Relaxed Memory Models: an Operational Approach *

Gérard Boudol Gustavo Petri
INRIA Sophia Antipolis
{Gerard.Boudol,Gustavo.Petri}@inria.fr

Abstract

Memory models define an interface between programs written in some language and their implementation, determining which behaviour the memory (and thus a program) is allowed to have in a given model. A minimal guarantee memory models should provide to the programmer is that well-synchronized, that is, data-race free code has a standard semantics. Traditionally, memory models are defined axiomatically, setting constraints on the order in which memory operations are allowed to occur, and the programming language semantics is implicit as determining some of these constraints. In this work we propose a new approach to formalizing a memory model in which the model itself is part of a weak operational semantics for a (possibly concurrent) programming language. We formalize in this way a model that allows write operations to the store to be buffered. This enables us to derive the ordering constraints from the weak semantics of programs, and to prove, at the programming language level, that the weak semantics implements the usual interleaving semantics for data-race free programs, hence in particular that it implements the usual semantics for sequential code.

1. Introduction

Optimizing the performance of computing systems has always been a concern for hardware and software technology, and therefore, given that the latency of memory operations is one of the key performance factors, it is not surprising that techniques have been developed very early to minimize the cost of these operations in program execution. As a typical optimization technique, shared memory multiprocessors use write buffers and caches [15], thus benefiting from a parallel architecture where memory operations run concurrently with other instructions. These techniques, as well as various optimizations performed by compilers, are naturally intended to preserve the semantics of sequential code.

However, problems arise when executing concurrent code sharing the memory. In particular, many algorithms designed for synchronization purposes (mutual exclusion, producer-consumer) fail to achieve their goal in an optimized implementation setting. A classical example (see for instance [3]) is with Dekker's mutual exclusion algorithm:

$$\begin{array}{lcl} \mathsf{flag}_0 := \mathsf{false} & \| & \mathsf{flag}_1 := \mathsf{false} \\ \mathsf{if}\ \mathsf{flag}_1\ \mathsf{then} & & \mathsf{if}\ \mathsf{flag}_0\ \mathsf{then} \\ \quad \mathit{critical\ section}_0 & & \quad \mathit{critical\ section}_1 \end{array}$$

that may fail – that is, there exists an execution where both threads can enter their critical section –, for various reasons. For instance, the compiler might have decided to reorder independent statements, making the assignments $\mathsf{flag}_0 := \mathsf{false}$ and $\mathsf{flag}_1 := \mathsf{false}$ to happen after, or in parallel with the subsequent conditional branchings. These assignments could also be delayed in some communication structure (write buffer or cache) before they actually affect the memory, there again opening the possibility for the conditional branchings to read a wrong value from the memory. Another standard example (again given in [3]) is the one of a producer-consumer algorithm. Namely, with the program

$$\begin{array}{lcl} \mathsf{data} := 1 & \| & \mathsf{while\ not\ flag\ do\ skip} \\ \mathsf{flag} := \mathsf{true} & & r := \mathsf{data} \end{array}$$

one could get a wrong value for r if the assignment $\mathsf{data} := 1$ is delayed with respect to $\mathsf{flag} := \mathsf{true}$, or if $r := \mathsf{data}$ is speculatively done before $\mathsf{while\ not\ flag\ do\ skip}$. To compensate for these failures, the hardware usually offers various synchronization features: memory barriers (fence), read-modify-write atomic instructions (test-and-set, compare-and-swap).

The optimizations implemented in commercial hardware or software work fine for sequential programs, but the failure of the standard interleaving semantics prompted researchers to understand which semantics for shared variable concurrency is actually supported by these optimized hardware architectures and compilers. Then the notion of a *weak*, or *relaxed memory model* was introduced [15], to serve as an abstract interface [4] (or a contract, as we will see) between the programmer and the implementation. (In the following we shall use "weak" and "relaxed" as synonymous.) A memory model is intended to specify which values a read, in the execution of a program, can return, and therefore it defines what the possible outcomes of a concurrent system are. There are usually more (possibly unwanted) outcomes for such a system than with the standard interleaving semantics – which, as a memory model, is known as Strong Consistency (SC) [27] –, and memory models can be compared from this point of view. Numerous relaxed (w.r.t. SC) memory models have been introduced: Weak Ordering (WO) [4, 15], Processor Consistency (PC) [21], Release Consistency (RCsc and RCpc) [19], Location Consistency (LC) [17], and many others (see [2] and [34] for a survey). These models are formulated, quite abstractly, as constraints on the ordering of memory operations.

The problem of understanding the semantics supported by shared memory architectures recently regained a lot of interest, prompted by the fact that some programming languages, e.g. JAVA, offer a multithreading facility at the application programming level. Defining an abstract memory model for a high-level programming language is not an easy task, and, despite considerable efforts have been made to propose a revision of the original JAVA Memory Model (JMM, see [29]), this is still a controversial matter [6, 14]. However, a consensus has emerged as regards the rôle of memory models: a memory model is now generally conceived as a *contract* between the (application) programmer and the implementor [4]. The latter should use it as delineating the allowed optimizations, while the former is ensured that, provided he/she writes "properly synchronized" programs, he/she will get a semantics which is equivalent to the usual interleaving semantics. Well-synchronized

* Work partially supported by the ANR-SETI-06-010 grant.

programs[1] are also *data-races free* (DRF), and therefore the programmer's side of the contract memory models should provide is also known as the "DRF guarantee." This DRF guarantee is a good property to have, because the interleaving semantics provides a reasonable basis for the formal understanding, analysis and verification of concurrent programs (and this is also a portability guarantee).

Memory models are usually defined by means of various partial orders relating actions, or more precisely events in an execution. This is the way the JMM [29] is (re)defined for instance. A proof of the DRF guarantee for this model was given in [29], and later formalized using proof assistants (Isabelle [5] and COQ [25]), which revealed some subtle problems. Some other proofs of a similar result were previously given in [4, 19, 20] using different formalisms, but it seems fair to say that all these proofs, including the most formal ones, only establish a very abstract version of the DRF guarantee, from which the notion of a program, in the sense of programming languages, is actually absent. Our aim here is to give a proof of the DRF guarantee, that is, a proof of the fact that a relaxed memory model implements the interleaving semantics for data-race free programs, *at the programming language level*. To this end, we introduce a new way of defining a memory model.

In the approach we propose, the memory model is defined as part of a *weak operational semantics* for the programming language (as opposed to the strong, i.e. interleaving semantics). Our weak operational semantics is quite concrete: to formalize the memory access reorderings supported by relaxed memory models, and to get a model similar to an architecture involving write buffers, we introduce a construct $\langle B\rangle T$ to describe (part of) a configuration involving a write buffer B, which, for some references (or pointers) $p_1, \ldots, p_k$, separately records sequences of values to be written in the memory. That is, B is a mapping

$$B = \{p_1 \mapsto v_1^1 \cdots v_{n_1}^1, \ldots, p_k \mapsto v_1^k \cdots v_{n_k}^k\}$$

where v_1^i is the first-in value for p_i, and $v_{n_i}^i$ is the last-in value. The syntax for thread systems with write buffers is given by

$$T ::= e \mid \langle B\rangle T \mid (T \parallel T')$$

where e is any program of the language. We should think of these as (downwards) trees, with unary nodes labeled by buffers, binary nodes putting together parallel components, and where each leaf is a thread, that is a program. The shared memory (or store) should be seen as sitting over the root of such trees. The buffer B in $\langle B\rangle T$ collects the write operations issued from the thread system T, and is shared among all the threads in T. Then in our weak operational semantics a write instruction $(p := v)$ does not directly affect the memory, but puts the value v for p in a buffer on the path to the store. For instance we have, using ML's notation $!\,p$ for dereferencing a pointer:

$$\begin{array}{rl} & x := 1\,; r_0 := !\,y \parallel y := 1\,; r_1 := !\,x \\ \xrightarrow{*} & \langle x \mapsto 1\rangle r_0 := !\,y \parallel \langle y \mapsto 1\rangle r_1 := !\,x \end{array}$$

Then the writes are propagated from the buffers to the store, in an asynchronous way (the precise semantical rules are given in Section 4). For each pointer, the propagation of values follows the FIFO order determined by the sequencing of thread instructions, but is otherwise performed in an interleaved manner, and is asynchronous with respect to the execution of threads. That is, the propagation of writes is parallel to the execution of threads. A read operation $!\,p$ returns the most recent value for p, as viewed from the thread issuing this operation, that is, the last-in value for p in the closest buffer on the path from the thread reading p to the store, if any, or the value stored for p in the shared memory otherwise. Then, assuming that the memory initially contains $\{x \mapsto 0, y \mapsto 0\}$, a possible execution, where the buffered writes are delayed with respect to the execution of the threads, is the following:

$$\begin{array}{rl} & x := 1\,; r_0 := !\,y \parallel y := 1\,; r_1 := !\,x \\ \xrightarrow{*} & \langle x \mapsto 1\rangle r_0 := 0 \parallel \langle y \mapsto 1\rangle r_1 := 0 \end{array}$$

That is, we get the outcome $\{r_0 \mapsto 0, r_1 \mapsto 0\}$, which we cannot get by the usual interleaving semantics. (This is similar to what happens with the mutual exclusion algorithm example given above.) In our model, there are queues of values to write in the memory for each reference, and not just sequences of writes. Then the writes on distinct references issued by a given thread may be reordered in the propagation process (this is called "jockeying" in [15]). For instance, consider the following system of threads:

$$\begin{array}{lcl} r_0 := !\,y\,; & \parallel & x := 1\,; \\ r_1 := !\,x & & y := 1 \end{array}$$

Then, starting with $\{x \mapsto 0, y \mapsto 0\}$, a possible outcome is $\{r_0 \mapsto 1, r_1 \mapsto 0\}$, since, representing termination by $()$, we have:

$$\begin{array}{rl} & r_0 := !\,y\,; r_1 := !\,x \parallel x := 1\,; y := 1 \\ \xrightarrow{*} & r_0 := !\,y\,; r_1 := !\,x \parallel \langle x \mapsto 1, y \mapsto 1\rangle() \\ \rightarrow & \langle y \mapsto 1\rangle\big(r_0 := !\,y\,; r_1 := !\,x \parallel \langle x \mapsto 1\rangle()\big) \end{array}$$

The various relaxations, that is, reorderings of memory operations that are allowed in our model, are *observed* from the behaviour of threads dictated by the rules of the weak operational semantics, rather than being prescribed a priori in an axiomatic manner. According to the classification of Adve and Gharachorloo's tutorial paper [3], we can say that our model implements:

W→R, that is reordering of a read of some reference w.r.t. writes on distinct references previously issued by the same thread;

W→W, that is reordering of writes issued by a given thread on distinct references;

Read others' write early, that is the possibility to read a write issued by another thread, even before this write has affected the memory;

Read own write early, the same, with respect to the writes issued by a given thread.

On the other hand, our model does not implement **R→RW**, that is, the reordering of reads with respect to subsequent memory operations in the same thread (this is explained in Section 4). One can also see that our model maintains "write atomicity" in the sense of [3].

To the best of our knowledge, with the exception of [33] that we will comment in Section 6, most of the approaches about memory models describe such a model using partial orders, prescribing in an axiomatic way which are the allowed (schemata of) executions, as sets of events equipped with partial order relations (see [29, 34] for instance, and [14] for an approach using the more abstract configuration structures). However, this notion of an acceptable execution is generally not used (with the exception of [13]) to give a weak semantics to programs. The paper [18] for instance argues that high-level memory models should be used to support an end-to-end approach; however, the proposed memory model (LC), which is based on partial order execution semantics, is not related to the semantics of some high-level concurrent programming language. In our proof of the DRF guarantee, we shall also use a partial order relation on events, to establish that data-race free programs are well-synchronized, that is the fact that two conflicting actions performed concurrently in a computation sequence of a DRF program are separated by a synchronization action (more precisely, an

[1] We deviate here from the standard terminology, where "well-synchronized" – or "properly labeled" [19] – is often taken as synonymous to "data-race free."

unlock action). However, here we directly extract this partial order from the standard interleaving semantics of threads, using "true concurrency" techniques [11, 12]. Then both our memory model and our proof of the DRF guarantee are based on descriptions of the strong (that is, interleaving) and weak semantics that follow the standard operational style which, we believe, is easier to understand, and to formally manipulate (as our proof shows), than an axiomatic partial order semantics.

To conclude this introduction, we must point out that we do not claim we are proposing a new relaxed memory model. Our contribution rather is to propose using an operational approach to such models, as this allows us to give a proof of the DRF guarantee at the programming language level. Our proof uses arguments that, we think, could be adapted to other models than the one with write buffers we are considering here. In particular, it would be interesting to apply similar techniques to more concrete models, closer to actual hardware designs, possibly with a different geometry than a tree as regards the interconnection network, and to the weak concurrency semantics introduced by compiler optimizations for sequential code. For such weak semantics one should also prove the DRF guarantee.

The rest of the paper is organized as follows: in Section 2 we introduce our language, which is Core ML (without typing) with some concurrent programming constructs, and we define the reference (interleaving) semantics of this language – the one a programmer is supposed to understand. This section also defines the data-race freeness property. Then, in Section 3 we use Berry and Lévy's equivalence by permutation of computations [7, 28] to show that DRF programs are properly synchronized. In Section 4 we introduce our main contribution, namely the weak operational semantics of the language, involving write buffering. Then in Section 5, using the bisimulation method, we show our main result, namely that the weak semantics does not introduce unwanted outcomes for DRF programs. In Section 6, we briefly discuss some related work, and finally conclude, indicating some possible future work.

2. The Language

Our language is basically Core ML (without typing), that is an imperative call-by-value λ-calculus, enriched with thread creation and a construct for ensuring mutual exclusion. The syntax is:

$$\begin{array}{rcll} e & ::= & v \mid (e_0 e_1) & \textit{expressions} \\ & \mid & (\mathsf{ref}\, e) \mid (\mathbf{!}\, e) \mid (e_0 := e_1) & \\ & \mid & (\mathsf{thread}\, e) \mid (\mathsf{with}\ \ell\ \mathsf{do}\ e) & \\ v & ::= & x \mid \lambda x e \mid () & \textit{values} \end{array}$$

where ℓ is a *lock*, that is a name from a given infinite set $\mathcal{L}ocks$. As usual, λ is a binder for the variable x in $\lambda x e$, and we shall consider expressions up to α-conversion, that is up to the renaming of bound variables. The capture-avoiding substitution of e_0 for the free occurrences of x in e_1 is denoted $\{x \mapsto e_0\} e_1$. We shall use some standard abbreviations like $(\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1)$ for $(\lambda x e_1 e_0)$, which is also denoted $e_0 \,;\, e_1$ whenever x does not occur free in e_1. We could easily add to the language standard constructs such as recursion, or conditional branching on boolean values.

To state the operational semantics of the language, we have to extend it with run-time constructs, in two ways. First, we introduce *references* (sometimes also referred to as memory locations, memory addresses, or pointers) p, $q, \ldots$ that are names from a given infinite set $\mathcal{R}ef$ (disjoint from $\mathcal{L}ocks$). These are (run-time) values. Then we use the construct $(\mathsf{holding}\ \ell\ \mathsf{do}\ e)$ to hold a lock for e. As it is standard with languages involving concurrency with shared variables, we follow a *small-step* style to describe the operational semantics, where an atomic transition consists in reducing a *redex* (reducible expression) within an *evaluation context*, while possibly performing a side effect. The syntax is then extended and enriched as follows:

$$\begin{array}{rcll} p,\ q \ldots & \in & \mathcal{R}ef & \textit{references} \\ v & ::= & \cdots \mid p & \textit{run-time values} \\ e & ::= & \cdots \mid (\mathsf{holding}\ \ell\ \mathsf{do}\ e) & \textit{run-time expressions} \\ r & ::= & (\lambda x e v) & \textit{redexes} \\ & \mid & (\mathsf{ref}\, v) \mid (\mathbf{!}\, p) \mid (p := v) & \\ & \mid & (\mathsf{thread}\, e) \mid (\mathsf{with}\ \ell\ \mathsf{do}\ e) \mid (\mathsf{holding}\ \ell\ \mathsf{do}\ v) & \\ \mathbf{E} & ::= & [\,] \mid \mathbf{E}[\mathbf{F}] & \textit{evaluation contexts} \\ \mathbf{F} & = & ([\,]\, e) \mid (v\, [\,]) & \textit{frames} \\ & \mid & (\mathsf{ref}\, [\,]) \mid (\mathbf{!}\, [\,]) \mid ([\,] := e) \mid (v := [\,]) & \\ & \mid & (\mathsf{holding}\ \ell\ \mathsf{do}\ [\,]) & \end{array}$$

As usual, we denote by $\mathbf{E}[e]$ the expression resulting from filling the hole in $\mathbf{E}$ by e. Every expression of the (run-time) language is either a value, or a redex in a position to be reduced, or faulty. More precisely, let us say that an expression is *faulty* if it has one of the following forms:

- $(v e)$ where the value v is not a function $\lambda x e'$;
- $(\mathbf{!}\, v)$ or $v := v'$ where the value v is not a reference.

Then we have:

LEMMA 2.1. *For any expression e of the run-time language, either e is a value, or there is a unique evaluation context $\mathbf{E}$ and a unique expression e' which either is a redex, or is faulty, such that $e = \mathbf{E}[e']$.*

(The proof, by induction on e, is immediate, once we observe that if $\mathbf{E}$ is an evaluation context and $\mathbf{F}$ a frame, then $\mathbf{F}[\mathbf{E}]$ is an evaluation context.)

The transitions in the operational semantics go from one configuration to another. A *configuration* is a triple (S, L, T) where S is the *store*, L is the *lock context*, and T is a *thread system*. Let us define these components, and introduce some notations. The store, also called here the memory, is a mapping from a finite set $\mathsf{dom}(S)$ of references to values. We denote by $S[p := v]$ the store obtained from S by updating the value of the reference p to v. The lock context L is a finite set of locks, those that are currently held by some thread. More concretely, this would be represented by a specific part of the store, giving to a finite number of locks a value (not directly accessible from the language) from the set $\{\mathsf{free}, \mathsf{busy}\}$. However, in the relaxed semantics we will have to maintain this distinct from the standard store. Finally T is a *thread system*. This could be given some structure, like a multiset, or a queue of threads (that is, expressions), but for our purposes it will be convenient to have some syntax for that, namely:

$$T \ ::= \ e \mid (T \parallel T')$$

That is, a thread system is a parallel composition of expressions. This syntax is rigid, in the sense that here parallel composition is *not* assumed to be commutative or associative.

We shall use the symbol C to denote configurations. These configurations (S, L, T) will be qualified as *strong*, or sometimes standard, in what follows (where there will also be a notion of weak configuration). As usual, we shall assume we consider only *well-formed* configurations, meaning that any reference that occurs somewhere in the configuration belongs to the domain of the store, that is, it is bound to a value in the memory – we shall not define this property, which is preserved in the operational semantics, more formally. For instance, if e is an expression of the source language, the initial configuration $(\emptyset, \emptyset, e)$ is well-formed.

$$
\begin{array}{rcll}
(S, L, \mathbf{T}[\mathbf{E}[(\lambda x e v)]]) & \rightarrow & (S, L, \mathbf{T}[\mathbf{E}[\{x \mapsto v\}e]]) & \\
(S, L, \mathbf{T}[\mathbf{E}[(\mathsf{ref}\ v)]]) & \rightarrow & (S \cup \{p \mapsto v\}, L, \mathbf{T}[\mathbf{E}[p]]) & p \notin \mathsf{dom}(S) \\
(S, L, \mathbf{T}[\mathbf{E}[(!\, p)]]) & \rightarrow & (S, L, \mathbf{T}[\mathbf{E}[v]]) & S(p) = v \\
(S, L, \mathbf{T}[\mathbf{E}[(p := v)]]) & \rightarrow & (S[p := v], L, \mathbf{T}[\mathbf{E}[()]]) & \\
(S, L, \mathbf{T}[\mathbf{E}[(\mathsf{thread}\ e)]]) & \rightarrow & (S, L, \mathbf{T}[(\mathbf{E}[()] \parallel e)]) & \\
(S, L, \mathbf{T}[\mathbf{E}[(\mathsf{with}\ \ell\ \mathsf{do}\ e)]]) & \rightarrow & (S, L \cup \{\ell\}, \mathbf{T}[\mathbf{E}[(\mathsf{holding}\ \ell\ \mathsf{do}\ e)]]) & \ell \notin L \\
(S, L, \mathbf{T}[\mathbf{E}[(\mathsf{holding}\ \ell\ \mathsf{do}\ v)]]) & \rightarrow & (S, L - \{\ell\}, \mathbf{T}[\mathbf{E}[v]]) &
\end{array}
$$

Figure 1: the Interleaving Semantics

Our last ingredient before defining the operational semantics is the one of *parallel* evaluation context, given by

$$\mathbf{T} ::= [] \mid (\mathbf{T} \parallel T) \mid (T \parallel \mathbf{T})$$

The operational semantics of the language is given in Figure 1. As one can see, this is the usual *interleaving* semantics. At each step one non-deterministically chooses a thread of the form $\mathbf{E}[r]$ to reduce, if any. Reducing a redex is always an atomic, and, apart from the possible choice we have when creating a new reference, deterministic step. This semantics is intended to serve as the *reference semantics* for a programmer writing in our language. Since this semantics is quite standard, we do not further comment it. Let us just note that $(\mathsf{with}\ \ell\ \mathsf{do}\ e)$ is a structured synchronization construct (which we borrow from [31]), the semantics of which is that it performs a test-and-set operation on the lock ℓ and, if this succeeds, proceeds up to the termination of e, upon which the lock is released.

As usual, we denote by $C \xrightarrow{*} C'$ the reflexive and transitive closure of the transition relation, and we say that C' is reachable from C if $C \xrightarrow{*} C'$. Our main result will be established for configurations that are reachable from an initial configuration of the form $(\emptyset, \emptyset, e)$ where e is an expression of the source language. More generally, we shall consider *regular* configurations, where at most one thread can hold a lock, and where a lock held by some thread is indeed in the lock context. This is defined as follows:

DEFINITION (REGULAR CONFIGURATION) 2.2. *A configuration $C = (S, L, T)$ is* regular *if and only if it satisfies*
(i) *if* $T = \mathbf{T}_0[\mathbf{E}_0[(\mathsf{holding}\ \ell\ \mathsf{do}\ e_0)]] = \mathbf{T}_1[\mathbf{E}_1[(\mathsf{holding}\ \ell\ \mathsf{do}\ e_1)]]$ *then* $\mathbf{T}_0 = \mathbf{T}_1\ \&\ \mathbf{E}_0 = \mathbf{E}_1\ \&\ e_0 = e_1$
(ii) $T = \mathbf{T}[\mathbf{E}[(\mathsf{holding}\ \ell\ \mathsf{do}\ e)]] \Rightarrow \ell \in L$

For instance, any configuration of the form $(\emptyset, \emptyset, e)$ where e is an expression of the source language is regular. The following should be obvious:

REMARK 2.3. *If C is regular and $C \rightarrow C'$ then C' is regular.*

Apart from regularity, the only property of programs (that is, expressions) we shall consider in the following is *data-race freeness*. This is a safety property, meaning that one cannot reach a configuration where there are simultaneous accesses to the same reference, one of them being a write access:

DEFINITION (DRF PROGRAMS) 2.4. *A configuration $C = (S, L, T)$ involves a* data-race *if $T = \mathbf{T}[\mathbf{E}[r]] = \mathbf{T}'[\mathbf{E}'[r']]$ with $\mathbf{T} \neq \mathbf{T}'$, where r and r' are both accesses to the same reference p, that is redexes $(!\, p)$ or $(p := v)$, at least one of them is a write (that is, an assignment). A configuration C is* data-race free *if one cannot reach from C a configuration that involves a data-race. An expression e is* data-race free *if the initial configuration $(\emptyset, \emptyset, e)$ is data-race free.*

To conclude this section, let us make a comment on the synchronization constructs. We could easily add to the language a construct new_lock, for creating a new lock, which is a run-time value. Then we would use $(\mathsf{with}\ e_0\ \mathsf{do}\ e_1)$ where e_0 is an expression. We could also use explicit locking $(\mathsf{lock}\ e)$ and unlocking $(\mathsf{unlock}\ e)$ constructs, but we think it is a better discipline to use a block-structured construct as the one proposed here.

It should intuitively be clear that if, in every computation of a given configuration, two concurrent conflicting accesses to the same memory location are separated by a synchronization action, then the configuration is data-race free. The next section is devoted to establish that the converse is true.

3. Concurrency, Conflict and Event Ordering

To prove our main result regarding DRF programs, we need to introduce a refined presentation of the operational semantics, where we make explicit *where* a reduction occurs, and *what* reduction occurs. In doing this, we take our inspiration from previous work on "true concurrency semantics" [11, 12], which in turn relied on Berry and Lévy's notion of *permutation equivalence* of computations [7, 28]. Our aim here is to be able to define in a rigorous way the notion of the "happens before" relation [26]. More generally, we aim at defining, with respect to a given execution sequence of a program, an ordering meaning that an event in the computation sequence "intrinsically precedes" another one, that is, we cannot make the latter occur before the former, whatever reordering we can make of the sequence, by commuting concurrent, non conflicting steps. In order to define this notion, we have to introduce some technical definitions. We shall use various kinds of *sequences* in the following, which we shall collectively denote by $\sigma,\ \xi \ldots$ (and later also γ), and therefore we fix a few notations regarding sequences: the empty sequence is always denoted ε, and the concatenation of the sequence σ' after the sequence σ is denoted $\sigma \cdot \sigma'$. The prefix ordering is denoted $\leq$, that is, $\sigma \leq \sigma'$ if $\sigma' = \sigma \cdot \sigma''$ for some σ''. The length of σ is $|\sigma|$.

Now, we first need a notion of occurrence, denoting a path from the root to a node in a binary tree made of expressions put in parallel. An *occurrence* is a sequence of symbols $\leftarrow$, meaning "on the left of a parallel composition," and $\rightarrow$, meaning "on the right." We denote by $\mathcal{O}cc$ the set of occurrences, that is the free monoid $\{\leftarrow, \rightarrow\}^*$, and we use occ, or sometimes simply o, to range over $\mathcal{O}cc$. For each thread system T and occurrence occ, we define the subsystem (subtree) T/occ of T at occurrence occ – if this is indeed an occurrence of a subtree –, in the obvious way, that is:

$$
\begin{array}{rcl}
T/\varepsilon & = & T \\
(T \parallel T')/\leftarrow \cdot occ & = & T/occ \\
(T \parallel T')/\rightarrow \cdot occ & = & T'/occ
\end{array}
$$

(otherwise undefined), and we define similarly $\mathbf{T}/occ$. The (unique) occurrence of the hole in a parallel context, that is occ satisfying $\mathbf{T}/occ = []$, is denoted $@\mathbf{T}$. Whenever occ is an occurrence in T, that is T/occ is defined, we denote by $T[occ := T']$ the thread

$$
\begin{array}{rcll}
(S, L, \mathbf{T}[\mathbf{E}[(\lambda x e v)]]) & \xrightarrow[@\mathbf{T}]{\beta} & (S, L, \mathbf{T}[\mathbf{E}[\{x \mapsto v\}e]]) & \\
(S, L, \mathbf{T}[\mathbf{E}[(\mathsf{ref}\ v)]]) & \xrightarrow[@\mathbf{T}]{\nu_p} & (S \cup \{p \mapsto v\}, L, \mathbf{T}[\mathbf{E}[p]]) & p \notin \mathsf{dom}(S) \\
(S, L, \mathbf{T}[\mathbf{E}[(!\, p)]]) & \xrightarrow[@\mathbf{T}]{\mathsf{rd}_p} & (S, L, \mathbf{T}[\mathbf{E}[v]]) & S(p) = v \\
(S, L, \mathbf{T}[\mathbf{E}[(p := v)]]) & \xrightarrow[@\mathbf{T}]{\mathsf{wr}_p} & (S[p := v], L, \mathbf{T}[\mathbf{E}[()]]) & \\
(S, L, \mathbf{T}[\mathbf{E}[(\mathsf{thread}\ e)]]) & \xrightarrow[@\mathbf{T}]{\mathsf{spw}} & (S, L, \mathbf{T}[(\mathbf{E}[()] \parallel e)]) & \\
(S, L, \mathbf{T}[\mathbf{E}[(\mathsf{with}\ \ell\ \mathsf{do}\ e)]]) & \xrightarrow[@\mathbf{T}]{\widehat{\ell}} & (S, L \cup \{\ell\}, \mathbf{T}[\mathbf{E}[(\mathsf{holding}\ \ell\ \mathsf{do}\ e)]]) & \ell \notin L \\
(S, L, \mathbf{T}[\mathbf{E}[(\mathsf{holding}\ \ell\ \mathsf{do}\ v)]]) & \xrightarrow[@\mathbf{T}]{\overleftarrow{\ell}} & (S, L - \{\ell\}, \mathbf{T}[\mathbf{E}[v]]) &
\end{array}
$$

Figure 2: the Decorated Operational Semantics

system obtained from T by replacing the subtree T/occ at occurrence occ by T' (we omit the formal definition, by induction on occ, which should be obvious).

With the notion of an occurrence, we are able to say where, that is, in which thread, a reduction occurs: an occurrence is similar to a thread identifier in a thread system (although in our setting this identity may dynamically change, upon thread creation). Now to say what occurs, we introduce the notion of an *action*. There are several kinds of actions: performing a β-reduction, creating a new reference p in the store, reading or writing a reference, spawning a new thread, and taking or releasing a lock ℓ. Then the syntax of actions is as follows:

$$a,\ b \ldots ::= \beta \mid \nu_p \mid \mathsf{rd}_p \mid \mathsf{wr}_p \mid \mathsf{spw} \mid \widehat{\ell} \mid \overleftarrow{\ell}$$

We denote by $\mathcal{A}ct$ the set of actions. We now are in a position where we can formulate our refined, that is, decorated operational semantics. This takes the form of transitions

$$(S, L, T) \xrightarrow[o]{a} (S', L', T')$$

where a is the action performed, and o the occurrence where it is done. This is described in Figure 2. The relation between the two presentations of the semantics should be obvious:

LEMMA 3.1. *For any configuration C, we have $C \to C'$ if and only if $C \xrightarrow[o]{a} C'$ for some action a and occurrence o.*

Two occurrences in a thread system T are *concurrent* if, intuitively, they lead to non-overlapping parts of the tree T, or in other words, they are diverging paths. This is easily formally defined:

DEFINITION (CONCURRENCY) 3.2. *Two occurrences o and o' are* concurrent, *in notation $o \smile o'$, if neither is prefix of the other: $\neg(o \leq o')$ & $\neg(o' \leq o)$.*

It is easy to see for instance that if occurrence o_2 follows occurrence o_1 in a sequence of (decorated) transitions, then either $o_1 \smile o_2$ or $o_1 \leq o_2$, that is, o_2 cannot be a strict prefix of o_1. One can also observe that a thread system T involves a data-race if there are concurrent occurrences of accesses to the same reference in T, one of which is a write. The following should be clear:

REMARK 3.3. *If $o_1 \smile o_2$, and T/o_1 and T/o_2 are both defined, then for any T_1 and T_2 we have $(T[o_1 := T_1])[o_2 := T_2] = (T[o_2 := T_2])[o_1 := T_1]$.*

We now define what it means for actions to be *conflicting*. With the idea that locking ($\widehat{\ell}$) and unlocking ($\overleftarrow{\ell}$) are both write actions in a sense, we have:

DEFINITION (CONFLICT) 3.4. *The* conflict *relation $\#$ is the binary relation on actions given by*

$$
\begin{array}{rcl}
\# & =_{\mathrm{def}} & \bigcup_{p \in \mathcal{R}ef} \{(\nu_p, \nu_p), (\mathsf{wr}_p, \mathsf{wr}_p), (\mathsf{wr}_p, \mathsf{rd}_p), (\mathsf{rd}_p, \mathsf{wr}_p)\} \\
 & & \cup \bigcup_{\ell \in \mathcal{L}ocks} \{\widehat{\ell}, \overleftarrow{\ell}\} \times \{\widehat{\ell}, \overleftarrow{\ell}\}
\end{array}
$$

This is a symmetric relation which in particular, says that accesses to the same reference in the store are conflicting and cannot be reordered, because this would in general result in a different state, unless, obviously, both accesses are made to read the memory.

We could prove, as in [11, 12], a *diamond lemma*, which is a conditional confluence property saying that two concurrent and non-conflicting transitions from a given configuration can be "pushed-out" (in the sense of category theory) to close the diagram, resulting in a common configuration. This was the basis for defining the equivalence by permutation of computations in the above mentioned papers. Here we shall use another property, which we call *asynchrony* (see [12] for references about this terminology). This property asserts that two consecutive steps in a computation can be commuted if the actions are not conflicting, and the occurrences where they are performed are disjoint:

LEMMA (ASYNCHRONY) 3.5. *If $C \xrightarrow[o_1]{a_1} C_1 \xrightarrow[o_2]{a_2} C_2$ where $\neg(a_1 \# a_2)$ and $o_1 \smile o_2$ then there exists a unique configuration C_1' such that $C \xrightarrow[o_2]{a_2} C_1' \xrightarrow[o_1]{a_1} C_2$.*

PROOF: for this proof we let $C = (S, L, T)$, $C_1 = (S_1, L_1, T_1)$ and $C_2 = (S_2, L_2, T_2)$. First we observe that if $T = \mathbf{T}_1[\mathbf{E}_1[r_1]]$ with $T_1 = \mathbf{T}_1[e_1] = \mathbf{T}_2[\mathbf{E}_2[r_2]]$, so that $o_1 = @\mathbf{T}_1$ and $o_2 = @\mathbf{T}_2$, with $T_2 = \mathbf{T}_2[e_2]$, then we have $T/o_2 = \mathbf{E}_2[r_2]$ since $o_1 \smile o_2$. Then if a_2 can be performed from (S, L, T) at occurrence o_2, which only depends on S and L, with

$$(S, L, T) \xrightarrow[o_2]{a_2} (S_1', L_1', T_1') = C_1'$$

where $T_1' = T[o_2 := e_2]$, then $T_1'/o_1 = \mathbf{E}_1[r_1]$ and we know by Remark 3.3 that, if the action a_1 can be performed at occurrence o_1 from (S_1', L_1', T_1') (which only depends on S_1' and L_1'), this will result in a configuration of the form (S_2', L_2', T_2') where $T_2' = T_2$. It remains to check that, thanks to the hypothesis $\neg(a_1 \# a_2)$, the steps can indeed be permuted, and that $S_2' = S_2$ and $L_2' = L_2$. Then we proceed by a case analysis on the transitions. Let us just examine a few cases and sketch the corresponding proof (all the numerous cases are actually equally easy). In particular, we do not investigate the cases where one of the actions has no side effect, that is, one of the actions is β or spw, where it is obvious that the two actions can be commuted.

• $a_1 = \nu_p$. In this case we cannot have $a_2 = \mathsf{rd}_p$ or $a_2 = \mathsf{wr}_p$, since this would imply $o_1 \leq o_2$ (for p does not occur in T, since the configuration (S, L, T) is well-formed), a contradiction. If $a_2 = \nu_q$ we must have $q \neq p$ since $p \in \mathsf{dom}(S_1)$, and in this case

$$(S, L, T) \xrightarrow[o_2]{a_2} (S_1', L, T_1')$$

with $\mathsf{dom}(S_1') = \mathsf{dom}(S) \cup \{q\}$, and therefore a_1 can be performed from the configuration (S_1', L, T_1') since $p \notin \mathsf{dom}(S_1')$, and performing a_1 results in (S_2, L_2, T_2) (where $L_2 = L$). All the other cases are easy.

• $a_1 = \mathsf{wr}_p$. In this case we know that a_2 is neither wr_p nor rd_p. If, for instance, $a_2 = \mathsf{wr}_q$ with $q \neq p$, then we have $S_1 = S[p := v_1]$ and $S_2 = S_1[q := v_2]$ for some values v_1 and v_2. Then if we let $S_1' = S[q := v_2]$, we clearly have $S_2 = S_1'[p := v_1]$, and we easily conclude in this case.

• $a_1 = \overset{\frown}{\ell}$. In this case $S_1 = S$ and $L_1 = L \cup \{\ell\}$ with $\ell \notin L$. The action a_2 cannot be neither $\overset{\frown}{\ell}$ nor $\overset{\curvearrowleft}{\ell}$, and it is easy to see that it can be performed from (S, T, L), resulting in (S_1', L, T_1'), and therefore a_1 can be performed from the latter configuration. The case where $a_1 = \overset{\curvearrowleft}{\ell}$ is similar. ❑

We can now define the equivalence by permutation of transitions on computations. A *computation* is a sequence of (decorated) transitions

$$\gamma = C_0 \xrightarrow[o_1]{a_1} C_1 \ \cdots \ C_{n-1} \xrightarrow[o_n]{a_n} C_n$$

More formally, γ is a sequence of steps $C_i \xrightarrow[o_{i+1}]{a_{i+1}} C_i'$ such that $C_{j+1} = C_j'$ for any j, but one should notice that, given an initial configuration C_0, the sequence of actions and occurrences is enough to determine the whole computation (and among the actions, only the ν_p's are actually necessary). Then the equivalence by permutation is the congruence (with respect to concatenation, which, on computations, is only partially defined) on such sequences generated by the asynchrony property:

DEFINITION (EQUIVALENCE BY PERMUTATION) 3.6. *The* equivalence by permutation of transitions *is the least equivalence $\simeq$ on computations such that*

(i) *if* $C \xrightarrow[o_1]{a_1} C_1 \xrightarrow[o_2]{a_2} C_2$ *where* $\neg(a_1 \# a_2)$ *and* $o_1 \smile o_2$ *then*

$$C \xrightarrow[o_1]{a_1} C_1 \xrightarrow[o_2]{a_2} C_2 \simeq C \xrightarrow[o_2]{a_2} C_1' \xrightarrow[o_1]{a_1} C_2$$

where C_1' *is determined as in the Asynchrony Lemma;*

(ii) $\gamma_0 \simeq \gamma_0' \ \& \ \gamma_1 \simeq \gamma_1' \ \Rightarrow \ \gamma_0 \cdot \gamma_1 \simeq \gamma_0' \cdot \gamma_1'$.

It should be clear that if $\gamma \simeq \gamma'$ then $|\gamma| = |\gamma'|$, and γ and γ' perform the same actions at the same occurrences (and in the same number for each of such pair), possibly in a different order. The main purpose of this definition is to allow us to formally define an event ordering relation with respect to a computation γ. To introduce this last notion, let us first see an example. Let $e_0 = (p := v)$, $e_0' = (p := v')$ $e_1 = (\lambda x(\lambda z z e_0')())$ and $T = (e_0 \parallel e_1)$. Then for S satisfying $p \in \mathsf{dom}(S)$ we have

$$\begin{aligned}\gamma = (S, \emptyset, T) &\xrightarrow[\curvearrowleft]{\mathsf{wr}_p} (S', \emptyset, (() \parallel e_1)) \xrightarrow[\curvearrowright]{\beta} (S', \emptyset, (() \parallel (\lambda z z e_0')))\\ &\xrightarrow[\curvearrowright]{\mathsf{wr}_p} (S'', \emptyset, (() \parallel (\lambda z z ())) \xrightarrow[\curvearrowright]{\beta} (S'', \emptyset, (() \parallel ()))\end{aligned}$$

This computation is equivalent to the following one:

$$\begin{aligned}(S, \emptyset, T) &\xrightarrow[\curvearrowright]{\beta} (S, \emptyset, (e_0 \parallel (\lambda z z e_0'))) \xrightarrow[\curvearrowleft]{\mathsf{wr}_p} (S', \emptyset, (() \parallel (\lambda z z e_0')))\\ &\xrightarrow[\curvearrowright]{\mathsf{wr}_p} (S'', \emptyset, (() \parallel (\lambda z z ())) \xrightarrow[\curvearrowright]{\beta} (S'', \emptyset, (() \parallel ()))\end{aligned}$$

where we have permuted the first two steps. We cannot go further, since the second step in this computation is conflicting with the third, which in turn is not concurrent with the last one (these last two steps are in "program order" since they are performed from the same thread). In this example we can say that the first wr_p "inherently precedes" (this will be formally defined below) the second such action, and also precedes the second β, and that the first β "inherently precedes" the second wr_p and the second β. This example shows that a given action, say wr_p, may have, in a given computation, like γ, several different relations with another action, like β. Then the notion of action, even if complemented with its occurrence, is not the right basis to define the ordering we are looking for. We have to introduce the notion of an *event*, as follows:

DEFINITION (EVENTS) 3.7. *An* event *in a computation γ is a pair $(a, o)^i$ decorated by a positive integer i, where the action a is performed at occurrence o in the computation γ, and i is less than the number of such pairs (a, o) in γ (that is, $(a, o)^i$ is the i-th occurrence of action a performed at occurrence o in γ). We denote by $\mathsf{Event}(\gamma)$ the set of events determined by the computation γ.*

For instance, for the computation γ above, we have

$$\mathsf{Event}(\gamma) = \{(\mathsf{wr}_p, \curvearrowleft)^1, (\mathsf{wr}_p, \curvearrowright)^1, (\beta, \curvearrowright)^1, (\beta, \curvearrowright)^2\}$$

For a given computation

$$\gamma = C_0 \xrightarrow[o_1]{a_1} C_1 \ \cdots \ C_{n-1} \xrightarrow[o_n]{a_n} C_n$$

we denote by $\partial(\gamma)$ the sequence $(a_1, o_1)^1 \cdots (a_n, o_n)^k$ of events of γ in temporal order, that is, as they appear successively in γ. We can finally define the *event ordering* determined by a computation:

DEFINITION (EVENT ORDERING) 3.8. *Given a computation γ, we say that an event $(a, o)^i \in \mathsf{Event}(\gamma)$* inherently precedes *$(a', o')^j \in \mathsf{Event}(\gamma)$ in γ, in notation $(a, o)^i \leq_\gamma (a', o')^j$, if and only if in any $\gamma' \simeq \gamma$, the i-th occurrence of (a, o) precedes the j-th occurrence of (a', o').*

It should be clear that this is indeed an ordering, that is, a reflexive, transitive and anti-symmetric relation. Moreover, two conflicting actions can never be permuted, and therefore if

$$\partial(\gamma) = \sigma_0 \cdot (a, o)^i \cdot \sigma_1 \cdot (a', o')^j \cdot \sigma_2$$

with $a \# a'$ then $(a, o)^i \leq_\gamma (a', o')^j$. For instance, in the computation γ above, we have

$$(\mathsf{wr}_p, \curvearrowleft)^1 \leq_\gamma (\mathsf{wr}_p, \curvearrowright)^1 \leq_\gamma (\beta, \curvearrowright)^2$$

and

$$(\beta, \curvearrowright)^1 \leq_\gamma (\mathsf{wr}_p, \curvearrowright)^1$$

The event ordering in a computation contains in particular the "program order," that relates actions (a, o) and (a', o') in temporal order, such that $o \leq o'$ (this ordering also takes into account the "creation of redexes" identified by Lévy [28] in the λ-calculus).

To conclude this section we prove a property of DRF programs that will be crucial in establishing our main result. This property shows in particular that if, in a computation starting from a DRF configuration, two conflicting actions are performed concurrently, then in between the two there must be a synchronization, that is an action $\overset{\curvearrowleft}{\ell}$ of releasing a lock. Let us define:

DEFINITION (WELL-SYNCHRONIZED) 3.9. *A configuration C is* well-synchronized *if, for any computation*

$$C = C_0 \xrightarrow[o_1]{a_1} C_1 \ \cdots \ C_{n-1} \xrightarrow[o_n]{a_n} C_n$$

where $a_i \# a_j$ (with $i < j$) and $o_i \smile o_j$ then there exists h such that $i \leqslant h \leqslant j$, $a_h = \overset{\curvearrowleft}{\ell}$ and $o_i \leq o_h$.

This is similar to the DRF0 property of [4]. To show that DRF programs are well-synchronized, we first need a lemma, stating that, in a computation, two events, one of which precedes, in serialization order, the other one, but not inherently, can be moved to occur in the reverse temporal order:

LEMMA (TRANSPOSITION) 3.10. *Let γ be a computation such that*

$$\partial(\gamma) = \sigma_0 \cdot (a,o)^i \cdot \sigma_1 \cdot (a',o')^j \cdot \sigma_2$$

with $(a,o)^i \not\leq_\gamma (a',o')^j$. Then there exists $\gamma' \simeq \gamma$ such that $\partial(\gamma') = \sigma_0 \cdot \sigma_1' \cdot (a',o')^j \cdot (a,o)^i \cdot \sigma_1'' \cdot \sigma_2$.

PROOF: by induction on $|\sigma_1|$. If $\sigma_1 = \varepsilon$, we have neither $o \leq o'$ nor $a \# a'$, since otherwise we would have $(a,o)^i \leq_\gamma (a',o')^j$, and therefore $o \smile o'$ and $\neg(a \# a')$. Then by definition of the permutation equivalence we can commute these two steps. If $\sigma_1 = (a'',o'')^h \cdot \xi$ there are two cases:

• $(a'',o'')^h \not\leq_\gamma (a',o')^j$. In this case we apply twice the induction hypothesis, to transpose first $(a'',o'')^h$ and $(a',o')^j$, and then the latter with $(a,o)^i$.

• $(a'',o'')^h \leq_\gamma (a',o')^j$. We do not have $o \leq o''$, since otherwise we would have $(a,o)^i \leq_\gamma (a',o')^j$ by transitivity, and therefore $o \smile o''$. Similarly, it is impossible that $a \# a''$, and then by definition of the permutation equivalence we can transpose $(a,o)^i$ with $(a'',o'')^h$, and conclude using the induction hypothesis. ❑

PROPOSITION 3.11. *DRF regular configurations are well-synchronized.*

PROOF: we show that for any computation γ starting from a DRF regular configuration C, if

$$\partial(\gamma) = \sigma_0 \cdot (a,o)^i \cdot \sigma_1 \cdot (a',o')^j \cdot \sigma_2$$

with $(a,o)^i \leq_\gamma (a',o')^j$ and $o \smile o'$, then there exist ℓ, o'', k, ξ_0 and ξ_1 such that $(a,o)^i \cdot \sigma_1 \cdot (a',o')^j = \xi_0 \cdot (\overset{\frown}{\ell}, o'')^k \cdot \xi_1$ with $o \leq o''$. We proceed by induction on $|\sigma_1|$.

• $\sigma_1 = \varepsilon$. In this case we must have $a \# a'$, since otherwise we could commute $(a,o)^i$ and $(a',o')^j$, contradicting $(a,o)^i \leq_\gamma (a',o')^j$. Then we proceed by case on $a \# a'$. It is impossible that $a = \nu_p = a'$, because one cannot create twice the same reference in a given computation. If $a, a' \in \{\mathsf{wr}_p, \mathsf{rd}_p\}$ for some p, where either a or a' is wr_p, then e is not data-race free, since $o \smile o'$. Finally the only possible case is $a, a' \in \{\overset{\frown}{\ell}, \overset{\frown}{\ell}\}$ for some ℓ. Since one cannot acquire twice the same lock consecutively in a computation, either a or a' is $\overset{\frown}{\ell}$. Moreover, since C is a regular configuration, the same holds for the configuration performing $(a,o)^i$, by Remark 2.3, and therefore either $a = \overset{\frown}{\ell}$, or $a = \overset{\frown}{\ell}$, $a' = \overset{\frown}{\ell}$ and $o' = o$.

• $\sigma_1 = (a'',o'')^h \cdot \xi$. We distinguish again two cases.

(1) $(a'',o'')^h \leq_\gamma (a',o')^j$. If $o'' \smile o'$ then we use the induction hypothesis to conclude. Otherwise, we have $o'' \leq o'$, and since $o \smile o'$ this implies $o'' \smile o$, for it cannot be the case that $o'' < o$, whereas $o \leq o''$ would imply $o \leq o'$, contradicting $o \smile o'$. Now if $a \# a''$ we argue as in the base case, and otherwise we can commute $(a,o)^i$ with $(a'',o'')^h$, and then apply the induction hypothesis.

(2) $(a'',o'')^h \not\leq_\gamma (a',o')^j$. In this case we use the Transposition Lemma above, and conclude using the induction hypothesis. ❑

4. The Weak Memory Model

In this section we introduce our main contribution, namely an operational formulation of a "relaxed" memory model. As indicated in the Introduction, we extend the syntax of thread systems with a semantical ingredient, namely the one of a (write) buffer. Our buffers are not simply FIFO files recording the memory updates issued by a thread. We adopt a weaker memory model, where the writes regarding each reference are independently buffered. Then a *buffer* B is a mapping from a finite set $\mathsf{dom}(B)$ of references to sequences of values. We use the symbol s to denote sequences $v_1 \cdots v_n$ of values. In such a sequence the head (first-in value v_1) is on the left, and the tail is on the right, so that the last-in value is v_n. In particular $\{p \mapsto v\}$ denotes a buffer which assigns to the single reference p the sequence made of the single value v. We denote by $\mathsf{W}(B)$ the set of references for which there is indeed some write operation buffered in B, that is

$$\mathsf{W}(B) =_{\mathrm{def}} \{\, p \mid p \in \mathsf{dom}(B) \;\&\; B(p) \neq \varepsilon \,\}$$

In order to formulate the weak semantics, we need some technical definitions regarding buffers. First, we denote by $B[p \leftarrow v]$ the buffer obtained from B by putting the value v at the end of the queue assigned to this reference in B. If there is no such queue, that is, if $p \notin \mathsf{dom}(B)$, this is simply $B \cup \{p \mapsto v\}$. Then we denote by $B \uparrow p$ the buffer obtained by removing the first-in value for p in B. We shall only use this in the case where $p \in \mathsf{W}(B)$, and therefore we consider this is not defined otherwise. The thread systems with write buffers are described as follows:

$$\Theta \;::=\; T \;\mid\; \langle B \rangle \Theta \;\mid\; (\Theta \parallel \Theta)$$

We extend the notation W to these trees, denoting by $\mathsf{W}(\Theta)$ the set of references for which there is a write operation buffered in Θ (the formal definition, which should be obvious, is omitted). A *weak* configuration is a configuration possibly involving write buffers, that is a triple of the form (S, L, Θ). We still only consider *well-formed* (weak) configurations, and this means in particular that if p is in the domain of a buffer occurring in Θ, then we must have $p \in \mathsf{dom}(S)$. We need to generalize the parallel contexts into weak contexts, as follows:

$$\mathbf{\Theta} \;::=\; [\,] \;\mid\; \langle B \rangle \mathbf{\Theta} \;\mid\; (\mathbf{\Theta} \parallel \Theta) \;\mid\; (\Theta \parallel \mathbf{\Theta})$$

As above, $\mathsf{W}(\mathbf{\Theta})$ denotes the set of references for which there is a write operation recorded in the context $\mathbf{\Theta}$. In the semantics we will use the *most recent* value for a reference p along a given path in a weak configuration. In fact we shall only use this for the path leading to a read instruction $(!\,p)$ in a weak context $\mathbf{\Theta}$. In order to define the most recent value for p in $\mathbf{\Theta}$, we first define the sequence $\mathsf{buff}(p, \mathbf{\Theta})$ of values buffered for p in $\mathbf{\Theta}$, as follows:

$$\mathsf{buff}(p, []) = \varepsilon$$

$$\mathsf{buff}(p, \langle B \rangle \mathbf{\Theta}) = \begin{cases} B(p) \cdot \mathsf{buff}(p, \mathbf{\Theta}) & \text{if } p \in \mathsf{W}(B) \\ \mathsf{buff}(p, \mathbf{\Theta}) & \text{otherwise} \end{cases}$$

$$\mathsf{buff}(p, (\mathbf{\Theta} \parallel \Theta)) = \mathsf{buff}(p, \mathbf{\Theta})$$

$$\mathsf{buff}(p, (\Theta \parallel \mathbf{\Theta})) = \mathsf{buff}(p, \mathbf{\Theta})$$

Then the most recent value for p in $(S, \mathbf{\Theta})$ is defined as follows:

DEFINITION (MOST RECENT VALUE) 4.1. *Given a pair $(S, \mathbf{\Theta})$ of a store S and a weak context $\mathbf{\Theta}$, for any reference $p \in \mathsf{dom}(S)$ the* most recent value *given to p in $(S, \mathbf{\Theta})$, denoted $(S, \mathbf{\Theta})(p)$, is the value v such that $\mathsf{buff}(p, \mathbf{\Theta}) = s \cdot v$, if such a value exists, and $S(p)$ otherwise.*

We denote by $\mathbf{\Theta}\dagger$ the fact that there is no buffered write (for any reference) in $\mathbf{\Theta}$ on the path from the root to the occurrence of the hole, that is:

$$\mathbf{\Theta}\dagger \;\Leftrightarrow_{\mathrm{def}}\; \forall p.\; \mathsf{buff}(p, \mathbf{\Theta}) = \varepsilon$$

One may notice that if $\mathbf{\Theta}\dagger$, then for any reference p the most recent value for p in $(S, \mathbf{\Theta})$ is the one in the store S. As a last ingredient

$$
\begin{array}{rcll}
(S, L, \mathbf{\Theta}[\mathbf{E}[(\lambda x e v)]]) & \xrightarrow[@\mathbf{\Theta}]{\beta} & (S, L, \mathbf{\Theta}[\mathbf{E}[\{x \mapsto v\}e]]) & \\
(S, L, \mathbf{\Theta}[\mathbf{E}[(\mathsf{ref}\ v)]]) & \xrightarrow[@\mathbf{\Theta}]{\nu_p} & (S \cup \{p \mapsto v\}, L, \mathbf{\Theta}[\mathbf{E}[p]]) & p \notin \mathsf{dom}(S) \\
(S, L, \mathbf{\Theta}[\mathbf{E}[(!\, p)]]) & \xrightarrow[@\mathbf{\Theta}]{\mathsf{rd}_p} & (S, L, \mathbf{\Theta}[\mathbf{E}[v]]) & (S, \mathbf{\Theta})(p) = v \\
(S, L, \mathbf{\Theta}[\mathbf{E}[(p := v)]]) & \xrightarrow[@\mathbf{\Theta}]{\mathsf{wr}_p} & (S, L, \mathbf{\Theta}[\langle\{p \mapsto v\}\rangle\mathbf{E}[()]]) & \\
(S, L, \mathbf{\Theta}[\mathbf{E}[(\mathsf{thread}\, e)]]) & \xrightarrow[@\mathbf{\Theta}]{\mathsf{spw}} & (S, L, \mathbf{\Theta}[(\mathbf{E}[()] \parallel e)]) & \\
(S, L, \mathbf{\Theta}[\mathbf{E}[(\mathsf{with}\ \ell\ \mathsf{do}\ e)]]) & \xrightarrow[@\mathbf{\Theta}]{\overset{\frown}{\ell}} & (S, L \cup \{\ell\}, \mathbf{\Theta}[\mathbf{E}[(\mathsf{holding}\ \ell\ \mathsf{do}\ e)]]) & \ell \notin L \\
(S, L, \mathbf{\Theta}[\mathbf{E}[(\mathsf{holding}\ \ell\ \mathsf{do}\ v)]]) & \xrightarrow[@\mathbf{\Theta}]{\overset{\frown}{\ell}} & (S, L - \{\ell\}, \mathbf{\Theta}[\mathbf{E}[v]]) & \mathbf{\Theta}\dagger \\
(S, L, \langle B\rangle\Theta) & \xrightarrow[\varepsilon]{} & (S[p := v], L, \langle B \uparrow p\rangle\Theta) & B(p) = v \cdot s \\
(S, L, \mathbf{\Theta}[\langle B_0\rangle\langle B_1\rangle\Theta]) & \xrightarrow[@\mathbf{\Theta}]{} & (S, L, \mathbf{\Theta}[\langle B_0[p \leftarrow v]\rangle\langle B_1 \uparrow p\rangle\Theta]) & B_1(p) = v \cdot s \\
(S, L, \mathbf{\Theta}[(\langle B\rangle\Theta \parallel \Theta')]) & \xrightarrow[@\mathbf{\Theta}]{} & (S, L, \mathbf{\Theta}[\langle\{p \mapsto v\}\rangle(\langle B \uparrow p\rangle\Theta \parallel \Theta')]) & B(p) = v \cdot s \\
(S, L, \mathbf{\Theta}[(\Theta \parallel \langle B\rangle\Theta')]) & \xrightarrow[@\mathbf{\Theta}]{} & (S, L, \mathbf{\Theta}[\langle\{p \mapsto v\}\rangle(\Theta \parallel \langle B \uparrow p\rangle\Theta')]) & B(p) = v \cdot s \\
(S, L, \mathbf{\Theta}[\langle B\rangle\Theta]) & \xrightarrow[@\mathbf{\Theta}]{} & (S, L, \mathbf{\Theta}[\Theta]) & \mathsf{W}(B) = \emptyset
\end{array}
$$

Figure 3: the Weak Operational Semantics

we need in order to define the weak semantics, we extend the notion of a path, i.e. occurrence, to the weak setting. We add a new symbol $\downarrow$ to denote a step through a buffer. Then the notion of subtree Θ/occ at occurrence occ is redefined in the obvious way, with

$$\langle B\rangle\Theta/\downarrow \cdot\ occ = \Theta/occ$$

(the remaining clauses do not change). Similarly, the definition of the occurrence of the hole in a weak context $\mathbf{\Theta}$, still denoted $@\mathbf{\Theta}$, is extended in the obvious way, that is $@\langle B\rangle\mathbf{\Theta} = \downarrow \cdot @\mathbf{\Theta}$. We now have all the technical definitions needed to define the weak operational semantics, which is described as transitions

$$(S, L, \Theta) \xrightarrow[o]{a} (S', L', \Theta')$$

possibly without any action label. This is given in Figure 3. There are several important differences with the reference semantics:

(1) when performing a read operation $(!\, p)$, we take the value as the most recent one for p along the path that leads to this operation. This value is taken from the store only when there is no write buffered for p on the path to the store, that is $\mathsf{buff}(p, \mathbf{\Theta}) = \varepsilon$. We could also adopt this as a condition for performing a read operation $(!\, p)$, thus defining a stronger memory model[2], without affecting the correctness result.

(2) a write operation $(p := v)$ issued by a thread is *delayed*, that is, the value v is put in a write buffer for p. Such an operation is therefore asynchronous, that is, non-blocking.

(3) an unlock action now operates as a *memory barrier*, since one cannot release a lock unless all the write buffers on the path to the store are empty. Indeed, in an extended language, one could define:

$$\mathsf{fence} =_{\mathrm{def}} (\mathsf{let}\ x = \mathsf{new\_lock}\ \mathsf{in}\ (\mathsf{with}\ x\ \mathsf{do}\ ()))$$

Notice that, as usual, the semantics of acquiring or releasing a lock are still "strong," that is, these (write) operations are not delayed, and are atomic.

[2] similar to the IBM 370 model, and the PC model, see [3].

In addition to that, the weak semantics involves operations to update, in an asynchronous way (but respecting the "program order" for each reference), the memory. This is done by means of the first four undecorated transitions, where no action is indicated – we also call these the *silent* transitions: the first one pushes a value from a topmost write buffer into the store, and the other three transfer a buffered value to an upper level.

With these rules, the write operations issued from a given thread on different references are made independent, because a buffer records a distinct queue for each reference, and therefore they may be "globally performed," that is, finally reach the store, in any order. Similarly, the writes issued by different threads may update the store with no particular order. We let the reader check that, for instance, a possible outcome of

$$\begin{array}{l} x := 1\,; \\ y := 2 \end{array} \quad \parallel \Big(\begin{array}{l} p_0 := !\, y\,; \\ x := 3 \end{array} \quad \parallel \quad \begin{array}{l} p_1 := !\, x\,; \\ p_2 := !\, x \end{array} \Big)$$

is $\{p_0 \mapsto 2, p_1 \mapsto 3, p_2 \mapsto 1\}$. This example is taken from [34], as well as the following one:

$$\begin{array}{lll} x := 1\,; & \parallel & p_2 := !\, x\,; \\ p_0 := !\, y\,; & & x := 2\,; \\ p_1 := !\, x & & y := 3 \end{array}$$

where, a possible outcome is $\{p_0 \mapsto 3, p_1 \mapsto 1, p_2 \mapsto 1\}$, as the reader may check. An execution that is not permitted in our model, but is allowed in the JAVA memory model for instance is the following one. Let us assume the store initially contains $\{x \mapsto 0, y \mapsto 0\}$. Then the outcome $\{p_0 \mapsto 1, p_1 \mapsto 1\}$ is not possible in our relaxed model for the program

$$p_0 := !\, y\,; x := 1 \parallel p_1 := !\, x\,; y := 1$$

because one of the statements $p_0 := !\, y$ or $p_1 := !\, x$ has to be issued first, and in our model this means that $!\, y$ or $!\, x$ has to be evaluated, returning the value 0. An argument for allowing this outcome is that the implementation might realize that the statements $p_0 := !\, y$ and $x := 1$ are independent (and similarly for $p_1 := !\, x$ with respect to $y := 1$), and therefore may be commuted, which brings us back to an example given in the Introduction. However,

in our view this transformation pertains to compiler optimizations rather than to memory operation optimizations. In our language a write operation $p := e$ does not necessarily reduce to $p := !\,x$, and therefore moving such an operation could only be done after some code analysis that is typically a compiler's task. In our weak memory model, as in [15], the value to store is known at the time of issuance, and the relaxation, w.r.t. the strong semantics, concerns memory operations (redexes) of the form $p := v$ and $!\,p$.

The alert reader may have noticed that our semantics is a little bit too rigid. For instance, with the program

$$(\mathsf{thread}\, p_0 := !\,x)\,;(\mathsf{thread}\, p_1 := !\,x)\,; x := 1$$

and the initial store $\{x \mapsto 0\}$, the outcome $\{p_0 \mapsto 1, p_1 \mapsto 0\}$ is not possible, because the second thread is always informed of the write $x := 1$ before the first one. As a remedy to this, we could adopt a non-deterministic semantics for thread creation, namely

$$(S, L, \mathbf{T}_0[\mathbf{T}_1[\mathbf{E}[(\mathsf{thread}\, e)]]]) \to (S, L, \mathbf{T}_0[(\mathbf{T}_1[\mathbf{E}[()]] \parallel e)])$$

However, the proof of our main result would be more complicated with this rule, and therefore we do not consider it, since we are mainly interested in the semantics of DRF programs. (The semantics of programs that run into a data-race is sometimes considered undefined, see [8, 31] for instance.)

Clearly, by modifying the architecture of the weak configurations, we could describe different memory models. For instance, the write buffers could be queues of pairs (p, v) meaning that an update for reference p has been issued, in which case reordering the writes issued from a given thread is impossible. One could also assume that there is only one, or a finite number of buffers at the top of the configuration, thus reflecting static architectures with a fixed number of processors each with their own write buffers. All these models are stronger, that is, less relaxed than the one we describe in Figure 3, and therefore they induce fewer weak behaviours for programs.

The last rule in Figure 3 allows one to remove empty buffers. In this way, we may reduce a weak configuration to a standard one. To see this, let us denote by $(S, L, \Theta) \xrightarrow{*} (S', L', \Theta')$ the transition relation inductively defined by the following rules:

$$\frac{}{C \xrightarrow{*} C} \qquad \frac{C \xrightarrow[o]{} C'' \quad C'' \xrightarrow{*} C'}{C \xrightarrow{*} C'}$$

and let $\Theta /\!\!/ occ$ be the buffer found at node occ in Θ, if any, that is:

$$\begin{array}{rcl} \langle B\rangle\Theta /\!\!/ \varepsilon & = & B \\ \langle B\rangle\Theta /\!\!/ \downarrow \cdot\, occ & = & \Theta /\!\!/ occ \\ (\Theta \parallel \Theta') /\!\!/ \,\lnot \cdot\, occ & = & \Theta /\!\!/ occ \\ (\Theta \parallel \Theta') /\!\!/ \,\urcorner \cdot\, occ & = & \Theta' /\!\!/ occ \end{array}$$

Then we define the size $|\Theta|$ of the thread system with buffers Θ as follows:

$$\begin{array}{rcl} |T| & = & 0 \\ |\langle B\rangle\Theta| & = & 2 + \|\Theta\| + |B| \\ |(\Theta_0 \parallel \Theta_1)| & = & 2 + \dfrac{(|\Theta_0| + |\Theta_1|)(|\Theta_0| + |\Theta_1| + 3)}{2} \end{array}$$

where

$$\begin{array}{rcl} |B| & = & \sum_{p \in \mathsf{W}(B)} |B(p)| \\ \|\Theta\| & = & \sum_{\{\, o \mid \exists B.\ \Theta /\!\!/ o = B \,\}} |\Theta /\!\!/ o| \end{array}$$

Then we have

LEMMA (TERMINATION) 4.2. $(S, L, \Theta) \twoheadrightarrow (S', L, \Theta') \Rightarrow |\Theta| > |\Theta'|$

(The proof is omitted.) One can also check that if $|\Theta| > 0$ then there is (S', L, Θ') such that $(S, L, \Theta) \twoheadrightarrow (S', L, \Theta')$, and therefore we have:

COROLLARY 4.3. *For any weak configuration (S, L, Θ) there exists a standard configuration (S', L, T) such that $(S, L, \Theta) \xrightarrow{*} (S', L, T)$.*

As a consequence, we observe that if there is an unlock operation to perform, one can always clear the buffers to reach a state where the lock can be released. In other words, there is no deadlock due to an unlock operation. (There are obviously deadlocked configurations, where threads are stuck on trying to acquire busy locks.) We shall use the following notation:

$$(S, L, \Theta) \Downarrow (S', L, T) \;\Leftrightarrow_{\mathrm{def}}\; (S, L, \Theta) \xrightarrow{*} (S', L, T)$$

Notice that, given $C = (S, L, \Theta)$, there are in general several distinct stores S' such that $C \Downarrow (S', L, T)$, because the writes buffered at disjoint occurrences, that is, issued by concurrent threads, can be used to update the memory in any order.

5. Correctness

To establish our correctness property, which says that the weak semantics agrees with the reference semantics for data-race free programs, we first introduce a property of weak configurations that is called *coherence*, which means that there are no concurrent buffered writes for the same reference in the configuration (there may be concurrent writes, but they concern distinct references).

DEFINITION (COHERENCE) 5.1. *A thread system with buffers Θ is* coherent *if and only if*

$$o \smile o' \;\Rightarrow\; \mathsf{W}(\Theta /\!\!/ o) \cap \mathsf{W}(\Theta /\!\!/ o') = \emptyset$$

for any $o, o' \in \mathcal{O}cc$. We also say that the configuration (S, L, Θ) is coherent *if Θ is coherent.*

In other words, Θ is coherent if for any given reference p the set

$$\{\, o \mid p \in \mathsf{W}(\Theta /\!\!/ o) \,\}$$

is totally ordered by the prefix order $\leq$. This property obviously holds for any standard configuration (S, L, T), hence in particular for initial configurations of the form $(\emptyset, \emptyset, e)$. It should be intuitively clear that the relation $\Downarrow$ is deterministic for coherent configurations. This is what we now prove. First we observe that the silent transitions preserve coherence:

LEMMA 5.2. *If Θ is coherent and $(S, L, \Theta) \xrightarrow[occ]{} (S', L', \Theta')$ then Θ' is coherent.*

PROOF: one checks that, for any reference q, if $q \in \mathsf{W}(\Theta' /\!\!/ o) \cap \mathsf{W}(\Theta' /\!\!/ o')$ then $o \leq o'$ or $o' \leq o$, by cases on the transition $(S, L, \Theta) \xrightarrow[occ]{} (S', L', \Theta')$. If $\Theta = \mathbf{\Theta}[(\langle B\rangle\Theta_0 \parallel \Theta_1)]$ and $\Theta' = \mathbf{\Theta}[\langle\{p \mapsto v\}\rangle(\langle B \uparrow p\rangle\Theta_0 \parallel \Theta_1)]$ we only consider the case of $q = p$. We see that, since Θ is coherent, if $p \in \mathsf{W}(\Theta' /\!\!/ o)$ then either $o \leq @\mathbf{\Theta}$, or $o = @\mathbf{\Theta} \cdot \downarrow \cdot o'$ with $p \in \mathsf{W}(\langle B \uparrow p\rangle\Theta_0 /\!\!/ o')$, and therefore the occurrences of writes for p in Θ' are totally ordered w.r.t. the prefix ordering. The case where $\Theta = \mathbf{\Theta}[(\Theta_0 \parallel \langle B\rangle\Theta_1)]$ and $\Theta' = \mathbf{\Theta}[\langle\{p \mapsto v\}\rangle(\Theta_0 \parallel \langle B \uparrow p\rangle\Theta_1)]$ is similar, and all the other cases are immediate. ❑

Next we show that the silent transitions on coherent configurations are locally confluent:

LEMMA (LOCAL CONFLUENCE) 5.3. *If C is a coherent weak configuration, and $C \xrightarrow[o_0]{} C_0$ and $C \xrightarrow[o_1]{} C_1$ then either $C_0 = C_1$ or there exists C' such that $C_0 \xrightarrow{*} C'$ and $C_1 \xrightarrow{*} C'$.*

PROOF: there are many cases to consider, which are all easy, and therefore we only examine a few of them. For instance, we may have, if the transitions $C \xrightarrow[o_0]{} C_0$ and $C \xrightarrow[o_1]{} C_1$ are performed at disjoint occurrences, $C = (S, L, \mathbf{\Theta}[(\langle B_0\rangle\Theta_0 \,\|\langle B_1\rangle\Theta_1)]$ and

$$\begin{aligned} C_0 &= (S, L, \mathbf{\Theta}[\langle\{p \mapsto v\}\rangle(\langle B_0 \uparrow p\rangle\Theta_0 \,\|\langle B_1\rangle\Theta_1)]) \\ C_1 &= (S, L, \mathbf{\Theta}[\langle\{q \mapsto v'\}\rangle(\langle B_0\rangle\Theta_0 \,\|\langle B_1 \uparrow q\rangle\Theta_1)]) \end{aligned}$$

Since C is coherent, we have $p \neq q$, and in this case we let

$$C' = (S, L, \mathbf{\Theta}[\langle\{p \mapsto v, q \mapsto v'\}\rangle(\langle B_0 \uparrow p\rangle\Theta_0 \,\|\langle B_1 \uparrow q\rangle\Theta_1)])$$

and we have $C_0 \xrightarrow{*}\!\!\!\rightarrow C'$ and $C_1 \xrightarrow{*}\!\!\!\rightarrow C'$ in three steps.

When the two transitions $C \xrightarrow[o_0]{} C_0$ and $C \xrightarrow[o_1]{} C_1$ are performed at occurrences that are related by the prefix order, we may have for instance $C = (S, L, \mathbf{\Theta}[\langle B_0\rangle\langle B_1\rangle\langle B_2\rangle\Theta])$ with

$$\begin{aligned} C_0 &= (S, L, \mathbf{\Theta}[\langle B_0[p \leftarrow v]\rangle\langle B_1 \uparrow p\rangle\langle B_2\rangle\Theta]) \\ C_1 &= (S, L, \mathbf{\Theta}[\langle B_0\rangle\langle B_1[q \leftarrow v']\rangle\langle B_2 \uparrow q\rangle\Theta]) \end{aligned}$$

(where possibly $p = q$). In this case we let

$$C' = (S, L, \mathbf{\Theta}[\langle B_0[p \leftarrow v]\rangle\langle (B_1 \uparrow p)[q \leftarrow v']\rangle\langle B_2 \uparrow q\rangle\Theta])$$

and we have $C_0 \xrightarrow{*}\!\!\!\rightarrow C'$ and $C_1 \xrightarrow{*}\!\!\!\rightarrow C'$ in one step. ❑

As a consequence of the lemmas 4.2, 5.2 and 5.3, the relation $\twoheadrightarrow$ is confluent on coherent configurations, and therefore

COROLLARY 5.4. *If C is a coherent configuration then*

$$C \Downarrow C_0 \;\&\; C \Downarrow C_1 \;\Rightarrow\; C_0 = C_1$$

To establish our main result, we need a technical definition. We denote by $\pi(o)$ the *projection* of the (weak) occurrence o, given as follows:

$$\begin{aligned} \pi(\varepsilon) &= \varepsilon \\ \pi(\downarrow \cdot o) &= \pi(o) \\ \pi(\lhd \cdot o) &= \lhd \cdot \pi(o) \\ \pi(\rhd \cdot o) &= \rhd \cdot \pi(o) \end{aligned}$$

DEFINITION (The BISIMULATION RELATION) 5.5. *For any given (strong) configuration C, we define the relation $\mathcal{R}(C)$ between weak and strong configurations as follows: $C' \,\mathcal{R}(C)\, C''$ if and only if there exists a sequence of weak transitions*

$$C_0 = C \xrightarrow[o_0]{}\!\!\!\!\rightarrow C_1 \cdots \xrightarrow[o_n]{}\!\!\!\!\rightarrow C_n = C'$$

such that

$$C'_0 = C \xrightarrow[\pi(o_0)]{} C'_1 \cdots \xrightarrow[\pi(o_n)]{} C'_n = C''$$

is a valid sequence of (strong) transitions, with $C_i \Downarrow C'_i$ for all i.

(Notice that since C is a standard configuration, we actually have, with the notations of the definition, $C \xrightarrow[o_0]{} C_1$.) We show that, if C is a DRF configuration, the relation $\mathcal{R}(C)$ is indeed a bisimulation. First, we observe that the weak semantics simulates the reference one:

PROPOSITION 5.6. *If $C' \,\mathcal{R}(C)\, C''$ and $C'' \xrightarrow[o]{a} \overline{C''}$ then there exist $\overline{C'}$ and o' such that $C' \xrightarrow[o']{}\!\!\!\!\rightarrow \overline{C'}$ with $o = \pi(o')$ and $\overline{C'} \,\mathcal{R}(C)\, \overline{C''}$.*

PROOF: this is immediate, because if

$$C_0 = C \xrightarrow[o_0]{}\!\!\!\!\rightarrow C_1 \cdots \xrightarrow[o_n]{}\!\!\!\!\rightarrow C_n = C'$$

is such that

$$C \xrightarrow[\pi(o_0)]{} C'_1 \cdots \xrightarrow[\pi(o_n)]{} C'_n = C''$$

with $C_i \Downarrow C'_i$ for all i, hence in particular $C' \Downarrow C''$, then we have $C' \xrightarrow{*}\!\!\!\rightarrow C'' \xrightarrow[o]{a} C'''$, and in all cases except $a = \mathsf{wr}_p$ we have $C''' = \overline{C''}$, hence obviously $C''' \Downarrow \overline{C''}$. It is easy to see that $C''' \Downarrow \overline{C''}$ also holds in the case where $a = \mathsf{wr}_p$, since there is only one buffered write (on p) in C'''. ❑

To prove that, conversely, the weak semantics does not deviate from the reference semantics as regards data-race free programs, we need the following lemma:

LEMMA 5.7. *Let C be a strong regular configuration such that*

$$C = C_0 \xrightarrow[o_0]{}\!\!\!\!\rightarrow C_1 \cdots \xrightarrow[o_n]{}\!\!\!\!\rightarrow C_n = (S, L, \Theta)$$

with $p \in \mathsf{W}(\Theta /\!\!/ o)$. Then there exists i such that $a_i = \mathsf{wr}_p$ with $\pi(o) \leq \pi(o_i)$, and for all $i < j \leqslant n$ if $\pi(o_i) \leq \pi(o_j)$ then $a_j \neq \overset{\frown}{\ell}$.

PROOF: by induction on n. First we observe that, due to the hypothesis $\mathsf{W}(\Theta) \neq \emptyset$, we must have $n \neq 0$, since C is a standard configurations. If $n = 1$, then it is easy to see that the only possibility, in order to have $\mathsf{W}(\Theta) \neq \emptyset$, is $a_1 = \mathsf{wr}_p$ with $\mathsf{W}(\Theta) = \{p\}$ (and $o = o_1$).

Otherwise ($n > 1$), we proceed by cases on a_n. We notice that if $a_n = \overset{\frown}{\ell}$ then $o \not\leq o_n$. The lemma is obvious in the case where $a_n = \mathsf{wr}_p$ and $o = o_n$. Otherwise, we have $C_{n-1} \xrightarrow{*}\!\!\!\rightarrow \xrightarrow[o_n]{a_n} C_n$, and if $C_{n-1} = (S', L', \Theta')$ we have $p \in \mathsf{W}(\Theta')$ with

$$p \in \mathsf{W}(\Theta /\!\!/ o) \;\Rightarrow\; \exists o'.\, \pi(o') \leq \pi(o) \;\&\; p \in \mathsf{W}(\Theta' /\!\!/ o')$$

and we conclude using the induction hypothesis. ❑

Now we show that, for data-race free regular configurations, the second half of our bisimulation result holds. Moreover, we show that in the bisimulation scenario, the coherence property is preserved by the weak semantics (not just the silent transitions as in Lemma 5.2):

PROPOSITION 5.8. *If C is a DRF regular configuration, $C' \,\mathcal{R}(C)\, C''$ where C' is coherent and $C' \xrightarrow{*}\!\!\!\rightarrow \xrightarrow[o]{a} \overline{C'}$ then $\overline{C'}$ is coherent and there exists $\overline{C''}$ such that $C'' \xrightarrow[\pi(o)]{a} \overline{C''}$ and $\overline{C'} \,\mathcal{R}(C)\, \overline{C''}$.*

PROOF: we have

$$C_0 = C \xrightarrow[o_0]{}\!\!\!\!\rightarrow C_1 \cdots \xrightarrow[o_n]{}\!\!\!\!\rightarrow C_n = C'$$

and

$$C \xrightarrow[\pi(o_0)]{} C'_1 \cdots \xrightarrow[\pi(o_n)]{} C'_n = C''$$

with $C_i \Downarrow C'_i$ for all i. Let D be such that $C' \xrightarrow{*}\!\!\!\rightarrow D \xrightarrow[o]{a} \overline{C'}$. Then D is coherent by Lemma 5.2. By Lemma 4.3 there exists $\overline{D}$ such that $D \Downarrow \overline{D}$, hence $C' \Downarrow \overline{D}$, and therefore $\overline{D} = C''$ by Corollary 5.4. We proceed by induction on the length of the sequence of $\twoheadrightarrow$-transitions from D to C''. If this length is 0, that is, $D = C''$, we have $\pi(o) = o$ since C'' is a strong configuration, and either $a \neq \mathsf{wr}_p$ and $D \xrightarrow[o]{a} \overline{C'}$, or $a = \mathsf{wr}_p$. In the first case, we may let $\overline{C''} = \overline{C'}$. In the second case there obviously exists $\overline{C''}$ such that $D \xrightarrow[o]{a} \overline{C''}$ and $\overline{C'} \Downarrow \overline{C''}$. Since there is exactly one write buffered in $\overline{C'}$, this configuration is coherent, and clearly $\overline{C'} \,\mathcal{R}(C)\, \overline{C''}$.

Otherwise let D' be such that $D \xrightarrow[o']{} D' \xrightarrow{*}\!\!\!\rightarrow C''$. We show that there exist $\overline{D}$ and u such that $\overline{D}$ is coherent and $D' \xrightarrow[u]{a} \overline{D}$ with $\pi(u) = \pi(o)$ (we shall then conclude using the induction hypothesis regarding D'). We proceed by cases on the transitions

$D \xrightarrow[o]{a} \overline{C'}$ and $D \xrightarrow[o']{} D'$. There are many cases to consider, most of which are immediate. We only examine the ones where $a = \mathsf{wr}_p$ or rd_p, that is $D = (S, L, \Theta)$ with $\Theta = \mathbf{\Theta}[\mathbf{E}[r]]$ where $r = (p := v)$ or $r = (!\, p)$ and $o = @\mathbf{\Theta}$.

• $r = (p := v)$. We have $\overline{C'} = (S, L, \mathbf{\Theta}[\langle\{p \mapsto v\}\rangle\mathbf{E}[\langle\rangle]])$. If $o \smile o'$, let us consider the case where $\mathbf{\Theta} = \mathbf{\Theta}'[\mathbf{\Theta}_0[\langle B_0\rangle\langle B_1\rangle\Theta'] \parallel \mathbf{\Theta}_1]$ with

$$D' = (S, L, \mathbf{\Theta}'[\mathbf{\Theta}_0[\langle B_0[q \leftarrow v']\rangle\langle B_1 \uparrow q\rangle\Theta'] \parallel \mathbf{\Theta}_1[\mathbf{E}[r]]])$$

and $o = @\mathbf{\Theta}' \cdot \Rsh \cdot @\mathbf{\Theta}_1$. Assume that $q = p$. Then $p \in \mathsf{W}(\Theta /\!\!/ o' \cdot \downarrow)$ with

$$o' = @\mathbf{\Theta}' \cdot \Lsh \cdot @\mathbf{\Theta}_0$$

and by Lemma 5.7 there exists i such that $a_i = \mathsf{wr}_p$ and $\pi(o' \cdot \downarrow) \leq \pi(o_i)$, with $a_j \neq \overset{\frown}{\ell}$ for $i < j \leqslant n$ if $\pi(o_i) \leq \pi(o_j)$. Then $\pi(o_i) \smile \pi(o)$, but this contradicts Proposition 3.11 since $a_i \# a$ and C is data-race free and regular. Then it must be the case that $q \neq p$, and if we let $\overline{D} = (S, L, \Theta'')$ where

$$\Theta'' = \mathbf{\Theta}'[\mathbf{\Theta}_0[\langle B_0[q \leftarrow v']\rangle\langle B_1 \uparrow q\rangle\Theta'] \parallel \mathbf{\Theta}_1[\langle\{p \mapsto v\}\rangle\mathbf{E}[\langle\rangle]]]$$

then we have $D' \xrightarrow[o]{a} \overline{D}$. It remains to see that $\overline{D}$ is coherent. Assume that $p \in \Theta'' /\!\!/ o''$ with $o'' \smile o$. Then by Lemma 5.7 there exists i such that $a_i = \mathsf{wr}_p$ and $\pi(o'') \leq \pi(o_i)$, with $a_j \neq \overset{\frown}{\ell}$ for $i < j \leqslant n$ if $\pi(o_i) \leq \pi(o_j)$, but, as above, this contradicts Proposition 3.11.

Still assuming $o \smile o'$, let us consider the case where $\mathbf{\Theta} = \mathbf{\Theta}'[(\langle B\rangle\Theta' \parallel \mathbf{\Theta}_1)]$ with $o = @\mathbf{\Theta}' \cdot \Rsh \cdot @\mathbf{\Theta}_1$ and

$$D' = (S, L, \mathbf{\Theta}'[\langle\{q \mapsto v'\}\rangle(\langle B \uparrow q\rangle\Theta' \parallel \mathbf{\Theta}_1[\mathbf{E}[r]])])$$

and $o' = @\mathbf{\Theta}' \cdot \Lsh$. Since $q \in \mathsf{W}(\Theta /\!\!/ o')$, we can show, using as in the previous case Lemma 5.7 and Proposition 3.11, that $q \neq p$ (since otherwise this would contradict the assumption that C is DRF). Then we let in this case $\overline{D} = (S, L, \Theta'')$ where

$$\Theta'' = \mathbf{\Theta}'[\langle\{q \mapsto v'\}\rangle(\langle B \uparrow q\rangle\Theta' \parallel \mathbf{\Theta}_1[\langle\{p \mapsto v\}\rangle\mathbf{E}[\langle\rangle]])]$$

We have $D \xrightarrow[u]{a} \overline{D}$ where $u = @\mathbf{\Theta}' \cdot \downarrow \cdot \Rsh \cdot @\mathbf{\Theta}_1$, and we conclude as in the previous case. All the other cases (with $o \smile o'$ or $o' \leq o$) are easy. As indicated above, we conclude the proof of the Proposition in the case where $r = (p := v)$ using the induction hypothesis regarding D'.

• $r = (!\, p)$. We have $\overline{C'} = (S, L, \mathbf{\Theta}[\mathbf{E}[v]])$ where $v = (S, \mathbf{\Theta})(p)$. We only examine the case where $\mathbf{\Theta} = \mathbf{\Theta}'[(\langle B\rangle\Theta' \parallel \mathbf{\Theta}_0)]$ with $o = @\mathbf{\Theta}' \cdot \Rsh \cdot @\mathbf{\Theta}_0$, $o' = @\mathbf{\Theta}' \cdot \Lsh$ and

$$D' = (S, L, \mathbf{\Theta}'[\langle\{q \mapsto v'\}\rangle(\langle B \uparrow q\rangle\Theta' \parallel \mathbf{\Theta}_0[\mathbf{E}[r]])]$$

(all the other cases are easy). Assume that $q = p$. Then $p \in \mathsf{W}(\Theta /\!\!/ o')$, and therefore by Lemma 5.7 there exists i such that $a_i = \mathsf{wr}_p$ with $\pi(o') \leq \pi(o_i)$ and $i < j \leqslant n \;\Rightarrow\; a_j \neq \overset{\frown}{\ell}$, but this contradicts Proposition 3.11, since $\pi(o_i) \smile \pi(o)$ and $a_i \# a$. Then we must have $q \neq p$ in this case, and it is easy to see that we then have $(S, \mathbf{\Theta}'')(p) = v = (S, \mathbf{\Theta})(p)$ where

$$\mathbf{\Theta}'' = \mathbf{\Theta}'[\langle\{q \mapsto v'\}\rangle(\langle B \uparrow q\rangle\Theta' \parallel \mathbf{\Theta}_0)]$$

Therefore if we let $u = @\mathbf{\Theta}''$ and $\overline{D} = (S, L, \mathbf{\Theta}''[\mathbf{E}[v]])$ we have $D' \xrightarrow[u]{a} \overline{D}$ and $\pi(u) = \pi(o)$. By Lemma 5.2 D' is coherent, hence so is $\overline{D}$. We conclude the proof, as above, using the induction hypothesis for D'. ❑

As an obvious consequence of the propositions 5.6 and 5.8 (and of Corollary 5.4), we finally obtain the correctness result:

THEOREM (CORRECTNESS) 5.9. *The weak memory model implements the reference semantics for data-race free programs. More precisely, the strong configurations reachable from a (strong) DRF regular configuration C in the weak semantics coincide with the configurations reachable from the same configuration C in the reference semantics.*

Notice that in particular the weak semantics correctly implements sequential programs, that do not use the $(\mathsf{thread}\ e)$ construct.

6. Some related work

In the Introduction we briefly surveyed part of the literature on memory models, where our main source of inspiration was [15]. In this area the work that is the closest to ours is [33], where the authors introduce a syntactic model for hardware architectures involving buffers (FIFO queues of write requests) and caches, with rewriting rules which, in effect, define an operational semantics for the basic memory operations, decomposed using commit/reconcile steps. The target of this model is hardware design, and therefore the syntactic structure is static, and the programming language side (DRF guarantee) is not investigated, but the approach is nevertheless similar to ours. For further references about memory models, especially as regards performance issues, and particular hardware models, we refer to [2, 3]. In the rest of this section we briefly discuss some works that are related to ours, though not always dealing explicitly with relaxed memory models.

A line of work which is related to ours, as regards the method we use and the kind of result we get, is the one on software transactional memory (STM). Several recent papers [1, 16, 30] study this topic from an operational point of view, defining a strong and weak semantics for a high-level language, and establishing a correctness result for a particular class of programs. (The notions of weak and strong transactions, or recoverable and non-interfering atomic actions, also appear in [9, 10].) So one can see that these works are similar to ours in spirit. Furthermore, since they are focusing on synchronization problems arising in STM, which is a sort of memory model, it would be interesting to see whether more formal connections could be established. Some research in that direction has been initiated in [22], where some of the issues arising from the interaction between atomic blocks and relaxed memory models, and more specifically the JMM [29], are investigated.

The paper [32] proposes "a theory of memory models." Although it seems to us that this work is more concerned with compiler optimizations than with memory models proper (the two are quite often mixed), it proposes an approach similar to ours, where allowed transformations are specified at the syntactic level. A DRF guarantee, there called "the fundamental property," is shown, for a class of programs restricted to "steps," that is simultaneous asignments similar to $p := e$, on which transformations are defined (whereas our model deals with memory operations $p := v$). This work adopts an axiomatic style however, where the possible relaxations of the strong semantics are determined a priori, relying on partial orders representing dependencies, and not by considering a weak semantics. This sounds quite appropriate as regards the kind of optimizations that the authors consider, and it would be worth investigating whether the two approaches could be combined. In a similar vein, let us mention the work [23], the overall aim of which is to establish and end-to-end approach to concurrent programming in C−. By this is meant a programming style where properties of programs are proved by means of separation logic, and where the semantics under consideration (as regards logical properties in particular) includes, or more precisely is intended to include compiler optimizations for sequential code. It seems to us that the use of separation logic is a way to deal only with well-synchronized programs, and therefore that a property similar to the DRF guarantee should hold in this case too.

7. Conclusion and Future Work

We have proposed a new approach to relaxed memory models, by formalizing such a model by means of a weak operational semantics. This allowed us to prove the correctness of the weak memory model for data-race free programs. There are several directions in which this work could be extended. Observing that our model is less abstract than the ones that use partial orders of events, and more abstract than particular hardware architectures, we think two different directions could be explored: first, we could try to extract a more abstract weak semantics from the operational presentation, using the same true-concurrency techniques that we used for the strong case. Indeed, in [12] we built an event structure semantics (for CCS) in that way. This would allow us to compare our model with others presented using partial orders, like [14, 29, 34], and to see which configurations, axiomatically prescribed in a partial order approach, are allowed or not. Second, in the opposite direction, we could try to make our model closer to real hardware implementations. For instance, changing our model to deal with buffers made of sequences of writes (p, v), thus forbidding the $\mathbf{W}{\rightarrow}\mathbf{W}$ relaxation, it seems that we get a model which is very close to the INTEL 64 architecture [24].

We think it is easy to extend our model with some other synchronization mechanisms, like volatile variables for instance: this would mean having another way of creating references, $\mathsf{vref}\ e$, which in the weak semantics creates a "strong" reference, with atomic writes, that is, no write buffering. A more ambitious goal would be, as indicated in the previous section, to deal with atomic transactions. We could also model the prefetching of reads by allowing the value stored at some pointer to be propagated downwards into the read buffers, or simulate the behaviour of a cache by putting a value read from the store in a buffer close to the thread that issued the read operation. We think the correctness result still holds in these cases, but the proof has to be adapted. Finally, as we suggested in the previous section, it would be interesting to integrate compiler optimizations in our approach.

References

[1] M. ABADI, A. BIRRELL, T. HARRIS, M. ISARD, *Semantics of transactional memory and automatic mutual exclusion*, POPL'08 (2008) 63-74.

[2] S. V. ADVE, *Designing Memory Consistency Models for Shared-Memory Multiprocessors*, PhD Thesis, Univ. of Wisconsin (1993).

[3] S. A. ADVE, K. GHARACHORLOO, *Shared memory consistency models: a tutorial*, IEEE Computer Vol. 29 No. 12 (1996) 66-76.

[4] S. ADVE, M. D. HILL, *Weak ordering – A new definition*, ISCA'90 (1990) 2-14.

[5] D. ASPINALL, J. ŠEVČÍK, *Formalising Java's data race free guarantee*, TPHOLs'07, Lecture Notes in Comput. Sci. 4732 (2007) 22-37.

[6] D. ASPINALL, J. ŠEVČÍK, *Java memory model examples: good, bad and ugly*, VAMP'07 (2007).

[7] G. BERRY, J.-J. LÉVY, *Minimal and optimal computations of recursive programs*, J. of ACM 26 (1979) 148-175.

[8] H.-J. BOEHM, S. V. ADVE, *Foundations of the C++ concurrency model*, PLDI'08 (2008) 68-78.

[9] C. BLUNDELL, E. C. LEWIS, M. M. K. MARTIN, *Subtleties of transactional memory atomicity semantics*, IEEE Comput. Architecture Letters Vol. 5 No. 2 (2006).

[10] G. BOUDOL, *Atomic actions*, INRIA Res. Rep. 1026 and EATCS Bull. 38 (1989) 136-144.

[11] G. BOUDOL, I. CASTELLANI, *A non-interleaving semantics for CCS based on proved transitions*, Fundamenta Informaticae XI (1988) 433-452.

[12] G. BOUDOL, I. CASTELLANI, *Flow models of distributed computations: three equivalent semantics for CCS*, Information and Computation Vol. 114 No. 2 (1994) 247-314.

[13] P. CENCIARELLI, A. KNAPP, B. REUS, M. WIRSING, *An event-based structural operational semantics of multi-threaded Java*, in Formal Syntax and Semantics of JAVA, Lecture Notes in Comput. Sci. 1523 (1999) 157-200.

[14] P. CENCIARELLI, A. KNAPP, E. SIBILIO, *The Java memory model: operationally, denotationally, axiomatically*, ESOP'07, Lecture Notes in Comput. Sci. 4421 (2007) 331-346.

[15] M. DUBOIS, CH. SCHEURICH, F. BRIGGS, *Memory access buffering in multiprocessors*, ISCA'86 (1986) 434-442.

[16] L. EFFINGER-DEAN, M. KEHRT, D. GROSSMAN, *Transactional events for ML*, to appear in the Proc. of ICFP'08 (2008).

[17] G. R. GAO, V. SARKAR, *Location consistency – a new memory model and cache consistency protocol*, IEEE Trans. on Computers Vol. 49 No. 8 (2000) 798-813.

[18] G. R. GAO, V. SARKAR, *On the importance of an end-to-end view of memory consistency in future computer systems*, ISHPC'97, Lecture Notes in Comput. Sci. 1336 (1997) 30-41.

[19] K. GHARACHORLOO, D. LENOSKI, J. LAUDON, P. GIBBONS, A. GUPTA, J. HENNESSY, *Memory consistency and event ordering in scalable shared-memory multiprocessors*, ACM SIGARCH Computer Architecture News Vol. 18 No. 3a (1990) 15-26.

[20] P. B. GIBBONS, M. MERRITT, K. GHARACHORLOO, *Proving sequential consistency of high-performance shared memories*, ACM Symp. on Parallel Algorithms and Architectures (1991) 292-303.

[21] J. R. GOODMAN, *Cache consistency and sequential consistency*, Techn. Rep. TR1006, University of Wisconsin (1991).

[22] D. GROSSMAN, J. MANSON, W. PUGH, *What do high-level memory models mean for transactions?*, MSPC'06 (2006) 62-69.

[23] A. HOBOR, A. W. APPEL, F. ZAPPA NARDELLI, *Oracle semantics for concurrent separation logic*, ESOP'08, Lecture Notes in Comput. Sci. 4960 (2008) 353-360.

[24] INTEL CORP., *Intel 64 architecture memory ordering white paper*, (2007).

[25] M. HUISMAN, G. PETRI, *The Java memory model: a formal explanation*, VAMP'07 (2007).

[26] L. LAMPORT, *Time, clocks, and the ordering of events in a distributed system*, CACM Vol. 21 No. 7 (1978) 558-565.

[27] L. LAMPORT, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Trans. on Computers Vol. 28 No. 9 (1979) 690-691.

[28] J.-J. LÉVY, *Optimal reductions in the lambda calculus*, in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism (J. P. Seldin, J. R. Hindley, Eds), Academic Press (1980) 159-191.

[29] J. MANSON, W. PUGH, S. A. ADVE, *The Java memory model*, POPL'05 (2005) 378-391.

[30] K. F. MOORE, D. GROSSMAN, *High-level small-step operational semantics for transactions*, POPL'08 (2008) 51-62.

[31] J. C. REYNOLDS, *Toward a grainless semantics for shared-variable concurrency*, FST-TCS'04, Lecture Notes in Comput. Sci. 3328 (2004) 35-48.

[32] V. SARASWAT, R. JAGADEESAN, M. MICHAEL, C. von PRAUN, *A theory of memory models*, PPOPP'07 (2007) 161-172.

[33] X. SHEN, ARVIND, L. RUDOLPH, *Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers*, ISCA'99 (1999) 150-161.

[34] R. C. STEINKE, G. J. NUTT, *A unified theory of shared memory consistency*, JACM Vol. 51 No. 5 (2004) 800-849.