On Thin Air Reads
Towards an Event Structures Model of Relaxed Memory

Alan Jeffrey* and James Riely†
*Bell Labs and Mozilla Research
†DePaul University

***Abstract*—This is the first paper to propose a pure event structures model of relaxed memory. We propose *confusion-free event structures* over an alphabet with a *justification* relation as a model. Executions are modeled by *justified* configurations, where every read event has a justifying write event. Justification alone is too weak a criterion, since it allows cycles of the kind that result in so-called thin-air reads. *Acyclic justification* forbids such cycles, but also invalidates event reorderings that result from compiler optimizations and dynamic instruction scheduling. We propose a notion *well-justification*, based on a game-like model, which strikes a middle ground.**

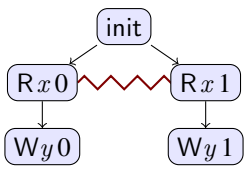
We show that well-justified configurations satisfy the *DRF theorem*: in any data-race free program, all well-justified configurations are sequentially consistent. We also show that *rely-guarantee reasoning* is sound for well-justified configurations, but not for justified configurations. For example, well-justified configurations are *type-safe*.

Well-justification allows many, but not all reorderings performed by relaxed memory. In particular, it fails to validate the commutation of independent reads. We discuss variations that may address these shortcomings.

1. Introduction

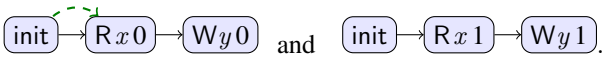
The last few decades have seen several attempts to define a suitable semantics for shared-memory concurrency under relaxed assumptions; see Batty et al. [2015] for a recent summary. *Event structures* [Winskel 1986] provide a way to visualize all of the conflicting executions of a program as a single semantic object. In this paper, we exploit the visual nature of event structures to provide a fresh approach to relaxed memory models.

Consider a simple programming language where all values are booleans, registers (ranged over by `r`) are thread-local and variables (ranged over by `x` and `y`) are global. In order to define the semantics compositionally, variable read is defined as a choice among the possible values that might be read. For example, the event structure for `(r=x; y=r;)` is as follows.



Register values are resolved via substitution and therefore do not appear in the event structure. The arrows represent program order, and the zigzag represents a primitive conflict. If two events are in conflict, then all following events are also in conflict.

This structure has two maximal conflict-free *configurations*, which represent a possible execution of the program:

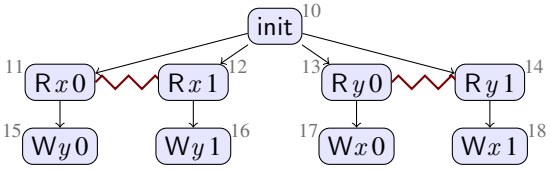


If we suppose that this code fragment is embedded in a larger program, the two configurations are equally sensible: x could be anything. However, if we take init to be the top-level initialization of the program and suppose that variables are initialized to 0, then the first configuration above seems sensible, whereas the second does not: x must be 0.

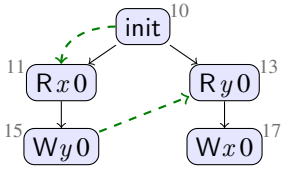
A read event is *justified* by a matching *visible* write, drawn with a dashed arrow in the above configurations. Writes are *hidden* if they occur later or are blocked by an intervening write. When modeling executions of whole programs, one expects that all reads in a configuration must be justified.

In a happens-before model [Manson et al. 2005], all concurrent writes are visible, making this notion of justification quite permissive. Consider a program with two threads and the corresponding event structure, with events numbered for reference.

`(r1=x; y=r1;) || (r2=y; x=r2;)` (P_1)

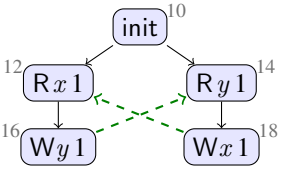


Here, the events that are neither ordered nor in conflict are concurrent. The event structure for P_1 has the following configuration, in which every read event is justified by a matching write that is either before it, or concurrent:



(C_0)

Unfortunately, the event structure also has a configuration in which there is a cycle in justification-and-program-order:



(C_1)

Due to the cycle, any available value can be so justified, thus arising "out of thin air". Some memory models have undefined semantics in the presence of such data races [Batty et al. 2011]. In the absence of such undefined behaviour, however, languages that claim memory safety must disallow thin-air values in order to preserve type safety.

Unfortunately, cycles such as that in configuration C_1 cannot be banned outright without also banning useful program transformations, such as instruction reordering. For example, consider the following program.

`(r1=x; y=1;) || (r2=y; x=1;)` (P_2)

The event structure for P_2 is the same as that for P_1 except that all writes have value 1. Thus, P_2 also allows configuration C_1. Clearly, if the order of the two instructions is swapped in either thread of P_2, then it is possible for both threads to read 1. Since program transformations may not introduce new behaviors, C_1 must also be considered a valid configuration of the original program.

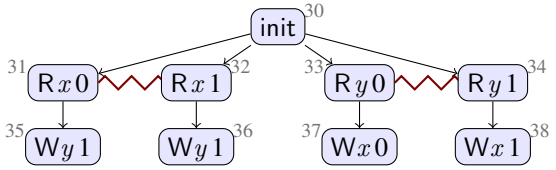
There are several models in the literature designed to allow configuration C_1 for P_2, yet deny it for P_1. Roughly these can be divided into two approaches: working with multiple executions [Manson et al. 2005; Jagadeesan et al. 2010] or working with axioms and rewrite rules [Cenciarelli et al. 2007; Saraswat et al. 2007; Pichon-Pharabod and Sewell 2016].

We propose a new approach, based on two-player games. The game is as follows: we start in configuration C, and the player's goal is to extend it to configuration D. The opponent picks a configuration C' which includes C, and whose new events are acyclically justified. The player then picks a configuration C'' which includes C', and whose new events are also acyclically justified. If C'' justifies D then the player has won, otherwise the opponent has won. If the player has a winning strategy for this game, we say that C *AE-justifies* D.

From this game, we can define the *well-justified* configurations inductively: $\emptyset$ is well-justified; if C is well-justified and C AE-justifies D then D is well-justified.

Consider the following program, P_3.

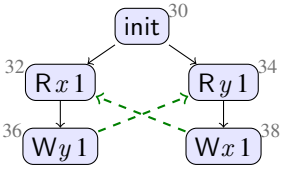
`(r1=x; y=1;) || (r2=y; x=r2;)` (P_3)



We show that both reads may be resolved to 1 in the well-justified configuration $\{30, 32, 34, 36, 38\}$. In this case the cyclic justifier models a valid execution, caused by a compiler or hardware optimization reordering `(r1=x; y=1;)` as `(y=1; r1=x;)`.

We first show that $\emptyset$ AE-justifies $\{30, 34, 38\}$. The opponent may choose any configuration acyclically justified from $\emptyset$; the interesting choices are the maximal configurations $\{30, 31, 33, 35, 37\}$ and $\{30, 31, 34, 35, 38\}$. Since both of these include 35, which justifies 34, the player does not have to add any events to justify $\{30, 34, 38\}$. Note that $\emptyset$ does *not* AE-justify $\{30, 32\}$, since the opponent can choose the configuration $\{30, 31, 35, 33, 37\}$.

We now show that the configuration $\{30, 34, 38\}$ AE-justifies $\{30, 32, 34, 36, 38\}$. The opponent may choose any configuration acyclically justified from $\{30, 34, 38\}$; since any choice includes 38, which justifies 32, the player does not have to add any events to justify $\{30, 32, 34, 36, 38\}$. We have thus shown a cyclic configuration similar to C_1 is well-justified for P_3.



This reasoning fails for configuration C_1 of P_1. In this case, the player is unable to establish that $\emptyset$ AE-justifies $\{10, 14, 18\}$. We provide a proof in §6. Intuitively, the only maximal configuration available to the opponent is $\{10, 11, 13, 15, 17\}$, and this fails to justify 14 since there is no write of 1 to y.

We review the literature on *confusion-free event structures* in §3. In §4 we define well-justification and provide further examples. We give the definition for a Java-like happens-before model [Manson et al. 2005]. We discuss synchronization actions, such as locks, in §7.

Perhaps the most important property of a relaxed memory model is *DRF*: that programs without data races behave as they would with strong memory—that is, as they would with sequentially consistent memory [Lamport 1979]. In §5, we describe our proof of the DRF theorem, which we have verified in Agda.

In §6, we show that invariant reasoning is possible using our definition. We state a general theorem—also verified in Agda—which is sufficient to establish type safety for static allocation.

We describe some of the limitations of our definition in §8. While the definition presented here is a step in the right direction, it fails to validate common reorderings, such as the reordering of reads on different variables. We give an alternative definition that is better behaved on the Java Memory Model causality test cases [Pugh 2004]. The induction principle used in our proof of DRF fails for this alternative definition.

The paper ends with a discussion of open problems.

The Agda development underlying this paper is available at http://asaj.org/papers/lics16.tgz.

2. Related work

Batty et al. [2015] describe the problem of thin-air executions and provide a detailed review of the literature. Lochbihler [2013] provides an encyclopedic survey and history of the Java Memory Model, in particular.

Event structures have appeared in previous attempts to formalized relaxed memory semantics. Cenciarelli et al. [2007] define a semantics for Java using ideas from event structures to describe the states of an operational transition relation. Pichon-Pharabod and Sewell [2016] define a semantics for C/C++ relaxed atomics using event structures as states used to define rewrite rules. In contrast, we define the semantics of programs using event structures alone, without requiring a additional layer of rewrite rules. Castellan [2016] presents an interleaving semantics for memory using event structures. In contrast, our semantics uses the standard non-interleaving interpretation of parallel composition.

The use of universal quantification over configurations in the definition of AE-justification is novel in this work. Other definitions of valid execution for weak memory, such as the JMM [Manson et al. 2005; Ševčík 2008; Lochbihler 2013], are purely existential in their quantification over possible executions.

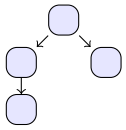
3. Event Structures

Event structures were introduced by Winskel [1989] as a non-interleaving model of concurrency. They are notable for providing a compact model of concurrent systems, for example an event structure model for n concurrent processes will often have only $O(n)$ events, compared to the $O(2^n)$ states in a labelled transition system.

In this section, we review the definitions associated with conflict-free labelled event structures, and their visualization as graphs. Readers familiar with event structures can skip to §4, where the new material begins.

A *partial order* $(E, \leq)$ is a set E (the *event set*) equipped with a reflexive, transitive, antisymmetric relation $\leq$ (the *causal order*). A *well order* is a partial order that has no infinite decreasing sequence.

We visualize partial orders as directed acyclic graphs where edges denote order. For example the order on $\{0,1,2,3\}$ where $0 \leq 1 \leq 2$ and $0 \leq 3$ is visualized on the right.

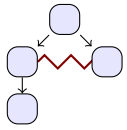


A *prime event structure* $(E, \leq, \#)$ is a well order together with a symmetric relation $\#$ on E (the *conflict* relation), such that if $c \# d \leq e$ then $c \# e$.

For any prime event structure, define the *primitive conflict* relation $\#_\mu$ on E as $d \#_\mu e$ whenever $d \# e$, and for any $d \geq b \# c \leq e$ we have $d = b$ and $c = e$. Primitive conflict is also known as *minimal* conflict. A prime event structure is *confusion-free* [Nielsen et al. 1979] whenever $\#_\mu$ is transitive, and if $c \leq d \#_\mu e$ then $c \leq e$.

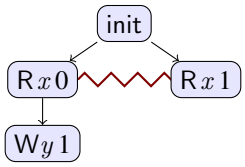
For any confusion-free event structure, define the *primitive conflict equivalence* $d \sim e$ whenever $d = e$ or $d \#_\mu e$. It is routine to show that primitive conflict equivalence is symmetric and transitive, and hence forms an equivalence on E.

We visualize confusion-free event structures by including the primitive conflict equivalence in the visualization. For example the event structure which extends the previous partial order with $1 \# 3$ and $2 \# 3$ has $1 \sim 3$, so is visualized as on the right.



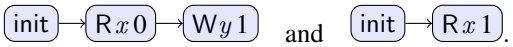
A *labelled event structure* $(E, \leq, \#, \lambda)$ over a *label set* Σ is a prime event structure together with a function $\lambda : E \to \Sigma$.

We visualize labelled event structures as node-labelled graphs. For example the labelled event structure which extends the previous event structure with labelling $\lambda(0) = \mathsf{init}$, $\lambda(1) = (\mathsf{R}\,x\,0)$, $\lambda(2) = (\mathsf{W}\,y\,1)$ and $\lambda(3) = (\mathsf{R}\,x\,1)$ is visualized as follows.



For any prime event structure, a set $C \subseteq E$ is *conflict-free* whenever there is no $d, e \in C$ such that $d \# e$. C is *down-closed* whenever $d \leq e \in C$ implies $d \in C$. A *configuration* is a set which is conflict-free and down-closed.

Since configurations are conflict-free, they can be visualized as node-labelled directed acyclic graphs, for example the two largest configurations for the previous labelled event structure are



Given labelled event structures ES_1 and ES_2 (without loss of generality, we assume event sets ES_1 and ES_2 are disjoint) define the *product* event structure $ES_1 \times ES_2$ as having:

- event set E is $E_1 \cup E_2$,
- causal order $\leq$ is $\leq_1 \cup \leq_2$,
- conflict $\#$ is $\#_1 \cup \#_2$, and
- labelling λ is $\lambda_1 \cup \lambda_2$.

The *sum* event structure $ES_1 + ES_2$ is the same except:

- conflict $\#$ is $\#_1 \cup \#_2 \cup (E_1 \times E_2) \cup (E_2 \times E_1)$.

We write 0 for the empty event structure with event set $\emptyset$.

For a label $\sigma \in \Sigma$, the prefix $\sigma \bullet ES_0$ introduces a new σ-labeled event ordered before all the events of ES_0. It is defined as having:

- event set E is $E_0 \cup \{\bot\}$,
- causal order $\leq$ is $\leq_0 \cup (\{\bot\} \times E)$,
- conflict $\#$ is $\#_0$, and
- labelling λ is $\lambda_0 \cup \{(\bot, \sigma)\}$.

Using an appropriate alphabet (discussed in more detail in §4), we can give the semantics of a simple shared-memory concurrent language. The construction uses sum, product, prefix and the empty event structure.

Let r range over *registers*. A *store* maps registers to values. Let ρ range over stores and ρ_0 be the initial store,

which maps all registers to 0. We write $\rho[r \mapsto v]$ for store update:

$$\rho[r \mapsto v](r') = \begin{cases} v & \text{if } r = r' \\ \rho(r') & \text{otherwise} \end{cases}$$

Let M range over *expressions*, which may include registers, but not variables. Let V be a set of *values*. Let $\mathscr{M}[\![\cdot]\!]$ be an interpretation that maps expressions and stores to values.

We give the semantics of a single-threaded program, featuring reads, writes and conditionals as an event structure:

$$\begin{aligned} \mathscr{T}[\![r{=}x;\ T]\!]\rho &\triangleq \textstyle\sum_{v \in V} (\mathsf{R}\, x\, v) \bullet \mathscr{T}[\![T]\!](\rho[r \mapsto v]) \\ \mathscr{T}[\![x{=}M;\ T]\!]\rho &\triangleq (\mathsf{W} x\, \mathscr{M}[\![M]\!]\rho) \bullet \mathscr{T}[\![T]\!]\rho \\ \mathscr{T}[\![\texttt{done}]\!]\rho &\triangleq 0 \\ \mathscr{T}[\![\texttt{if } M\ T_1\ \texttt{else}\ T_0]\!]\rho &\triangleq \begin{cases} \mathscr{T}[\![T_0]\!]\rho & \text{if } \mathscr{M}[\![M]\!]\rho = 0 \\ \mathscr{T}[\![T_1]\!]\rho & \text{otherwise} \end{cases} \end{aligned}$$

A *program* is an collection of threads $T_1 \,\|\, \cdots \,\|\, T_n$, interpreted as product of event structures with an initial event:

$$\mathscr{P}[\![T_1 \,\|\, \cdots \,\|\, T_n]\!] \triangleq \mathsf{init} \bullet (\mathscr{T}[\![T_1]\!]\rho_0 \times \cdots \times \mathscr{T}[\![T_n]\!]\rho_0)$$

We use standard abbreviations. For example, the program

```
if (x==0) {y=1;}
```

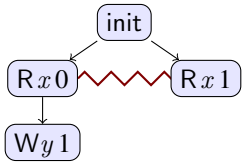
desugars to the following.

```
r=x; if (r==0) {y=1; done} else {done}
```

If we take $V = \{0, 1\}$, then this has semantics

$$\mathsf{init} \bullet (((\mathsf{R}\, x\, 0) \bullet (\mathsf{W}\, y\, 1) \bullet 0) + ((\mathsf{R}\, x\, 1) \bullet 0))$$

visualized as follows.



Note that in this semantics, conflict is only introduced by reads. Each conflicting event represents the read of a distinct value; since only one value can be read, the events are in primitive conflict.

4. Memory Event Structures

A *memory alphabet* (Σ, R, W, J, K) consists of

- a set Σ (the *actions*),
- set $R \subseteq \Sigma$ (the *read actions*)
- set $W \subseteq \Sigma$ (the *write actions*),
- binary relation $J \subseteq (W \times R)$ (*justification*), and
- binary relation $K \subseteq J$ (*synchronized justification*).

When $(a, b) \in J$, we say that *a justifies b*. Synchronization does not play a role in this section; we return to it in §7.

A *memory event structure* over such a memory alphabet is a confusion-free labelled event structure over Σ.

The prototypical memory alphabet consists of an initial action, and read and write actions over some set of *variables* X, and some set of *values* V:

$$\begin{aligned} \Sigma &= R \cup W \\ R &= \{(\mathsf{R}\, x\, v) \mid x \in X, v \in V\} \\ W &= \{(\mathsf{W} x\, v) \mid x \in X, v \in V\} \cup \{\mathsf{init}\} \end{aligned}$$

In §3 we saw that such an alphabet can be used to give the semantics for a simple shared-memory concurrent language. The justification relation for this alphabet is that init justifies a read of 0, and that a write of v justifies a read of v to the same variable:

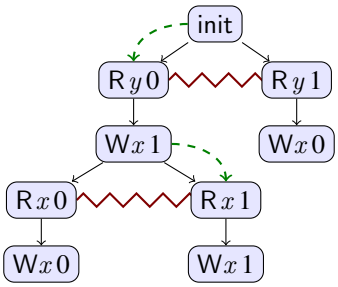
$$\begin{aligned} J = \{&(\mathsf{init}, (\mathsf{R}\, x\, 0)) \mid x \in X\} \\ \cup \{&((\mathsf{W} x\, v), (\mathsf{R}\, x\, v)) \mid x \in X, v \in V\} \end{aligned}$$

In a memory event structure, an event e is a *read event* whenever $\lambda(e) \in R$, and a *write event* whenever $\lambda(e) \in W$. The sets R and W need not be disjoint; thus, a memory alphabet may include read-modify-write actions such as exchange, compare-and-set or increment. The semantics of these operators as event structures is straightforward, following the style given in §3.

In a memory event structure, we can lift justification from labels to events, but this is not just a matter of looking at the labelling, since events should not be justified by later events, by events in conflict, or by events with an intervening event in read-write conflict. For example, in the program

```
if (y) { x=0; } else { x=1; x=x; }
```

with event structure semantics



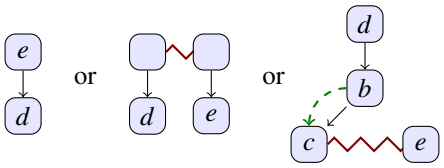
the only justified read of x is 1, not 0, and the only justified read of y is 0, not 1. We visualize justification as a dashed edge.

In a memory event structure, we say write event d *justifies* read event e whenever:

(1) $(\lambda(d), \lambda(e)) \in J$,
(2) we do not have $e < d$,
(3) we do not have $d \# e$, and
(4) there is no $d < b < e \sim c$ such that $\lambda(b)$ justifies $\lambda(c)$.

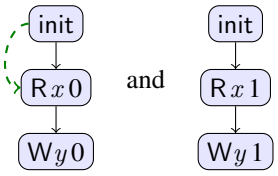
The following statement is equivalent to item (4): "there is no $d < b < e \sim c$ such that b justifies c." However, we cannot use this as the definition without worrying about circularity.

Visually these conditions are that d cannot justify e when:

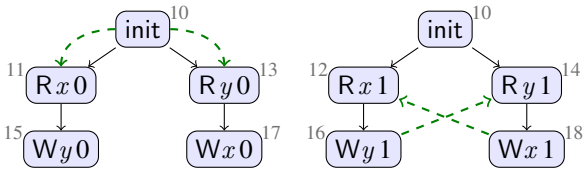


Definition 4.1 (Justified). A configuration C is *justified* whenever every read event in C is justified by an event in C. □

For example, the program `y=x;` has two maximal configurations, but only one of them is justified:



Unfortunately, justified configurations, although necessary, are not a sufficient condition for modeling valid executions, as they allow cycles in the union of causal order and justification, which cause *thin air reads*. For example, the program P_1 from the introduction includes the justified configurations $\{10, 11, 13, 15, 17\}$ and $\{10, 12, 14, 16, 18\}$:



In the latter, there is a cycle in causal+justification order. It is straightforward to ban such cycles.

Definition 4.2 (Acyclically justified). On configurations, define C *justifies* D whenever for any read event $d \in D$ there exists a write event $c \in C$ such that c justifies d.

Write $C \lesssim D$ whenever $C \subseteq D$ and C justifies D.

Write $\lesssim^*$ for the reflexive, transitive closure of $\lesssim$.

Define C is *acyclically justified* whenever $\emptyset \lesssim^* C$. □

Any acyclically justified configuration is also justified. In addition, any justified configuration with acyclic causal+justification order is acyclically justified.

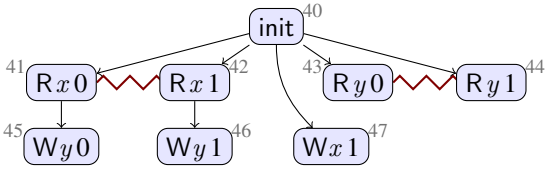
For example, for P_1, we have that $\emptyset \lesssim \{10\}$, since 10 is not a read. In addition, we have $\{10\} \lesssim \{10, 11\}$ and $\{10\} \lesssim \{10, 13\}$ since 10 justifies both 11 and 13. Taking a maximal configuration at each step, we have:

$$\emptyset \lesssim \{10\} \lesssim \{10, 11, 13, 15, 17\}$$

However, there is no such chain leading from $\emptyset$ to $\{10, 12, 14, 16, 18\}$.

Consider the following program.

```
(y=x; || x=1 || r=y;)
```
(P_4)



In this case the write to x is immediately available, since it is not causally dependent on any read. Thus:

$$\emptyset \lesssim \{40, 47\} \lesssim \{40, 47, 41, 45\} \lesssim \{40, 47, 41, 45, 43\}$$
$$\emptyset \lesssim \{40, 47\} \lesssim \{40, 47, 42, 46\} \lesssim \{40, 47, 42, 46, 44\}$$

Here the read of x is a coin-toss, which determines whether it is possible to read $(\mathsf{R}\, y\, 1)$: A configuration that contains 41 cannot also contain 44.

The second of these sequences can be seen in Figure 1, read from top to bottom. The events included in each successive configuration are highlighted using a darker, blue background. Events that are in conflict with an included event are covered in white. Thus in a maximal configuration, such as the last configuration in Figure 1, all events are either highlighted or covered.

Acyclic justification rules out cycles, since in any acyclically justified C, there must be configurations $\emptyset = C_0 \lesssim \cdots \lesssim C_n = C$, and for any read event $e \in C$ there must be a j such that $e \in C_{j+1}$ and a $d \in C_j$ which justifies e. Since configurations are $\leq$-closed, this means that there is no infinite sequence $d_1 \leq e_1, d_2 \leq e_2, \ldots$, where d_i justifies e_{i+1}, and in particular there are no cycles.

Unfortunately, acyclic justification is too strong a requirement, as it rules out some valid executions in the presence of optimizations which reorder memory accesses. For example, the program $P_3 =$ `(r=x; y=1; || x=y;)` has event structure given the introduction. In this case the cyclic justifier models a valid execution, caused by a compiler or hardware optimization reordering `(r=x; y=1;)` as `(y=1; r=x;)`. If we are going to admit such reorderings, we cannot model valid executions by a property of configurations, and must look at the entire event structure (this observation was made, in a different model, by Batty et al. [2015]).

Definition 4.3 (Well-justified). On configurations, define C *always eventually justifies (AE-justifies)* D whenever for any $C \lesssim^* C'$ there exists a $C' \lesssim^* C''$ such that C'' justifies D.

Write $C \sqsubseteq D$ whenever $C \subseteq D$ and C AE-justifies D.

Write $\sqsubseteq^*$ for the reflexive, transitive closure of $\sqsubseteq$.

Define C is *well-justified* whenever C is justified and $\emptyset \sqsubseteq^* C$. □

A well-justified configuration must be both justified and AE-justified. The notion of AE-justification describes when a read event is justified by some write event no matter which execution path is chosen. AE-justification has the flavor of a two-player game: in a configuration C_i, the opponent chooses a $C_i \lesssim C_i'$, after which the player chooses a $C_i' \lesssim C_i''$ which justifies C_{i+1}. If the player can justify C_{i+1} regardless of the opponent move, then the player wins the round. The player well-justifies C if they can repeat this game to move from the initial configuration $\emptyset$ to the final configuration C.

Any acyclically justified configuration is AE-justified.

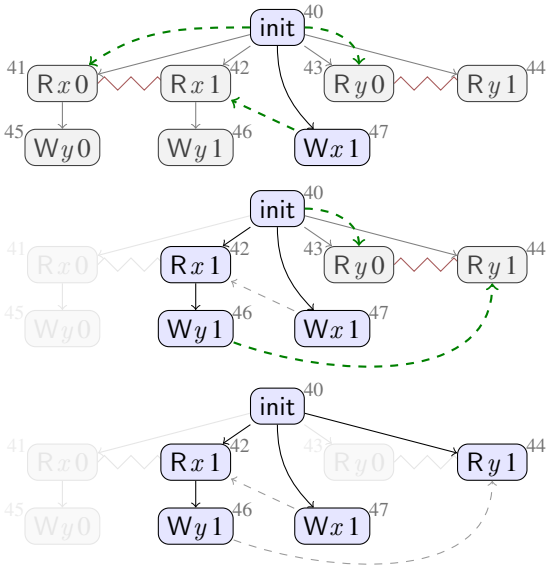


Figure 1: Acyclic- and AE-justification for P_4

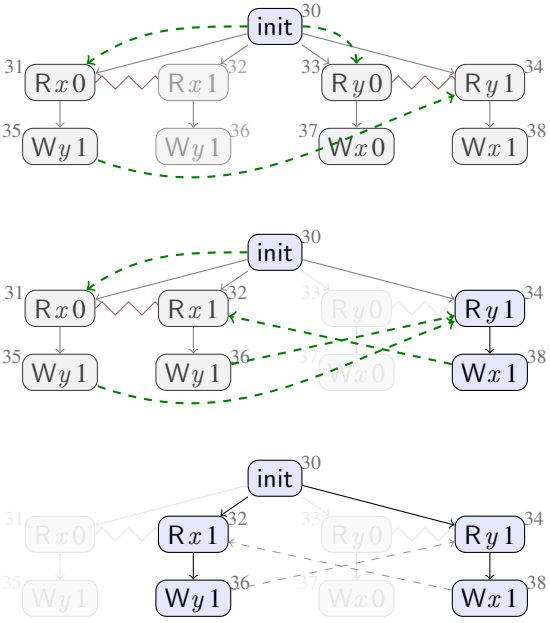


Figure 2: AE-order for P_3

Example 4.4. Consider the proof of acyclic justification given in Figure 1. Starting from $C_0 = \emptyset$, the player follows the proof of acyclic justification to select $C_1 = \{40, 47\}$, then $C_2 = \{40, 47, 42, 46\}$ and finally $C_3 = \{40, 47, 42, 46, 44\}$. In each case, the events in C_i are justified by an extension C''_{i-1} of C_{i-1}, regardless of the opponent's choice of C'_{i-1}.

For any opponent move $\emptyset \lesssim^* C'_0$, the player must choose $C'_0 \lesssim^* C''_0$ so that C''_0 justifies C_1. In this case, the player can always choose $C''_0 \supseteq \{40, 47\}$, since 40 and 47 conflict with no event and do not require justification. The first two moves of the player can be collapsed, choosing C_1 to be $\{40, 47, 42, 46\}$, since 42 can be justified regardless of the opponent move. However, the last two moves cannot be collapsed. The player cannot initially select 44; in this case the opponent would win by choosing $C'_0 = \{40, 41, 43, 45\}$. □

Example 4.5. We now consider the proof that P_3 is well-justified to read all ones, given in Figure 2. The previous strategy does not work, since the goal configuration is not acyclically justified.

The player chooses $C_1 = \{30\}$, $C_2 = \{30, 34, 38\}$ and finally $C_3 = \{30, 34, 38, 32, 36\}$. As in the previous example, the first two player moves can be collapsed, but not the last two. We show the first two player moves separately to make the opponent choices clear. The opponent can choose C'_1 to include any events except 32 (and therefore 36); there is no acyclically justified configuration that includes 32. For this reason events 32 and 36 are gray in the top configuration of Figure 2. The opponent option to include $33 \in C'_1$ prevents the player from selecting $32 \in C_1$. The $(\mathsf{R}\,x\,1)$ cannot be justified in this case. However, $(\mathsf{R}\,y\,1)$ can be justified regardless of the opponent's choice. Thus 34 can be included in C_1 (or C_2, as shown).

Once the player has won the round including 34, the opponent is no longer at liberty to include 33—the choice has been made. Thus the player may include 32 in the next round. □

Example 4.6. As noted in the introduction, configuration C_1 of P_1 fails to be well-justified. We provide a proof in §6. Intuitively, the player is unable to select $14 \in C_1$, because the opponent can choose $11 \in C'_0$. □

In the process of revising the Java Memory Model, Pugh [2004] developed a set of twenty *causality test cases*. Using hand calculation, we tested our semantics against nineteen of these cases. (TC9 is based on the idea that an execution should be allowed if there exists an augmentation, such as thread inlining, that allows it. This is a non-goal for our semantics; therefore, we do not consider TC9.)

Our semantics agrees with sixteen of the test cases and disagrees with three: TC3, TC7 and TC11. In §8, we discuss TC7, which best elucidates the issues.

Also by hand calculation, we found that our semantics gives the desired results for all examples in Batty et al. [2015, §4] and all but one in Ševčík [2008, §5.3]: redundant-write-after-read-elimination—this counterexample applies to any sensible non-coherent semantics.

5. Data-race-free event structures

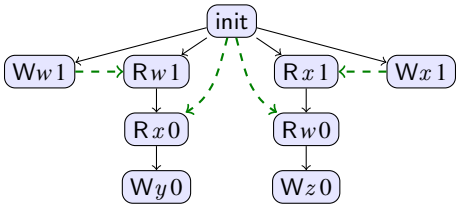
We say that ES' is an augmentation of ES if it has same events, conflict and labels, and possibly more order. Formally, $(E, \leq', \#, \lambda)$ is an augmentation of $(E, \leq, \#, \lambda)$ if $\leq \subseteq \leq'$.

It is straightforward to show that justification, acyclic justification and well-justification are all reflected by augmentation. For example, if ES' augments ES and C is a well-justified configuration of ES' then C is a well-justified configuration of ES.

A *sequential* memory event structure is one where, for any events d and e, either $d \leq e$, $e \leq d$ or $d \# e$. A *sequentially consistent* configuration of a memory event structure is a justified configuration of a sequential augmentation of it. That is, a configuration C of ES is sequentially consistent if there exists an augmentation ES' of ES such that ES' is sequential and C is a justified configuration of ES'.

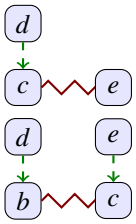
Note that in a sequential memory event structure, if d justifies e then $d \leq e$. It follows that any justified configuration of a sequential memory event structure is well-justified, and hence that any sequentially consistent configuration of a memory event structure is well-justified.

The converse is not true. There are well-justified configurations that are not sequentially consistent, due to data races. For example, the program `(w=1; || y=(w<=x); || z=(x<=w); || x=1;)` has semantics with configuration:



This is acyclically justified, and hence well-justified, but not sequentially consistent. In this section, we shall show that such data races are the only source of configurations which are well-justified but not sequentially consistent.

In a memory event structure, define concurrent events d and e to be a *read-write race* whenever there is some $c \sim e$ such that d justifies c, as shown on the right.

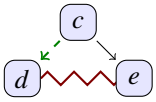


Define concurrent events d and e to be a *write-write race* whenever there is some $b \sim c$ such that d justifies b and e justifies c, as shown on the right.

Define a configuration to be *data-race-free* when it contains no read-write or write-write races.

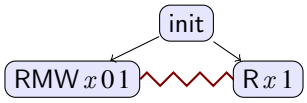
We state the theorem generally for any memory event structure that is read-enabled and commutative. The prototypical example, given in §4, satisfies both these criteria.

A memory event structure is *read-enabled* whenever, for any read event e there exists some $c \leq e \sim d$ such that c justifies d, as shown on the right. Any event structure that is the semantics of a program is read enabled. For example, if e is $(\mathsf{R}\,x\,1)$, then it suffices to take c to be init, since init justifies $(\mathsf{R}\,x\,0)$, which is in primitive conflict with $(\mathsf{R}\,x\,1)$.



A memory event structure is *commutative* whenever $c \sim d$ and d justifies e implies there exists $b \sim e$ where c justifies b, that is:

If read and write actions are disjoint, then it follows immediately that the resulting event structure will be commutative. Read-modify-write operators such as swap and fetch-and-add are commutative. Compare-and-set (CAS) is commutative if we interpret a failed CAS as both read and write (of the old value), but not if we consider a failed CAS only as a read. Under this interpretation, for example, CAS(x,0,1) generates the following event structure for bit register x.



The $(\mathsf{RMW}\,x\,0\,1)$ event may justify some other read of x; however, the minimal conflicting event $(\mathsf{R}\,x\,1)$ is a plain read. We leave weakening commutativity as future work.

Theorem 5.1 (DRF). In any commutative read-enabled memory event structure, if all sequentially consistent configurations are data-race-free, then all well-justified configurations are sequentially consistent.

PROOF. Define a configuration C to be *pre-justified* if every read action $e \in C$ is justified by a write action $d \in C$ where $d \leq e$. It is routine to show that any pre-justified configuration is sequentially consistent. The core lemma of the proof follows [Lochbihler 2013], which is that if C is pre-justified, and C justifies D, then D is pre-justified. After this, the proof is routine: we first show that if C is pre-justified and $C \lesssim^* D$ then D is pre-justified; then that if C is pre-justified and $C \sqsubseteq^* D$ then D is pre-justified. The result follows, since $\emptyset$ is trivially pre-justified. This proof has been mechanized in Agda. □

6. Invariants

While there is no formal definition of "thin-air read" [Batty et al. 2015], the examples point to a failure of inductive reasoning, typically due to a cycle in the union of the causal and data dependency orders. In order to establish that these forms of thin-air read are impossible, it is sufficient to show that is possible to reason inductively. In this section, we show that well-justification enables inductive reasoning.

We consider a limited form of invariant reasoning, which is strong enough to capture non-temporal safety properties, such as type safety. Given a suitable notion of formula, ϕ, we show that if, in every configuration of ES, the *read* events satisfy ϕ, then, in every configuration of ES, *all* events satisfy ϕ. Significantly, the result can be applied without reasoning about well-justification.

To keep the setting as simple as possible, we consider logics over labels rather than events. In order to establish the result, we must restrict attention to logics that are subset closed. This allows the expression of certain safety properties such as $x \neq 1$, but not liveness properties such as $x=1$. For a label set Σ, a *program logic* $(\Phi, \vDash)$ consists of:

- a set Φ (the *formulae*), and
- a binary relation $\vDash$ between $\mathscr{P}(\Sigma)$ and Φ (*satisfaction*).

A formula ϕ is *subset closed* whenever $A \subseteq B \vDash \phi$ implies $A \vDash \phi$. It is *satisfiable* whenever $A \vDash \phi$ for some A. It

respects justification whenever A justifies B and $A \vDash \phi$ implies $B \vDash \phi$.

For any configuration C, let $\Sigma(C)$ be the labels of C:

$$\Sigma(C) = \{\lambda(e) \mid e \in C\}$$

A formula ϕ is an *invariant* of a memory event structure whenever $\Sigma(C) \cap R \vDash \phi$ implies $\Sigma(C) \vDash \phi$ for any configuration C (recalling that R is the set of all read actions).

A formula ϕ is a *tautology* of a memory event structure whenever $\Sigma(C) \vDash \phi$ for any well-justified configuration C.

Theorem 6.1. For any satisfiable, subset-closed ϕ which respects justification, if ϕ is an invariant of ES then ϕ is a tautology of ES. □

PROOF. Mechanized in Agda. □

In the remainder of this section, we consider an example logic. Let T be a set of *type names*, ranged over by τ. The set Φ of formulae is generated by the following BNF.

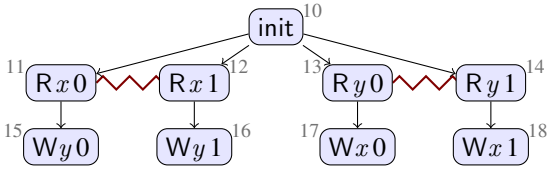
$$\phi, \psi ::= x{\neq}v \mid x{:}\tau \mid \mathsf{true} \mid \mathsf{false} \mid \phi \wedge \psi \mid \phi \vee \psi$$

Given a semantics $V_\tau \subseteq V$ for each type τ, let $\vDash$ be the obvious satisfaction relation generated by the following rules for the atoms.

- $A \vDash x{\neq}v$ when for any $a \in A$, if $a = (\mathsf{R}\,x\,w)$ or $a = (\mathsf{W}x\,w)$, then $w \neq v$.
- $A \vDash x{:}\tau$ when for any $a \in A$, if $a = (\mathsf{R}\,x\,v)$ or $a = (\mathsf{W}x\,v)$, then $v \in V_\tau$.

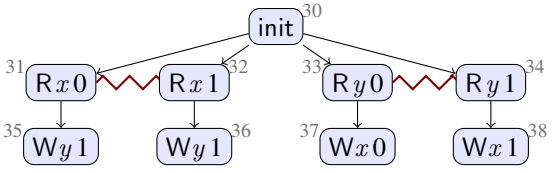
Note that the logic is satisfiable and subset closed and thus satisfies the criteria of Theorem 6.1.

Suppose that we attempt to show that $\phi_1 = (x{\neq}1 \wedge y{\neq}1)$ is a tautology for $P_1 =$ `(y=x; || x=y;)`. Recall the event structure for this program, given in the introduction.



Note that any configuration which includes write events 16 or 18, must also include read events 12 or 14. Thus, if the read events satisfy ϕ_1 then the write events satisfy ϕ_1, and so ϕ_1 is invariant for P_1. Thus, by Theorem 6.1, ϕ_1 is a tautology for P_1.

For P_3=`(r=x; y=1; || x=y;)`, instead, ϕ_1 fails. Recall the event structure for this program, also given in the introduction.



The configuration $\{30, 31, 35\}$ fails to satisfy ϕ_1 even though its only read event 31 satisfies ϕ_1.

These examples can be adapted to show reasoning using types. Let 0 be the unique value of type Unit. Then P_1 satisfies the typing $x{:}\mathsf{Unit} \wedge y{:}\mathsf{Unit}$, but P_2 does not.

7. Fencing

In §4, we noted that a memory alphabet includes *synchronized justification*, such as lock release and acquire; however, we have not made any use of synchronization up to now. In this section, we develop a notion of fencing, in which synchronized justifications contribute to causal order.

We model lock-based synchronization, in a very simple setting. We assume that there is only one, statically allocated lock and that lock release always occurs in the same thread as the previous acquire. The latter assumption ensures that each release causally follows the corresponding acquire.

The memory alphabet from §4 is modified to include acquire and release actions, which are considered both read and write actions. (We comment on this design in §9.)

$$\begin{aligned}\Sigma &= R \cup W\\ R &= \{(\mathsf{R}\,x\,v), \mathsf{Aq}, \mathsf{Rl} \mid x \in X, v \in V\}\\ W &= \{(\mathsf{W}x\,v), \mathsf{Aq}, \mathsf{Rl} \mid x \in X, v \in V\} \cup \{\mathsf{init}\}\end{aligned}$$

The semantics of locking actions are as follows.

$$\begin{aligned}\mathscr{T}[\![\texttt{acq;}T]\!]\rho &\triangleq \mathsf{Aq} \bullet \mathscr{T}[\![T]\!]\rho\\ \mathscr{T}[\![\texttt{rel;}T]\!]\rho &\triangleq \mathsf{Rl} \bullet \mathscr{T}[\![T]\!]\rho\end{aligned}$$

The justification relation, J, now includes lock actions:

$$\begin{aligned}J = \{&(\mathsf{init}, (\mathsf{R}\,x\,0)), ((\mathsf{W}x\,v), (\mathsf{R}\,x\,v)) \mid x \in X, v \in V\}\\ &\cup \{(\mathsf{init}, \mathsf{Aq}), (\mathsf{Aq}, \mathsf{Rl}), (\mathsf{Rl}, \mathsf{Aq})\}\end{aligned}$$

The synchronized justification relation, K, is restricted to lock actions.

$$K = \{(\mathsf{init}, \mathsf{Aq}), (\mathsf{Aq}, \mathsf{Rl}), (\mathsf{Rl}, \mathsf{Aq})\}$$

Recall from §4 that event d *justifies* event e whenever (1) $(\lambda(d), \lambda(e)) \in J$, (2) d does not follow e, (3) d is not in conflict with e, and (4) there is no intervening b between d and e that justifies an event in primitive conflict with e.

We say event d *synchronously justifies* event e whenever d justifies e and $(\lambda(d), \lambda(e)) \in K$.

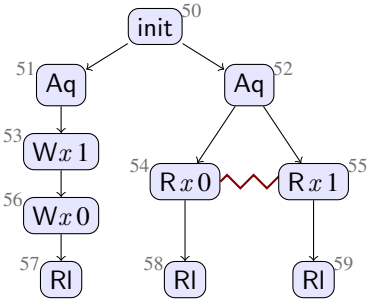
A *fenced* memory event structure is one where, for any events d and e, if d synchronously justifies e then $d \leq e$.

A *well-fenced* configuration of a memory event structure is a well-justified configuration of a fenced augmentation of it. That is, a configuration C of ES is well-fenced if there exists a augmentation ES' of ES such that ES' is fenced and C is a justified configuration of ES'.

To show that a configuration is well-fenced, the player must first choose a fencing. Then the inductive argument for well-justification proceeds as before.

For example, consider the following program, P_5.

```
   (acq; x=1; x=0; rel;)
|| (acq; r=x; rel;)                    (P5)
```



Whereas the second thread can read 1 in a well-justified configuration, this is not possible in a well-fenced configuration. There are two possible fencings. One includes the augmentation $57 \leq 52$, and the other includes $58 \leq 51$ and $59 \leq 51$. In either case, 54 can be justified, but 55 cannot. Therefore, there is no well-fenced configuration that includes 55.

Note that any sequential memory event structure is fenced. It follows that any sequential augmentation of a memory event structure is a fenced augmentation of it, and hence that any sequentially consistent configuration is a well-fenced configuration.

Now, if every justification is synchronized (that is $J = K$) we have that every well-fenced configuration is sequentially consistent, but in general this is not true. In particular, if there is no synchronization (that is $K = \emptyset$) then every well-justified configuration is well-fenced.

Fortunately, the proof of the DRF Theorem for well-fenced configurations follows directly from the DRF theorem for well-justified configurations: we just use DRF on each fencing.

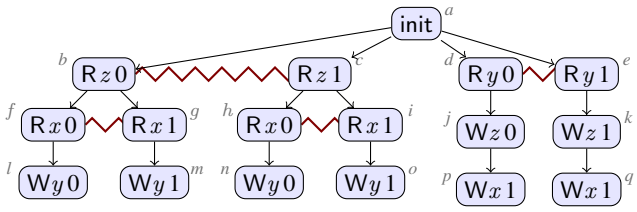
Theorem 7.1 (DRF). In any commutative read-enabled memory event structure, if all sequentially consistent configurations are data-race-free, then all well-fenced configurations are sequentially consistent.

PROOF. Follows directly from Theorem 5.1. □

8. Limitations

As noted at the end of §4, our semantics agrees with sixteen test cases from [Pugh 2004] and disagrees with three: TC3, TC7 and TC11. We now discuss TC7, which best elucidates the issues.

```
(r=z; y=x;) || (z=y; x=1;)        (TC7)
```

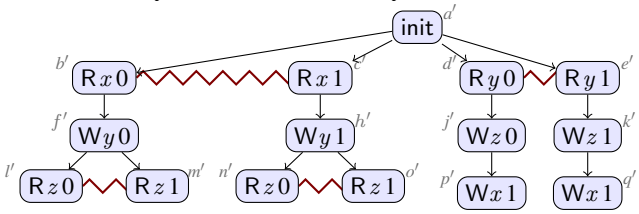


The question is whether all of the reads can be resolved to 1. This fails under our semantics: there is no well-justified configuration that includes events c, e and i, all of which read 1. In order to well-justify such a configuration, one must first resolve the conflict on x, then y and finally z. But this strategy fails immediately after resolving x.

Starting from the empty configuration, all acyclically justified configurations can be extended to include either p or q, and thus the player can select a configuration that includes either g or i. Suppose the player selects i. Since configurations are downclosed, the player must also select c; however c is not acyclically justified when the opponent selects p. Symmetrically, b is not acyclically justified when the opponent selects q. Thus the player cannot resolve the conflict on y.

The failure of TC7 indicates a failure to validate the reordering of independent reads. To see this, consider the program in which the first thread is rewritten.

```
(y=x; r=z;) || (z=y; x=1;)
```



In this case, the player can choose c', then e', then o', as required.

We now sketch a proposal to address this issue. On sets of events, define C is *compatible* with D whenever there is no $c \in C$ and $d \in D$ such that $c \#_\mu d$. Define C is *consistent* whenever C is compatible with C.

Proposal. Modify Definition 4.2 and Definition 4.3 to range over consistent sets, rather than configurations.

A configuration C is *alt-well-justified* whenever C is justified and there exists a consistent set D such that $\emptyset \sqsubseteq^* D \supseteq C$. □

By this proposal, the definition of alt-AE-justifies is as follows: On consistent sets, C *alt-AE-justifies* D whenever for any $C \lesssim^* C'$ there exists a $C' \lesssim^* C''$ such that C'' justifies D.

The configuration of TC7 that always reads 1 is alt-well-justified. Starting from the empty set, $C_1 = \{a, g, i\}$ is alt-AE-justified, since every consistent set that extends $\emptyset$ (via $\lesssim^*$) can be further extended to include either p or q, both labeled $(\mathsf{W}x\,1)$. Although g and i are in conflict, they are not in primitive conflict, and therefore may both be included in a consistent set—this is the key difference between well- and alt-well-justification. From C_1, $C_2 = \{a, g, i, e\}$ is alt-AE-justified, since we can always extend to include either m or o, both labeled $(\mathsf{W}y\,1)$. From C_2, $C_3 = \{a, g, i, e, c\}$ is alt-AE-justified, since we can always extend to include k, labeled $(\mathsf{W}z\,1)$. Thus $\emptyset \sqsubseteq^* C_3$ and therefore, since writes do not require justification, $\emptyset \sqsubseteq^* \{a, g, m, i, o, e, k, q, c\} \supseteq \{a, i, o, e, k, q, c\}$, as required.

By hand calculation, alt-well-justification agrees with all nineteen test cases; therefore, the definition looks quite promising. The question of whether DRF holds for alt-well-justification remains open. The inductive structure of the proof of DRF relies on the fact that configurations are

down closed. Since consistent sets are not necessarily down closed, the proof strategy fails.

9. Open problems

This paper has introduced the first model for relaxed memory based event structures. This fresh approach creates many opportunities for future work.

Our model does not enforce *coherence*: that writes on a single variable appear to occur in some global order. For example, in a coherent semantics for `(x=1;) || (x=2;) || (r1=x; r2=x; r3=x;)` it is not possible to have $\mathtt{r}_1 = \mathtt{r}_3 \neq \mathtt{r}_2$, whereas our semantics allows this. To enforce coherence, it appears to be necessary to distinguish the causal order from the order used to determine visibility.

We have modeled synchronization using a restricted form of locks. A coherent semantics is required to model Java's volatile variables, and would also allow a more satisfactory treatment of locks. In the formalization of §7, release and acquire are both read/write actions. This guarantees that, for example, a single release does not enable two parallel acquires. In the standard treatment of locks, assuming coherence, release is a write, and acquire is a read; thus, release justifies acquire, but acquire does not justify release. The order between acquire and release is usually guaranteed by thread order, which we have assumed. This assumption guarantees that the order we have required from acquire to release is redundant. With a coherent semantics for locks, the causal order from acquire to release can be dropped.

Separate from the concerns of §8, variations on the definition of well-justification may be worth exploring. For example, we define of $\sqsubseteq^*$ in terms of $\lesssim^*$: when exploring extensions of the current configuration, both player and opponent are restricted to using acyclic justification. It is natural to ask about a further definition, which uses $\sqsubseteq^*$ in place of $\lesssim^*$. If we let $\sqsubseteq^*_0 = \lesssim^*$ and $\sqsubseteq^*_1 = \sqsubseteq^*$, we can see this as hierarchy where each $\sqsubseteq^*_i$ uses $\sqsubseteq^*_{i-1}$. It is not the case that $\sqsubseteq^*_i$ contains $\sqsubseteq^*_{i-1}$, since the definition uses $\sqsubseteq^*_{i-1}$ in both positive and negative position: positive for the player, and negative for the opponent. The redundant-read-elimination counterexample of [Ševčík 2008, §5.3.2] and TC18 of [Pugh 2004] are both interesting in this regard. If the opponent is allowed to pick out of thin air, then the player is unable to well-justify the desired configuration. Well-justification becomes possible, however, if the opponent is restricted to acyclically chosen configurations.

We have investigated a very simple program logic, which establishes a restricted form of safety. It would be interesting to investigate more powerful logics, such as that of Turon et al. [2014]. One of the primary purposes of a memory model is to support program transformation. To this end, it would be useful to have a refinement relation over memory event structures that preserves well-justification.

Our approach to type safety is novel, in that we have not required a static association between variables and types as in prior work [Lochbihler 2013; Goto et al. 2012]. It would be interesting to extend our approach to model dynamic allocation and deallocation.

Acknowledgement. We thank the anonymous referees for their suggestions, and the members of the JMM working group for useful discussions.

References

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. Principals of Programming Languages*, pages 55–66, 2011.

M. Batty, K. Memarian, K. Nienhuis, J. Pichon-Pharabod, and P. Sewell. The problem of programming language concurrency semantics. In *Proc. European Symp. on Programming*, pages 283–307, 2015.

S. Castellan. Weak memory models using event structures. In *Journées Francophones des Langages Applicatifs*, 2016.

P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proc. European Symp. on Programming*, pages 331–346, 2007.

M. Goto, R. Jagadeesan, C. Pitcher, and J. Riely. Types for relaxed memory models. In *Proc. Types in Language Design and Implementation*, pages 25–37, 2012.

R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *Proc. European Symp. on Programming*, 2010.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

A. Lochbihler. Making the java memory model safe. *ACM Trans. Program. Lang. Syst.*, 35(4):12, 2013.

J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Proc. Principals of Programming Languages*, pages 378–391, 2005.

M. Nielsen, G. D. Plotkin, and G. Winskel. Petri nets, event structures and domains. In *Proc. Int. Symp. Semantics of Concurrent Computation*, pages 266–284, 1979.

J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proc. Principals of Programming Languages*, 2016. To appear.

W. Pugh. Causality test cases, 2004.

V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *Proc. Principles and Practice of Parallel Programming*, pages 161–172, 2007.

A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, pages 691–707, 2014.

J. Ševčík. *Program Transformations in Weak Memory Models*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 2008.

G. Winskel. Event structures. In *Advances in Petri Nets*, pages 325–392, 1986.

G. Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 364–397, 1989.