highlighted by previous work [23, 10].

In [6], the first author and others proposed a framework for defining and reasoning about a family of memory models, comprising SC, the Sparc hierarchy [25, 26], *i.e.* TSO (which is x86's model [21]), PSO and RMO, Alpha [7], and a significant fragment of Power [1]. The goal was to define formally whether a given execution of a concurrent program is valid on one of these models.

This framework describes an execution of a program *via* partial orders over the memory events yielded by the instructions of the program. These partial orders model for example the program order per thread, the communication through memory (*e.g.* which read reads from which write), and the orderings induced by the *memory barriers*, which are special instructions provided by architectures to prevent some of the aforementioned optimisations. The validity of an execution boils down to the acyclicity of several combinations of these partial orders. This formalisation is generic, in the sense that it allows to generalise or reuse from one model to another the subtle reasonings required by weak memory models. Thus, as exposed in [5], it becomes possible to obtain generic results on the semantics of synchronisation primitives in a program running on a model of this framework, to force this program to behave as if running on SC.

Yet, as in many other areas of program verification, the proofs involved in this formalisation are tedious and intricate, even if they require a relatively small, essentially combinatorics, mathematical background and even if the mathematical content of the proofs is most often uninteresting. We believe the use of a proof assistant to be crucial for managing these investigations, both to craft the most concise specifications of the models and to provide a high guarantee on the correctness of the proofs and the consistency of the framework. The latter has been demonstrated by impressive previous pieces of work in the formalisation of semantics of instruction sets [19, 11, 12] or compiler issues [17, 24].

In [6, 5], the authors present the theoretical results they have obtained and the testing tool they have developed, mentioning that all these results have been formalised in the Coq [27, 8] system without describing their libraries. In the constructive type theory underlying Coq, the main issue raised by such a formalisation is to find a convenient representation for finite sets and to benefit from a body of formalised theory of boolean binary relations. The formalisation mentioned in [6, 5] is based on the standard library of Coq augmented with the excluded middle axiom. In particular, it relies on the generic library on sets which offers little support for our needs. These naive choices actually revealed inappropriate, and the code is now difficult both to maintain and to extend.

The purpose of this paper is to present our ongoing rewriting of this formalisation, which covers so far the theoretical results described in [6]. The design of the code has been improved along two directions: first we stand on a better library on finite sets, second we use a small scale reflection methodology [13], as coined by G. Gonthier after his formalisation of the proof of the Four Colour Theorem. The first issue could have been addressed by choosing a better suited library in the standard distribution of the Coq system. The second point consists in choosing carefully the data-structures of the formalised objects so that proofs can best benefit from the computational abilities of the Coq type theory. We get rid of the excluded middle axiom by adopting boolean specifications, which significantly eases the formalisation.

Since the last aspect had a positive impact on our formalisation, we used the SSReflect extension [14] to the Coq system which provides a set of tactics dedicated to this methodology. We also benefited from the Coq libraries de-

veloped by the Mathematical Component project [1], which include in particular a theory of finite sets, finite graphs, and boolean relations. We have extended the material already available in the existing Mathematical Component libraries with some extra results needed for the present work, like a pigeon-hole lemma and the formal correctness proof of a topological sorting procedure.

We first briefly overview in section 2 the data-structures that we adopted. In section 3, we present the formalisation of the main objects of our study: event structures, executions and architectures. In section 4, we show how to encode a memory model in this framework. Section 5 presents the formal proof of the main theorem of [6] on barrier placement. In section 6, we conclude by presenting first some related work, then some directions of future work. The old implementation and our ongoing development are available online[2].

2 Boolean relations, finite sets, pigeon hole arguments

We present here the features of the Mathematical Component library that we rely on in the present work. Thus, most of the material is not new, except the pigeon-hole lemma—suggested by C. Cohen—and the correctness of the topological sorting procedure.

Boolean reflection. In the standard Coq related literature, *reflection* usually refers to a proof methodology, where a symbolic representation—usually an inductive type—models a fragment of the inhabitants of a given logical type. This symbolic representation then becomes the input of some decision or simplification procedure and enables relatively large scale computations. Most often, this symbolic representation is hidden to the user, and translated back to its logical form once the desired computation has been performed. By contrast, the small-scale reflection [13] leaves the symbolic representations explicit in the definitions and lemmas, and the translations between symbolic and logical worlds should be locally and explicitly performed by the user when needed in the proof.

Boolean reflection is an instance of small-scale reflection, where the `bool` data type reflects the fragment of the Coq `Prop` sort on which the excluded middle holds. Thus, any statement whose truth value can be effectively computed should be modeled by a boolean predicate, coerced to the `Prop` sort. The excluded middle principle on a boolean statement boils down to case analysis on the value of the boolean, and equivalences between boolean statements are boolean equalities. Hence, the simplification of large boolean conjunctions or disjunctions becomes very fluid, whereas it otherwise imposes a tedious splitting and casing bureaucracy. Rewriting becomes prominent in the proof scripts: we replace subparts of the statements by simpler, equivalent, boolean expressions, up to the point they get an explicit value—`true` or `false`—then partially evaluate the whole boolean expression.

[1] `http://www.msr-inria.inria.fr/math-components`
[2] `http://www.comlab.ox.ac.uk/people/jade.alglave/wmm/`

The current standard tactic language of the Coq system actually provides little support for small scale methodology, specially regarding the pervasive translations between logical and boolean worlds. For this reason, the present development makes use of the SSReflect tactic extension language distributed with the Mathematical Component libraries.

In what follows, `(pred T)` is the type of boolean unary predicates on the arbitrary type `T` and `(rel T)` is the type of boolean binary relations on `T`. We use the notations provided by the library; for example, `[pred x : T | E]` denotes the predicate `(fun x => E)` and `[rel x y | E]` denotes the relation `(fun x y => E)`. The notation `(relU r s)` is the union of two relations operating on the same type, and `(subrel r s)` denotes the (non-boolean) property that the relation `r` is included in the relation `s`.

Equality types, finite types, finite sets. We call *equality type* a type equipped with a boolean comparison operator. This operator, denoted by a generic infix `(_ == _)` notation, should be proved equivalent to the default structural `Prop` equality provided by Coq on this type, denoted `(_ = _)`. In this paper, we will in fact only work with data (values, locations, processor indexes,...) modeled by types with a finite, abstract and arbitrary large number of inhabitants, called *the size of the universe*. The Mathematical Component distribution defines a finite type `(T : finType)` by the duplicate free enumeration of its inhabitants. A finite ordinal `('I_n : finType)`, which is the type of natural numbers smaller than `n`, is a typical example of such finite type. A finite type can be canonically equipped with a structure of equality type. The distributed library provides a comprehensive corpus of operators and formalised theory for these finite types, which we introduce when necessary. Sets of elements on a finite type carrier `T` have the type `{set T}` (see [13] for more details).

Finite types benefit constructively from classical logic and choice principles, since comparison and testing boolean properties become effective on such types. Thus, properties arbitrarily quantified on a finite type can be tested, hence modeled by boolean predicates. The notation `(forallb x : T, P)` denotes the *boolean* proposition: *for all* `(x : T)`, *the boolean property* `P` *holds*. The notation `(existsb x : T, P)` refers to the existential boolean quantifier. The classical lemmas on the negation of quantified formulas become provable, stated as an equality between two boolean values:

```
Lemma negb_forall (T : finType)(P : pred T) :
  ~~ (forallb x, P x) = (existsb x, ~~ P x).
```

Equivalence lemmas allow to switch back and forth between the boolean quantifiers and the usual logical quantifiers in the `Prop` sort when possible and needed.

When a function `(f : aT -> rT)` has a finite domain `(aT : finType)`, and an equality type codomain `(rT : eqType)`, its injectivity becomes testable, by checking whether the sequence obtained by mapping `f` on the enumeration of `aT` is duplicate-free. In addition, we can exhibit two distinct elements in the domain of `f` which have the same image when `f` is not injective. This is the cornerstone

of pigeon-hole based arguments, a frequent key point of our proofs (see sections 4 and 5). We have proved formally the following version of pigeon-hole:

```
Lemma pigeon_hole (T : finType)(f : nat -> T) :
  {i : nat & {d : nat | i + d <= #|T| & f i = f (i + d.+1)}}.
```

which we found the most useful generic formulation in our setting. This lemma states that for any function `(f : nat -> T)` where `T` is a finite type, it is possible to compute two natural numbers `i` and `d`, such that `(i + d)` is smaller than the number of elements of `T` and such that `(f i = f (i + d + 1))`. We first prove that the restriction `(g : 'I_#|T|.+1 -> T)` of `f` to the natural numbers smaller than `(#|T|.+1)` cannot be injective, hence the existence of two numbers mapped to the same image. We typically use this lemma to show that for a function `(phi : T -> T)` where `T` is a finite type, the iteration of `phi` starting from an `(x0 : T)` is necessarily cyclic. In that case, we apply the `pigeon_hole` lemma to the function associating a number `n` with the `n`-th iteration of `phi` on `x0`.

2.1 Order extensions

The Mathematical Component libraries also contain libraries on sequences (lists), paths and finite graphs which originate from G. Gonthier's proof of the Four Colour Theorem. Consider an arbitrary equality type `T`, a relation `(e : rel T)`, an element `(x : T)` and a sequence `(p : seq T)`. Let $[x_0, \ldots, x_n]$ denote the sequence `(x :: p)` where x_0 is `x` and x_n is `(last x p)`, the last element of `p` with default value `x`. The sequence `(x :: p)` is by definition a path for the relation `e`, denoted `(path e x p)`, if `(e` x_i x_{i+1}`)` holds for any $i < n$. For instance, `(traject f x n)` is the sequence of length `n` obtained by successive iterations of the function `f` starting from `x`. It is a path for the relation which links a point to its image by `f`. A sequence `p` is a cycle for the relation `e`, denoted `(cycle e p)`, if `(path e x (rcons p x))` holds, where `(rcons p x)` adds `x` at the end of `p`.

The Mathematical Component library provides a specific library devoted to boolean relations on a finite type domain, which can be seen as a formalisation of finite graphs. This library provides in particular the construction of the transitive closure `(connect e)` of a relation `(e : rel T)`, where `T` is a finite type, by implementing and certifying a depth-first search algorithm (see [13]). Following the same approach, we have implemented the construction of a total acyclic relation from an acyclic relation by implementing a topological sorting algorithm. To do so, we use the choice operator on finite types, which inspects all the elements of the finite type in the order provided by the underlying enumeration: `[pick x \in A]` (resp. `[pick x \in A | P]`) is an element in `(option T)`, which is `(Some x)` if a witness of `A` (resp. a witness of the boolean predicate `P` satisfying `A`) has been found among the elements of `T` and `None` otherwise. We define the topological sorting as follows:

```
Variable T : finType.
Fixpoint topo_rec (e : rel T) (s : {set T}) (n : nat) :=
```

```
  if n is n'.+1 then
    if [pick x \in s | forallb y, (y \in s) ==> ~~ e y x]
       is Some x
    then
      let e' a b :=
      if (a == x) && (b \in (s :\ x)) then true else e a b
      in topo_rec e' (s :\ x) n'
    else e
  else e.
```

We define this function recursively on a counter `n` and the topological sorting is local to the set `s`. The counter is imposed by the `Fixpoint` construction of the Coq system, which requires a syntactic termination check. If the counter is non-zero, then the function looks for an element in the set `s` which has no predecessor by the relation `e`. If there is no such element, the relation is `e` itself. If there is a witness `x`, the relation is the union of the topological sorting of the set `(s :\ x)` (`s` where `x` has been removed), and of the relation which relates `x` to any element of `(s :\ x)`. Since since one element is removed from the initial set at each recursive call, the complete topological sorting is performed as soon as the counter is equal to or larger than the cardinal of the set `s`. We prove easily that this relation is always an extension of its argument:

```
Lemma topo_rec_subrel (e : rel T) (s : {set T}) (n : nat) :
   subrel e (topo_rec e s n).
```

We also prove by induction on the counter that it is asymmetric (resp. transitive) as soon as its argument is asymmetric (resp. transitive) on the input set:

```
Lemma topo_rec_anti (e : rel T) (s : {set T}) (s' : {set T})
  (ae : {in s' & &, asymmetric e}) (n : nat) :
  {in s' & &, asymmetric (topo_rec e s n)}.
```

where `{in s & &, p}` means that the binary (non necessarily boolean) predicate `p` holds when both its arguments are in `s`. For the last two proofs, we distinguish the set on which the sorting is performed from the set on which the property is transmitted. Thus, we obtain directly an induction hypothesis which is strong enough for the induction on the counter to solve the proof easily. From its asymmetry, we trivially deduce that topological sorting is irreflexive and since it is also transitive, it is acyclic.

Finally if the value of the counter is large enough, the output relation is total on its input set as soon as its input relation is acyclic:

```
Lemma topo_rec_total (e : rel T) (s : {set T}) :
  acyclic_on s e -> {in s & &, strict_total (topo_rec e s #|s|)}.
```

where the `acyclic_on Prop` predicate ensures that no sequence of elements in `s` is a cycle for `e`, and the `strict_total Prop` predicate on boolean relations ensures that for any two distinct elements `x` and `y`, either `(e x y)` or `(e y x)`. For this property, the value of the counter matters, because if the counter is not

large enough, the sorting stops at an intermediate state, and there is no reason why the computed relation should be already total. This proof mainly consists in showing that the acyclicity hypothesis ensures the existence of a predecessor-free witness at each recursive call: if every element of the set on which we operate has a predecessor, then we can find a cycle for the relation among the elements of this finite set using the pigeon-hole principle.

We were able to find only one reference which explicitly mentions a formal proof of correctness of a topological sorting [22], available as part of the Coq CoLoR library [9]. Its computational content is only available through extraction and not as a computable Coq program. However, the underlying algorithm is not realistic, in particular due to repeated calls to a transitive closure process to ensure transitivity: computation was certainly not the motivation of the authors but rather the theoretical existence of the linearisation process. By contrast, we wanted to describe an ideal implementation in Coq of the algorithm, with naive data structures (such as the generic `pick`), but with a code fairly close to a realistic functional implementation (see section 6). We have however proved the specification of the non-tail recursive version because it was easier to handle in the proof. Proving the equivalence of the above program with its tail recursive counterpart is then a straightforward induction on the counter.

3 Event structures, executions, architectures

The framework proposed in [3, 6] abstracts from the semantic of assembly code instructions by reasoning only about the *memory events* issued by the instructions of a given program. We first define the finite types `Value`, `Proc`, `Location`, `Eiid` and `poi` which are respectively modelling values, a set of unique identifiers for processors, locations, a set of unique identifiers for events and the type of indexes in programs order, as natural numbers bounded by a global constant.

Event structures A *memory event* `(m : Event)` is a triple which consists of a unique identifier `(e : Eiid)`, an instruction identifier `(i : Iiid)` and the action `(a : Action)` it performs. An instruction identifier consists of a pair of a processor identifier `(p : Proc)` and the index of its line `(l : poi)`. An *action* is an access in the memory and is encoded as a triple containing the direction `Dirn` (read `R` or write `W`) of the action, its location and its value:

```
Definition Action := (Dirn * Location * Value).
Definition Access d l v : Action := (d, l, v).
```

For instance, the event *"At the line* 1 *of the program, the value* 2 *is read from location* x *on processor* P_0*"* uniquely identified by the label 12 is represented by the term: `(mkEvent 12 (mkiid 0 1)(Access R x 2))`.

Programs are abstracted by *event structures*. An event structure $E \triangleq (\mathbb{E}, \overset{\text{dp}}{\rightarrow})$, is composed of a (finite) set of events $\mathbb{E}$ and the dependency (boolean) relation $\overset{\text{dp}}{\rightarrow}$:

```
Definition Event_struct := {set Event} * (rel Event).
```

```
Definition events_of_ES E := let: (evts, _) := E in evts.
Definition dp_of_ES E := let: (_, dp) := E in dp.
```

The $\overset{\text{dp}}{\rightarrow}$ relation abstracts the instruction semantics[3]. The $\overset{\text{dp}}{\rightarrow}$ relation models the dependencies between instructions, *e.g.* when we compute the address of a load or store from the value of a preceding load.

We can compute the *program order* relation `(po E)` of an event structure `E` by the comparing the program indexes of events occurring on the same processor:

```
Definition po (E : Event_struct) : rel Event := [rel e1 e2 |
same_proc_same_ES E e1 e2 &&(poi_of_Event e1 <= poi_of_Event e2)].
```

A *well-formed event structure* E requires $\overset{\text{dp}}{\rightarrow}$ to be a subset of the program order. Moreover, since dependencies arise between instructions only when the first one is a load [25, 26, 7, 1], we impose the source of $\overset{\text{dp}}{\rightarrow}$ to be always a read.

Execution witnesses A given event structure corresponds to several executions of the same program. To model one of these executions, we define *execution witnesses*. An execution witness $X \triangleq (\overset{\text{ws}}{\rightarrow}, \overset{\text{rf}}{\rightarrow})$ provides two relations on events, $\overset{\text{ws}}{\rightarrow}$ (for *write serialisation*) and $\overset{\text{rf}}{\rightarrow}$ (for *read-from map*):

```
Definition Execution_witness := (rel Event * rel Event).
Definition ws_of_EW (X : Execution_witness) := X.1.
Definition rf_of_EW (X : Execution_witness) := X.2.
```

A *well-formed execution witness* requires its write serialisation to be a per-location total order on the write events. This models the *memory coherence* assumed by modern architectures [7, 1, 25, 26, 15], linking a write w to any write w' to the same location hitting the memory after w. In addition, the read-from map $\overset{\text{rf}}{\rightarrow}$ should link a write w to a read r from the same location that reads from w. We also require that for every read from a given location, there exists a unique write to the same location from which the read takes its value.

Note that until now, all the relations that we defined would also be necessary to describe SC executions. To integrate weak memory models considerations into the picture, we derive an additional relation $\overset{\text{fr}}{\rightarrow}$ (for *from-read map*), from the write serialisation and the read-from map relations, which relates two events e_r and e_w such that there is a write $e_{w'}$ from which e_r reads (ie. $e_{w'} \overset{\text{rf}}{\rightarrow} e_r$) and $e_{w'}$ hits the memory before e_w does (ie. $e_{w'} \overset{\text{ws}}{\rightarrow} e_w$).

The semantics intuition of $\overset{\text{fr}}{\rightarrow}$ is as follows. Some of the weaknesses of memory models arise in our framework because the read events may take their values not only from the main memory, but also from store buffers and caches. This means

[3] We also implement the relation `iico` between events coming from the same instruction, which we omit here for brevity. For example, a load instruction reads from a location x then writes the contents of x into a register r. In this case, the read event from x and the write event to r are linked by `iico`. This requires to add register events to the formalism, which is straightforward.

that we cannot determine at which instant of time the value read by a given read event is actually in the main memory. We can only determine an interval of time in which this value is in the memory, *i.e.* between the write that wrote this value hit the memory and the next write to the same location in the write serialisation. Thus, the $\overset{\text{fr}}{\rightarrow}$ relation allows us to represent this interval, for it relates a given read event to both the write from which it reads its value (*via* the $\overset{\text{rf}}{\rightarrow}$ relation) and the next write to the same location in $\overset{\text{ws}}{\rightarrow}$.

Architectures Finally we describe a given memory model by implementing what we call its *architecture*:

```
Record Archi : Type := mkArchi{
  ppo : Event_struct -> rel Event;
  grf : Execution_witness -> rel Event;
  ab : Event_struct -> Execution_witness -> rel Event;
  grf_valid : forall X, subrel (grf X) (rf_of_EW X);
  ppo_valid : forall E, subrel (ppo E) (po E);
  ab_evts : forall E X,
    subrel (ab E X) [rel x y \in (events_of_ES E)]}.
```

An architecture is characterized by its *preserved program order* $\overset{\text{ppo}}{\rightarrow}$, its *global read-from map* relation $\overset{\text{grf}}{\rightarrow}$ and its *barrier semantics* $\overset{\text{ab}}{\rightarrow}$.

In a shared-memory multiprocessor, some pairs of instructions in the program order may be reordered. Hence in an event structure e, for two events e_1 and e_2 such that $e_1 \overset{\text{po}}{\rightarrow} e_2$, we only know that they will happen in this order if the architecture ensures that $e_1 \overset{\text{ppo}}{\rightarrow} e_2$. TSO for example authorises write-read pairs to be reordered, but nothing else [25, 26]: $\overset{\text{ppo}}{\rightarrow} = \overset{\text{po}}{\rightarrow} \setminus (\mathbb{W} \times \mathbb{R})$.

On any architecture, in any execution, as soon as $e_1 \overset{\text{ws}}{\rightarrow} e_2$ or $e_1 \overset{\text{fr}}{\rightarrow} e_2$, we know that the event e_1 happens before event e_2. However, in most if not all shared-memory multiprocessors, a write may be committed first into a store buffer, then into memory; this optimisation is known as *store buffering* [2]. In some architectures, like PowerPC, a write can be written first to a cache, then to the main memory; in this case we say that the write is not *atomic*, for this optimisation is known as *store atomicity relaxation* [2]. In practice, these two optimisations mean that while a write transits in store buffers and caches, a processor may read a past value. Hence in an execution x, for two events e_1 and e_2 such that $e_1 \overset{\text{rf}}{\rightarrow} e_2$, we only know that the event e_1 will happen before e_2 if the architecture ensures that $e_1 \overset{\text{grf}}{\rightarrow} e_2$. This captures the specification of TSO which authorises store buffering but considers stores to be atomic or the specification of Power which relaxes the atomicity of writes.

Last, architectures provide barrier instructions to order on certain pairs of events. We gather the orderings induced by barriers in the global relation $\overset{\text{ab}}{\rightarrow}$: as soon as $e_1 \overset{\text{ab}}{\rightarrow} e_2$, we know that the event e_1 happens before event e_2.

We are now able to define a *global happens before* relation $\overset{\text{ghb}}{\rightarrow}$. If two events e_1 and e_2 in an execution witness of a given event structure are ordered by $e_1 \overset{\text{ghb}}{\rightarrow} e_2$,

then the architecture ensures that the event e_1 happens before event e_2. On an architecture A, and an execution X of the event structure E, this relation $\overset{\text{ghb}}{\rightarrow}$ is defined as the union of the relations known to be global on A:

```
Definition ghb A E X : rel Event :=
relU (relU (relU (relU (grf A X) (ab A E X)) (ws_of_EW X)) (fr X))
```

In our framework, the main validity condition of an execution consists of the acyclicity of the $\overset{\text{ghb}}{\rightarrow}$ relation, which means that we can think of an execution as exhibiting a global timeline of certain events from the memory point of view.

Validity We now need to determine when a given execution of an event structure is considered as valid on a given architecture. For this purpose, we first define a relation gathering all the communication arrows between events of an execution:

```
Definition com (X : Execution_witness) : rel Event :=
  relU (relU (rf_of_EW X) (fr X)) (ws_of_EW X).
```

We need to define also the $\overset{\text{po-loc}}{\rightarrow}$ relation, which corresponds to the program order restricted to events that share the same memory location. The $\text{uniproc}(E, X) \triangleq \text{acyclic}(\overset{\text{com}}{\rightarrow} \cup \overset{\text{po-loc}}{\rightarrow})$ condition models the fact that all the values that a given memory location can take on a weak memory model are in fact already accessible on SC[4]. This is how we model the *memory coherence property*, as widely assumed by the existing architectures [7, 1, 25, 26, 15].

The $\text{thin}(E, X) \triangleq \text{acyclic}(\overset{\text{rf}}{\rightarrow} \cup \overset{\text{dp}}{\rightarrow})$ condition prevents executions where values seem to come *out of thin air* [18]. We only need to add the thin check on our framework so as to embrace Alpha in our framework: for all the other models, because $\overset{\text{dp}}{\rightarrow} \subseteq \overset{\text{ppo}}{\rightarrow}$, the thin check is subsumed by the acyclicity check of $\overset{\text{ghb}}{\rightarrow}$ [3].

Finally, an execution (E, X) is *valid* on an architecture A when the relation $A.\text{ghb}(E, X)$ is acyclic, together with the two above checks. We model this in Coq by the following three component conjunction:

```
Definition valid_execution A E X : Prop :=
  [/\ acyclic (relU (com X) (po_loc E)),
      acyclic (relU (rf_of_EW X) (dp_of_ES E)) &
      acyclic (ghb A E X)].
```

Amongst these three requirements, only the last one actually depends on the architecture. We consider an architecture as a filter over executions, determining whether they are valid or not, whereas the well-formedness check gathers constraints over programs. Hence, since the first two checks model constraints over executions and not on programs, we include them in the execution filter.

[4] All the results presented here and in [6, 5] hold with a weaker version of uniproc, that allows us to embrace Sun's RMO in our framework. We omit this restriction for clarity and brevity, but more details can be found in [3, p.47-48].

4 Implementations of existing architectures

The *native definition* A_n of a real-world architecture as exposed in the vendor documentation, never matches the form of an architecture as defined in Sec. 3. Thus, we should check that A_n and its encoding A in this framework define the same validity conditions over executions. All the documentations or model descriptions that we studied (*i.e.* SC, Sun TSO, PSO, RMO, Alpha, Power and x86) share a common style, as exposed below, except Power and x86. The x86 model has been discussed to be a TSO model [21], thus what follows apply. We discuss the case of Power at the end of this section. We now explain how we model each architecture in a generic style close to the documentation, then how we encode it into the framework presented in section 3.

A common description of native architectures The description of SC [16] and the documentations of Sun TSO, PSO, RMO [25, 26] and Alpha [7] all describe an execution as an order ex on events. They define the execution ex to be valid if it is a strict partial order on events which moreover contains a certain (architecture dependant) subrelation of the program order[5]. The relation ex should be understood as our global happens-before relation.

For convenience, and since we proved formally in Sec. 2 that any partial order can be extended to a total order, we describe the native models in terms of a total order. This means that for an execution defined as a partial order, we consider all its linear extensions to be valid native executions.

For a given architecture A_n, and an event structure E, we write $A_n\,.\,\mathrm{ppo}(E)$ for the subrelation of the program order distinguished by the architecture. We write $A\,.\,\mathrm{native}(E, \mathrm{ex})$ when ex satisfies the conditions imposed by A_n. The literal formalisation of a vendor documentation is hence of the form:

$$A\,.\,\mathrm{native}(E, \mathrm{ex}) \;\triangleq\; \text{total-order}(\mathrm{ex}, \mathrm{evts}(E)) \wedge (A_n\,.\,\mathrm{ppo}(E) \subseteq \mathrm{ex}))$$

Equivalence proofs Given a native architecture A_n, we want to show that we can build an instance of the framework which defines a validity criterium $A\,.\,\mathrm{valid}$ on execution witnesses equivalent to the check $A\,.\,\mathrm{native}$. This means that whenever an execution ex satisfies $A\,.\,\mathrm{native}$, we can build an execution witness (E, X) which is valid on A, such that ex and (E, X) have the same events and the same communication relations $\xrightarrow{\mathrm{ws}}$ and $\xrightarrow{\mathrm{fr}}$. Conversely, from an execution (E, X) valid on A, we should be able to build an execution ex valid on A_n with the same events and communication relations. Formally, we need to build an instance A of the generic framework such that:

$$\forall EX,\;\; A\,.\,\mathrm{valid}(E, X) \Leftrightarrow \exists\, \mathrm{ex},\;\; A\,.\,\mathrm{native}(E, \mathrm{ex}) \wedge \mathrm{wit}(\mathrm{ex}) = X$$

where wit(ex) stands for an execution witness built from ex.

[5] In fact, the execution ex also satisfies the uniproc and thin checks as defined in Sec. 3, but we omit the details for brevity.

Candidate execution witnesses We now explain how to extract execution witness candidate from the order relation ex, *i.e.* how to build wit(ex).

For this purpose, we need to extract $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$ from an order ex. We write rf(ex) (resp. ws(ex)) for the $\xrightarrow{\text{rf}}$ (resp. $\xrightarrow{\text{ws}}$) extracted from ex. We define (x, y) to be in rf(ex) when x is a write and y a read, both to the same location, such that x is a maximal previous write to this location before y in ex, written $\text{pw}(\text{ex}, r)$. Formally, we have $\text{pw}(\text{ex}, r) \triangleq \{w \mid \text{loc}(w) = \text{loc}(r) \wedge w \,\text{ex}\, r\}$, writing $\text{loc}(m)$ for the location of an event e, and $\text{rf}(\text{ex}) \triangleq \{(w, r) \mid w = \max_{\text{ex}}(\text{pw}((\text{ex}), r))\}$.

We define (x, y) to be in ws(ex) when x and y are writes to the same location and $x \,\text{ex}\, y$. Formally, writing $\mathbb{W}_\ell$ for the set of write events to the location ℓ, we have $\text{ws}(\text{ex}) \triangleq \bigcup_\ell (\mathbb{W}_\ell \times \mathbb{W}_\ell) \cap \text{ex}$. We derive the from-read map fr(ex) following the definition given in Sec. 3.

Note that the extracted $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$ are well-formed in a finite execution, hence the execution witness $\text{wit}(\text{ex}) = (\text{ws}(\text{ex}), \text{rf}(\text{ex}))$ is well-formed as well.

From native architectures to the framework At that point, finding a suitable candidate well-formed instance of the generic framework capturing the native architecture causes no difficulty. We simply take $A = (A_n \,.\, \text{ppo}, A_n \,.\, \text{grf}, \emptyset)$, where $A_n \,.\, \text{grf}(\text{ex})$ returns the fragment of rf(ex) defined to be global by A_n. For example, on TSO, the store atomicity relaxation is not allowed, but store buffering is, hence $A_n \,.\, \text{grf}(\text{ex})$ will return the *internal* $\xrightarrow{\text{rf}}$, *i.e.* the fragment of rf(ex) that relates events from the same processor.

Now we need to show that, for a given event structure E, we can build an execution X, associated with E and valid on A. It follows from their definition that the extracted read-from maps rf(ex), write serialisation ws(ex) and the extracted from-read maps fr(ex) are included in ex. Moreover, since ex contains $A_n \,\text{ppo}(E)$, we know that ex contains the $\xrightarrow{\text{ghb}}$ relation of the extracted execution witness wit(ex). Thus, the acyclicity of $\xrightarrow{\text{ghb}}$ follows from ex being a total order[6]:

$$\forall E \,\text{ex}, A \,.\, \text{native}(E, \text{ex}) \Rightarrow A \,.\, \text{valid}(E, \text{wit}(\text{ex}))$$

From the framework to native architectures This step causes no difficulty as well: the ex relation is the topological sort of the transitive closure of the global-happens-before $\xrightarrow{\text{ghb}}$ relation provided by the candidate architecture. By well-formedness hypothesis (see section 3) of the execution, $\xrightarrow{\text{ghb}}$ should be acyclic on the events. This acyclicity property is preserved by the transitive closure, which is hence transitive and irreflexive. Hence we can apply the topological sorting construction described in section 2, which results in the appropriate ex.

Implemented models In the original formalisation, we have defined SC, Sun TSO, PSO, RMO and Alpha as instances of our framework and proved in Coq the equivalence of our definitions with the original definitions. The code can be browsed online. At the time of submission, only the SC model has been re-implemented. We plan to add the re-implementation of the other models online

[6] Again, we omit the uniproc and thin checks for brevity.

during the review process, which should be without difficulty as these formal proofs are rather routine work.

The framework of Sec. 3 embraces a fragment of Power as well. For Power, since no formal definition existed, the first author and others presented in [6] an intensive testing experiment against three generations of Power machines. This lead to the design of a fragment of a Power model, given in [6], as an instance of the framework presented in Sec. 3.

5 Semantics of barriers

We now present the formalisation of the main theorem presented in [6], which addresses the semantics and placement of memory barriers in a program, to ensure that it only has strong executions, for example SC ones. We say that an architecture A_1 is *weaker* than an architecture A_2 if A_1 exhibits at least all the behaviors of the stronger one A_2:

$$\forall A_1 A_2,\ (A_1 \leq A_2) \triangleq (\forall EX, A_2^{\epsilon}\,.\,\mathrm{valid}(E,X) \Rightarrow A_1^{\epsilon}\,.\,\mathrm{valid}(E,X))$$

where A^{ϵ} represents A without barriers, *i.e.* $\overset{\mathrm{ab}}{\rightarrow} = \emptyset$. For instance, the SC model is stronger than any weak architecture we have mentioned so far. Observe that this definition can be implemented as: $A_1 \leq A_2 \triangleq (\overset{\mathrm{ppo}_1}{\rightarrow} \subseteq \overset{\mathrm{ppo}_2}{\rightarrow}) \wedge (\overset{\mathrm{grf}_1}{\rightarrow} \subseteq \overset{\mathrm{grf}_2}{\rightarrow})$.

We now want to study how to simulate the behavior of a strong architecture over a weaker one, by studying the semantic of the barrier instructions. For instance we may want to show that the semantic of barrier instructions, together with a correct placement of these barriers in the code make possible to forbid non-SC executions on a weak architecture. For this purpose, we define the predicate fb (*fully barriered*) on $A_1 \leq A_2$ as:

$$A_1\,.\,\mathrm{fb}_{A_2}(E,X) \triangleq \left((\overset{\mathrm{ppo}_2 \setminus 1}{\rightarrow}) \cup (\overset{\mathrm{grf}_2 \setminus 1}{\rightarrow}; \overset{\mathrm{ppo}_2}{\rightarrow})\right) \subseteq \overset{\mathrm{ab}_1}{\rightarrow}$$

where $\overset{\mathrm{r}_2 \setminus 1}{\rightarrow} \triangleq \overset{\mathrm{r}_2}{\rightarrow} \setminus \overset{\mathrm{r}_1}{\rightarrow}$ is the set difference, and $x \overset{\mathrm{r}_1}{\rightarrow}; \overset{\mathrm{r}_2}{\rightarrow} y \triangleq \exists z.\, x \overset{\mathrm{r}_1}{\rightarrow} z \wedge z \overset{\mathrm{r}_2}{\rightarrow} y$ stands for the sequence.

The fb predicate provides an insight on the strength that the barriers of the architecture A_1 should have to restore the stronger A_2. They should (i) maintain the pairs preserved in the program order on A_2 and not on A_1 and (ii) compensate for the fact that some writes may not atomic on A_1 while they are on A_2, which we model by (some subrelation of) $\overset{\mathrm{rf}}{\rightarrow}$ not being global on A_1 while it is on A_2. The main theorem of [6] shows that the above condition on $\overset{\mathrm{ab}_1}{\rightarrow}$ regains A_2^{ϵ} from A_1, a property that we call the *barrier guarantee*:

$$\forall A_1 A_2, (A_1 \leq A_2) \Rightarrow (\forall EX, A_1\,.\,\mathrm{valid}(E,X) \wedge A_1\,.\,\mathrm{fb}_{A_2}(E,X) \Rightarrow A_2^{\epsilon}\,.\,\mathrm{valid}(E,X))$$

The only difficulty of this proof is to show the acyclicity of the global happens before relation of the stronger architecture A_2 under the assumptions of the theorem. We have simplified the argument previously formalised, and described in [3], to reduce it the (somehow ad-hoc) following acyclicity criterium:

```
Lemma acyclicity_crit : forall (T : finType)(r1 r2 : rel T),
  acyclic r1 ->
  (forall x y z, r2 x y -> r2 y z -> r1 x y || r1 x z) ->
  acyclic r2.
```

This lemma is again a simple application of the pigeon-hole principle: if a given sequence is a cycle for the relation `r2`, then the hypothesis of the criterium ensures that any element in the sequence has a successor by relation `r1` among the other elements of the sequence: either its successor in the `r2`-cycle, or the successor of its successor. Therefore, the pigeon-hole principle ensures the existence of a `r1`-cycle built from elements in the initial `r2`-cycle. But this violates the acyclicity hypothesis on `r1`.

Let us now go back to the proof of our the barrier guarantee. Let $\overset{\text{ghb}_1}{\rightarrow}$ be the global happens before relation on A_1. Since (E, X) is valid on A_1, this relation is acyclic. Let $\overset{\text{ghb}_2}{\rightarrow}$ be the global happens before relation on A_2^ϵ. We only need to show that these relations satisfy the hypothesis of `acyclicity_crit`. Recall from Sec. 3 that for any architecture A we have $A.\text{ghb}(E, X) \triangleq \overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{ppo}}{\rightarrow} \cup \overset{\text{grf}}{\rightarrow} \cup \overset{\text{ab}}{\rightarrow}$ and that for $\overset{\text{ghb}_2}{\rightarrow}$, $\overset{\text{ab}}{\rightarrow}$ is empty by definition.

Now suppose that $e_1 \overset{\text{ghb}_2}{\rightarrow} e_2 \overset{\text{ghb}_2}{\rightarrow} e_3$ but that $e_1 \overset{\text{ghb}_1}{\rightarrow} e_2$ does not hold. In that case, under the hypothesis of a valid execution, the only possibility is that $e_1 \overset{\text{grf}_2}{\rightarrow} e_2$, but not $e_1 \overset{\text{grf}_1}{\rightarrow} e_2$, and that $e_2 \overset{\text{ppo}_2}{\rightarrow} e_3$. Other possibilities are excluded, for instance by the fact that e_2 cannot be simultaneously a write and a read event. But then we have exactly $e_1 \overset{\text{grf}_2 \setminus 1}{\rightarrow} e_2$, followed by $e_2 \overset{\text{ppo}_2}{\rightarrow} e_3$, which means that e_1 and e_3 are related by $\overset{\text{grf}_2 \setminus 1}{\rightarrow}; \overset{\text{ppo}_2}{\rightarrow}$ which is included in $\overset{\text{ab}_1}{\rightarrow}$ under the hypothesis $A_1.\text{fb}_{A_2}(E, X)$. Hence $e_1 \overset{\text{ab}_1}{\rightarrow} e_3$ thus $e_1 \overset{\text{ghb}_1}{\rightarrow} e_3$. Finally, the hypotheses of the criterium are satisfied with `r1` set at $A_1.\text{ghb}(E, X)$ and `r2` set at $A_2^\epsilon.\text{ghb}(E, X)$, which completes the formal proof of the barrier guarantee.

6 Related Work and Conclusion

There is extensive related work on the semantics issues associated with memory models. For the sake of brevity, we focus here exclusively on the formalisation of hardware models inside a proof assistant. The work presented in [11, 12] studies the formalisation of the ARM architecture, with a particular focus on the instruction set. In our context, we abstract away from the instruction semantics *via* for example the $\overset{\text{dp}}{\rightarrow}$ relation, and leave the integration of instruction sets for further work. The work presented in *e.g.* [23, 21] focus on the formalisation the x86 model in a proof assistant. A few semantics results are proved on top of the formalisation, as a sanity check. Later on, S. Owens proposed in [20] a generalisation of the DRF guarantee, built on top of the formalisation given in [21]. Thus, all these previous pieces of work consider one architecture at a time, except [19], but in this case, only the sequential case is studied, and from the point of view of the instruction set's semantics. By contrast to these previous pieces of work, we provide both a unifying and formalised framework, which nonetheless captures the store atomicity relaxation.

After completing the re-formalisation of the architectures mentioned in section 4, we plan to extend our libraries to the results described in [4, 5, 3], in a small-scale reflection fashion. We do not expect significant difficulties, since the proofs of these extra results rely on the same techniques as used in the formalisation of [6]. Hence the formalisation methods successfully employed in the new library described in the present paper should scale.

Following the approach of M. Myreen *et al.* [19, 11, 12], one of the main motivation of the present re-development is actually the design a certifying layer for the tools described in [6, 4, 5, 3], *e.g.* memevents [23] and offence [5].

The memevents tool [23] is a simulator written in OCaml, which, given a small Power or x86 program and a memory model (amongst which the x86 ones from [23, 21] and the Power one from [6]), outputs all the executions (*i.e.* the event structures and executions witnesses) of this programm that are allowed by the specified memory model. We would like to validate the specifications implemented in memevents, *i.e.* make sure that they actually implement the memory model they are supposed to.

We proposed in [5] a generalisation of the barrier guarantee (also formalised in Coq), which extends to both lock-based and lock-free synchronisation. We implemented our approach in a tool called offence, which places (upon user's choice) either lock-based or lock-free synchronisation in a program in x86 or Power assembly code. When the chosen primitives are barriers, offence should implement the barrier guarantee, but we would like to validate this. This means that we would like to check that the tool actually places enough barriers in a program to ensure that it will behave as if running on SC.

Yet, while M. Myreen *et al.* built their tools inside the proof assistant, we would like to use our formalised framework as a correctness checker for the output of the existing external testing tools. The design of such a (large scale) reflexive certificate checker requires to implement and certify inside the Coq proof assistant some efficient algorithms on graphs, *e.g.* a topological sorting procedure or an acyclicity check. We believe that the preliminary certification of the functional code presented in this paper is a significant step towards the formal certification of a more efficient code, since the remaining effort only lies in the gap between naive and efficient data-structures.

Acknowledgements. We wish to thank Cyril Cohen for his significant help on simplifying the pigeon hole lemma and the proof of the topological sorting.

References

1. *Power ISA Version 2.06*. 2009.
2. S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1995.
3. J. Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7 and INRIA, 2010. `http://moscova.inria.fr/~alglave/these`.
4. J. Alglave, D. Longuet, and L. Maranget. Testing for Weak Memory Model Exploration. In *`http://moscova.inria.fr/~alglave/ddiy`*.

5. J. Alglave and L. Maranget. Stability in Weak Memory Models. In *CAV 2011*.
6. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models. In *CAV 2010*.
7. Alpha Architecture Reference Manual, Fourth Edition, 2002.
8. Y. Bertot and P. Casteran. *Coq'Art*. Springer Verlag, EATCS Texts in Theoretical Computer Science.
9. F. Blanqui and A. Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Sciences*, to appear.
10. S. Burckhardt and M. Musuvathi. Memory Model Safety of Programs. In *ECA-2 2008*.
11. A. Fox, M. Gordon, and M. Myreen. Specification and verification of ARM hardware and software. In *Design and Verification of Microprocessor Systems for High-Assurance Applications 2010*.
12. A. Fox and M. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *ITP 2010*.
13. G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3:95–152, 2010.
14. G. Gonthier, A. Mahboubi, and Enrico Tassi. *A small scale reflection extension for the Coq system*. INRIA Technical report, `http://hal.inria.fr/inria-00258384`.
15. Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3A, rev. 30, March 2009.
16. L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1979.
17. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL 2006*.
18. J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL 2005*.
19. M. Myreen and M. Gordon. Verified LISP implementations on ARM, x86, and PowerPC. In *TPHOL 09*.
20. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP 2010*.
21. S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *TPHOL 2009*.
22. S. Le Roux. Acyclic Preferences and Existence of Sequential Nash Equilibria: A Formal and Constructive Equivalence. In *TPHOLs*, 2009.
23. S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The Semantics of x86-CC Multiprocessor Machine Code. In *POPL 2009*.
24. J. Sevcik, V. Vafeiadis, F. Zappa Nardelli, P. Sewell, and S. Jagannathan. Relaxed-memory concurrency and verified compilation. In *POPL 2011*.
25. Sparc Architecture Manual Version 8, 1992.
26. Sparc Architecture Manual Version 9, 1994.
27. Coq Development team. *The Coq System*, 2011. `http://coq.inria.fr`.