Model Checking for ARMv8

Adithya Bhaskar

May 2021

1 Introduction and Motivation

In the paper [1], the authors demonstrated a general scheme by which one could ascertain decidability for a wide range of memory models, which resulted in acknowledgement of the undecidability for a wide range of memory models, including AArch64. This called for an efficient under-approximation to the model, and accordingly an efficient model checking scheme which can use it. While existing models such as Herd [2] and the Flat [3] model do a good job of expressing the memory model, neither is well suited to a model checking approach. This is primarily because the former is in the form of predicates over allowed abstractions which is not easily extended to operational semantics, while the latter is not expressed in a formal notation and also does not include many oft-used instructions such as load and store exclusives. Motivated by recent approaches to using context-bounding as an efficient under-approximation to reachability, such as for POWER [4], we make the following contributions in this document: 1) We simplify the above two models into a form directly usable for the purposes of model checking, and prove it's equivalence to the herd model; and 2) We develop an under-approximation scheme to check reachability by bounding the number of contexts of execution, in the spirit of the work first championed by Lal and Reps [5]. We demonstrate that it is feasible to check moderately complex multithreaded programs running under power in acceptable amounts of time, opening the door to further optimizations. We open source our code so that others can build upon it.

2 Syntax for programs (simpler version)

We use the following syntax for the program.

$$
\begin{aligned}
Prog ::= \ & \texttt{vars:} \ \ x^* \\
& \texttt{procs:} \ \ p^*
\end{aligned}
$$

$$
\begin{aligned}
p ::= \ & \texttt{regs:} \ \ \$r^* \\
& \texttt{instrs:} \ \ i^*
\end{aligned}
$$

$$
i ::= l : \texttt{s}
$$

$$
\begin{aligned}
\texttt{s} ::= \ & x \leftarrow exp \\
| \ & \$r \leftarrow x \\
| \ & \$r \leftarrow exp \\
| \ & \texttt{if } exp \texttt{ then } i^* \texttt{ else } i^* \\
| \ & \texttt{while } exp \texttt{ do } i^* \\
| \ & \texttt{assume } exp \\
| \ & \texttt{assert } exp \\
| \ & \texttt{term}
\end{aligned}
$$

A program *Prog* first declares a set $\mathcal{X}$ of (shared) variables followed by the code of a set $\mathcal{P}$ of processes. Each process in turn declares a set of registers $\$r$ and it's code in the form of a sequence of instructions denoted by i^*. Each such instruction consists of a label l and a statement s. A *write* instruction has a statement of the form $x \leftarrow exp$ where $x \in \mathcal{X}$ is a variable and exp is an *expression*. In all the above rules, exp is an expression containing only constants and registers, and no shared variables. A *read* instruction in a process $p \in \mathcal{P}$ is a statement of the form $\$r \leftarrow x$, where $\$r$ is a register in p and $x \in \mathcal{X}$ is a variable. An *assign* instruction in a process $p \in \mathcal{P}$ has the form $\$r \leftarrow exp$, where $\$r$ is a register and exp is an expression. Conditional, iterative, assume and assert instructions, collectively called aci instructions describe the rest of the instructions. The special instruction `term` is a syntactic sugar to mark the end of the code for the process.

3 Definitions for the rules

We first define a few helper functions:

- We refer to the statement corresponding to an instruction `i` by the function `stmt(i)`.
- For a write instruction i whose statement is of the form $x \leftarrow exp$, or a read instruction i whose statement is of the form $\$r \leftarrow x$, we define $\texttt{var}(i) := x$ to be the variable corresponding to the instruction. For all other types of instructions we define $\texttt{var}(i) = \bot$.
- For a instruction i which is of one of the following forms: a write instruction with $\texttt{stmt}(i) = \$r \leftarrow exp$, an assign instruction with $\$r \leftarrow exp$, an aci instruction with either $\texttt{stmt}(i) = \texttt{if } exp \texttt{ then } i^* \texttt{ else } i^*$ or $\texttt{stmt}(i) = \texttt{while } exp \texttt{ do } i^*$, or $\texttt{stmt}(i) = \texttt{assume } exp$, or $\texttt{stmt}(i) = \texttt{assert } exp$, we define the expression occuring in the statement as $\texttt{exp}(i) := exp$. For all other types of instructions we define $\texttt{exp}(i) := \bot$.
- For an instruction instance i belonging to the process p, we define $\texttt{proc}(i) := p$. Further, we denote the set of instructions of process p by $\mathcal{J}_p$. Also, we denote the statement at which process p begins execution (i.e. the first instruction of the process), by i_p^{init}.
- For an instruction instance $i \in \mathcal{J}_p$, we denote by $\texttt{next}(i)$ the set of instructions that could immediately follow i in program order. In particular, for an ACI instruction i, we denote the set of instructions that could follow i upon evaluation of $\texttt{exp}(i)$ to True, as $\texttt{Tnext}(i)$, and the set of those that could follow on its evaluation to False as $\texttt{Fnext}(i)$.
- We define the closest write $CW(\mathbb{c}, \mathbb{e}) := \mathbb{e}'$ where $\mathbb{e}'$ is the unique event such that (here and in the rest of the document, an *event* is a single execution instance of an instruction),

  1. $\mathbb{e}' \in \mathbb{E}_p^W$ ($\mathbb{E}_p^W$ denotes the set of write events as defined below),
  2. $\mathbb{e}' \prec_{\texttt{poloc}} \mathbb{e}$ (The poloc program order $\prec_{\texttt{poloc}}$ will be defined below),
  3. there is no event $\mathbb{e}''$ such that $\mathbb{e}'' \in \mathbb{E}_p^W$ and $\mathbb{e}' \prec_{\texttt{poloc}} \mathbb{e}'' \prec_{\texttt{poloc}} \mathbb{e}$.

  If no such events exists, we write $CW(\mathbb{e}) := \bot$.
- We define by $\mathcal{R}(i)$ the set of registers appearing in $\texttt{exp}(i)$. In the special case that $\texttt{exp}(i) = \bot$, we write $\mathcal{R}(i) := \emptyset$.

We define a *configuration* $\mathbb{c}$ as the tuple $\langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}, \texttt{rf}, \texttt{Prop} \rangle$. Here, $\mathbb{E} \in \mathcal{E}$ is the set of events that have been created up to that point in the program. For any process p, we denote by $\mathbb{E}_p$ the events of process p that have been created so far. By $\mathbb{E}^W$ we denote the subset of $\mathbb{E}$ containing precisely the write events. We similarly define $\mathbb{E}^R$, $\mathbb{E}^{ACI}$ and so on. The program order relation $\prec \subseteq \mathbb{E} \times \mathbb{E}$ is an irreflexive partial order that describes for each process $p \in \mathcal{P}$ the order in which events are fetched from the code of p. Note that $\mathbb{e}_1 \not\prec \mathbb{e}_2$ if $\texttt{proc}(\mathbb{e}_1) \neq \texttt{proc}(\mathbb{e}_2)$, i.e. if they belong to different processes, and $\prec$ is a total order on each of the $\mathbb{E}_p$ individually. The function $\texttt{status} : \mathbb{E} \mapsto \{\texttt{fetch}, \texttt{init}, \texttt{com}\}$ defines the current status of each event, which is one of fetched, initialized, and committed. The function $\texttt{Prop} : \mathcal{X} \mapsto \mathbb{E}^W \cup \mathcal{E}^{\texttt{init}}$ defines for each variable the latest value of that variable that has been propagated to the main memory. The function $\texttt{rf} : \mathbb{E}^R \mapsto \mathbb{E}^W \cup \mathcal{E}^{\texttt{init}}$ defines for each read event $\mathbb{e}$ the event $\texttt{rf}(\mathbb{e})$ from which $\mathbb{e}$ gets its value. We introduce a number of dependency relations to help us frame the rules. The *per-location program order* $\prec_{\texttt{poloc}} \subseteq \mathbb{E} \times \mathbb{E}$ is such that $\mathbb{e}_1 \prec_{\texttt{poloc}} \mathbb{e}_2$ if and only if $\mathbb{e}_1 \prec \mathbb{e}_2$ and $\texttt{var}(\mathbb{e}_1) = \texttt{var}(\mathbb{e}_2)$. Further, we define the data dependency order $\prec_{\texttt{data}}$ such that $\mathbb{e}_1 \prec_{\texttt{data}} \mathbb{e}_2$ if

- $\mathbb{e}_1 \in \mathbb{E}^R \cup \mathbb{E}^A$, i.e. $\mathbb{e}_1$ is a read or assign event.
- $\mathbb{e}_2 \in \mathbb{E}^W \cup \mathbb{E}^A \cup \mathbb{E}^{ACI}$, i.e. $\mathbb{e}_2$ is a write, assign or ACI event.
- $\mathbb{e}_1 \prec \mathbb{e}_2$
- $\texttt{stmt}(\texttt{ins}(\mathbb{e}_1))$ is of the form $\$r \leftarrow x$ or $\$r \leftarrow exp$,
- $\$r \in \mathcal{R}(\texttt{ins}(\mathbb{e}_2))$
- There is no event $\mathbb{e}_3 \in \mathbb{E}^R \cup \mathbb{E}^A$ such that $\mathbb{e}_1 \prec \mathbb{e}_3 \prec \mathbb{e}_2$ is of the form $\$r \leftarrow y$ or $\$r \leftarrow exp'$. That is, we want that $\mathbb{e}_1$ be responsible for "supplying" the data to $\mathbb{e}_2$.

Finally, we define the control dependency $\prec_{\texttt{ctrl}}$ as $\mathbb{e}_1 \prec_{\texttt{ctrl}} \mathbb{e}_2$ if $\mathbb{e}_1 \in \mathbb{E}^{ACI}$ and $\mathbb{e}_1 \prec \mathbb{e}_2$.
We also define the transition relation as a relation $\longrightarrow \subseteq \mathbb{C} \times \mathcal{P} \times \mathbb{C}$. For configurations $\mathbb{c}_1, \mathbb{c}_2 \in \mathbb{C}$ and a process $p \in \mathcal{P}$, we write $\mathbb{c}_1 \xrightarrow{p} \mathbb{c}_2$ to denote that $\langle \mathbb{c}_1, p, \mathbb{c}_2 \rangle \in \longrightarrow$.

4 Helper predicates for next section

Predicate	Definition	Meaning
$\mathbb{e} \in \mathbb{E}^W$: $\texttt{ComCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}$: $((\mathbb{e}' \prec_{\texttt{data}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{ctrl}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{poloc}} \mathbb{e}))$ $\implies$ $(\texttt{status}(\mathbb{e}') = \texttt{com})$	All events preceeding $\mathbb{e}$ in $\prec_{\texttt{data}}$, $\prec_{\texttt{ctrl}}$ or $\prec_{\texttt{poloc}}$ have already been commited.
$\mathbb{e} \in \mathbb{E}^R \cup \mathbb{E}^A$: $\texttt{RdAssignComCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}$: $(\mathbb{e}' \prec_{\texttt{data}} \mathbb{e})$ $\implies$ $(\texttt{status}(\mathbb{e}') = \texttt{com})$	All events preceeding $\mathbb{e}$ in $\prec_{\texttt{data}}$, have already been commited.
$\mathbb{e} \in \mathbb{E}^W$: $\texttt{InitCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^R \cup \mathbb{E}^A$: $(\mathbb{e}' \prec_{\texttt{data}} \mathbb{e})$ $\implies$ $((\texttt{status}(\mathbb{e}') = \texttt{init}) \vee (\texttt{status}(\mathbb{e}') = \texttt{com}))$	All instructions on which $\mathbb{e}$ is data-dependent on have already been initialized.
$\mathbb{e} \in \mathbb{E}^{\texttt{ACI}}$: $\texttt{ValidCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{ACI}}$: $((\mathbb{e} \prec \mathbb{e}') \wedge (\nexists \mathbb{e}'' \in \mathbb{E} : \mathbb{e} \prec \mathbb{e}'' \prec \mathbb{e}'))$ $\implies$ $(((\texttt{Val}(\mathbb{c}, \mathbb{e}) = true) \wedge (\texttt{ins}(\mathbb{e}') = \texttt{Tnext}(\texttt{ins}(\mathbb{e}))))$ $\vee$ $((\texttt{Val}(\mathbb{c}, \mathbb{e}) = false) \wedge (\texttt{ins}(\mathbb{e}') = \texttt{Fnext}(\texttt{ins}(\mathbb{e})))))$	The instruction that was fetched right after the ACI instruction was consistent with its truth value.

5 Rules without synchronization instructions

In each of these rules, $\mathbb{c} \equiv \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}, \texttt{rf}, \texttt{Prop} \rangle$ denotes the "current" configuration, and the rule describes one possible way in which this configuration may evolve.

$$\frac{\mathbb{e} \notin \mathbb{E}, \quad \prec' = \prec \cup \{\langle \mathbb{e}', \mathbb{e} \rangle | \mathbb{e}' \in \mathbb{E}\}, \quad i \in \texttt{MaxI}(\mathbb{c}, p)}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E} \cup \{\mathbb{e}\}, \prec', \texttt{ins}[\mathbb{e} \leftarrow i], \texttt{status}[\mathbb{e} \leftarrow \texttt{fetch}], \texttt{rf}, \texttt{Prop} \rangle} \texttt{Fetch}$$

$$\frac{\mathbb{e} \in \mathbb{E}^R_p, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad CW(\mathbb{c}, \mathbb{e}) = \mathbb{e}', \quad \texttt{status}(\mathbb{e}') = \texttt{com}, \quad \texttt{RdAssignComCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{rf}[\mathbb{e} \leftarrow \mathbb{e}'], \texttt{Prop} \rangle} \texttt{ComReadFromLocal}$$

$$\frac{\mathbb{e} \in \mathbb{E}^R_p, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad (CW(\mathbb{c}, \mathbb{e}) = \bot) \vee (CW(\mathbb{c}, \mathbb{e}) = \mathbb{e}' \wedge \texttt{status}(\mathbb{e}') = \texttt{com}), \quad \texttt{RdAssignComCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}[\mathbb{e} \leftarrow \texttt{Prop}(\texttt{var}(\mathbb{e}))], \texttt{Prop} \rangle} \texttt{ComReadFromProp}$$

$$\frac{\mathbb{e} \in \mathbb{E}^W_p, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad \texttt{InitCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{init}], \texttt{rf}, \texttt{Prop} \rangle} \texttt{InitWrite}$$

$$\frac{\mathbb{e} \in \mathbb{E}^W_p, \quad \texttt{status}(\mathbb{e}) = \texttt{init}, \quad \texttt{ComCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}[\texttt{var}(\mathbb{e}) \leftarrow \mathbb{e}] \rangle} \texttt{ComWrite}$$

$$\frac{\mathbb{e} \in \mathbb{E}^A_p, \quad \texttt{status}(\mathbb{e}) = \texttt{init}, \quad \texttt{RdAssignComCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop} \rangle} \texttt{ComAssign}$$

$$\frac{\mathbb{e} \in \mathbb{E}^{\texttt{ACI}}_p, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad \texttt{ComCnd}(\mathbb{c}, \mathbb{e}), \quad \texttt{ValidCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop} \rangle} \texttt{ComACI}$$

The rule `Fetch` chooses the next instruction to be executed from the code of a process $p \in \mathcal{P}$. To do this it should be able to select the next instruction i given the configuration $\mathbb{c}$. To do this, we define the function $\texttt{MaxI}(\mathbb{c}, p)$ to be the set of instructions that can (intuitively) follow the current one. Formally,

- If $\mathbb{E}_p = \emptyset$, then we define $\texttt{MaxI}(\mathbb{c}, p) := \{i^{init}_p\}$, i.e. the first instruction to be fetched by p is i^{init}_p.
- If $\mathbb{E}_p \neq \emptyset$ let $\mathbb{e}' \in \mathbb{E}_p$ be the maximal event in $\mathbb{E}_p$ w.r.t $\prec$ in the configuration $\mathbb{c}$. Define $\texttt{MaxI}(\mathbb{c}, p) := \texttt{next}(\texttt{ins}(\mathbb{e}'))$

The remaining rules can be explained as follows:

- Once a read instruction is fetched, it can be satisfied by reading from either the latest `po`-previous write instruction (from the same process) that has not yet been propagated to memory, or can be satisfied by reading the value from memory if no such instruction exists. The rule `ComReadFromLocal` captures the former case, while the rule `ComReadFromProp` captures the latter. In both cases, we update the status of $\mathbb{e}$ to show that it has been 'committed', and update `rf` to indicate the instruction that satisfied it.

- A write instruction can be initialized if all events it is data-dependent on have been initialized. This rule is captured in the form of `InitWrite`. At this point events of the same thread can read from this write.
- Once all `po`-previous instructions on which `e` depends in some way (via a data, control or poloc dependency) have been committed (i.e. been propagated to memory), an initialized read can be propagated to memory and be committed. At this point events from all threads can read from this write. This forms `ComWrite`.
- The rules for an assign event are similar to that of a read event.
- An ACI event undergoes only one state transition after being fetched, i.e. being committed. However, such an event can be committed if and only if it is *valid*, i.e., e.g. the predicted branch turns out to be correct, or a loop is entered, etc. Of course, along with this it must satisfy the `ComCnd` which says that all events that `e` depends upon must have already been committed. This is captured as the rule `ComACI`.

6 Syntax for programs (complete version)

We introduce a number of instructions resembling those that are used in the AArch64. We explain them in the next section.

$$
\begin{aligned}
Prog &::= \texttt{procs:}\ \ p^* \\
p &::= \texttt{regs:}\ \ \$r^* \\
&\quad\ \ \texttt{instrs:}\ \ i^* \\
i &::= l : \texttt{s} \\
\texttt{s} &::= \ \ \$r \leftarrow exp \\
&\quad |\texttt{LD}\ \$r' \leftarrow [\$r] \\
&\quad |\texttt{ST}\ [\$r'] \leftarrow \$r \\
&\quad |\texttt{LDA}\ \$r' \leftarrow [\$r] \\
&\quad |\texttt{STL}\ [\$r'] \leftarrow \$r \\
&\quad |\texttt{LDX}\ \$r' \leftarrow [\$r] \\
&\quad |\texttt{STX}\ \$r'',\ [\$r'] \leftarrow \$r \\
&\quad |\texttt{LDAX}\ \$r' \leftarrow [\$r] \\
&\quad |\texttt{STLX}\ \$r'',\ [\$r'] \leftarrow \$r \\
&\quad |\texttt{if}\ exp\ \texttt{then}\ i^*\ \texttt{else}\ i^* \\
&\quad |\texttt{while}\ exp\ \texttt{do}\ i^* \\
&\quad |\texttt{assume}\ exp \\
&\quad |\texttt{assert}\ exp \\
&\quad |\texttt{dmb.ld} \\
&\quad |\texttt{dmb.st} \\
&\quad |\texttt{dmb.sy} \\
&\quad |\texttt{isb} \\
&\quad |\texttt{term}
\end{aligned}
$$

We comment about a few things here: the first thing to notice is that we have removed variables, and instead introduced addressed memory: the latter is exactly the same as the former, with the added complexity that there can now be address dependencies between instructions, which did not exist for variables. We refer to all kinds of loads collectively as "reads", and all kinds of stores collectively as "writes". Second, the assign instruction is closer to a read than it is a write: the primary reason is that it writes to registers and not memory (and thus it's side effects affect the former). In fact, an assign may be represented as a store to a "private" variable followed by a load from it, hence it is in a sense a compound instruction. However, since it does not directly interface with the memory at all, any run is indistinguishable from another wherein an assign is initialized or committed at a different time, as long as the read-from relations to and from it are preserved. Hence, we do not include the assign statement in the statement of the predicates. Third, for simplicity we assume that the addresses of loads and stores are registers, and not expressions of registers. The reason for this is that one, exactly the same rules given in the next-to-next section apply if we use expressions instead, and this, two, we can simply assume that we assign the expression to a reserved register and use it as the address immediately after (this is the same under this model).

7 Definitions and Helper predicates for next section

Along with the usual load/store and read/write operations, we also introduce three "special" types of loads and stores. First, a load or a store can be exclusive. The significance of a load exclusive that loads from an address and a store exclusive that writes back to that address is that the store succeeds if and only if no other event (from any process) has written to that address since the load exclusive. Also, corresponding to whether the store exclusive succeeded or not, the success value is written to a specified register. Note that such a primitive may be used to implement, e.g. locks or `CAS` instructions. There are also load-acquire and store-release instructions. These are special in that it is forbidden for any instruction that appears `po`-after the load-acquire to be reordered with it. Similarly, it is forbidden for any instruction that appears `po`-before the store-release to be reordered with it. In effect, the combination of a load-acquire and store-release serves like a `dmb.sy`. Finally there are `LDAX` and `STLX` instructions which are simply the combination of both the acquire/release and exclusive type of instructions, i.e. they provide the strictest guarantees. One point to observe here is that by nature of its definition the exclusive versions of stores are multi-copy atomic: every process (including the local one) sees the write at the same time; consequently we can consider it to have only a commit rule (since the `init` rules were meant to portray the period where effects were visible only to the local process).

We also introduce a few more helpful definitions. For any event $\mathtt{e}$, $\mathtt{RegWritten}(\mathtt{e})$ is the register that $\mathtt{e}$ writes to, or $\bot$ if there is no such register. This is non-`bot` for read, exclusive store and assign events. Further, for an assign event $\mathtt{e}$ of the form $\$r \leftarrow exp$, we define as $\mathtt{OpRegs}(\mathtt{e})$ to be the set of registers appearing in exp.

It is also worth pointing out that any kind of read has only a commit rule: since the point of local and global visibility for a read is same, we can assume that the `init` and `commit` point are the same; consequently we let them directly commit after being fetched. This applies to all four kinds of reads.

Since we now have address dependencies, we note that at any point in an execution, for an event $\mathtt{e}$, $\mathtt{addr}(\mathtt{e})$ may not be defined, which we write as $\mathtt{addr}(\mathtt{e}) = \top$. Then, for two events $\mathtt{e}_1 \prec \mathtt{e}_2$, if $\mathtt{addr}(\mathtt{e}_1) = \top$, it is possible that upon resolution, we get later that $\mathtt{e}_1 \prec_{\mathtt{poloc}} \mathtt{e}_2$. Subsequently, we include in `poloc`-dependencies all event pairs $(\mathtt{e}_1, \mathtt{e}_2)$ such that $\mathtt{addr}(\mathtt{e}_1) = \top$.

Coming to the barriers, there are four types: three are `dmb.ld`,`dmb.st` and `dmb.sy`. The `ld` variant stops the reordering of (local) load instructions (and variants therof, including "reads" from a variable) with it, while the `st` variant does that with stores and writes. The `sy` variant does both. The `isb` instruction is a bit different: all we need is that when it is committed, all `po`-preceeding events have their addresses (to which they write or from which they read) fully determined, i.e. any instruction that they are `addr`-dependent on must have been initialized (or committed).

We introduce events corresponding to these new instructions: we retain our nomenclature of their partition, with the corresponding $\mathbb{E}^{\mathtt{name}}$ being the subset of $\mathbb{E}$ corresponding to `name` type of instructions. We should also revise our definition of CW. The events that can now write to a memory location are stores and store releases. Note however that Store Releases will write the success/failure indicator bit (0 is success) into a *register* (called $\$r$ above), hence we need to consider them for the definition of CW as well. Thus, our definition of CW is revised to be the following: for reads and loads $\mathtt{e}$, $CW(\mathtt{e})$ is defined to the `po`-latest store/write or variant therof that is `poloc`-before the store release. Note that this works even if that instruction is exclusive, because since a store exclusive has only a commit rule, that any read that is `poloc` after it has to wait for it to commit; by which time due to the rule, all events `po`-before the SRel are committed as well. Hence, the effects of the `SRel` are already well defined and visible to all processes. We also comment that we refer to both the old type of reads as well as the normal load instructions as "reads", and the old type of writes as well as plain stores as "writes".

Since we need to be able to determine if an intervening write overwrote the value written by an exclusive load, we introduce into our concept of a configuration a mapping function $\mathtt{Mark} : \mathcal{X} \mapsto \mathcal{P} \cup \{\bot\}$. Here $\mathcal{X}$ includes addressable memory locations instead of variables as it did before: both are the same, except that variables do not have address dependencies. Accordingly, we also reuse the function `var` to return the address of a load/store event. The idea is that any exclusive load "marks" the memory location with its originating process, while all other store based instructions simply overwrite the marker with $\bot$ to record that the location is dirty.

We also define the function $\mathtt{GetUpdate}(\mathtt{e}) : (\mathbb{E}^{\mathtt{STX}} \cup \mathbb{E}^{\mathtt{STLX}}) \mapsto \mathbb{E}$, which, depending on whether the exclusive event $\mathtt{e}_2$ succeeds, updates the memory. It is defined as follows: if $\mathtt{Mark}(\mathtt{var}(\mathtt{e})) = \mathtt{proc}(\mathtt{e})$, then the function equals $\mathtt{e}_2$, and otherwise $\mathtt{Prop}(\mathtt{var}(\mathtt{e}))$. Note that this correctly captures the idea of the success-value of a store-exclusive: a store exclusive can be successful only if the last event propagated to memory is an exclusive load from the same process. We also define $\mathtt{GetMark}(\mathtt{e}_2) : \times(\mathbb{E}^{\mathtt{STX}} \cup \mathbb{E}^{\mathtt{STLX}}) \mapsto \mathcal{P} \cup \{\bot\}$. This function determines whether to update the mark of a location based on whether an exclusive store is successful. Similarly to `GetUpdate`, if $\mathtt{Mark}(\mathtt{var}(\mathtt{e})) = \mathtt{proc}(\mathtt{e})$, then the function equals $\bot$, and otherwise $\mathtt{Mark}(\mathtt{var}(\mathtt{e}))$. Similarly, when an exclusive load is committed, we must ensure that the event we read from is the one that is on "top" at the memory, since we may have had some other process' writes intervening in after we read the value. Hence we define, $\mathtt{GetLoadMark}(\mathtt{e}) : (\mathbb{E}^{\mathtt{LDX}} \cup \mathbb{E}^{\mathtt{LDAX}}) \mapsto \mathcal{P} \cup \{\bot\}$: if $\mathtt{Prop}(\mathtt{var}(\mathtt{e}_2)) = \mathtt{rf}(\mathtt{e})$, then the function equals $\mathtt{proc}(\mathtt{e})$, and otherwise $\mathtt{Mark}(\mathtt{var}(\mathtt{e}))$.

There is one more note of importance: the *Fetch* rule is purely cosmetic: it is more intuitive to present a Fetch rule so as to encapsulate the out-of-order execution, however *it could simply be merged with the init/commit rules.* Subsequently, our code-to-code translation scheme (presented later) **does not make use of the fetch rule** (i.e. incorporates it as a part of the other rules).

We next define a bunch of predicates which will prove to be useful in stating the transition rules.

Predicate	Definition	Meaning
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllDmbLds}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{DmbLd}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous `dmb.ld`'s have been commited.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllDmbSts}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{DmbSt}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous `dmb.st`'s have been commited.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllDmbSys}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{DmbSy}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous `dmb.sy`'s have been commited.
$\mathbb{e} \in \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STLX}} \cup \mathbb{E}^{\texttt{DmbSt}}$: $\texttt{AllWrites}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{ST}} \cup \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STX}} \cup \mathbb{E}^{\texttt{STLX}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All `po`-previous write-variants have been commited.
$\mathbb{e} \in \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STLX}} \cup \mathbb{E}^{\texttt{DmbLd}}$: $\texttt{AllReads}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{LD}} \cup \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDX}} \cup \mathbb{E}^{\texttt{LDAX}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All `po`-previous read-variants have been commited.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllSyncs}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{DmbSy}} \cup \mathbb{E}^{\texttt{Isb}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All po-previous `dmb.sy`'s and `isb`'s have been commited.
$\mathbb{e} \in \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDAX}}$: $\texttt{AllSRels}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e} \in \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STLX}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All `po`-previous Store-Releases have been commited.
$\mathbb{e} \in \mathbb{E}^{\texttt{ST}} \cup \mathbb{E}^{\texttt{STX}}$: $\texttt{AllLAcqs}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDAX}}$: $((\mathbb{e}' \prec \mathbb{e}) \implies \texttt{status}(\mathbb{e}') = \texttt{com}))$	All `po`-previous Load-Acquires have been committed.
$\mathbb{e} \in \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STLX}}$: $\texttt{AllMem}(\mathbb{c}, \mathbb{e})$	$\texttt{AllWrites}(\mathbb{c}, \mathbb{e}) \wedge \texttt{AllReads}(\mathbb{c}, \mathbb{e})$	All `po`-previous events that access memory have been committed.
$\mathbb{e} \in \mathbb{E}$: $\texttt{AllBarriers}(\mathbb{c}, \mathbb{e})$	$\texttt{AllDmbLds}(\mathbb{c}, \mathbb{e}) \wedge \texttt{AllDmbSts}(\mathbb{c}, \mathbb{e}) \wedge \texttt{AllSyncs}(\mathbb{c}, \mathbb{e})$	All `po`-previous barriers have been committed.
$\mathbb{e} \in \mathbb{E}^{\texttt{ST}} \cup \mathbb{E}^{\texttt{STL}}$: $\texttt{InitCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}$: $(\mathbb{e}' \prec_{\texttt{data}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{addr}} \mathbb{e}) \implies (\texttt{status}(\mathbb{e}') = \texttt{com}))$	All events on which $\mathbb{e}$ is dependent on have already been committed.
$\mathbb{e} \in \mathbb{E}^{\texttt{Isb}} \cup \mathbb{E}^{\texttt{ST}} \cup \mathbb{E}^{\texttt{STL}} \cup \mathbb{E}^{\texttt{STX}} \cup \mathbb{E}^{\texttt{STLX}}$: $\texttt{AddrCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}^{\texttt{LD}} \cup \mathbb{E}^{A} \cup \mathbb{E}^{\texttt{LDA}} \cup \mathbb{E}^{\texttt{LDX}} \cup \mathbb{E}^{\texttt{LDAX}}, \mathbb{e}'' \in \mathbb{E}$: $(\mathbb{e}' \prec_{\texttt{addr}} \mathbb{e}'') \wedge (\mathbb{e}'' \prec_{\texttt{po}} \mathbb{e}) \implies ((\texttt{status}(\mathbb{e}') = \texttt{init}) \vee (\texttt{status}(\mathbb{e}') = \texttt{com}))$	All events `po`-before $\mathbb{e}$ have fully defined memory footprints.
$\mathbb{e} \in \mathbb{E}$: $\texttt{IsWrite}(\mathbb{e})$	$\mathbb{e} \in \mathbb{E}^{\texttt{ST}} \vee \mathbb{e} \in \mathbb{E}^{\texttt{STL}} \vee \mathbb{e} \in \mathbb{E}^{\texttt{STX}} \vee \mathbb{e} \in \mathbb{E}^{\texttt{STLX}}$	Event $\mathbb{e}$ is a write.
$\mathbb{e} \in \mathbb{E}$: $\texttt{IsSync}(\mathbb{e})$	$\mathbb{e} \in \mathbb{E}^{\texttt{DmbSy}} \vee \mathbb{e} \in \mathbb{E}^{\texttt{Isb}}$	Event $\mathbb{e}$ is a `dmb.sy` or `isb`.
$\mathbb{e} \in \mathbb{E}$: $\texttt{ComCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}$: $(((\mathbb{e}' \prec_{\texttt{data}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{ctrl}} \mathbb{e} \wedge (\texttt{IsWrite}(\mathbb{e}) \vee \texttt{IsSync}(\mathbb{e}))) \vee (\mathbb{e}' \prec_{\texttt{poloc}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{addr}} \mathbb{e})) \implies (\texttt{status}(\mathbb{e}') = \texttt{com}))$	All events preceding $\mathbb{e}$ in $\prec_{\texttt{data}}$, $\prec_{\texttt{addr}}$, $\prec_{\texttt{poloc}}$ and if $\mathbb{e}$ is a write or a barrier, in $\prec_{\texttt{ctrl}}$, have already been commited.
$\mathbb{e} \in \mathbb{E}$: $\texttt{RdComCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}$: $((\mathbb{e}' \prec_{\texttt{data}} \mathbb{e}) \vee (\mathbb{e}' \prec_{\texttt{addr}} \mathbb{e}) \implies (\texttt{status}(\mathbb{e}') = \texttt{com})$	All events on which $\mathbb{e}$ is dependent on have already been initialized.
$\mathbb{e} \in \mathbb{E}^{\texttt{A}}$: $\texttt{AssignComCnd}(\mathbb{c}, \mathbb{e})$	$\forall \mathbb{e}' \in \mathbb{E}$: $((\mathbb{e}' \prec \mathbb{e}) \wedge (\texttt{RegWritten}(\mathbb{e}') \in \texttt{OpRegs}(\mathbb{e})) \wedge \nexists \mathbb{e}'' \texttt{s.t.} ((\mathbb{e}' \prec \mathbb{e}'' \prec \mathbb{e}) \wedge (\texttt{RegWritten}(\mathbb{e}') = \texttt{RegWritten}(\mathbb{e}''))) \implies (\texttt{status}(\mathbb{e}') = \texttt{com})$	The closest events writing to any register that $\mathbb{e}$ reads from have all been committed.

8 Rules with synchronization conditions

For each of the rules below, $\mathbb{c}$ is the configuration before the rule, and is of the form $\langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}, \mathtt{rf}, \mathtt{Prop}, \mathtt{Mark} \rangle$. The rules describe one possible way in which this configuration may evolve. The initial configuration is $\langle \emptyset, \emptyset, \lambda i.\emptyset, \lambda i.\emptyset, \lambda i.\emptyset, \lambda i.0, \lambda i.\bot \rangle$.

$$\frac{\mathbb{e} \in \mathbb{E}_p, \quad \prec' = \prec \cup \{\langle \mathbb{e}', \mathbb{e} \rangle | \mathbb{e}' \in \mathbb{E}_p\}, \quad i \in \mathtt{MaxI}(\mathbb{c}, p)}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E} \cup \mathbb{e}, \prec', \mathtt{ins}[\mathbb{e} \leftarrow i], \mathtt{status}[\mathbb{e} \leftarrow \mathtt{fetch}], \mathtt{rf}, \mathtt{Prop}, \mathtt{Mark} \rangle} \mathtt{Fetch}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{LD}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{RdComCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllLacqs}(\mathbb{c}, \mathbb{e}), \quad \mathbb{e}' = CW(\mathbb{c}, \mathbb{e}), \quad \mathtt{status}(\mathbb{e}') = \mathtt{init}}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}[\mathbb{e} \leftarrow \mathbb{e}'], \mathtt{Prop}, \mathtt{Mark} \rangle} \mathtt{ComLDFromLocal}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{LD}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllLacqs}(\mathbb{c}, \mathbb{e}), \quad (CW(\mathbb{c}, \mathbb{e}) = \bot) \vee (CW(\mathbb{c}, \mathbb{e}) = \mathbb{e}' \wedge \mathtt{status}(\mathbb{e}') = \mathtt{com}), \quad \mathtt{RdComCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}[\mathbb{e} \leftarrow \mathtt{Prop}(\mathtt{Var}(\mathbb{c}, \mathbb{e}))], \mathtt{Prop}, \mathtt{Mark} \rangle} \mathtt{ComLDFromProp}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{LDX}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{RdComCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllLacqs}(\mathbb{c}, \mathbb{e}), \quad \mathbb{e}' = CW(\mathbb{c}, \mathbb{e}), \quad \mathtt{status}(\mathbb{e}') = \mathtt{init}}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}[\mathbb{e} \leftarrow \mathbb{e}'], \mathtt{Prop}, \mathtt{Mark}[\mathtt{var}(\mathbb{e}) \leftarrow \mathtt{GetLoadMark}(\mathbb{e})] \rangle} \mathtt{ComLDXFromLocal}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{LDX}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllLacqs}(\mathbb{c}, \mathbb{e}), \quad (CW(\mathbb{c}, \mathbb{e}) = \bot) \vee (CW(\mathbb{c}, \mathbb{e}) = \mathbb{e}' \wedge \mathtt{status}(\mathbb{e}') = \mathtt{com}), \quad \mathtt{RdComCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}[\mathbb{e} \leftarrow \mathtt{Prop}(\mathtt{Var}(\mathbb{c}, \mathbb{e}))], \mathtt{Prop}, \mathtt{Mark}[\mathtt{var}(\mathbb{e}) \leftarrow \mathtt{GetLoadMark}(\mathbb{e})] \rangle} \mathtt{ComLDXFromProp}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{LDA}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllSRels}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllLacqs}(\mathbb{c}, \mathbb{e}), \quad \mathbb{e}' = CW(\mathbb{c}, \mathbb{e}), \quad \mathtt{status}(\mathbb{e}') = \mathtt{init}, \quad \mathtt{RdComCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}[\mathbb{e} \leftarrow \mathbb{e}'], \mathtt{Prop}, \mathtt{Mark} \rangle} \mathtt{ComLDAFromLocal}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{LDA}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllLacqs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{RdComCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllSRels}(\mathbb{c}, \mathbb{e}), \quad (CW(\mathbb{c}, \mathbb{e}) = \bot) \vee (CW(\mathbb{c}, \mathbb{e}) = \mathbb{e}' \wedge \mathtt{status}(\mathbb{e}') = \mathtt{com})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}[\mathbb{e} \leftarrow \mathtt{Prop}(\mathtt{Var}(\mathbb{c}, \mathbb{e}))], \mathtt{Prop}, \mathtt{Mark} \rangle} \mathtt{ComLDAFromProp}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{LDAX}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllSRels}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllLacqs}(\mathbb{c}, \mathbb{e}), \quad \mathbb{e}' = CW(\mathbb{c}, \mathbb{e}), \quad \mathtt{status}(\mathbb{e}') = \mathtt{init}, \quad \mathtt{RdComCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}[\mathbb{e} \leftarrow \mathbb{e}'], \mathtt{Prop}, \mathtt{Mark}[\mathtt{var}(\mathbb{e}) \leftarrow \mathtt{GetLoadMark}(\mathbb{e})] \rangle} \mathtt{ComLDAXFromLocal}$$

$$\frac{\mathbb{e} \in \cup \mathbb{E}_p^{\mathtt{LDAX}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllLacqs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{RdComCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllSRels}(\mathbb{c}, \mathbb{e}), \quad (CW(\mathbb{c}, \mathbb{e}) = \bot) \vee (CW(\mathbb{c}, \mathbb{e}) = \mathbb{e}' \wedge \mathtt{status}(\mathbb{e}') = \mathtt{com})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}[\mathbb{e} \leftarrow \mathtt{Prop}(\mathtt{Var}(\mathbb{c}, \mathbb{e}))], \mathtt{Prop}, \mathtt{Mark}[\mathtt{var}(\mathbb{e}) \leftarrow \mathtt{GetLoadMark}(\mathbb{e})] \rangle} \mathtt{ComLDAXFromProp}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{ST}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{InitCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllLAcqs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbSts}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{init}], \mathtt{rf}, \mathtt{Prop}, \mathtt{Mark} \rangle} \mathtt{InitST}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{STL}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{InitCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllMem}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbSts}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{init}], \mathtt{Prop}, \mathtt{Mark} \rangle} \mathtt{InitSTL}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{ST}} \cup \mathbb{E}_p^{\mathtt{STL}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{init}, \quad \mathtt{ComCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AddrCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop}[\mathtt{var}(\mathbb{e}) \leftarrow \mathbb{e}], \mathtt{Mark}[\mathtt{var}(\mathbb{e}) \leftarrow \bot] \rangle} \mathtt{ComNonExclusiveWrite}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{STX}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AddrCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllLAcqs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbSts}(\mathbb{c}, \mathbb{e}), \quad \mathtt{ComCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{init}], \mathtt{rf}, \mathtt{Prop}[\mathtt{var}(\mathbb{e}) \leftarrow \mathtt{GetUpdate}(\mathbb{e})], \mathtt{Mark}[\mathtt{var}(\mathbb{e}) \leftarrow \mathtt{GetMark}(\mathbb{e})] \rangle} \mathtt{ComSTX}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{STLX}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{AddrCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{ComCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllSyncs}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbLds}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllMem}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbSts}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop}[\mathtt{var}(\mathbb{e}) \leftarrow \mathtt{GetUpdate}(\mathbb{e})], \mathtt{Mark}[\mathtt{var}(\mathbb{e}) \leftarrow \mathtt{GetMark}(\mathbb{e})] \rangle} \mathtt{ComSTLX}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{DmbSy}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{ComCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllMem}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllBarriers}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop} \rangle} \mathtt{ComDmbSy}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\mathtt{Isb}}, \quad \mathtt{status}(\mathbb{e}) = \mathtt{fetch}, \quad \mathtt{ComCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AddrCnd}(\mathbb{c}, \mathbb{e}), \quad \mathtt{AllDmbSys}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \mathtt{ins}, \mathtt{status}[\mathbb{e} \leftarrow \mathtt{com}], \mathtt{rf}, \mathtt{Prop} \rangle} \mathtt{ComIsb}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{DmbLd}}, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad \texttt{ComCnd}(\mathbb{c}, \mathbb{e}), \quad \texttt{AllReads}(\mathbb{c}, \mathbb{e}), \quad \texttt{AllDmbSys}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}, \texttt{Mark} \rangle} \texttt{ComDmbLd}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{DmbSt}}, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad \texttt{ComCnd}(\mathbb{c}, \mathbb{e}), \quad \texttt{AllWrites}(\mathbb{c}, \mathbb{e}), \quad \texttt{AllDmbSys}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}, \texttt{Mark} \rangle} \texttt{ComDmbSt}$$

$$\frac{\mathbb{e} \in \mathbb{E}_p^{\texttt{ACI}}, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad \texttt{ComCnd}(\mathbb{c}, \mathbb{e}), \quad \texttt{ValidCnd}(\mathbb{c}, \mathbb{e}), \quad \texttt{AllSyncs}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop}, \texttt{Mark} \rangle} \texttt{ComACI}$$

We explain the above rules as thus:

- The `Fetch` rule is the same as before (and is cosmetic as explained above).
- We need the following pre-conditions to hold for committing a non-acquire read (exclusive as well as not) event $\mathbb{e}$ (from either local uncommitted events or memory):
  1. For all events $\mathbb{e}'$ that are poloc-predecessors of $\mathbb{e}$, $\mathbb{e}'$ must have been initialized.
  2. All `dmb.sy`, `isb` and `dmb.ld` instructions that are po-predecessors of $\mathbb{e}$ have been committed.
  3. All po-preceeding load acquire instructions have been committed.
  4. All `data` and `addr` dependents have committed.
- The rule for committing an acquire-read (exclusive as well as not) is similar to that above, but with the extra condition that all `po`-previous Store Releases have been committed.
- One condition required for committing for any kind of write needs that we know the "memory footprint", i.e. address of all `po`-previous memory access events is known.
- To initialize a (non-exclusive, non-release) write, along with an `InitCnd` that we had previously (but which also includes address dependencies now), we need the following condition to hold: all po-previous `dmb.sy`, `isb`, `dmb.ld`, `dmb.st`, and Load Acquire instructions have been committed.
- To initialize a release- but non-exclusive write (`STL`), instead of just load-acquires, we need *all* `po`-preceding memory access instructions to have been committed.
- To commit any write, two of the conditions required are `ComCnd` and `AddrCnd`. The latter postulates that all `po`-previous events have well defined memory footprints.
- The exclusive versions of the two kinds of writes are to be directly committed: an `STX` can be committed only if it satisfies all the conditions mentioned above for simple write - initialization, along with the `ComCnd`. On the other hand, an `STLX` instruction can be committed only if all `po`-preceeding instructions excluding `ACI` events have been committed.
- The condition for committing a non-exclusive write is similar to before, but with the added condition that all `po`-previous synchronization instructions(`dmb.sy` and `isb`) are committed.
- All ACI and barrier instructions have only one transition: from fetch to commit. Across all of them, the `ComCnd` and `ValidCnd` are common conditions, as is the condition that all `dmb.sy`'s have been committed (for `ACI`'s and `dmb.sy`'s we also need `po`-previous `isb`'s to have been committed). Apart from this,
  1. For `dmb.sy`'s we need all `po`-previous memory access instructions to have committed.
  2. For `isb`'s we need all `po`-previous memory access instructions to have fully defined memory footprints.
  3. For `dmb.ld`'s we need all `po`-previous load (read and load acquire) instructions to have been committed.
  4. For `dmb.st`'s we need all `po`-previous stores (write and store release) instructions to have been committed.

Assign instructions. The Herd or the Flat model do not explicitly handle assign instructions. Consider the hypothetical scenario where we assign the value of a register $\$r_1$ to a register $\$r_2$. We would then want the latter to be able to `init`/`commit` just after but no before the former. Generalizing this to any assign, we postulate that an assign can `commit` once all of it's operands (unless they are literals) have `commited`. We also corroborate the validity of this rule via litmus tests. Note that this rule is, in a sense, not a central rule: it is not directly modelled by Herd of Flat, and it's only purpose is to act as a bridge connected many producing `LD`*'s to an event. Thus, it is, in effect, a syntactic sugar and does not represent a "real" (i.e. interfacing with memory) event.

$$\frac{\mathbb{e} \in \mathbb{E}_p^{A}, \quad \texttt{status}(\mathbb{e}) = \texttt{fetch}, \quad \texttt{AssignComCnd}(\mathbb{c}, \mathbb{e})}{\mathbb{c} \xrightarrow{p} \langle \mathbb{E}, \prec, \texttt{ins}, \texttt{status}[\mathbb{e} \leftarrow \texttt{com}], \texttt{rf}, \texttt{Prop} \rangle} \texttt{ComAssign}$$

9 Proof of equivalence to the Herd Model

We give here the proof of equivalence of the above model to the Herd model, as found in section B2.3 of the ARM developer manual. There are two main conditions for the equivalence to hold: the internal and the external visibility requirements. We do not take into account tag reads or tag writes since our model does not include them. We use the `.cat` file available at [6] for our purposes of specification: we note that the requirements are stated as predicates over possible orderings, and are as follows:

```
(* Internal visibility requirement *)
let po-loc =
  let Exp = M\domain(tob) in
  (po-loc & Exp*Exp) | (po & tlo)
acyclic po-loc | ca | rf as internal

(* External visibility requirement *)
irreflexive ob as external

(* Observed-by *)
let obs = rfe | fre | coe
```

```
(* Ordered-before *)
let rec ob = obs; si
         | lob
         | ob; ob

(* Locally-ordered-before *)
let rec lob = lws; si
            | dob
            | aob
            | bob
            | tob
            | lob; lob
```

Above, `po-loc`, `ca` and `rf` stand for meanings consistent with our definitions, and `lob` stands for "locally-ordered-before". Further, `rfe` stands for "read-from-*external*", and so on. The constituents of `lob` are further elaborated as:

```
(* Local write successor *)
let lws = po-loc; [W]

(* Atomic-ordered-before *)
let aob = rmw
        | [range(rmw)]; lrs; [A | Q]

(* Barrier-ordered-before *)
let bob = po; [dmb.full]; po
        | po; ([A];amo;[L]); po
        | [L]; po; [A]
        | [R\NoRet]; po; [dmb.ld]; po
        | [A | Q]; po
        | [W]; po; [dmb.st]; po; [W]
        | po; [L]
```

```
(* Local read successor *)
let lrs = [W]; (po-loc \ (po-loc;[W];po-loc)); [R]

(* Read-modify-write *)
let rmw = lxsx | amo

(* Dependency-ordered-before *)
let dob = addr | data
        | ctrl; [W]
        | (ctrl | (addr; po)); [ISB]; po; [R]
        | addr; po; [W]
        | (addr | data); lrs
```

Above, `si` relates events belonging to the same instruction, and range refers to all the events constituting the `rmw` block (i.e., the read, the comparison and the write). Also, `[A]` and `[L]` refer to acquire and release instructions, respectively.
We further comment that we do not support `amo`-type instructions for simplicity. However, it is straightforward to extend the model to create rules for `amo` instruction. We call our model as `ARM'` and the model presented above as `ARM`. We wish to show that `ARM'` $\equiv$ `ARM`. For this, we prove subsumption in both directions below. Note that we do not consider the `tob` condition since we do not support tags in our model. Further, for our purposes `rmw` is equivalent to a `lx+sx` pair due to non-existence of `amo` instructions under our model. We omit assign events in the following proof since the Herd model does not talk about it; however note that since an assign can take any transition once all of it's operands have, adding an assign is simply equivalent to conjunction of multiple `data` dependencies. As mentioned above, we also verify this through extensive litmus testing.

9.1 All execution traces valid under `ARM'` are valid under `ARM`

First consider the internal visibility requirements. Note that due to the rules, read events always read from either the main memory or from the closest `po`-before write. We also have that writes need all dependents to have been committed before the former are initialized. Consequently, from the above two, the locally observed `poloc`-order is acyclic. Further, `ca` simply stems from `poloc`-order in the case of writes. For `rf`, note that when a read is committed, it's closest write must have either initialized (if it reads via forwarding) or have been committed (if it reads from main memory). In either case, if the event it reads from belongs to the same thread must have already at least been initialized, preserving the `rf` order. Thus no cycle may exist among these pairs: hence the internal order is acyclic, as needed.
Next we come to the external visibility constraints. For `rfe`, note that an event can only read from an event via main memory if the latter has been committed already, preserving `rfe`. Further, an event cannot read from an event that commits after it, preserving the `fre` order. In addition, `coe` is obviously preserved since after a write event e_1 "overwrites" another write event e_2, no event can read from e_2 ever again. These imply that the partial order `obs` is preserved.
Further, from `ComCnd` we conclude that `lws` is preserved.

Next we tackle dob. The first line is obviously satisfied since ComCnd/RdComCnd enforces preservation of addr and data dependencies. Further, note that ComCnd also enforces the second line of dob. In addition, the AddrCnd appears in all the write-commit rules and ensures that the fourth line of dob is respected. The third line holds because a) When committing an ISB event, the AddrCnd and ComCnd ensure the first half of the order holds; and b) We require all ISB's to have been committed while committing reads. Note: We require this even for writes, but note that anyway writes have to commit after any event they are dependent on via ctrl or addr;po so this is a redundant clause and causes no extra conditions for writes. For the final line, note that we need reads to commit not before a write whose lrs the read is, is initialized. Further, said write can init only after it's address and data dependents have committed, effectively proving the last line of dob.
In effect, we have proved that dob is preserved under ARM'.
Next we come to aob. Since we do not model amo instructions, we can focus on lx-sx pairs exclusively. Note that any acquire-load that is a local-read-successor of an exclusive store will require the store to be at least initialized when the former is committed. However, since exclusive stores do not have an init phase, it must have been committed: and when it is committed, it's effects (if any) have been already recorded in main memory. This effect is nil iff the exclusive store was unsuccessful. Hence we conclude that aob holds.
Next we come to bob. The first line follows since dmb.sy's require virtually every preceding event (other than assigns) to have committed, and all events require every po-previous dmb.sy to have committed. The second line is not of consequence since we do not model amo instructions. The third follows from the AllSRels predicate appearing in the commit-rules for acquire-loads. The next follows from the predicate AllReads of ComDmbLd and AllDmbLds that appears while committing any read/write event. The line after that is respected since AllLAcqs appears in all read-write events. Similarly, the sixth line too is respected. Finally, the last line follows from the AllMem predicate that appears in the commit-rules for store-releases. This completes bob. We do not model tags and so tob is skipped.
The final line of lob is transitivity which obviously holds here.
Since lws,dob,aob and bob are preserved, so is lob. Further, since obs is also preserved, we conclude that ob is preserved. Hence, we have that ARM' $\subseteq$ ARM.

9.2 All execution traces valid under ARM are valid under ARM'

To prove this direction, we must reconstruct all of the rules of our model from the predicates of the Herd model. The Fetch rule is not relevant, since given any run that is valid under ARM, one can imagine an equivalent run under ARM', where every instruction is fetched just before it is initialized (so we only need to prove the validity of other rules).
Consider the ComLDFrom* rules. The AllSyncs predicate is justified by the first line of bob and the third line of dob. The AllDmbLds is justified by the fourth line of bob. Further, the AllLacqs predicate is justified by the fifth line of bob. Then, the condition that the closest read has been atleast initialized is derived from the local visibility condition on ca. Further, RdComCnd has two parts: addr and data. Both are justified using the first line of dob.
Next, in the ComLDAFrom* rules, the only extra predicate is AllSRels. This is justified using the third line of bob.
For ComLDXFrom* and ComLDAX* rules, the same arguments hold as above.
The rule InitCnd in InitST follows from the internal visibility condition, for the following reason: if writes could initialize before a read event that supplies, e.g. their data is committed, then we could have a scenario where this same read event reads-from that write, violating the acyclicity of local visibility. The AllSyncs, AllDmbLds and AllLacqs are explained as for reads. The AllDmbSts predicate follows from the sixth line of bob. Also, the part of ComCnd about ACI events is due to the second line of dob.
The only extra predicate in the case of InitSTL is AllMem (which subsumes AllLacqs). This is justified by referring to the last line of bob.
Further, the AddrCnd in ComNonExclusiveWrite is a consequence of the fourth line of dob. The ComCnd is as before, with the poloc order added in to respect the ca order (coe as appears in obs).
The rules for committing exclusive writes are in fact exactly the same for their non exclusive counterparts, with the condition that init and commit happen at the same time: this assumption is sound since those writes may be discarded: their validity is confirmed only when they are committed.
For the ComDmb* rules, the respective lines of bob are enough justification. Further, for dmb.sy's and isb's, the first line of bob and the third line of dob imply the commit of ACI instructions as well.
Thus, we have that ARM $\subseteq$ ARM'.
Combining this with the result of the previous subsection, we conclude that

$$\boxed{\text{ARM}' \equiv \text{ARM}}$$

10 Context Bound Model Checking for ARM

In the spirit of Context Bound Model Checking, we propose a code-to-code translation from source code written under the ARMv8 model to one that runs under Sequential Consistency (SC), but with a bound on the maximum number of contexts of execution. Here, we define a context as a stretch of execution (across time) in which only one process is active. Thus, inside a single context, the program behaves exactly like SC to the executing process (and the other processes are not

executing). This context-bound reduction was first pioneered by Lal and Reps [5], and we adapt their approach (with the required changes) to fit the ARMv8 model. In particular, our code guesses the global state (the state of the main memory) at all context switches. Then, our code simulates, for each process, the contexts in which it is active, starting from the respective guessed global states. In the end, we verify, that for each context k, the process active in context k *transforms* the global state from that guessed at the end of context $k-1$ to that guessed at the end of context k. The advantage of this approach over naive simulation is that it avoids having to deal with cross-products of local spaces, and hence avoids the exponential blowup associated with it. As an end result, we have code that when run under SC successfully simulates all possible runs under the ARM v8 memory model that incur at most k context switches. That is, Context Bound Model Checking is an efficient under-approximation to state reachability. Since it has been demonstrated before that most bugs are (as a general rule of thumb) reachable in a small number of context switches, our approach is an effective means of tackling the undecidability of the ARM v8 memory model.
We first demonstrate the above method on the simple model introduced at the beginning of the document, and then proceed to adapt it to the whole model.

11 Code-to-code translation for reduction to SC (for the simple model)

For the sake of performing context-bound model checking, we next define the scheme for the code-to-code translation.

11.1 Scheme for the first part of model (without synchronization instructions)

Our translation scheme translates a program $Prog$ into a program $Prog^{\clubsuit}$ using the map function $[\![\cdot]\!]_K$. Let $\mathcal{P}$ and $\mathcal{X}$ be the set of processes and shared variables in our program. Then, the map $[\![\cdot]\!]_K$ replaces the variables of $Prog$ by $|\mathcal{P}| \cdot K$ copies of the set $\mathcal{X}$, along with a finite set of finite data structures defined below. Below, the function `gen` takes in a finite set and returns a randomly chosen element of the set.

$$[\![Prog]\!]_K \stackrel{\text{def}}{=} \begin{array}{ll} \texttt{vars:} & x^* \ \langle \texttt{addvars} \rangle_K \\ \texttt{procs:} & ([\![p]\!]_K)^* \ \langle \texttt{initProc} \rangle_K \ \langle \texttt{verProc} \rangle_K \end{array}$$

$$[\![p]\!]_K \stackrel{\text{def}}{=} \begin{array}{ll} \texttt{regs:} & \$r^* \\ \texttt{instrs:} & ([\![i]\!]^p_K)^* \end{array}$$

$$[\![i]\!]^p_K \stackrel{\text{def}}{=} l: \ [\![\texttt{s}]\!]^p_K$$

$$[\![x \leftarrow exp]\!]^p_K \stackrel{\text{def}}{=} [\![x \leftarrow exp]\!]^{\texttt{p,Write}}_K$$

$$[\![\$r \leftarrow x]\!]^p_K \stackrel{\text{def}}{=} [\![\$r \leftarrow x]\!]^{\texttt{p,Read}}_K$$

$$[\![\$r \leftarrow exp]\!]^p_K \stackrel{\text{def}}{=} [\![\$r \leftarrow exp]\!]^{\texttt{p,Assign}}_K$$

$$[\![\texttt{if } exp \texttt{ then } i^* \texttt{ else } i^*]\!]^p_K \stackrel{\text{def}}{=} \texttt{if } exp \texttt{ then } ([\![i]\!]^p_K)^* \texttt{ else } ([\![i]\!]^p_K)^*; \ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{while } exp \texttt{ do } i^*]\!]^p_K \stackrel{\text{def}}{=} \texttt{while } exp \texttt{ do } ([\![i]\!]^p_K)^*; \ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{assume } exp]\!]^p_K \stackrel{\text{def}}{=} \texttt{assume } exp; \ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{assert } exp]\!]^p_K \stackrel{\text{def}}{=} \texttt{assert } exp; \ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{term}]\!]^p_K \stackrel{\text{def}}{=} \texttt{term}$$

$$\begin{array}{c} \langle \texttt{addvars} \rangle_K \stackrel{\text{def}}{=} \mu(|\mathcal{X}|, K), \mu^{init}(|\mathcal{X}|, K), \nu(|\mathcal{P}|, |\mathcal{X}|), \\ \texttt{cR}(|\mathcal{P}|, |\mathcal{X}|), \\ \texttt{iW}(|\mathcal{P}|, |\mathcal{X}|), \texttt{cW}(|\mathcal{P}|, |\mathcal{X}|), \\ \texttt{cReg}(|\mathcal{P}|, |\mathcal{R}|), \\ \texttt{ctrl}(|\mathcal{P}|), \texttt{active}(K) \\ \texttt{assume}(cnt \leq K) \end{array}$$

$$\langle \texttt{control} \rangle^p_K \stackrel{\text{def}}{=} \begin{array}{l} \texttt{ctrl}(p) \leftarrow \texttt{ctrl}(p) + \texttt{gen}(0, \cdots, K-1); \\ \texttt{assume}(\texttt{ctrl}(p) \leq K) \end{array}$$

11.2 Data Structures (for the model without synchronization)

We denote by $\mathcal{D}$ the domain of all possible values of expressions and variables. Our simulation maintains, as described above, a finite set of finite data structures. We explain each in turn. First, for each context k, we store the ID of the active process p in the context k, using the mapping $\texttt{active} : [1, \cdots, K] \mapsto \mathcal{P}$. The mapping $\mu^{init} : \mathcal{X} \times [1, \cdots, K] \mapsto \mathcal{D}$ maintains, for each variable x and context k the last value of the variable x that has been propagated to memory by any process *upto the beginning of context k*. We also define the mapping $\mu : \mathcal{X} \times [1, \cdots, K] \mapsto \mathcal{D}$ as the counterpart of the above mapping that actually changes (gets updated) through the course of context k, i.e. at any given point in time covered by context k, $\mu(x, k)$ gives the latest value of variable x propagated to memory until that point. Further, the mapping $\nu : \mathcal{P} \times \mathcal{X} \mapsto \mathcal{D}$ denotes for each process p and variable x the latest value that has been written to x by p. We also maintain the maps

$$\texttt{iW} : \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]$$

$$\texttt{cW} : \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]$$

$$\texttt{cR} : \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]$$

The first map indicates, for each process p and variable x, the latest context in which a write on x has been initialized by p. Similarly, the second one indicates for each such pair the latest context in which a write was committed by p. Similarly, we define cR for committing reads.
Since registers are shared among all processes, we need to define special mappings for them. Thus we define $\texttt{cReg} : \mathcal{P} \times \mathcal{R} \mapsto [1, \cdots, K]$ captures for each register $\$r$ the committing context of the latest read or assign event loading a value to $\$r$. We also extend this to expressions, by defining $\texttt{cReg}(p, exp) = \max\{\texttt{cReg}(p, \$r) | \$r \in \mathcal{R}(exp)\}$. Finally, the mapping $\texttt{ctrl} : \mathcal{P} \mapsto [1, \cdots, K]$ gives for each process p the committing context of the latest aci event in p. The variable *cnt* tracks the current context.
The function gen is assumed to return, given a set S, a random element of S.

11.3 The initializing process

We next present the algorithm $\langle \texttt{initProc} \rangle_K$.

Algorithm 1: Algorithm $\langle \texttt{initProc} \rangle_K$.

```
1  for p ∈ P ∧ x ∈ X do
2  |  iR(p, x) ← 1;
3  |  iW(p, x) ← 1;
4  |  cR(p, x) ← 1;
5  |  ν(p, x) ← 0;
6  |  μ(p, x, 1) ← 0;
7  |  for k ∈ [2, ··· , K] do
8  |  |  μ^init(x, k) ← gen(D);
9  |  |  μ(x, k) ← μ^init(p, x, k);
10 |  end
11 end
12 for p ∈ P do
13 |  ctrl(p) ← 1;
14 end
15 for $r ∈ R do
16 |  cReg(p, $r) ← 1;
17 end
18 for k ∈ [1, ··· , K] do
19 |  active(k) ← gen([1, ··· , K]);
20 end
21 cnt ← 1;
```

The initial process's job is to initialize all the data structures. We assume that in the beginning all variables are assigned the value 0. If needed we can replace this by a symbolic variable depicting all possible values. The structures which record contexts are all initialized to 1, since that is the earliest context possible. We also assign to each context a randomly chosen active process.

11.4 Write instructions

Algorithm 2: $[\![x \leftarrow exp]\!]_K^{p,\texttt{Write}}$

```
1  //Guess
2  iW(p, x) ← gen([1, ··· , K]);
3  old-cW ← cW(p, x);
4  cW(p, x) ← gen([1, ··· , K]);
5  //Check
6  assume(active(iW(p, x)) = p);
7  assume(iW(p, x) ≥ cReg(p, exp));
8  assume(cW(p, x) ≥ iW(p, x));
9  assume(active(cW(p, x)) = p);
10 assume(cW(p, x) ≥ max(old-cW, cR(p, x), ctrl(p));
11 //Update
12 μ(x, cW(p, x)) ← exp;
13 ν(p, x) ← exp;
```

The above listing shows how write instructions are handled. First, we guess the contexts in which the write will be initialized and committed respectively. Next, we perform a set of sanity checks to ensure that the write conforms to the rule `InitWrite`: line 7 ensures that p is the active process in that context. We then make sure that `InitCnd` is satisfied in line 8. Further, the write must be initialized before committed, as expressed in line 9. Finally, we must ensure that the write is committed not before any `poloc`-preceeding write or read, and also not before any `po`-previous aci instruction or before any instruction which supplies the data for this write. That is expressed in line 10. Then, we update the new values of the global values of x in the respective context, and the latest local value of x.

11.5 Read Instructions

Algorithm 3: $[\![\$r \leftarrow x]\!]_K^{p,\texttt{Read}}$

```
1  //Guess
2  cR(p, x) ← gen([1, ··· , K]);
3  cReg(p, $r) ← cR(p, x);
4  //Check
5  assume(active(cR(p, x)) = p);
6  assume(cR(p, x) ≥ iW(p, x));
7  //Update
8  if iR(p, x) < cW(p, x) then
9  |  $r ← ν(p, x);
10 else
11 |  $r ← μ(x, iR(p, x));
12 end
```

As for writes, we start by guessing the context in which the read is committed. We then perform some sanity checks in the lines 5-6, which we explain next. Line 5 ensures that p is the active process during the `commit` of the read. Line 6 ensures that the read is committed only after the closest po-before write is; this is needed due to the model. If the read is initialized before the closest `po`-before write (say w) is committed, then we must follow the rule `ComReadFromLocal`, as dictated by line 9. Otherwise, the global memory is at least as up-to-date, and we must use the rule `ComReadFromProp` to read from it. This is captured by the update of rule 11.

11.6 Assign Instructions

Algorithm 4: $[\![\$r \leftarrow exp]\!]_K^{p,\texttt{Assign}}$

```
1  //Guess
2  cReg(p, $r) ← gen([1, ··· , K]);
3  //Check
4  for $r' ∈ R(exp) do
5  |  assume(cReg(p, $r) ≥ cReg(p, $r'));
6  end
7  //Update
8  $r ← exp;
```

We first guess the commiting context of the assign statement. We then check that it commits no earlier than any of the events that supply it with the operand values, in line 5. The update step is simple: we just record the new value of the register $\$r$.

11.7 Verifying process

Algorithm 5: $\langle \texttt{verProc} \rangle_K$.

```
for p ∈ 𝒫 ∧ x ∈ 𝒳 ∧ k ∈ [1, ··· , K − 1] do
    assume(μ(p, x, k) = μ^init(p, x, k + 1));
end
if l is reachable then
    error;
end
```

The verifying process simply makes sure that the modifications to global state by a process in a context leaves it exactly in the state that the process of the next context finds it. If this is satisfied, then the execution is valid and the *error* state is reachable if and only if a bad state can be reached in this reduced SC program.

12 Code-to-code translation for reduction to SC (for the complete model)

12.1 Scheme for the complete model

Our translation scheme translates a program $Prog$ into a program $Prog^{\clubsuit}$ using the map function $[\![\cdot]\!]_K$. Let $\mathcal{P}$ and $\mathcal{X}$ be the set of processes and shared memory locations in our program. Then, the map $[\![\cdot]\!]_K$ replaces the memory of $Prog$ by $|\mathcal{P}| \cdot K$ copies of the set $\mathcal{X}$, along with a finite set of finite data structures defined below. Below, the function `gen` takes in a finite set S and returns a randomly chosen element of S.

$$[\![Prog]\!]_K \stackrel{\text{def}}{=} \langle \texttt{addvars} \rangle_K \quad \texttt{procs:}\ ([\![p]\!]_K)^*\ \langle \texttt{initProc} \rangle_K\ \langle \texttt{verProc} \rangle_K$$

$$[\![p]\!]_K \stackrel{\text{def}}{=} \texttt{regs:}\ \ \$r^* \quad \texttt{instrs:}\ \ ([\![i]\!]^p_K)^*$$

$$[\![i]\!]^p_K \stackrel{\text{def}}{=} l : [\![\texttt{s}]\!]^p_K$$

$$[\![\$r \leftarrow exp]\!]^p_K \stackrel{\text{def}}{=} [\![\$r \leftarrow exp]\!]^{\texttt{p,Assign}}_K$$

$$[\![\texttt{LD}\ \$r' \leftarrow [\$r]]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{LD}\ \$r' \leftarrow [\$r]]\!]^{\texttt{p,LD}}_K$$

$$[\![\texttt{ST}\ [\$r'] \leftarrow \$r]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{ST}\ [\$r'] \leftarrow \$r]\!]^{\texttt{p,ST}}_K$$

$$[\![\texttt{LDA}\ \$r' \leftarrow [\$r]]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{LDA}\ \$r' \leftarrow [\$r]]\!]^{\texttt{p,LDA}}_K$$

$$[\![\texttt{STL}\ [\$r'] \leftarrow \$r]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{STL}\ [\$r'] \leftarrow \$r]\!]^{\texttt{p,STL}}_K$$

$$[\![\texttt{LDX}\ \$r' \leftarrow [\$r]]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{LDX}\ \$r' \leftarrow [\$r]]\!]^{\texttt{p,LDX}}_K$$

$$[\![\texttt{STX}\ \$r'',\ [\$r'] \leftarrow \$r]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{STX}\ \$r'',\ [\$r'] \leftarrow \$r]\!]^{\texttt{p,STX}}_K$$

$$[\![\texttt{LDAX}\ \$r'',\ \$r' \leftarrow [\$r]]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{LDAX}\ \$r' \leftarrow [\$r]]\!]^{\texttt{p,LDAX}}_K$$

$$[\![\texttt{STLX}\ [\$r'] \leftarrow \$r]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{STLX}\ \$r'',\ [\$r'] \leftarrow \$r]\!]^{\texttt{p,STLX}}_K$$

$$[\![\texttt{if}\ exp\ \texttt{then}\ i^*\ \texttt{else}\ i^*]\!]^p_K \stackrel{\text{def}}{=} \texttt{if}\ exp\ \texttt{then}\ ([\![i]\!]^p_K)^*\ \texttt{else}\ ([\![i]\!]^p_K)^*;\ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{while}\ exp\ \texttt{do}\ i^*]\!]^p_K \stackrel{\text{def}}{=} \texttt{while}\ exp\ \texttt{do}\ ([\![i]\!]^p_K)^*;\ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{assume}\ exp]\!]^p_K \stackrel{\text{def}}{=} \texttt{assume}\ exp;\ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{assert}\ exp]\!]^p_K \stackrel{\text{def}}{=} \texttt{assert}\ exp;\ \langle \texttt{control} \rangle^p_K$$

$$[\![\texttt{acquire}\ \$\texttt{r} \leftarrow [exp]]\!] \stackrel{\text{def}}{=} [\![\$r \leftarrow [exp]]\!]^{\texttt{p,LAcq}}_K$$

$$[\![\texttt{release}\ [\texttt{exp'}] \leftarrow exp]\!]^p_K \stackrel{\text{def}}{=} [\![[exp'] \leftarrow exp]\!]^{\texttt{p,SRel}}_K$$

$$[\![\texttt{dmb.ld}]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{dmb.ld}]\!]^{\texttt{p,DmbLd}}_K$$

$$[\![\texttt{dmb.st}]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{dmb.st}]\!]^{\texttt{p,DmbSt}}_K$$

$$[\![\texttt{dmb.sy}]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{dmb.sy}]\!]^{\texttt{p,DmbSy}}_K$$

$$[\![\texttt{isb}]\!]^p_K \stackrel{\text{def}}{=} [\![\texttt{isb}]\!]^{\texttt{p,isb}}_K$$

$$[\![\texttt{term}]\!]^p_K \stackrel{\text{def}}{=} \texttt{term}$$

$$\begin{aligned}\langle \texttt{addvars} \rangle_K \stackrel{\text{def}}{=} &\mu(|\mathcal{X}|, K), \mu^{init}(|\mathcal{X}|, K), \nu(|\mathcal{P}|, |\mathcal{X}|),\\ &\delta(|\mathcal{X}|, K), \delta^{init}(|\mathcal{X}|, K),\\ &\texttt{cR}(|\mathcal{P}|, |\mathcal{X}|),\\ &\texttt{iW}(|\mathcal{P}|, |\mathcal{X}|), \texttt{cW}(|\mathcal{P}|, |\mathcal{X}|),\\ &\texttt{cReg}(|\mathcal{P}|, |\mathcal{R}|),\\ &\texttt{cL}(|\mathcal{P}|, |\mathcal{R}|),\\ &\texttt{iS}(|\mathcal{P}|, |\mathcal{X}|), \texttt{cS}(|\mathcal{P}|, |\mathcal{X}|),\\ &\texttt{cDY}(|\mathcal{P}|), \texttt{cDL}(|\mathcal{P}|), \texttt{cDS}(|\mathcal{P}|), \texttt{cISB}(|\mathcal{P}|),\\ &\texttt{ctrl}(|\mathcal{P}|), \texttt{cAddr}(|\mathcal{P}|), \texttt{active}(K), cnt\end{aligned}$$

$$\langle \texttt{activeCnt} \rangle^p_K \stackrel{\text{def}}{=} \texttt{assume}(\texttt{active}(\texttt{cnt}) = p)$$

$$\langle \texttt{closeCnt} \rangle^p_K \stackrel{\text{def}}{=} cnt \leftarrow cnt + \texttt{gen}([0, \cdots, K-1]);\ \texttt{assume}(cnt \leq K)$$

$$\langle \texttt{control} \rangle^p_K \stackrel{\text{def}}{=} \texttt{ctrl}(p) \leftarrow \texttt{ctrl}(p) + \texttt{gen}(0, \cdots, K-1);\ \texttt{assume}(\texttt{ctrl}(p) \leq K)$$

12.2 Additional data structures for the complete model

We use all of the data structures described in the section of the simple model, with a few minor changes. One, the set $\mathcal{X}$ now refers to the set of shared memory locations instead of shared variables. Further, functions such as cR are updated by all four types of reads, and similar conditions hold for the other three analogous functions. Since some predicates in this model are over just load-acquires or store-releases, we also define the predicates:

$$\begin{aligned}\texttt{cL} &: \mathcal{P} \times \mathcal{R} \mapsto [1, \cdots, K]\\ \texttt{iS} &: \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]\\ \texttt{cS} &: \mathcal{P} \times \mathcal{X} \mapsto [1, \cdots, K]\\ \texttt{cDY} &: \mathcal{P} \mapsto [1, \cdots, K]\\ \texttt{cDS} &: \mathcal{P} \mapsto [1, \cdots, K]\\ \texttt{cDL} &: \mathcal{P} \mapsto [1, \cdots, K]\\ \texttt{cISB} &: \mathcal{P} \mapsto [1, \cdots, K]\end{aligned}$$

The first four describe, for each process-register or process-variable pair, the initializing and committing contexts of the latest STL or instruction that loads to, and the latest SRel instruction that stores from that register/to that variable. There is however one change that we must keep in mind: now the set $\mathcal{X}$ consists of addressable memory regions (as opposed to variables). Also note that since exclusive stores directly commit, we use their committing context to also update iW, since later instructions need them to be "at least" initialized, and so we need these two functions to take into account their committing (it's as if they initialized and spontaneously committed). Also, noting that the predicates on LAcq or SRel that appear in our rules, such as AllLAcqs, are not about those to a particular variable but over all of them, it will be helpful to define for $p \in \mathcal{P}$:

$$\texttt{iS}(p) = \texttt{max}_{x \in \mathcal{X}} \texttt{iS}(p, x)$$

We similarly define $\texttt{iL}(p)$, $\texttt{cL}(p)$ and $\texttt{cS}(p)$. The last four are the analogues for the four types of barrier instructions respectively: dmb.sy, dmb.st, dmb.ld and isb.

We shall also need to track the tags of memory locations. To this end, we introduce two new data structures, $\delta^{init} : \mathcal{X} \times [2, \cdots, K] \mapsto \mathcal{P} \cup \{\bot\}$, and $\delta : \mathcal{X} \times [1, \cdots, K] \mapsto \mathcal{P} \cup \{\bot\}$. These give the value of the tag of a memory location in a particular context: the first at the beginning of the context, and the second at any given moment in time (i.e. it gets updated as more and more transitions are taken in the corresponding context). During implementation, we can use, e.g., 0 to represent $\bot$.

With these structures defined, we now elaborate upon the components of the code-to-code translation above. Below, when we refer to a register in the pseudo-code, such as $\$r$, we are actually referring to the *value* stored in the register. Thus, for example, $\texttt{cW}(p, [\$r])$ refers to the context of latest write-commit that writes to the location whose address is given by the then-present-value in $\$r$.

Finally, we need to track, for the sake of isb and write events, the latest committing context of any instruction that supplies addr-data to a read or write: this is because when an isb is committed we need all po-preceding instructions to have well defined memory footprints. So, we define

$$\texttt{cAddr} : \mathcal{P} \mapsto [1, \cdots, K]$$

which gives for each process P, the maximum initializing context of an instruction that supplies the address to another instruction seen so far.

12.3 The initializing process

Algorithm 6: Algorithm $\langle \texttt{initProc} \rangle_K$.

```
1  for p ∈ 𝒫 do
2      for x ∈ 𝒳 do
3          iR(p, x) ← 1;
4          iW(p, x) ← 1;
5          cW(p, x) ← 1;
6          cR(p, x) ← 1;
7          ν(p, x) ← 0;
8          iS(p, x) ← 1;
9          cS(p, x) ← 1;
10     end
11     for $r ∈ ℛ do
12         cL(p, r) ← 1;
13         iS(p, r) ← 1; cS(p, r) ← 1;
14         cReg(p, $r) ← 1;
15     end
16     ctrl(p) ← 1;
17     cAddr(p) ← 1;
18     cDY(p) ← 1;
19     cDS(p) ← 1;
20     cDL(p) ← 1;
21 end
22 for k ∈ [1, ··· , K] do
23     active(k) ← gen(𝒫);
24     for x ∈ 𝒳 do
25         if k ≠ 1 then
26             μ^init(x, k) ← ⊥;
27             δ^init(x, k) ← ⊥;
28         end
29         μ(x, k) ← ⊥;
30         δ(x, k) ← ⊥;
31     end
32 end
```

The algorithm is largely the same as before. We add statements that initialize the values of the new structures defined above to 1.

Implementation detail. In case certain registers and/or memory contents are to be set to initial values, the statements for doing so are added at the end of `initProc` as well. Further, we can use -1 as a representative for $\perp$ in the implementation.

12.4 Write Statements

First we show the algorithm for ST statements.

Algorithm 7: $[\![\texttt{ST}\ \ [\$r'] \leftarrow \$r]\!]_K^{p,\texttt{ST}}$

```
1  //Guess
2  iW(p, [$r']) ← gen([1, ··· , K]);
3  old-cW ← cW(p, [$r']);
4  cW(p, [$r']) ← gen([1, ··· , K]);
5  //Check
6  assume(active(iW(p, [$r'])) = p);
7  assume(iW(p, [$r']) ≥ max(cReg(p, $r'), cReg(p, $r)));
8  assume(iW(p, [$r']) ≥ max(cDY(p), cISB(p)));
9  assume(iW(p, [$r']) ≥ max(cDS(p), cDL(p)));
10 for $r ∈ R do
11 |  assume(iW(p, [$r']) ≥ cL(p, $r));
12 end
13 assume(active(cW(p, [$r'])) = p);
14 assume(cW(p, [$r']) ≥ old-cW);
15 assume(cW(p, [$r']) ≥ iW(p, [$r']));
16 assume(cW(p, [$r']) ≥ cR(p, [$r']));
17 assume(cW(p, [$r']) ≥ max(cAddr(p), ctrl(p));
18 //Update
19 cAddr(p) ← max(cAddr(p), cReg(p, $r'));
20 μ([$r'], cW(p, [$r'])) ← $r;
21 ν(p, [$r']) ← $r;
22 δ([$r'], cW(p, [$r'])) ← ⊥;
```

We first guess the initializing and committing contexts of the ST in lines 1-3. Then, we run it through a bunch of sanity checks. At lines 6 and 13 we check that the process p is active during init and commit, respectively. Then we need that the two register values are ready, i.e. the events providing them have been committed, at the moment when the event is initialized: this is expressed by line 7. Line 15 posits that the instruction is committed no sooner than it is initialized, and lines 8 and 9 ensure that all po-preceeding barriers are committed. We also need all po-previous load-acquires to have been initalized no later than this write is: that is checked in line 11. We then ensure the in-order commit of poloc-related instructions by comparing with old-cW and cR(...) in lines 14 and 16. Finally, line 17 checks that 1) any po-preceding ACI instruction has been committed, and 2) all po-previous memory-access instructions have fully determined addresses ("memory footprints"). Once all the checks pass, we update the respective data structures in lines 21 through 24.
We next present the other three types of writes. Since the changes are minor, we only highlight the differences in text.

Algorithm 8: $[\![\texttt{STL}\ \ [\$r'] \leftarrow \$r]\!]_K^{p,\texttt{STL}}$

1 //Guess
2 $\texttt{iW}(p, [\$r']) \leftarrow \texttt{gen}([1, \cdots, K])$;
3 $\texttt{old-cW} \leftarrow \texttt{cW}(p, [\$r'])$;
4 $\texttt{cW}(p, [\$r']) \leftarrow \texttt{gen}([1, \cdots, K])$;
5 $\texttt{iS}(p, [\$r']) \leftarrow \texttt{iW}(p, [\$r'])$;
6 $\texttt{cS}(p, [\$r']) \leftarrow \texttt{cW}(p, [\$r'])$;
7 //Check
8 $\texttt{assume}(\texttt{active}(\texttt{iW}(p, [\$r'])) = p)$;
9 $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq \texttt{max}(\texttt{cReg}(p, \$r'), \texttt{cReg}(p, \$r)))$;
10 $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq \texttt{max}(\texttt{cDY}(p), \texttt{cISB}(p)))$;
11 $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq \texttt{max}(\texttt{cDS}(p), \texttt{cDL}(p)))$;
12 **for** $x \in \mathcal{X}$ **do**
13 | $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq \texttt{max}(\texttt{cR}(p, x), \texttt{cW}(p, x)))$;
14 **end**
15 $\texttt{assume}(\texttt{active}(\texttt{cW}(p, [\$r'])) = p)$;
16 $\texttt{assume}(\texttt{cW}(p, [\$r']) \geq \texttt{old-cW})$;
17 $\texttt{assume}(\texttt{cW}(p, [\$r']) \geq \texttt{iW}(p, [\$r']))$;
18 $\texttt{assume}(\texttt{cW}(p, [\$r']) \geq \texttt{cR}(p, [\$r']))$;
19 $\texttt{assume}(\texttt{cW}(p, [\$r']) \geq \texttt{max}(\texttt{cAddr}(p), \texttt{ctrl}(p)))$;
20 //Update
21 $\texttt{cAddr}(p) \leftarrow \texttt{max}(\texttt{cAddr}(p), \texttt{cReg}(p, \$r'))$;
22 $\mu([\$r'], \texttt{cW}(p, [\$r'])) \leftarrow \$r$;
23 $\nu(p, [\$r']) \leftarrow \$r$;
24 $\delta([\$r'], \texttt{cW}(p, [\$r'])) \leftarrow \bot$;

The difference of this from the previous listing is only that instead of just load-acquires, we need all memory-access instructions that precede this one in program order to have been committed: this change results in line 11.

Algorithm 9: $[\![\texttt{STX}\ \ \$r'',\ [\$r'] \leftarrow \$r]\!]_K^{p,\texttt{STX}}$

1 //Guess
2 $\texttt{old-cW} \leftarrow \texttt{cW}(p, [\$r'])$;
3 $\texttt{new-cW} \leftarrow \texttt{gen}([1, \cdots, K])$;
4 //Check
5 $\texttt{assume}(\texttt{active}(\texttt{new-cW}) = p)$;
6 $\texttt{assume}(\texttt{new-cW} \geq \texttt{max}(\texttt{cDY}(p), \texttt{cISB}(p)))$;
7 $\texttt{assume}(\texttt{new-cW} \geq \texttt{max}(\texttt{cDS}(p), \texttt{cDL}(p)))$;
8 **for** $\$r \in \mathcal{R}$ **do**
9 | $\texttt{assume}(\texttt{iW}(p, [\$r']) \geq \texttt{cL}(p, \$r))$;
10 **end**
11 $\texttt{assume}(\texttt{new-cW} \geq \texttt{max}(\texttt{old-cW}, \texttt{cR}(p, [\$r'])))$;
12 $\texttt{assume}(\texttt{new-cW} \geq \texttt{max}(\texttt{cReg}(p, \$r), \texttt{cReg}(p, \$r')))$;
13 $\texttt{assume}(\texttt{new-cW} \geq \texttt{max}(\texttt{cAddr}(p), \texttt{ctrl}(p)))$;
14 //Update
15 $\texttt{cAddr}(p) \leftarrow \texttt{max}(\texttt{cAddr}(p), \texttt{cReg}(p, \$r'))$;
16 **if** $\delta([\$r'], \textit{new-cW}) = p$ **then**
17 | $\mu([\$r'], \texttt{new-cW}) \leftarrow \$r$;
18 | $\nu(p, [\$r']) \leftarrow \$r$;
19 | $\delta([\$r'], \texttt{new-cW}) \leftarrow \bot$;
20 | $\$r'' \leftarrow 0$;
21 | $\texttt{cW}(p, [\$r']) \leftarrow \texttt{new-cW}$;
22 | $\texttt{iW}(p, [\$r']) \leftarrow \texttt{cW}(p, [\$r'])$;
23 **else**
24 | $\$r'' \leftarrow 1$;
25 **end**
26 $\texttt{cReg}(p, \$r'') \leftarrow \texttt{new-cW}$;

This listing is still very similar to the first one: the difference is that all changes (except those to $\$r''$ and cAddr) are predicated on the "success" of the store-exclusive instruction. The same difference applied to the second listing leads to the one below.

Algorithm 10: $[\![\texttt{STLX}\ \$r'',\ [\$r']\leftarrow \$r]\!]_K^{p,\texttt{STLX}}$

1 //Guess
2 old-cW $\leftarrow$ cW$(p,[\$r'])$;
3 new-cW $\leftarrow$ gen$([1,\cdots,K])$;
4 //Check
5 assume(active(new-cW) $= p$);
6 assume(new-cW $\geq$ max(cDY(p), cISB(p)));
7 assume(new-cW $\geq$ max(cDS(p), cDL(p)));
8 **for** $\$r \in \mathcal{R}$ **do**
9 | assume(iW$(p,[\$r'])\geq$ max(cR(p,x), cW(p,x)));
10 **end**
11 assume(new-cW $\geq$ max(old-cW, cR$(p,[\$r'])$);
12 assume(new-cW $\geq$ max(cReg$(p,\$r)$, cReg$(p,\$r')$);
13 assume(new-cW $\geq$ max(cAddr(p), ctrl(p));
14 //Update
15 cAddr$(p)\leftarrow$ max(cAddr(p), cReg$(p,\$r')$);
16 **if** $\delta([\$r'],$ *new-cW*$) = p$ **then**
17 | $\mu([\$r'],$ new-cW$)\leftarrow \$r$;
18 | $\nu(p,[\$r'])\leftarrow \$r$;
19 | $\delta([\$r'],$ new-cW$)\leftarrow\perp$;
20 | $\$r''\leftarrow 0$;
21 | cW$(p,[\$r'])\leftarrow$ new-cW;
22 | iW$(p,[\$r'])\leftarrow$ cW$(p,[\$r'])$;
23 | iS$(p,[\$r'])\leftarrow$ iW$(p,[\$r'])$;
24 | cS$(p,[\$r'])\leftarrow$ cW$(p,[\$r'])$;
25 **else**
26 | $\$r''\leftarrow 1$;
27 **end**
28 cReg$(p,\$r'')\leftarrow$ new-cW;

Implementation detail. For write (and read) instructions with offsets, we reserve a private register. We first `assign` the sum of the base and offset to this private register followed by one of the above listings, in order to replicate the effect of a write-to-offset style instruction. This preserves correctness since, for an assign instruction, the only condition is that it is not initialized(committed) before any of its operands are initialized(committed): hence, every execution trace valid without this hack still remains so.

12.5 Assign Instructions

Algorithm 11: $[\![\$r\leftarrow exp]\!]_K^{p,\texttt{Assign}}$

1 //Guess
2 cReg$(p,\$r)\leftarrow$ gen$([1,\cdots,K])$;
3 //Check
4 **for** $\$r' \in \mathcal{R}(exp)$ **do**
5 | assume(cReg$(p,\$r)\geq$ cReg$(p,\$r')$);
6 **end**
7 //Update
8 $\$r\leftarrow exp$;

Assign instructions are simple: the only condition we have on them is that they cannot `commit` before every one of their operands (which are part of exp) have done so, as expressed in line 5.

12.6 Read Instructions

We show, in order, the listings for LD, LDA, LDX and LDAX respectively.

Algorithm 12: $[\![\texttt{LD}\ \$r' \leftarrow [\$r]]\!]_K^{p,\texttt{LD}}$

```
1  //Guess
2  cR(p, [$r]) ← gen([1, ··· , K]);
3  //cReg assigned later since may be needed during check
4  //Check
5  assume(active(cR(p, [$r])) = p);
6  assume(cR(p, [$r]) ≥ iW(p, [$r]));
7  assume(cR(p, [$r]) ≥ cReg(p, $r));
8  assume(cR(p, [$r]) ≥ max(cDY(p), cISB(p));
9  assume(cR(p, [$r]) ≥ cDL(p));
10 for $r'' ∈ R do
11 |  assume(cR(p, [$r]) ≥ cL(p, $r''));
12 end
13 //Update
14 cReg(p, $r') ← cR(p, [$r]);
15 cAddr(p) ← max(cAddr(p), cReg(p, $r));
16 if cR(p, [$r]) < cW(p, [$r]) then
17 |  $r' ← ν(p, [$r]);
18 else
19 |  $r' ← μ([$r], cR(p, [$r]));
20 end
```

The conditions for read instructions, apart from the obvious one about process p being active during commmit, are: 1) we need all dmb.sy's, isb's and dmb.ld's that precede this instruction in program order, to have been committed before it is committed, as lines 8-9 describe. We also need all events supplying the address to have been committed before it, as in line 7. We also require that all poloc-previous writes have initialized, as in line 6. Once the sanity checks are done, we update the required data structures to indicate that the LD instruction has committed.
We next present the listings for the other three types of reads. The modifications as compared to the above listing are minor and obvious, so we do not elaborate upon them.

Algorithm 13: $[\![\texttt{LDA}\ \$r' \leftarrow [\$r]]\!]_K^{p,\texttt{LDA}}$

```
//Guess
cR(p, [$r]) ← gen([1, ··· , K]);
//cReg,cL assigned later since may be needed during check
//Check
assume(active(cR(p, [$r])) = p);
assume(cR(p, [$r]) ≥ iW(p, [$r]));
assume(cR(p, [$r]) ≥ cReg(p, $r));
assume(cR(p, [$r]) ≥ max(cDY(p), cISB(p));
assume(cR(p, [$r]) ≥ cDL(p));
for $r'' ∈ R do
    assume(cR(p, [$r]) ≥ cL(p, $r''));
end
for x ∈ X do
    assume(cR(p, [$r]) ≥ cS(p, x));
end
//Update
cReg(p, $r') ← cR(p, [$r]);
cL(p, $r') ← cR(p, [$r]);
cAddr(p) ← max(cAddr(p), cReg(p, $r));
if cR(p, [$r]) < cW(p, [$r]) then
    $r' ← ν(p, [$r]);
else
    $r' ← μ([$r], cR(p, [$r]));
end
```

Algorithm 14: $[\![\texttt{LDX}\ \$r' \leftarrow [\$r]]\!]_K^{p,\texttt{LDX}}$

```
//Guess
cR(p, [$r]) ← gen([1, ··· , K]);
//cReg assigned later since may be needed during check
//Check
assume(active(cR(p, [$r])) = p);
assume(cR(p, [$r]) ≥ iW(p, [$r]));
assume(cR(p, [$r]) ≥ cReg(p, $r));
assume(cR(p, [$r]) ≥ max(cDY(p), cISB(p));
assume(cR(p, [$r]) ≥ cDL(p));
for $r'' ∈ R do
    assume(cR(p, [$r]) ≥ cL(p, $r''));
end
//Update
cReg(p, $r') ← cR(p, [$r]);
cAddr(p) ← max(cAddr(p), cReg(p, $r));
if cR(p, [$r]) < cW(p, [$r]) then
    $r' ← ν(p, [$r]);
else
    $r' ← μ([$r], cR(p, [$r]));
end
δ([$r], cR(p, [$r])) ← p;
```

Algorithm 15: $[\![\texttt{LDAX } \$r' \leftarrow [\$r]]\!]_K^{p,\texttt{LDAX}}$

```
1  //Guess
2  cR(p, [$r]) ← gen([1, ··· , K]);
3  //cReg,cL assigned later since may be needed during check
4  //Check
5  assume(active(cR(p, [$r])) = p);
6  assume(cR(p, [$r]) ≥ iW(p, [$r]));
7  assume(cR(p, [$r]) ≥ cReg(p, $r));
8  assume(cR(p, [$r]) ≥ max(cDY(p), cISB(p));
9  assume(cR(p, [$r]) ≥ cDL(p));
10 for $r'' ∈ R do
11 |  assume(cR(p, [$r]) ≥ cL(p, $r''));
12 end
13 for x ∈ X do
14 |  assume(cR(p, [$r]) ≥ cS(p, x));
15 end
16 //Update
17 cReg(p, $r') ← cR(p, [$r]);
18 cL(p, $r') ← cR(p, [$r]);
19 cAddr(p) ← max(cAddr(p), cReg(p, $r));
20 if cR(p, [$r]) < cW(p, [$r]) then
21 |  $r' ← ν(p, [$r]);
22 else
23 |  $r' ← μ([$r], cR(p, [$r]));
24 end
25 δ([$r], cR(p, [$r])) ← p;
```

Implementation detail. The procedure for offset-ed reads is the same as that for offset-ed writes: see the corresponding subsection for details.

12.7 Barrier Instructions

Algorithm 16: $[\![\texttt{dmb.sy}]\!]_K^{p,\texttt{DmbSy}}$

```
1  //Guess
2  old-cDY ← cDY(p);
3  cDY(p) ← gen[1, ··· , K];
4  //Check
5  assume(cDY(p) ≥ max(old-cDY, cISB(p)));
6  assume(cDY(p) ≥ max(cDL(p), cDS(p)));
7  assume(cDY(p) ≥ ctrl(p));
8  for x ∈ X do
9  |  assume(cDY(p) ≥ cW(p, x));
10 |  assume(cDY(p) ≥ cR(p, x));
11 end
```

We first guess the context in which the `dmb.sy` will be committed, and then run it through some sanity checks. The `AllBarriers` predicate is spread across lines 5 and 6. The `AllMem` predicate appears in lines 9-10. Line 7 ensures that all `po`-previous ACI instructions have been committed.
Next up is the ISB instruction:

Algorithm 17: $[\![\texttt{isb}]\!]_K^{p,\texttt{Isb}}$

```
1  //Guess
2  cISB(p) ← gen[1, ··· , K];
3  //Check
4  assume(cISB(p) ≥ max(cDY(p), ctrl(p)));
5  assume(cISB(p) ≥ cAddr(p));
```

For an ISB instruction, at the time of commit we need that all `po`-preceding `dmb.sy` and ACI instructions are committed; and also that all `po`-preceding memory access instructions have fully defined addresses: these conditions are a part of lines 4,4 and 5 respectively.

Algorithm 18: $[\![\texttt{dmb.ld}]\!]_K^{p,\texttt{DmbLd}}$

```
1 //Guess
2 cDL(p) ← gen[1, ··· , K];
3 //Check
4 assume(cDL(p) ≥ cDY(p));
5 for x ∈ X do
6 |  assume(cDY(p) ≥ cR(p, x));
7 end
```

Again, for a `dmb.ld` we first guess the committing context for it, in line 2. Then, we run it through the `AllDmbSys` and `AllReads` predicates. The first is in line 4, and the second is in line 6.

Algorithm 19: $[\![\texttt{dmb.st}]\!]_K^{p,\texttt{DmbSt}}$

```
1 //Guess
2 cDS(p) ← gen[1, ··· , K];
3 //Check
4 assume(cDS(p) ≥ cDY(p));
5 for x ∈ X do
6 |  assume(cDY(p) ≥ cW(p, x));
7 end
```

12.8 ACI statements

As shown in the translation schemes, the ACI statements themselves are retained as `if-then-else` C-statements. However, apart from the ⟨`control`⟩, we also need the condition that `control`(p) is greater than or equal to the committing context of the register it depends on (not committing). Thus, for example, if we have a `CBNZ` instruction:

```
CBNZ $r, label
```

Then we need that after modification, `ctrl`$(p) \geq$ `cReg`$(p, \$r)$.
Implementation detail. For instructions such as `BEQ` which follow a `CMP`, we do the following: during the `CMP` we simply assign the two operand-registers to two fixed private registers of the process, and consider `BEQ` and other such instructions as ACI instructions that use these fixed private registers.

12.9 Verifying process

Algorithm 20: ⟨`verProc`⟩$_K$.

```
1 for x ∈ X ∧ k ∈ [1, ··· , K − 1] do
2 |  assume(μ(x, k) = μ^init(x, k + 1));
3 |  assume(δ(x, k) = δ^init(x, k + 1));
4 end
5 if l is reachable then
6 |  error;
7 end
```

The verifying process is the same as in section 10: it simply ensures that each context leaves the global state in a consistent state for the next one.
Implementation detail. The checking for the final conditions (for which we are to determine reachability) are the contents of line 5.

13 Implementation and Evaluation

We first implemented the memory model in the form of a harness which could run a given concurrent program, and could search the state space both randomly and exhaustively. While this is fast enough for small litmus tests, it happens to be too slow for large programs. To this end, we implemented the code-to-code translation presented above in the form of a python script which reads in input multithreaded programs and translates them into a c program running under SC which can be directly fed to CBMC. Since ARMv8 is known to be undecidable, we apply, as presented in the above sections, a context bound (set to 10 by default) on the translated process. We evaluated the correctness of the model and translation on two sets of litmus suites: one of 26 litmus tests from the Herd model [2], and another set of 757 litmus tests generated using the diy7 tool [7]. The former tests the model on simple statements such as `ST`, `LD` and `BEQ` statements, while the latter introduces more complex instructions such as exclusive memory access instructions and loops. Both the harness and the original herd model take a significant amount of time on these tests: the harness takes well over 4 hours on some while herd7 takes a

few 10s of minutes in the worst case. In comparison, the worst runtime of the translated code output from our translation scheme was below 30 seconds.
The source code of both the model and the translation is available at https://github.com/testzer0/Arm2SC/ .

14 Conclusion

In this document, we first presented an operational semantics-version of the AArch64/ARMv8 memory model which we then showed to accurately mirror the official semantics via both a formal proof and using litmus tests. Then we developed an efficient under-approximation technique to perform model checking for programs under this model, inspired by the context-bounded model checking approach taken by Lal and Reps [5]. Finally, we demonstrated that this approach has acceptable runtimes for use in larger scale projects.
This being a work in progress, in future work, we intend to explore the lazy approach to context bounded model checking and optimize further the runtimes of the model checker.

References

[1] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. What's decidable about weak memory models? In Helmut Seidl, editor, *Programming Languages and Systems*, pages 26–46, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats - modelling, simulation, testing, and data-mining for weak memory, 2014.

[3] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: Multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. Context-bounded model checking for power, 2019.

[5] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 37–51, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[6] INRIA. The .cat file for the AArch64 (ARMv8) memory model. `http://diy.inria.fr/www/weblib/aarch64-v06.cat.html`, 2020. [Online].

[7] INRIA. The diy tool. `http://diy.inria.fr/`, 2015. [Online].