A wide-spectrum language for verification of programs on weak memory models

Robert J. Colvin and Graeme Smith

School of Information Technology and Electrical Engineering
University of Queensland

Abstract. Modern processors deploy a variety of weak memory models, which for efficiency reasons may (appear to) execute instructions in an order different to that specified by the program text. The consequences of instruction reordering can be complex and subtle, and can impact on ensuring correctness. Previous work on the semantics of weak memory models has focussed on the behaviour of assembler-level programs. In this paper we utilise that work to extract some general principles underlying instruction reordering, and apply those principles to a wide-spectrum language encompassing abstract data types as well as low-level assembler code. The goal is to support reasoning about implementations of data structures for modern processors with respect to an abstract specification.
Specifically, we encode a weak memory model in a pair-wise reordering relation on instructions. Some architectures require an additional definition of the behaviour of the global storage system if it does not provide multi-copy atomicity. In this paper we use the reordering relation in an operational semantics. We derive some properties of program refinement under weak memory models, and encode the semantics in the rewriting engine Maude as a model-checking tool. The tool is used to validate the semantics against the behaviour of a set of litmus tests (small assembler programs) run on hardware, and also to model check implementations of data structures from the literature against their abstract specifications.

1 Introduction

Modern processor architectures provide a challenge for developing efficient and correct software. Performance can be improved by parallelising computation to utilise multiple cores, but communication between threads is notoriously error-prone. Weak memory models go further and improve overall system efficiency through sophisticated techniques for batching reads and writes to the same variables and to and from the same processors. However, code that is run on such memory models is not guaranteed to take effect in the order specified in the program code, creating unexpected behaviours for those who are not forewarned [1]. For instance, the instructions $x := 1\ ;\ y := 1$ may, from the perspective of another process, taken effect in the order $y := 1\ ;\ x := 1$. Architectures typically provide *memory barrier/fence* instructions which can enforce ordering –

so that $x := 1$; **fence** ; $y := 1$ can not be reordered – but reduce performance improvements (and so should not be overused).

Previous work on formalising weak memory models has resulted in abstract formalisations which were developed incrementally through communication with processor vendors and rigorous testing on real machines [2–4]. A large collection of "litmus tests" have been developed [5, 6] which demonstrate the sometimes confusing behaviour of hardware. We utilise this existing work to provide a wide-spectrum programming language and semantics that runs on the same relaxed principles that apply to assembler instructions. When these principles are specialised to the assembler of ARM and POWER processors our semantics gives behaviour consistent with existing litmus tests. Our language and semantics, therefore, connect instruction reordering to higher-level notions of correctness. This enables verification of low-level code targeting specific processors against abstract specifications.

We begin in Sect. 2 with the basis of an operational semantics that allows reordering of instructions according to pair-wise relationships between instructions. In Sect. 3 we describe the semantics in more detail, focussing on its instantiation for the widely used ARM and POWER processors. In Sect. 4 we give a summary of the encoding of the semantics in Maude and its application to model-checking concurrent data structures. We discuss related work in Sect. 5 before concluding in Sect. 6.

2 Instruction reordering in weak memory models

2.1 Thread-local reorderings

It is typically assumed processes are executed in a fixed sequential order (as given by sequential composition – the "program order"). However program order may be inefficient, e.g., when retrieving the value of a variable from main memory after setting its value, as in $\mathtt{x} := \mathtt{1}$; $\mathtt{r} := \mathtt{x}$, and hence weak memory models sometimes allow execution out of program order to improve overall system efficiency. While many reorderings can seem surprising, there are basic principles at play which limit the number of possible permutations, the key being that the new ordering of instructions preserves the original sequential intention.

A classic example of weak memory models producing unexpected behaviour is the "store buffer" pattern below [5]. Assume that all variables are initially 0, and that thread-local variables (registers) are named r, r_1, r_2, etc., and that x and y are shared variables.

$$(x := 1\ ;\ r_1 := y) \parallel (y := 1\ ;\ r_2 := x) \tag{1}$$

It is possible to reach a final state in which $r_1 = r_2 = 0$ in several weak memory models: the two assignments in each process are independent (they reference different variables), and hence can be reordered. From a sequential semantics perspective, reordering the assignments in process 1, for example, preserves the final values for r_1 and x.

Assume that c and c' are programs represented as sequences of atomic actions $\alpha\,;\ \beta\,;\ \ldots$, as in a sequence of instructions in a processor or more abstractly a semantic trace. Program c may be reordered to c', written $c \rightsquigarrow c'$, if the following holds:

1. c' is a permutation of the actions of c, possibly with some modifications due to *forwarding* (see below).
2. c' preserves the sequential semantics of c. For example, in a weakest preconditions semantics [7], for all predicates P, $wp(c, P) \Rightarrow wp(c', P)$.
3. c' preserves *coherence-per-location* with respect to c (cf. `po-loc` in [3]). This means that the order of updates and accesses of each shared variable, considered individually, is maintained.

We formalise these constraints in the context of pair-wise reordering of instructions below. The key challenge for reasoning about programs executed on a weak memory model is that the behaviour of $c \parallel d$ is in general quite different to the behaviour of $c' \parallel d$, even if $c \rightsquigarrow c'$. We focus in this paper on the principles for ARM and POWER processors; for space reasons we do not address TSO [8], which has fewer relevant instruction types (e.g., only one type of fence) and stricter conditions on reordering.

2.2 Reordering and forwarding instructions

We write $\alpha \overset{\text{R}}{\Leftarrow} \beta$ if instruction β may be reordered before instruction α. It is relatively straightforward to define when two assignment instructions (encompassing stores, loads, and register operations at the assembler level) may be reordered. Below let x nfi f mean that x does not appear free in the expression f, and say expressions e and f are *load-distinct* if they do not reference any common shared variables.

$$x := e \overset{\text{R}}{\Leftarrow} y := f \quad \textit{if} \quad \begin{array}{l} 1)\ x, y \text{ are distinct};\ \ 2)\ x\ \mathsf{nfi}\ f;\ \ 3)\ y\ \mathsf{nfi}\ e; \text{ and} \\ 4)\ e, f \text{ are load-distinct}; \end{array} \tag{2}$$

Note that $\overset{\text{R}}{\Leftarrow}$ as defined above is symmetric, however when calculated after the effect of forwarding is applied (as described below) there are instructions that may be reordered in one direction but not the other. The relation is neither reflexive nor transitive. In TSO processors a load may be reordered before a store, but not vice versa [8], and hence the general condition for TSO is stronger and not reflexive.

Provisos 1), 2) and 3) ensure executing the two assignments in either order results in the same final values for x and y, and proviso 4) maintains order on accesses of the shared state. If two updates do not refer to any common variables they may be reordered. The provisos allow some reordering when they share common variables. Proviso 1) eliminates reorderings such as $(x := 1\,;\ x := 2) \rightsquigarrow (x := 2\,;\ x := 1)$ which would violate the sequential semantics (the final value of x). Proviso 2) eliminates reorderings such as $(x := 1\,;\ r := x) \rightsquigarrow (r := x\,;\ x := 1)$ which again would violate the sequential semantics (the final value of r). Proviso

3) eliminates reorderings such as $(r := y\ ;\ y := 1) \rightsquigarrow (y := 1\ ;\ r := y)$ which again would violate the sequential semantics (the final value of r). Proviso 4), requiring the update expressions to be load-distinct, preserves coherence-per-location, eliminating reorderings such as $(r_1 := x\ ;\ r_2 := x) \rightsquigarrow (r_2 := x\ ;\ r_1 := x)$, where r_2 may receive an earlier value of x than r_1 in an environment which modifies x.

The instructions used in the above examples, where each instruction references at most one global variable and uses simple integer values, correspond to the basic load and store instruction types of ARM and POWER processors. We may instantiate (2) to such instructions, giving reordering rules such as the following, which states that a store may be reordered before a load if they are to different locations ($r_1 := y \overset{\mathrm{R}}{\Leftarrow} x := r_2$). We use ARM syntax to emphasise the application to a real architecture.

$$\texttt{LDR r}_1\texttt{, y} \overset{\mathrm{R}}{\Leftarrow} \texttt{STR r}_2\texttt{, x} \tag{3}$$

In practice, proviso 2) may be circumvented by *forwarding*[1]. This refers to taking into account the effect of the earlier update on the expression of the latter. We write $\beta_{[\alpha]}$ to represent the effect of forwarding the (assignment) instruction α to the instruction β. For assignments we define

$$(y := f)_{[x := e]} \quad = \quad y := (f_{[x \setminus e]}) \quad \textit{if} \quad e \text{ does not refer to global variables} \tag{4}$$

where the term $f_{[x \setminus e]}$ stands for the syntactic replacement in expression f of references to x with e. The proviso of (4) prevents additional loads of globals being introduced by forwarding.

We specify the reordering and forwarding relationships with other instructions such as branches and fences in Sect. 3.3.

2.3 General operational rules for reordering

The key operational principle allowing reordering is given by the following transition rules for a program $(\alpha\ ;\ c)$, i.e., a program with initial instruction α.

$$(\alpha\ ;\ c) \xrightarrow{\alpha} c \quad (a) \qquad\qquad \frac{c \xrightarrow{\beta} c' \quad \alpha \overset{\mathrm{R}}{\Leftarrow} \beta_{[\alpha]}}{(\alpha\ ;\ c) \xrightarrow{\beta_{[\alpha]}} (\alpha\ ;\ c')} \quad (b) \tag{5}$$

Rule (5a) is the straightforward promotion of the first instruction into a step in a trace, similar to the basic prefixing rules of CCS [9] and CSP [10]. Rule (5b), however, states that, unique to weak memory models, an instruction of c, say β, can happen before α, provided that $\beta_{[\alpha]}$ can be reordered before α according to the rules of the architecture. Note that we forward the effect of α to β before deciding if the reordering is possible.

[1] We adopt the term "forwarding" from ARM and POWER [3]. The equivalent effect is referred to as *bypassing* on TSO [8].

Applying Rule (5b) then Rule (5a) gives the following reordered behaviour of two assignments.

$$(r := 1\ ;\ x := r\ ;\ \mathbf{nil}) \xrightarrow{x:=1} (r := 1\ ;\ \mathbf{nil}) \xrightarrow{r:=1} \mathbf{nil} \tag{6}$$

We use the command **nil** to denote termination. The first transition above is possible because we calculate the effect of $r := 1$ on the update of x before executing that update, i.e., $x := r_{[r:=1]} = x := 1$.

The definitions of instruction reordering, $\alpha \overset{\mathrm{R}}{\Leftarrow} \beta$, and instruction forwarding, $\beta_{[\alpha]}$ are architecture-specific, and are the only definitions required to specify an architecture's instruction ordering.[2] The instantiations for *sequentially consistent* processors (i.e., those which do not have a weak memory model) are trivial: $\alpha \overset{\mathrm{R}}{\not\Leftarrow} \beta$ for all α, β, and there is no forwarding. Since reordering is not possible Rule (5b) never applies and hence the standard prefixing semantics is maintained. TSO is relatively straightforward: loads may be reordered before stores (provided they reference different shared variables). In our framework there is no need to explicitly model local buffers, as the forwarding (bypassing) mechanism ensures that only the most recently stored value for a global x is used locally (or x's value is retrieved from the storage system). In this paper we focus on the more complex ARM and POWER memory models. These memory models are very similar, the notable difference being the inclusion of the *lightweight fence* instruction in POWER. Due to space limitations, we omit lightweight fences in this paper but see the appendix of [11] for a full definition.

2.4 Reasoning about reorderings

The operational rules allow a standard trace model of correctness to be adopted, that is, we say program c *refines to* program d, written $c \sqsubseteq d$, iff every trace of d is a trace of c. Let the program $\alpha \centerdot c$ have the standard semantics of prefixing, that is, the action α always occurs before any action in c (Rule (5a)). Then we can derive the following laws that show the interplay of reordering and true prefixing.

$$\alpha\ ;\ c \sqsubseteq \alpha \centerdot c \tag{7}$$

$$\alpha\ ;\ (\beta \centerdot c) \sqsubseteq \beta_{[\alpha]} \centerdot (\alpha\ ;\ c) \qquad \text{if } \alpha \overset{\mathrm{R}}{\Leftarrow} \beta_{[\alpha]} \tag{8}$$

Note that in Law (8) α may be further reordered with instructions in c. A typical interleaving law is the following.

$$(\alpha \centerdot c) \parallel d \sqsubseteq \alpha \centerdot (c \parallel d) \tag{9}$$

We may use these laws to show how the "surprise" behaviour of the store buffer pattern above arises.[3] In derivations such as the following, to save space, we abbreviate a thread $\alpha\ ;$ **nil** or $\alpha \centerdot$ **nil** to α, that is, we omit the trailing **nil**.

[2] Different architectures may have different storage subsystems, however, and these need to be separately defined (see Sect. 3.2).

[3] To focus on instruction reorderings we leave local variable declarations and process ids implicit, and assume a multi-copy atomic storage system (see Sect. 3.2).

$$
\begin{array}{ll}
 & (x := 1\ ;\ r_1 := y) \parallel (y := 1\ ;\ r_2 := x) \\
\sqsubseteq & \text{From Law (8) (twice), since } x := 1 \overset{\text{R}}{\Leftarrow} r_1 := y \text{ from (2).} \\
 & (r_1 := y \centerdot x := 1) \parallel (r_2 := x \centerdot y := 1) \\
\sqsubseteq & \text{Law (9) (four times) and commutativity of } \parallel. \\
 & r_1 := y \centerdot r_2 := x \centerdot x := 1 \centerdot y := 1
\end{array}
$$

If initially $x = y = 0$, a standard sequential semantics shows that $r_1 = r_2 = 0$ is a possible final state in this behaviour.

3 Semantics

3.1 Formal language

The elements of our wide-spectrum language are actions (instructions) α, commands (programs) c, processes (local state and a command) p, and the top level system s, encompassing a shared state and all processes. Below x is a variable (shared or local) and e an expression.

$$
\begin{array}{l}
\alpha ::= x := e \ \mid\ [e] \ \mid\ \textbf{fence} \ \mid\ \textbf{cfence} \ \mid\ \alpha^* \\
c ::= \textbf{nil} \ \mid\ \alpha\ ;\ c \ \mid\ c_1 \sqcap c_2 \ \mid\ \textbf{while}\ b\ \textbf{do}\ c \\
p ::= (\textbf{lcl}\ \sigma \bullet c) \ \mid\ (\textbf{tid}_{\text{N}}\ p) \ \mid\ p_1 \parallel p_2 \\
s ::= (\textbf{glb}\ \sigma \bullet p) \ \mid\ (\textbf{stg}\ W \bullet p)
\end{array}
\tag{10}
$$

An action may be an update $x := e$, a guard $[e]$, a (full) fence, a control fence (see Sect. 3.3), or a finite sequence of actions, α^*, executed atomically. Throughout the paper we denote an empty sequence by $\langle\rangle$, and construct a non-empty sequence as $\langle \alpha_1\ ,\ \alpha_2 \ldots \rangle$.

A command may be the empty command **nil**, which is already terminated, a command prefixed by some action α, a choice between two commands, or an iteration (for brevity we consider only one type of iteration, the while loop). Conditionals are modelled using guards and choice.

$$
\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mathrel{\widehat{=}} ([b]\ ;\ c_1) \sqcap ([\neg b]\ ;\ c_2) \tag{11}
$$

A well-formed process is structured as a process id $\text{N} \in \text{PID}$ encompassing a (possibly empty) local state σ and command c, i.e., a term $(\textbf{tid}_{\text{N}}\ \textbf{lcl}\ \sigma \bullet c)$. We assume that all local variables referenced in c are contained in the domain of σ.

A system is structured as the parallel composition of processes within the global storage system, which may be either a typical global state, σ, that maps all global variables to their values (modelling the storage systems of TSO, the most recent version of ARM, and abstract specifications), or a storage system, W, formed from a list of "writes" to the global variables (modelling the storage systems of older versions of ARM and POWER). The storage W injects more nondeterminism into the system than the typical global state approach. A top-level system is in one of the two following forms.

$$
\begin{array}{r}
(\textbf{glb}\ \sigma \bullet (\textbf{tid}_1\ \textbf{lcl}\ \sigma_1 \bullet c_1) \parallel (\textbf{tid}_2\ \textbf{lcl}\ \sigma_2 \bullet c_2) \parallel \ldots) \\
(\textbf{stg}\ W \bullet (\textbf{tid}_1\ \textbf{lcl}\ \sigma_1 \bullet c_1) \parallel (\textbf{tid}_2\ \textbf{lcl}\ \sigma_2 \bullet c_2) \parallel \ldots)
\end{array}
\tag{12}
$$

$$(\alpha\,;\ c) \xrightarrow{\alpha} c \quad (a) \qquad \frac{c \xrightarrow{\beta} c' \quad \alpha \overset{\mathrm{R}}{\Leftarrow} \beta_{[\alpha]}}{(\alpha\,;\ c) \xrightarrow{\beta_{[\alpha]}} (\alpha\,;\ c')} \quad (b) \quad (13) \qquad \begin{array}{l} c \sqcap d \xrightarrow{\tau} c \\ c \sqcap d \xrightarrow{\tau} d \end{array} \quad (14)$$

$$\textbf{while } b \textbf{ do } c \xrightarrow{\tau} ([b]\,;\ c\,;\ \textbf{while } b \textbf{ do } c) \sqcap ([\neg b]\,;\ \textbf{nil}) \qquad (15)$$

$$\frac{c \xrightarrow{r := v} c'}{(\textbf{lcl } \sigma \bullet c) \xrightarrow{\tau} (\textbf{lcl } \sigma_{[r := v]} \bullet c')} \quad (16) \qquad \frac{c \xrightarrow{x := r} c' \quad \sigma(r) = v}{(\textbf{lcl } \sigma \bullet c) \xrightarrow{x := v} (\textbf{lcl } \sigma \bullet c')} \quad (17)$$

$$\frac{c \xrightarrow{r := x} c'}{(\textbf{lcl } \sigma \bullet c) \xrightarrow{[x=v]} (\textbf{lcl } \sigma_{[r := v]} \bullet c')} \quad (18) \qquad \frac{c \xrightarrow{[e]} c'}{(\textbf{lcl } \sigma \bullet c) \xrightarrow{[e_\sigma]} (\textbf{lcl } \sigma \bullet c')} \quad (19)$$

$$\frac{p \xrightarrow{\alpha} p'}{(\textbf{tid}_{\mathrm{N}}\ p) \xrightarrow{\mathrm{N}:\alpha} (\textbf{tid}_{\mathrm{N}}\ p')} \quad (20) \qquad \frac{p_1 \xrightarrow{\alpha} p_1'}{p_1 \parallel p_2 \xrightarrow{\alpha} p_1' \parallel p_2} \quad \frac{p_2 \xrightarrow{\alpha} p_2'}{p_1 \parallel p_2 \xrightarrow{\alpha} p_1 \parallel p_2'} \quad (21)$$

$$\frac{p \xrightarrow{\mathrm{N}:x := e} p'}{(\textbf{glb } \sigma \bullet p) \xrightarrow{\tau} (\textbf{glb } \sigma_{[x := e_\sigma]} \bullet p')} \quad (22) \qquad \frac{p \xrightarrow{\mathrm{N}:[e]} p' \quad e_\sigma \equiv \mathit{true}}{(\textbf{glb } \sigma \bullet p) \xrightarrow{\tau} (\textbf{glb } \sigma \bullet p')} \quad (23)$$

Fig. 1. Semantics of the language

3.2 Operational semantics

The meaning of our language is formalised using an operational semantics, summarised in Fig. 1. Given a program c the operational semantics generates a *trace*, i.e., a possibly infinite sequence of steps $c_0 \xrightarrow{\alpha_1} c_1 \xrightarrow{\alpha_2} \ldots$ where the labels in the trace are actions, or a special label τ representing a silent or internal step that has no observable effect.

The terminated command **nil** has no behaviour; a trace that ends with this command is assumed to have completed. The effect of instruction prefixing in Rule (13) is discussed in Sect. 2.3. Note that actions become part of the trace. We describe an instantiation for reordering and forwarding corresponding to the semantics of ARM and POWER in Sect. 3.3.

A nondeterministic choice (the *internal choice* of CSP [10]) can choose either branch, as given by Rule (14). The semantics of loops is given by unfolding, e.g., Rule (15) for a 'while' loop. Note that *speculative execution*, i.e., early execution of instructions which occur after a branch point [12], is theoretically unbounded, and loads from inside later iterations of the loop could occur in earlier iterations.

For ease of presentation in defining the semantics for local states, we give rules for specific forms of actions, i.e., assuming that r is a local variable in the domain of σ, and that x is a global (not in the domain of σ). The more general version can be straightforwardly constructed from the principles below.

Rule (16) states that an action updating variable r to value v results in a change to the local state (denoted $\sigma_{[r:=v]}$). Since this is a purely local operation there is no interaction with the storage subsystem and hence the transition is promoted as a silent step τ. Rule (17) states that a *store* of the value in variable r to global x is promoted as an instruction $x := v$ where v is the local value for r. Rule (18) covers the case of a *load* of x into r. The value of x is not known locally. The promoted label is a guard requiring that the value read for x is v. This transition is possible for any value of v, but the correct value will be resolved when the label is promoted to the storage level. Rule (19) states that a guard is partially evaluated with respect to the local state before it is promoted to the global level. The notation e_σ replaces x with v in e for all $(x \mapsto v) \in \sigma$.

Rule (20) simply tags the process id to an instruction, to assist in the interaction with the storage system, and otherwise has no effect. Instructions of concurrent processes are interleaved in the usual way as described by Rule (21).

Other straightforward rules which we have omitted above include the promotion of fences through a local state, and that atomic sequences of actions are handled inductively by the above rules.

Multi-copy atomic storage subsystem. Traditionally, changes to shared variables occur on a shared global state, and when written to the global state are seen instantaneously by all processes in the system. This is referred to as *multi-copy atomicity* and is a feature of TSO and the most recent version of ARM [13]. Older versions of ARM and POWER, however, lack such multi-copy atomicity and require a more complex semantics. We give the simpler case (covered in Fig. 1) first.[4]

Recall that at the global level the process id N has been tagged to the actions by Rule (20). Rule (22) covers a store of some expression e to x. Since all local variable references have been replaced by their values at the process level due to Rules (16)-(19), expression e must refer only to shared variables in σ. The value of x is updated to the fully evaluated value, e_σ.

Rule (23) states that a guard transition $[e]$ is possible exactly when e evaluates to true in the global state. If it does not, no transition is possible; this is how incorrect branches are eliminated from the traces. If a guard does not evaluate to *true*, execution stops in the sense that no transition is possible. This corresponds to a false guard, i.e., **magic** [14, 15], and such behaviours do not terminate and are ignored for the purposes of determining behaviour of a real system. Interestingly, this straightforward concept from standard refinement theory allows us to handle speculative execution straightforwardly. In existing approaches, the semantics is complicated by needing to restart reads if speculation proceeds down the wrong path. Treating branch points as guards works because speculation should have no effect if the wrong branch was chosen.

To understand how this approach to speculative execution works, consider the following derivation. Assume that (a) loads may be reordered before guards

[4] In this straightforward model of shared state there is no global effect of fences, and we omit the straightforward promotion rule.

if they reference independent variables, and (b) loads may be reordered if they reference different variables. Recall that we omit trailing **nil** commands to save space.

$$
\begin{array}{ll}
 & r_1 := x \,;\ (\textbf{if } r_1 = 0 \textbf{ then } \underline{r_2 := y}) \\
= & \text{Definition of } \textbf{if} \ (11) \\
 & r_1 := x \,;\ (([r_1 = 0] \,;\ \underline{r_2 := y}) \sqcap [r_1 \neq 0]) \\
\sqsubseteq & \text{Resolve to the first branch, since } (c \sqcap d) \sqsubseteq c \\
 & r_1 := x \,;\ [r_1 = 0] \,;\ \underline{r_2 := y} \\
\sqsubseteq & \text{From Law (8) and assumption (a)} \\
 & r_1 := x \,;\ \underline{r_2 := y} \cdot [r_1 = 0] \\
\sqsubseteq & \text{From Law (8) and assumption (b)} \\
 & \underline{r_2 := y} \cdot r_1 := x \,;\ [r_1 = 0]
\end{array}
$$

This shows that the inner load (underlined) may be reordered before the branch point, and subsequently before an earlier load. Note that this behaviour results in a terminating trace only if $r_1 = 0$ holds when the guard is evaluated, and otherwise becomes **magic** (speculation down an incorrect path). On ARM processors, placing a control fence (**cfence**) instruction inside the branch, before the inner load, prevents this reordering (see Sect. 3.3).

Non-multi-copy atomic storage subsystem. Some versions of ARM and POWER allow processes to communicate values to each other without accessing the heap. That is, if process p_1 is storing v to x, and process p_2 wants to load x into r, p_2 may preemptively load the value v into r, before p_1's store hits the global shared storage. Therefore different processes may have different views of the values of global variables; see litmus tests such as the `WRC` family [3].

Our approach to modelling this is based on that of the operational model of [2]. However, that model maintains several partial orders on operations reflecting the nondeterminism in the system, whereas we let the nondeterminism be represented by choices in the operational rules. This means we maintain a simpler data structure, a single global list of writes. The shared state from the perspective of a given process is a particular view of this list. There is no single definitive shared state. In addition, viewing a value in the list causes the list to be updated and this affects later views. To obtain the value of a variable this list is searched starting with the most recent write first. A process p_1 that has already seen the latter of two updates to a variable x may not subsequently then see the earlier update. Hence the list keeps track of which processes have seen which stores. Accesses of the storage subsystem are also influenced by fences.

A write w has the syntactic form $(x \mapsto v)^{\mathrm{N}}_{\mathcal{S}}$, where x is a global variable being updated to value v, N is the process id of the process from which the store originated, and $\mathcal{S}$ is the set of process ids that have "seen" the write. For such a w, we let $w.var = x$, $w.thread = \mathrm{N}$ and $w.seen = \mathcal{S}$. For a write $(x \mapsto v)^{\mathrm{N}}_{\mathcal{S}}$ it is always the case that $\mathrm{N} \in \mathcal{S}$. The storage W is a list of writes, initially populated with writes for the initial values of global variables, which all processes have "seen".

$$\frac{p \xrightarrow{\text{N}:[x=v]} p' \qquad \forall\, w \in \mathrm{ran}(W_1) \bullet x = w.var \Rightarrow \text{N} \notin w.seen}{(\mathbf{stg}\ W_1 \frown (x \mapsto v)^{\text{M}}_{\mathcal{S}} \frown W_2 \bullet p) \xrightarrow{\text{N}:[x=v]} (\mathbf{stg}\ W_1 \frown (x \mapsto v)^{\text{M}}_{\mathcal{S}\cup\{\text{N}\}} \frown W_2 \bullet p')} \tag{24}$$

$$\frac{p \xrightarrow{\text{N}:x:=v} p' \qquad \forall\, w \in \mathrm{ran}(W_1) \bullet \text{N} \neq w.thread \wedge (x = w.var \Rightarrow \text{N} \notin w.seen)}{(\mathbf{stg}\ W_1 \frown W_2 \bullet p) \xrightarrow{\text{N}:x:=v} (\mathbf{stg}\ W_1 \frown (x \mapsto v)^{\text{N}}_{\{\text{N}\}} \frown W_2 \bullet p')} \tag{25}$$

$$\frac{p \xrightarrow{\text{N}:\mathbf{fence}} p'}{(\mathbf{stg}\ W \bullet p) \xrightarrow{\text{N}:\mathbf{fence}} (\mathbf{stg}\ \mathit{flush}_{\text{N}}(W) \bullet p')} \tag{26}$$

where

$$\mathit{flush}_{\text{N}}(\langle\rangle) = \langle\rangle \qquad \mathit{flush}_{\text{N}}(w \frown W) = \begin{cases} w_{[seen\,:=\,\text{PID}]} \frown \mathit{flush}_{\text{N}}(W) & \mathit{if}\ \text{N} \in w.seen \\ w \frown \mathit{flush}_{\text{N}}(W) & \mathit{otherwise} \end{cases}$$

Fig. 2. Rules for the non-multi-copy atomic subsystem of ARM and POWER

We give two specialised rules (for a load and store) in Fig. 2.[5] Rule (24) states that a previous write to x may be seen by process N if there are no more recent writes to x that it has already seen. Its id is added to the set of processes that have seen that write. Rule (25) states that a write to x may be added to the system by process N, *appearing earlier than existing writes in the system*, if the following two conditions hold for each of those existing writes w: they are not by N ($\text{N} \neq w.thread$, local coherence), and $x = w.var \Rightarrow \text{N} \notin w.seen$, i.e., writes to the same variable are seen in a consistent order (although not all writes need be seen). A **fence** action by process N 'flushes' all previous writes by and seen by N. The *flush* function modifies W so that all processes can see all writes by N, effectively overwriting earlier writes. This is achieved by updating the write so that all processes have seen it, written as $w_{[seen\,:=\,\text{PID}]}$.

3.3 Reordering and forwarding for ARM and POWER

Our general semantics is instantiated for ARM and POWER processors in Fig. 3 which provides particular definitions for the reordering relation and forwarding that are generalised from the orderings on stores and loads in these processors.[6]

[5] To handle the general case of an assignment $x := e$, where e may contain more than one shared variable, the antecedents of the rules are combined, retrieving the value of each variable referenced in e individually and accumulating the changes to W.

[6] We have excluded address shifting, which creates *address dependencies* [3], as this does not affect the majority of high-level algorithms in which we are interested. However, address dependencies are accounted for in our tool as discussed in [11].

$$\alpha \overset{R}{\not\Leftarrow} \textbf{fence} \quad (27)$$

$$\textbf{fence} \overset{R}{\not\Leftarrow} \alpha \quad (28)$$

$$[b] \overset{R}{\not\Leftarrow} \textbf{cfence} \quad (29)$$

$$\textbf{cfence} \overset{R}{\not\Leftarrow} r := e \quad (30)$$

$$[b_1] \overset{R}{\Leftarrow} [b_2] \quad (31)$$

$$[b] \overset{R}{\not\Leftarrow} \varphi := e \quad (32)$$

$$[b] \overset{R}{\Leftarrow} r := e \quad \text{iff } r \text{ nfi } b \quad (33)$$

$$x := e \overset{R}{\Leftarrow} [b] \quad \text{iff } x \text{ nfi } b \quad (34)$$

$$\alpha \overset{R}{\Leftarrow} \beta \quad \text{in all other cases}$$

$$x := e \overset{R}{\Leftarrow} y := f \text{ iff} \quad (35)$$
$x \neq y$, x nfi f, y nfi e, and
e, f are load-distinct

$$x := e_{[y:=f]} = x := e_{[y \setminus f]} \text{ if} \quad (36)$$
e has no shared variables

$$[e]_{[y:=f]} = [e_{[y \setminus f]}] \text{ if} \quad (37)$$
e has no shared variables

$$\beta_{[\alpha]} = \beta \quad \text{otherwise}$$

Fig. 3. Reordering and forwarding following ARM assembler semantics. Let x, y denote any variable, r a local variable, and φ a global variable.

Fences prevent all reorderings (27, 28). Control fences prevent speculative loads when placed between a guard and a load (29, 30). Guards may be reordered with other guards (31), but stores to shared variables may not come before a guard evaluation (32). This prevents speculative execution from modifying the global state, in the event that the speculation was down the wrong branch. An update of a local variable may be reordered before a guard provided it does not affect the guard expression (33). Guards may be reordered before updates if those updates do not affect the guard expression (34).

Assignments may be reordered as shown in (35) and discussed in Sect. 2.2. Forwarding is defined straightforwardly so that an earlier update modifies the expression of a later update or guard (36, 37), provided it references no shared variables.

4 Model checking concurrent data structures

Our semantics has been encoded in the Maude rewriting system [16]. We have used the resulting prototype tool to validate the semantics against litmus tests which have been used in other work on ARM (348 tests) [4] and POWER (758 tests) [2]. As that research was developed through testing on hardware and in consultation with the processor vendors themselves we consider compliance with those litmus tests to be sufficient validation. With two exceptions, as discussed in Sect. 5, our semantics agrees with those results.

We have employed Maude as a model checker to verify that a (test-and-set) lock provides mutual exclusion on ARM and POWER, and that a lock-free stack algorithm, and a deque (double-ended queue) algorithm, satisfy their abstract specifications on ARM and POWER. We describe the verification of the deque below, in which we found a bug in the published algorithm.

4.1 Chase-Lev deque

Lê et. al [17] present a version of the Chase-Lev deque [18] adapted for ARM and POWER. The deque is implemented as an array, where elements may be *put* on or *taken* from the tail, and additionally, processes may *steal* an element from the head of the deque. The *put* and *take* operations may be executed by a single process only, hence there is no interference between these two operations (although instruction reordering could cause consecutive invocations to overlap). The *steal* operation can be executed by multiple processes concurrently.

The code we tested is given in Fig. 4 where L is the maximum size of the deque which is implemented as a cyclic array, with all elements initialised to some irrelevant value. The original code includes handling array resizing, but here we focus on the insert/delete logic. For brevity we omit trailing **nil**s. We have used a local variable *return* to model the return value, and correspondingly have refactored the algorithm to eliminate returns from within a branch. A $CAS(x, r, e)$ (compare-and-swap) instruction atomically compares the value of global x with the value r and if the same updates x to e. We model a conditional statement with a CAS as follows.

$$\textbf{if } CAS(x, r, e) \textbf{ then } c_1 \textbf{ else } c_2 \mathrel{\widehat{=}} (\langle [x = r]\ ,\ x := e\rangle\ ;\ c_1) \sqcap ([x \neq r]\ ;\ c_2) \quad (38)$$

The *put* operation straightforwardly adds an element to the end of the deque, incrementing the *tail* index. It includes a full fence so that the tail pointer is not incremented before the element is placed in the array. The *take* operation uses a CAS operation to atomically increment the head index. Interference can occur if there is a concurrent *steal* operation in progress, which also uses CAS to increment *head* to remove an element from the head of the deque. The *take* and *steal* operation return empty if they observe an empty deque. In addition the *steal* operation may return the special value *fail* if interference on *head* occurs. Complexity arises if the deque has one element and there are concurrent processes trying to both *take* and *steal* that element at the same time.

Operations *take* and *steal* use a **fence** operation to ensure they have consistent readings for the head and tail indexes, and later use CAS to atomically update the head pointer (only if necessary, in the case of *take*). Additionally, the *steal* operation contains two **cfence** barriers (`ctrl_isync` in ARM).

Verification. We use an abstract model of the deque and its operations to specify the allowed final values of the deque and return values. The function $last(q)$ returns the last element in q and $front(q)$ returns q excluding its last element.

$$\begin{aligned} put(v) &\mathrel{\widehat{=}} q := q \frown \langle v\rangle \\ take &\mathrel{\widehat{=}} \textbf{lcl } return := none \bullet \\ &\qquad \langle [q = \langle\rangle]\ ,\ return := empty\rangle \sqcap \\ &\qquad \langle [q \neq \langle\rangle]\ ,\ return := last(q)\ ,\ q := front(q)\rangle \end{aligned}$$

Initial state: $\{head \mapsto 0, tail \mapsto 0, tasks \mapsto \langle \_, \_, \ldots \rangle\}$

```
put(v) ≙
  lcl t ↦ _ •
  t := tail;
  tasks[t mod L] := v;
  fence;
  tail := t + 1
```

```
take ≙
  lcl h ↦ _, t ↦ _, return ↦ _ •
  t := tail − 1 ;
  tail := t ;
  fence ;
  h := head ;
  if h ≤ t then
    return := tasks[t mod L] ;
    if h = t then
      if ¬CAS(head, h, h + 1) then
        return := empty
      tail := t + 1
  else
    return := empty ;
    tail := t + 1
```

```
steal ≙
  lcl h ↦ _, t ↦ _, return ↦ _ •
  h := head ;
  fence ;
  t := tail ;
  cfence ; // unnecessary
  if h < t then
    return := tasks[h mod L] ;
    cfence ; // incorrectly placed
    if ¬CAS(head, h, h + 1) then
      return := fail
  else
    return := empty
```

Fig. 4. A version of Lê et. al's work-stealing deque algorithm for ARM [17]

$$\begin{aligned} steal \mathrel{\widehat{=}} \textbf{lcl}\ return := none \\ \langle [q = \langle\rangle]\ ,\ return := empty \rangle \sqcap \\ \langle [q \neq \langle\rangle]\ ,\ return := head(q)\ ,\ q := tail(q) \rangle \end{aligned}$$

For simplicity the abstract specification for *steal* does not attempt to detect interference and return *fail*, and as such we exclude corresponding behaviours of the concrete code from the analysis. We could encode this special failure case for *steal*, requiring additional data to track which processes are active.

We ran several contextual programs calling the abstract model alongside the same programs calling the concrete model, comparing final states after applying a straightforward simulation relation between abstract and concrete states. The contextual programs were combinations of concurrent processes – 1, 2 or 3 – each sequentially making one or two calls to the three operations. This exposed a bug in the code which may occur when a *put* and *steal* operation execute in parallel on an empty deque. The load $return := tasks[h \bmod L]$ can be speculatively executed before the guard $h < t$ is evaluated, and hence also before the load of *tail*. Thus the steal process may load *head*, load an irrelevant *return* value, at

which point a *put* operation may complete, storing a value and incrementing *tail*. The *steal* operation resumes, loading the new value for *tail* and observing a non-empty deque, succeeding with its *CAS* and returning the irrelevant value, which was loaded before the *put* operation had begun.

Swapping the order of the second **cfence** with the load of $tasks[h \bmod L]$ eliminates this bug, and our analysis did not reveal any other problems. In addition, eliminating the first **cfence** does not change the possible outcomes.

5 Related work

This work makes use of an extensive suite of tests elucidating the behaviour of weak memory models in ARM and POWER via both operational and axiomatic semantics [3, 2, 19, 4]. Those semantics were developed and validated through testing on real hardware and in consultation with processor vendors themselves. Our model is validated against their results, in the form of the results of litmus tests. The hardware vendor does not provide a formal specification of the assembler language, and hence the results of the litmus tests and their abstraction to axiomatic relations in the above work is the most reliable validation benchmark. However, as identified by Alglave et al. [3], some chips have different behaviour to others, contain bugs, and do not implement certain features; in addition given that instructions sets and definitions may change over time it is difficult to achieve a single canonical specification.

Excluding two tests involving "shadow registers", which appear to be processor-specific facilities which are not intended to conform to sequential semantics (they do not correspond to higher-level code), all of the 348 ARM litmus tests run on our model agreed with the results in [4], and all of the 758 POWER litmus tests run on our model agreed with the results in [2], which the exception of litmus test `PPO015`, which we give below, translated into our formal language.[7]

$$\begin{array}{l} x := 1 \,;\ \mathbf{fence} \,;\ y := 1 \quad \| \\ r_0 := y \,;\ z := (r_0 \ \texttt{xor}\ r_0) + 1 \,;\ z := 2 \,;\ r_3 := z; \\ \qquad\qquad (\mathbf{if}\ r_3 = r_3\ \mathbf{then\ nil\ else\ nil}) \,;\ \mathbf{cfence} \,;\ r_4 := x \end{array} \tag{39}$$

The tested condition is $z = 2 \wedge r_0 = 1 \wedge r_4 = 0$, which asks whether it is possible to load x (the last statement of process 2) before loading y (the first statement of process 2). At a first glance the control fence prevents the load of x happening before the branch. However, as indicated by litmus tests such as `MP+dmb.sy+fri-rfi-ctrisb`, [4, Sect 3,*Out of order execution*], under some circumstances the branch condition can be evaluated early, as discussed in the

[7] We simplified some of the syntax for clarity, in particular introducing a higher-level **if** statement to model a jump command and implicit register (referenced by the compare (`CMP`) and branch-not-equal (`BNE`) instructions). We have also combined some commands, retaining dependencies, in a way that is not possible in the assembler language. The `xor` operator is exclusive-or; its use here artificially creates a *data dependency* [3] between the updates to r_0 and z.

speculative execution example. We expand on this below by manipulating the second process, taking the case where the success branch of the **if** statement is chosen. To aid clarity we underline the instruction that is the target of the (next) refinement step.

$$
\begin{array}{ll}
 & r_0 := y \,;\; z := (r_0 \;\texttt{xor}\; r_0) + 1 \,;\; z := 2 \,;\; \underline{r_3 := z} \,;\; [r_3 = r_3] \,;\; \textbf{cfence} \,;\; r_4 := x \\
\sqsubseteq & \text{Promote load with forwarding (from } z := 2\text{), from Laws (7) and (8)} \\
 & r_3 := 2 \;\centerdot\; r_0 := y \,;\; z := (r_0 \;\texttt{xor}\; r_0) + 1 \,;\; z := 2 \,;\; \underline{[r_3 = r_3]} \,;\; \textbf{cfence} \,;\; r_4 := x \\
\sqsubseteq & \text{Promote guard by Laws (7) and (8) (from (34))} \\
 & r_3 := 2 \;\centerdot\; [r_3 = r_3] \;\centerdot\; r_0 := y \,;\; z := (r_0 \;\texttt{xor}\; r_0) + 1 \,;\; z := 2 \,;\; \underline{\textbf{cfence}} \,;\; r_4 := x \\
\sqsubseteq & \text{Promote control fence by Laws (7) and (8) ((29) does not now apply)} \\
 & r_3 := 2 \;\centerdot\; [r_3 = r_3] \;\centerdot\; \textbf{cfence} \;\centerdot\; r_0 := y \,;\; z := (r_0 \;\texttt{xor}\; r_0) + 1 \,;\; z := 2 \,;\; \underline{r_4 := x} \\
\sqsubseteq & \text{Promote load by Laws (7) and (8)} \\
 & r_3 := 2 \;\centerdot\; [r_3 = r_3] \;\centerdot\; \textbf{cfence} \;\centerdot\; r_4 := x \;\centerdot\; r_0 := y \,;\; z := (r_0 \;\texttt{xor}\; r_0) + 1 \,;\; z := 2
\end{array}
$$

The load $r_4 := x$ has been reordered before the load $r_0 := y$, and hence when interleaved with the first process from (39) it is straightforward that the condition may be satisfied.

In the Flowing/POP model of [4], this behaviour is forbidden because there is a data dependency from the load of y into r_0 to r_3, via z. This appears to be because of the consecutive stores to z, one of which depends on r_0. In the testing of real processors reported in [4], the behaviour that we allow was never observed, but is allowed by the model in [3]. As such we deem this discrepancy to be a minor issue in Flowing/POP (preservation of transitive dependencies) rather than a fault in our model.

Our model of the storage subsystem is similar to that of the operational models of [2, 4]. However our thread model is quite different, being defined in terms of relationships between actions. The key difference is how we handle branching and the effects of speculative execution. The earlier models are complicated in the sense that they are closer to the real execution of instructions, involving restarting reads if an earlier read invalidates the choice taken at a branch point.

The axiomatic models, as exemplified by Alglave et al. [3], define relationships between instructions in a whole-system way, including relationships between instructions in concurrent processes. This gives a global view of how an architecture's reordering rules (and storage system) interact to reorder instructions in a system. Such global orderings are not immediately obvious from our pair-wise orderings on instructions. On the other hand, those globals orderings become quite complex and obscure some details, and it is unclear how to extract some of the generic principles such as (2).

6 Conclusion

We have utilised earlier work to devise a wide-spectrum language and semantics for weak memory models which is relatively straightforward to define and extend, and which lends itself to verifying low-level code against abstract specifications. While abstracting away from the details of the architecture, we believe it provides

a complementary insight into why some reorderings are allowed, requiring a pairwise relationship between instructions rather than one that is system-wide.

A model-checking approach based on our semantics exposed a bug in an algorithm in [17] in relation to the placement of a control fence. The original paper includes a hand-written proof of the correctness of the algorithm based on the axiomatic model of [19]. The possible traces of the code were enumerated and validated against a set of conditions on adding and removing elements from the deque (rather than with respect to an abstract specification of the deque). The conditions being checked are non-trivial to express using final state analysis only. An advantage of having a semantics that can apply straightforwardly to abstract specifications, rather than a proof technique that analyses behaviours of the concrete code only, is that we may reason at a more abstract level.

We have described the ordering condition as syntactic constraints on atomic actions, which fits with the low level decisions of hardware processors. However our main reordering principle (2) is based on semantic concerns, and as such may be applicable as a basis for understanding the interplay of software memory models, compiler optimisations and hardware memory models [20].

Future work. A feature of our framework is that we can potentially reason about reordering of abstract instructions (i.e., those working with abstract data types), and not only low-level assembler instructions. This allows the potential for stepwise verification techniques to be applied, in particular potentially capturing the complex interaction of the environment using rely-guarantee reasoning [21–24]. In this paper we consider assignments as the fundamental command, which is sufficient for specifying many concurrent programs. However we hope to extend the language to encompass more general constructs such as the specification command [25], which may modify and access multiple global variables. Refinement laws for decomposing a (non-atomic) rely-guarantee specification into a sequence of atomic steps will have proof obligations referencing the reordering relation to ensure that any reordering of the actions does not affect the guarantee; alternatively, where reordering would affect the guarantee, the law could specify one or more fences in the implementation sequence.

Acknowledgements We thank Kirsten Winter, Ian Hayes, and the anonymous reviewers for feedback on this work. It was supported by Australian Research Council Discovery Grant DP160102457.

References

1. Adve, S.V., Boehm, H.J.: Memory models: A case for rethinking parallel languages and hardware. Commun. ACM **53**(8) (August 2010) 90–101
2. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. SIGPLAN Not. **46**(6) (June 2011) 175–186
3. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2) (July 2014) 7:1–7:74

4. Flur, S., Gray, K.E., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., Sewell, P.: Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '16, New York, NY, USA, ACM (2016) 608–621
5. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Litmus: Running tests against hardware. In Abdulla, P.A., Leino, K.R.M., eds.: Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2011) 41–44
6. Mador-Haim, S., Alur, R., Martin, M.M.K.: Generating litmus tests for contrasting memory consistency models. In Touili, T., Cook, B., Jackson, P., eds.: Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2010) 273–287
7. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8) (August 1975) 453–457
8. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7) (July 2010) 89–97
9. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag New York, Inc. (1982)
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
11. Colvin, R.J., Smith, G.: A wide-spectrum language for verification of programs on weak memory models. CoRR **abs/1802.04406** (2018)
12. Sorin, D.J., Hill, M.D., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence. 1st edn. Morgan & Claypool Publishers (2011)
13. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM Press (2018) To appear.
14. Morgan, C.: Programming from Specifications. Second edn. Prentice Hall (1994)
15. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer-Verlag (1998)
16. Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science **285**(2) (2002) 187 – 243
17. Lê, N.M., Pop, A., Cohen, A., Zappa Nardelli, F.: Correct and efficient work-stealing for weak memory models. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '13, New York, NY, USA, ACM (2013) 69–80
18. Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: SPAA'05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures, New York, NY, USA, ACM Press (2005) 21–28
19. Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M.M.K., Sewell, P., Williams, D.: An axiomatic memory model for POWER multiprocessors. In: Proceedings of the 24th International Conference on Computer Aided Verification. CAV'12, Berlin, Heidelberg, Springer-Verlag (2012) 495–512

20. Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017, New York, NY, USA, ACM (2017) 175–189
21. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress. (1983) 321–332
22. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5** (October 1983) 596–619
23. Hayes, I.J., Colvin, R.J., Meinicke, L.A., Winter, K., Velykis, A.: An algebra of synchronous atomic steps. In Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A., eds.: FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings, Cham, Springer International Publishing (2016) 352–369
24. Colvin, R.J., Hayes, I.J., Meinicke, L.A.: Designing a semantic model for a wide-spectrum language with concurrency. Formal Aspects of Computing **29**(5) (Sep 2017) 853–875
25. Morgan, C.: The specification statement. ACM Trans. Program. Lang. Syst. **10** (July 1988) 403–419