Operational Semantics of a Weak Memory Model with Channel Synchronization

Daniel Schnetzer Fava,[1] Martin Steffen[1] and Volker Stolz[1,2]

[1] Dept. of Informatics, University of Oslo
[2] Western Norway University of Applied Sciences

Abstract. A multitude of weak memory models exists supporting various types of relaxations and different synchronization primitives. On one hand, such models must be lax enough to account for hardware and compiler optimizations; on the other, the more lax the model, the harder it is to understand and program for. Though the right balance is up for debate, a memory model should provide what is known as the *SC-DRF guarantee*, meaning that data-race free programs behave in a sequentially consistent manner.
We present a weak memory model for a calculus inspired by the Go programming language. Thus, different from previous approaches, we focus on a memory model with buffered channel communication as the sole synchronization primitive. We formalize our model via an operational semantics, which allows us to prove the SC-DRF guarantee using a standard simulation technique. Contrasting against an axiomatic semantics, where the notion of a program is abstracted away as a graph with memory events as nodes, we believe our semantics and simulation proof can be clearer and easier to understand. Finally, we provide a concrete implementation in $\mathbb{K}$, a rewrite-based executable semantic framework, and derive an interpreter for the proposed language.

1 Introduction

A *memory model* dictates which values may be observed when reading from memory, thereby affecting how concurrent processes communicate through shared memory. One of the simplest memory models, called *sequentially consistent*, stipulates that operations must appear to execute one at a time and in program order [25]. SC was one of the first formalizations to be proposed and, to this day, constitutes a baseline for well-behaved memory. However, for efficiency reasons, modern hardware architectures do not guarantee sequential consistency. SC is also considered much too strong to serve as the underlying memory semantics of programming languages; the reason being that sequential consistency prevents many established compiler optimizations and robs from the compiler writer the chance to exploit the underlying hardware for efficient parallel execution. The research community, however, has not been able to agree on exactly what a proper memory model should offer. Consequently, a bewildering array of *weak* or *relaxed memory models* have been proposed, investigated, and implemented. Different taxonomies and catalogs of so-called *litmus tests*, which highlight specific aspects of memory models, have also been researched [1].

Memory models are often defined axiomatically, meaning via a set of rules that constrain the order in which memory events are allowed to occur. The *candidate execution*

approach falls in this category [6]. These formalizations are not without controversy. For example, despite many attempts, there does not exist an well-accepted comprehensive specification of the C^{++}11 [7, 8] or Java memory models [5, 27, 34]. Luckily, more recently, one fundamental principle of relaxed memory has emerged, namely: no matter how much relaxation is permitted by a memory model, if a program is *data-race free* or *properly synchronized*, then the program must behave in a sequentially consistent manner [2, 27]. This is known as the *SC-DRF* guarantee.

We present an *operational* semantics for a weak memory. Similar to Boudol and Petri [10], we favor an operational semantics because it allows us to prove the SC-DRF guarantee using a standard simulation technique. Compared to axiomatic semantics in which the notion of a program is abstracted away (often in the form of a graph with nodes as memory events), we think that our formalism leads to an easier to understand proof of the SC-DRF guarantee. The lemmas we build up in the process of constructing the proof highlight meaningful invariants and give insight into the workings of the memory model.

Our calculus is inspired by the Go programming language: similar to Go, our model focuses on channel communication as the main synchronization primitive. Go's memory model, however, is described, albeit succinctly and precisely, in prose [18]. we provide a formal semantics instead.

The main contributions of our work are:

- There are few studies on channel communication as synchronization primitive for weak memory. We give an operational theory for a weak memory with bounded channel communication by leveraging thread-local happens-before information.
- We prove that the proposed memory upholds the *sequential consistency guarantee for data-race free* programs using a standard conditional simulation proof.
- We implement the operational semantics in the $\mathbb{K}$ executable semantics framework [22, 35] and make the source code publicly available via a git-repository [14].

The remaining of the paper is organized as follows. Section 2 presents background information directly related to the formalization of our memory model. Sections 3 and 4 provide the syntax and the semantics of the calculus with relaxed memory and channel communication. Section 6 establishes the SC-DRF guarantee. Sections 7 and 8 conclude with related and future work.

2 Background

Go's memory model. The Go language [17, 13] recently gained traction in networking applications, web servers, distributed software and the like. It prominently features goroutines, that is, asynchronous execution of function calls resembling lightweight threads, and buffered channel communication in the tradition of CSP [20] (resp. the π-calculus [30]) or Occam [21]. While encouraging message passing as the prime mechanism for communication and synchronization, threads can still exchange data via shared variables. Consequently, Go's specification includes a memory model which spells out, in precise but informal English, the few rules governing memory interaction at the language level [18].

Concerning synchronization primitives, the model covers goroutine creation and destruction, channel communication, locks, and the `once`-statement. Our semantics will concentrate on thread creation and channel communication because lock-handling and the `once` statement are *not* language primitives but part of the `sync`-library. Thread destruction, i.e. termination, comes with *no* guarantees concerning visibility: it involves no synchronization and thus the semantics does not treat thread termination in any special way. In that sense, our semantics treats all of the *primitives* covered by Go's memory model specification. As will become clear in the next sections, our semantics does not, however, relax read events. Therefore, our memory model is stronger than Go's. On the plus side, this prevents a class of undesirable behavior called *out-of-thin-air* [9]. On the negative, the absence of relaxed reads comes at the expense of some forms of compiler optimizations.

Languages like Java and C^{++} go to great lengths not only to offer the crucial SC-DRF guarantee for well-synchronized programs, but beyond that, strive to clarify the resulting non-SC behavior when the program is *ill*-synchronized. This involves ruling out definitely unwelcome behavior. Doing this precisely, however, is far from trivial. One class of unwanted behavior that is particularly troublesome is the so called *out-of-thin-air* behavior [9]. In contrast, Go's memory model is rather "laid back." Its specification [18] does not even mention "out-of-thin-air" behavior.

Happens-before relation and observability. Like Java's [27, 34], C^{++}11's [7, 8], and many other memory models, ours centers around the definition of a *happens-before* relation. The concept dates back to 1978 [24] and was introduced in a pure *message-passing* setting, i.e., without shared variables. The relation is a technical vehicle for defining the semantics of memory models. It is important to note that just because an instruction or event is in a *happens-before* relation with a second one, it does not necessarily mean that the first instruction *actually* "happens" before the second in the operational semantics. Consider the sequence of assignments $x := 1; y := 2$ as an example. The first assignment "happens-before" the second as they are in program order, but it does not mean the first instruction is actually "done" before the second,[3] and especially, it does not mean that the effect of the two writes become observable in the given order. For example, a compiler might choose to change the order of the two instructions. Alternatively, a processor may rearrange memory instructions so that their effect may not be visible in program order. Conversely, the fact that two events happen to occur one after the other in a particular schedule does not imply that they are in happens-before relationship, as the order may be coincidental. To avoid confusion between the technical happens-before relation and our understanding of what happens when the programs runs, we speak of event e_1 "happens-before" e_2 in reference to the technical definition (also abbreviated as $e_1 \rightarrow_{\mathsf{hb}} e_2$ in this section) as opposed to its natural language interpretation. Also, when speaking about steps and events in the operational semantics, we avoid talking about something happening before something else, and rather say that a step or transition "occurs" in a particular order.

[3] Assuming that x and y are not aliases in the sense that they refer to the same or "overlapping" memory locations.

Listing (1.1) Erroneous synchronization

```
1  var a string
2  var done bool
3
4  func setup() {
5    a = "hello, world"
6    done = true
7  }
8
9  func main() {
10   go setup()
11   for !done {  } // try waiting
12   print(a)
13 }
```

Listing (1.2) Channel synchronization

```
var a string
var c = make(chan int, 10)

func setup() {
  a = "hello, world"
  c <- 0   // send
}

func main() {
  go setup()
  <-c      // receive
  print(a)
}
```

Fig. 1: Synchronization via channel communication [18]

The happens-before relation regulates observability, and it does so very liberally. It allows a read r from a shared variable to *possibly observe* a particular write w to said variable *unless* one of the following two conditions hold:

$$r \rightarrow_{\mathsf{hb}} w \qquad \text{or} \tag{1}$$

$$w \rightarrow_{\mathsf{hb}} w' \rightarrow_{\mathsf{hb}} r \qquad \text{for some other write } w' \text{ to the same variable.} \tag{2}$$

For the sake of discussion, let's concentrate on the following two constituents for the happens-before relation: 1) *program order* and 2) the order stemming from channel communication. According to the Go memory model [18], we have the following constraints related to a channel c with capacity k:

A send on c happens-before the corresponding receive from c completes. (3)

The i^{th} receive from c happens-before the $(i+k)^{th}$ send on c. (4)

To illustrate, consider the example on Listing 1.1. The main function spawns an asynchronous execution of `setup`, at which point `main` and `setup` can run concurrently. In the thread or goroutine executing `setup`, the write to variable `a` happens-before the write to `done`, as they are in program order. For the same reason, the read(s) of `done` happen-before the read of `a` in the main thread. Without synchronization, the variable accesses are ordered locally *per thread* but not across threads. Since neither condition (1) or (2) applies, the main procedure may or may not observe writes performed by `setup`; it is possible for main to observe the initial value of `a` instead. This makes the writes to `a` and `done` performed by `setup` to potentially appear out-of-order from the main thread's perspective.

Replacing the use of `done` by channel synchronization properly synchronizes the two functions (cf. Listing 1.2). As the receive happens-after the send, an order is established between events belonging to the two threads. One can think of the main thread as receiving not only a value but also the knowledge that the write event to `a` in `setup` has taken place. With condition (3), channels implicitly communicate the happens-before relation from the sender to the receiver. Then, with condition (2), we can conclude that

once main receives a message from `setup`, the initial value of `a` is no longer observable from main's perspective.

Condition (4) is not shown in the example. This condition accounts for the boundedness of channels by transmitting happens-before information in the backward direction for some receiver to some sender. For *synchronous* channels, where $k = 0$, the two threads participating in the rendezvous symmetrically exchange their happens-before information.

In summary, the operational semantics captures the following principles:

Immediate positive information: a *write* is globally observable instantaneously.

Delayed negative information: in contrast, negative information overwriting previously observable *writes* is *not* immediately effective. Referring back to the example of Figure 1, the fact that `setup` has overwritten the initial value of variable `a` is not immediately available to other threads. Instead, the information is spread via message passing in the following way:

Causality: information regarding condition (3) travels with data through channels.

Channel capacity: *backward channels* are used to account for condition (4).

Local view: Each thread maintains a local view on the happens-before relationship of past write events, i.e. which events are unobservable. Thus, the semantics does not offer multi-copy atomicity.

3 Abstract syntax

The abstract syntax of the calculus is given in Table 1. *Values* v can be of two forms: r is used to denote the value of local variables or registers, while n in used to denote references or names in general and, in specific, c for channel names. We do not explicitly list values such as the unit value, booleans, integers, etc. We also omit compound local expressions like $r_1 + r_2$. Shared variables are denoted by x, z etc, `load` z represents the reading the shared variable z into the thread, and $z := v$ denotes writing to z.

References are dynamically created and are, therefore, part of the *run-time* syntax. Run-time syntax is highlighted with an underline as $\underline{n}$ in the grammar. A new channel is created by `make` (`chan` T, v) where T represents the type of values carried by the channel and v a non-negative integer specifying the channel's capacity. Sending a value over a channel and receiving a value as input from a channel are written respectively as $v_1 \leftarrow v_2$ and $\leftarrow v$. After the operation `close`, no further values can be sent on the specified channel. Attempting to send values on a closed channel leads to a panic.

Starting a new asynchronous activity, called goroutine in Go, is done using the `go`-keyword. In Go, the `go`-statement is applied to function calls only. We omit function calls, asynchronous or otherwise, since they are orthogonal to the memory model's formalization. See Steffen [36] for an operational semantics dealing with goroutines and closures in a purely functional setting, that is, without shared memory.

The select-statement, here written using the $\sum$-symbol, consists of a finite set of branches which are called communication clauses by the Go specification [17]. These branches act as guarded threads. General expressions in Go can serve as guards. Our calculus, however, imposes the restriction that only communication statements (i.e., channel sending and receiving) and the `default`-keyword can serve as guards. This

$v ::= r \mid \underline{n}$	values
$e ::= t \mid v \mid \texttt{load}\, z \mid z := v \mid \texttt{if}\, v\, \texttt{then}\, t\, \texttt{else}\, t \mid \texttt{go}\, t$	expressions
$\quad \mid \texttt{make}\,(\texttt{chan}\, T, v) \mid \leftarrow v \mid v \leftarrow v \mid \texttt{close}\, v$	
$g ::= v \leftarrow v \mid \leftarrow v \mid \texttt{default}$	guards
$t ::= \texttt{let}\, r = e\, \texttt{in}\, t \mid \sum_i \texttt{let}\, r_i = g_i\, \texttt{in}\, t_i$	threads

Table 1: Abstract syntax

restriction is in line with the A-normal form representation and does not impose any actual reduction in expressivity. Both in Go and in our formalization, there is at most one branch guarded by `default` in each select-statement. The same channel can be mentioned in more than one guard. "Mixed choices" [31, 32] are also allowed, meaning that sending and receiving guards can both be used in the same select-statement. We use `stop` as syntactic sugar for the empty select statement; it represents a permanently blocked thread. The `stop`-thread is also the only way to syntactically "terminate" a thread, meaning that it is the only element of t without syntactic sub-terms.

The `let`-construct $\texttt{let}\, r = e\, \texttt{in}\, t$ combines sequential composition and the use of scopes for local variables r: after evaluating e, the rest t is evaluated where the resulting value of e is handed over using r. The let-construct is seen as a binder for variable r in t. When r does not occur free in t, `let` then boils down to *sequential composition* and, therefore, is replaced by a semicolon.

4 Operational semantics

In this section we define the operational semantics of the calculus. We fix the run-time configurations of a program before giving the operational rules in Section 4.2.

4.1 Local states, events, and configurations

Let X represent a set of shared variables such as x, $z\ldots$ and let N represent an infinite set of names or identifiers with typical elements n, $n_2'\ldots$ As mentioned earlier, for readability, we will use names like c, $c_1,\ldots$ for channels, and p, $p_1'\ \ldots$ for goroutines or processes. A run-time configuration is then given by the following syntax:

$$P ::= n\langle\sigma, t\rangle \mid n(\!|z{:=}v|\!) \mid n[q] \mid \bullet \mid P \parallel P \mid \nu n\, P\,. \tag{5}$$

Configurations consist of the parallel composition of goroutines $p\langle\sigma,t\rangle$, write events $n(\!|z{:=}v|\!)$, and channels $c[q]$; $\bullet$ represents the empty configuration. The ν-binder, known from the π-calculus, indicates dynamic scoping [30]. Goroutines or processes $p\langle\sigma,t\rangle$ contain, besides the code t to be executed, a local view $\sigma = (E_{hb}, E_s)$ detailing the observability of write events from the perspective of p. Local observability is formulated "negatively," meaning that all write events are observable by default. It is possible for an event to no longer be visible from a thread's perspective; such events are called

$\texttt{let}\, x = v\, \texttt{in}\, t \rightsquigarrow t[v/x]$ R-RED

$\texttt{let}\, x_1 = (\texttt{let}\, x_2 = e\, \texttt{in}\, t_1)\, \texttt{in}\, t_2 \rightsquigarrow \texttt{let}\, x_2 = e\, \texttt{in}\, (\texttt{let}\, x_1 = t_1\, \texttt{in}\, t_2)$ R-LET

$\texttt{if true then}\, t_1\, \texttt{else}\, t_2 \rightsquigarrow t_1$ R-COND$_1$ $\quad$ $\texttt{if false then}\, t_1\, \texttt{else}\, t_2 \rightsquigarrow t_2$ R-COND$_2$

Table 2: Operational semantics: Local steps

shadowed and are tracked in σ, specifically in E_s. Note that, in order to properly update the list of shadowed events, σ must also contain thread-local information about the "happens-before" relationship between write events. This information is kept in E_{hb}.

Definition 1 (Local state). *A* local state σ *is a tuple of type* $2^{(N\times X)} \times 2^N$. *We use the notation* (E_{hb}, E_s) *to refer to the tuples and abbreviate their type by* Σ. *Let's furthermore denote by* $E_{hb}(z)$ *the set* $\{n \mid (n,z) \in E_{hb}\}$. *We write* $\sigma_\perp$ *for the local state* $(\emptyset,\emptyset)$ *containing neither happens-before nor shadow information.*

4.2 Reduction steps

The operational semantics is given in several stages. We start with local steps, that is, steps not involving shared variables.

4.2.1 Local steps The reduction steps are given modulo structural congruence $\equiv$ on configurations. The congruence rules are standard and thus omitted here (see the report [15] for details). Local steps $\rightsquigarrow$ (cf. Table 2) reduce a thread t without touching shared variables, and $t_1 \rightsquigarrow t_2$ implies $\langle \sigma, t_1 \rangle \rightarrow \langle \sigma, t_2 \rangle$.

4.2.2 Global steps Writing a value records the corresponding event $n(\!|z{:=}v|\!)$ in the global configuration, with n freshly generated (cf. rule R-WRITE). The write events are remembered without keeping track of the order of their issuance. Therefore, as far as the global configuration is concerned, no write event ever invalidates an "earlier" write event or overwrites a previous value in a shared variable. Instead, the global configuration accumulates the "positive" information about all available write events which potentially can be observed by reading from shared memory.

The *local* state σ of a goroutine captures which events are actually observable from a thread-local perspective. Its primary function is to contain "negative" information: A read can observe all write events *except* for those shadowed, that is, write events whose identifiers are contained in E_s (see rule R-READ). In addition, the local state keeps track of write events that are thread-locally known to have *happened-before*. These are stored in E_{hb}. So, issuing a write command (rule R-WRITE) with a write event labeled n updates the local E_{hb} by adding (n,z). Additionally, it marks all previous writes to the variable z as shadowed, thus enlarging E_s.

$$\frac{\sigma = (E_{hb}, E_s) \qquad \sigma' = (E_{hb} + (n,z), E_s + E_{hb}(z)) \qquad \mathit{fresh}(n)}{p\langle \sigma, z := v; t\rangle \;\to\; \nu n\, (p\langle \sigma', t\rangle \parallel n(\!|z{:=}v|\!))}\ \text{R-WRITE}$$

$$\frac{\sigma = (\_, E_s) \qquad n \notin E_s}{p\langle \sigma, \mathtt{let}\ r = \mathtt{load}\ z\ \mathtt{in}\ t\rangle \parallel n(\!|z{:=}v|\!) \;\to\; p\langle \sigma, \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle \parallel n(\!|z{:=}v|\!)}\ \text{R-READ}$$

$$\frac{q = [\sigma_\perp, \ldots, \sigma_\perp] \qquad |q| = v \qquad \mathit{fresh}(c)}{p\langle \sigma, \mathtt{let}\ r = \mathtt{make}\ (\mathtt{chan}\ T, v)\ \mathtt{in}\ t\rangle \;\to\; \nu c\, (p\langle \sigma, \mathtt{let}\ r = c\ \mathtt{in}\ t\rangle \parallel c_f[] \parallel c_b[q])}\ \text{R-MAKE}$$

$$\frac{\neg \mathit{closed}(c_f[q_2]) \qquad \sigma' = \sigma + \sigma''}{c_b[q_1 :: \sigma''] \parallel p\langle \sigma, c \leftarrow v; t\rangle \parallel c_f[q_2] \;\to\; c_b[q_1] \parallel p\langle \sigma', t\rangle \parallel c_f[(v,\sigma) :: q_2]}\ \text{R-SEND}$$

$$\frac{v \neq \perp \qquad \sigma' = \sigma + \sigma''}{\begin{array}{l} c_b[q_1] \parallel p\langle \sigma, \mathtt{let}\ r = \leftarrow c\ \mathtt{in}\ t\rangle \parallel c_f[q_2 :: (v,\sigma'')] \;\to \\ c_b[\sigma :: q_1] \parallel \quad p\langle \sigma', \mathtt{let}\ r = v\ \mathtt{in}\ t\rangle \quad \parallel c_f[q_2] \end{array}}\ \text{R-REC}$$

$$\frac{\sigma' = \sigma + \sigma''}{p\langle \sigma, \mathtt{let}\ r = \leftarrow c\ \mathtt{in}\ t\rangle \parallel c_f[(\perp,\sigma'')] \;\to\; p\langle \sigma', \mathtt{let}\ r = \perp\ \mathtt{in}\ t\rangle \parallel c_f[(\perp,\sigma'')]}\ \text{R-REC}_\perp$$

$$\frac{\sigma' = \sigma_1 + \sigma_2}{\begin{array}{l} c_b[] \parallel p_1\langle \sigma_1, c \leftarrow v; t\rangle \parallel p_2\langle \sigma_2, \mathtt{let}\ r = \leftarrow c\ \mathtt{in}\ t_2\rangle \parallel c_f[] \;\to \\ c_b[] \parallel \quad p_1\langle \sigma', t\rangle \quad \parallel p_2\langle \sigma', \mathtt{let}\ r = v\ \mathtt{in}\ t_2\rangle \quad \parallel c_f[] \end{array}}\ \text{R-SEND-REC}$$

$$\frac{\neg \mathit{closed}(c_f[q])}{p\langle \sigma, \mathtt{close}\ (c); t\rangle \parallel c_f[q] \;\to\; p\langle \sigma, t\rangle \parallel c_f[(\perp,\sigma) :: q]}\ \text{R-CLOSE}$$

$$\frac{\mathit{fresh}(p_2)}{p_1\langle \sigma, \mathtt{go}\ t'; t\rangle \;\to\; \nu p_2\, (p_1\langle \sigma, t\rangle \parallel p_2\langle \sigma, t'\rangle)}\ \text{R-GO}$$

Table 3: Operational semantics: Global steps

Channels in Go are the primary mechanism for communication and synchronization. They are typed and assure FIFO communication from a sender to a receiver sharing a channel. Channels can be dynamically created and closed. Channels are *bounded,* i.e., each channel has a finite capacity fixed upon creation. Channels of capacity 0 are called *synchronous.* Our semantics largely ignores that channel values are typed and that only values of an appropriate type can be communicated over a given channel.

Definition 2 (Channels). *A channel is of the form $c[q_1, q_2]$, where c is a name and (q_1, q_2) a pair of queues. The first queue, q_1, contains elements of type $(\mathit{Val} \times \Sigma) + (\{\perp\} \times \Sigma)$, where $\perp$ is a distinct, separate value representing the "end-of-transmission"; the second queue, q_2, contains elements of type Σ. We write (v, σ), $(\perp, \sigma)$ resp. (σ) for the respective queue values. The queues are also referred to as* forward *resp.* backward *queue. Furthermore, we use the following notational convention: We write $c_f[q]$ to refer*

to the forward queue of the channel and $c_b[q]$ to the backward queue. We also speak of the forward channel and the backward channel. We write $[]$ for an empty queue, $e :: q$ for a queue with e as the element most recently added into q, and $q :: e$ for the queue where e is the element to be dequeued next. We denote with $|q|$ the number of elements in q. A channel is closed*, written closed$(c[q])$, if q is of the form $\perp :: q'$. Note that it is possible for a non-empty queue to be closed.*

When creating a channel (cf. rule R-MAKE) the forward direction is initially empty but the backward is not: it is initialized to a queue of length v corresponding to the channel's capacity. The backward queue contains *empty* happens-before and shadow information, represented by the elements $\sigma_\perp$. The rule R-MAKE covers both synchronous and asynchronous channels. An asynchronous channel is created with empty forward $c_f[]$ and backward queue $c_b[]$.

Channels can be closed, after which no new values can be sent. Values "on transit" in a channel when it is being closed are *not* discarded and can be received as normal. The special value $\perp$ indicates the end-of-transmission. Note that there is a difference between an empty open channel $c[]$ and an empty closed one $c[\perp]$. The value $\perp$ is relevant to the forward channel only. Rules R-SEND and R-REC govern asynchronous channel communication while R-SEND-REC implements synchronous communication. In an asynchronous send, a process places a value on the forward channel along with its local state (provided the channel is not full, i.e., the backward queue is non-empty). In the process of sending, the sender's local state is updated with the knowledge that the previous k^{th} receive has completed; this is captured by $\sigma' = \sigma + \sigma''$ in the R-SEND rule. To receive a value from a (non-empty) asynchronous channel (cf. rule R-REC), the communicated value v is stored locally (in the rule, ultimately in variable r). Additionally, the local state of the receiver is updated by adding the previously sent local-state information. Furthermore, the state of the receiver before the update is sent back via the backward channel. In synchronous communication, the receiver obtains a value from the sender and together they exchange local state information. The R-CLOSE rule closes both sync and async channels. Executing a receive on a *closed* channel results in receiving the end-of-transmission marker $\perp$ (cf. rule R-REC$_\perp$) and updating the local state σ in the same way as when receiving a properly sent value. The "value" $\perp$ is not removed from the queue, so that all clients attempting to receive from the closed channel obtain the communicated happens-before synchronization information. Furthermore, there is no need to communicate happens-before constraints from the receiver to a potential future sender on the closed channel: after all, the channel is closed. Closing a channel resembles sending the special end-of-transmission value $\perp$ (cf. rule R-CLOSE). An already closed channel cannot be closed again. In Go, such an attempt would raise a panic. Here, this is captured by the absence of enabled transitions.

Thread creation leads to a form of a synchronization where the spawned goroutine inherits the local state of the parent (cf. rule R-GO). Finally, rules dealing with the select statement are given in the accompanying report [15].

Starting from an *initial weak configuration*, as far as the sizes of the queues of a channel in connection with the channel's capacity are concerned, the semantics assures the following invariant.

Definition 3 (Initial weak configuration). *An* initial *weak configuration is of the form* $\nu\vec{n}\ (\langle\sigma_0, t_0\rangle \parallel n_0(\!|z_0{:=}v_1|\!) \parallel \ldots \parallel n_k(\!|z_k{:=}v_k|\!))$ *where* $z_0, \ldots z_k$ *are all shared variables of the program,* $\vec{n}$ *represents* $n_0, \ldots, n_k$*, and* $\sigma_0 = (E^0_{hb}, E^0_s)$ *where* $E^0_{hb} = \{(n_0, z_0), \ldots, (n_k, z_k)\}$ *and* $E^0_s = \emptyset$*.*

Lemma 4 (Invariant for channel queues). *The following global invariant holds for a channel c created with capacity k:* $|q_f| + |q_b| - p = k$*.* □

5 Strong semantics

The strong semantics can be seen as a simpler version of the weak one. It represents a standard interleaving semantics, i.e., write and reads immediately interact with a shared global state. Therefore, there is no need for local thread information σ.

$$S ::= p\langle t\rangle \ \mid\ (\!|z{:=}v|\!) \ \mid\ \bullet \ \mid\ S \parallel S \ \mid\ n[q] \ \mid\ \nu n\ P\ . \tag{6}$$

Structural congruence $\equiv$ and the local transition steps $\rightsquigarrow$ remain unchanged (cf. Table 2). Apart from leaving out the events and other information, the only rules that conceptually change are the ones for read and write. These are included on Table 4.

$$\frac{}{p\langle z := v; t\rangle \parallel (\!|z{:=}v'|\!) \rightarrow p\langle t\rangle \parallel (\!|z{:=}v|\!)}\ \text{R-WRITE}$$

$$\frac{}{p\langle \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel (\!|z{:=}v|\!) \rightarrow p\langle \texttt{let}\ r = \ v\ \texttt{in}\ t\rangle \parallel (\!|z{:=}v|\!)}\ \text{R-READ}$$

Table 4: Strong operational semantics: read and write steps

Definition 5 (Initial configuration). *Initially, a strong configuration is of the form* $p\langle t_0\rangle \parallel (\!|z_0{:=}v_1|\!) \parallel \ldots \parallel (\!|z_k{:=}v_k|\!)$*, where* $z_0, \ldots z_k$ *are all shared variables of the program and* t_0 *contains no run-time syntax.*

Cf. also the "weak" version from Definition 3.

Definition 6 (Well-formed strong configuration). *An strong configuration S is* well-formed *if, for every variable* $z \in V_s$*, there exists exactly one write event* $(\!|z{:=}v|\!)$ *in S. We write* $\vdash_s S : ok$ *for such well-formed configurations.*

6 Relating the strong and the weak semantics

Let's recall the definition of simulation [29] relating states of labeled transition systems. The set of transition labels and the information carried by the labels may depend on the

specific steps or transitions done by a program and/or the observations one wishes to attach to those steps. This leads to a distinction between internally and externally visible steps. Let's write α for arbitrary transition labels. Later we will use a for visible labels and τ as the label of invisible or internal steps.

Definition 7 (Simulation). *Assume two labeled transition systems over the same set of labels and with state sets S and T. A binary relation $\mathcal{R} \subseteq S \times T$ is a simulation relation between the two transition systems if $s_1 \xrightarrow{\alpha} s_2$ and $s_1 \mathcal{R} t_1$ implies $t_1 \xrightarrow{\alpha} t_2$ for some state t_2. A state t* simulates *s, written $t \gtrsim s$, if there exists a simulation relation $\mathcal{R}$ such that $s \mathcal{R} t$.*

We use formulations like "s is simulated by t" interchangeably, and $\lesssim$ as the corresponding symbol. Also, we subscript the operational rules for disambiguation; for example, R-READ$_s$ refers to the strong version of the read while R-WRITE$_w$ to the weak version of the write operation. The rules of the strong semantics are simplifications of the weak rules given in Section 4.

The operational semantics is given as unlabeled global transitions $\rightarrow$. To establish the relationship between the strong and the weak semantics, we make the steps of the operational semantics more "informative" by labeling them appropriately: For read steps by rule R-READ$_s$ and R-READ$_w$, when reading a value v from a variable z, the corresponding step takes the form $\xrightarrow{(z?v)}$. All other steps, $\rightarrow$ as well as $\rightsquigarrow$ steps, are treated as invisible and noted as $\xrightarrow{\tau}$ in the simulation proofs. We make use of the following "alternative" labeling for the purpose of defining races and for some of the technical lemmas: we label write and read steps with the identity of the goroutine responsible for the action and the affected shared variable, i.e. $\xrightarrow{p_2(z!)}$ and $\xrightarrow{p_2(z?)}$. Note that the identity of the write event is omitted as well as the value exchanged; they will not be needed in the proofs. We often use subscripts when distinguishing the strong from the weak semantics; e.g. $\xrightarrow{p(z!)}_w$ and $\xrightarrow{p(z!)}_s$.

6.1 The strong semantics conditionally simulates the weak one

That the weak semantics "contains" the sequentially consistent strong one as special case, i.e., the weak semantics simulates the strong one, should be intuitively clear and expected. Equally clear is that the opposite direction —the strong semantics simulates the weak— does *not* hold in general. If a simulation relation would hold in both directions, the two semantics would be equivalent, thus obviating the whole point of a weak or relaxed memory model.

Simulation of the weak semantics by the strong one can only be guaranteed "conditionally." The standard condition is that the program is "well-synchronized." We take that notion to represent the absence of data races, where a data race is a situation in which two different threads have access to the same shared variable "simultaneously," with at least one of the accesses being a write.

Definition 8 (Data race). *A well-formed configuration S_i contains a* manifest data race *if $S_i \xrightarrow{p_1(z!)}_s$ and $S_i \xrightarrow{p_2(z!)}_s$ for some $p_1 \neq p_2$ (a manifest write-write race on z), or if*

$S_i \xrightarrow{p_1(z?)}_s$ *and* $S_i \xrightarrow{p_2(z!)}_s$ *(a manifest read-write race on z). We say a program S* has a data race *if a manifest data race is* reachable *from the initial configuration* S_0.

Definition 9 (Data race). *A well-formed configuration* S_i *contains a* manifest data race *if either hold:*

$$S_i \xrightarrow{p_1(z!)}_s \text{ and } S_i \xrightarrow{p_2(z!)}_s \text{ for some } p_1 \neq p_2 \qquad \text{(manifest write-write race on } z\text{)}$$

$$S_i \xrightarrow{p_1(z?)}_s \text{ and } S_i \xrightarrow{p_2(z!)}_s \qquad \text{(manifest read-write race on } z\text{)}$$

We say a program S has a data race *if a manifest data race is* reachable *from the initial configuration* S_0.

The definition is used analogously for the weak semantics. We also say a program is data-race free or *properly synchronized* if it does not have a data race.

Definition 10 (Observable and concurrent writes). *Let* W_P *stand for the set of all write events* $n(\!|z{:=}v|\!)$ *in a weak configuration P and let* $W_P(z)$ *stand for the set of identifiers of writes events to the variable z, i.e.* $W_P(z) = \{n \mid n(\!|z{:=}v|\!) \in W_P\}$. *Given a well-formed configuration P, the sets of writes that* happens-before*, that are* concurrent*, and that are* observable *by process p for a variable z are defined as follows:*

$$W_P^{\mathsf{hb}}(z@p) = E_{hb}(z@p) \tag{7}$$

$$W_P^{|||}(z@p) = W_P(z) \setminus E_{hb}(z@p) \tag{8}$$

$$W_P^{\circ}(z@p) = W_P(z) \setminus E_s(z@p)\ . \tag{9}$$

We also use notations like $W_P^{\circ}(\_@p)$ *to denote the set of observable write events in P for any shared variable.*

6.1.1 General invariant properties

Lemma 11 (Invariants about write events). *The weak semantics has the following invariants.*

1. *For all local states* (E_{hb}, E_s) *of all processes,* $E_s \subset E_{hb}(z)$.
2. $W_P^{|||}(z@p) \subseteq W_P^{\circ}(z@p)$.
3. $W_P^{|||}(z@p) \neq W_P^{\circ}(z@p)$.
4. $W_P^{\mathsf{hb}}(z@p) \cap W_P^{\circ}(z@p) \neq \emptyset$.

As $W_P^{\circ}(z@p)$ is a proper superset of $W_P^{|||}(z@p)$ (by part (2) and (3)), each thread can observe at least one value held by a variable. This means, unsurprisingly, that no thread will encounter an "undefined" variable. More interesting is the following generalization, namely that at each point and for each variable, some value is *jointly* observable by all processes. The property holds for arbitrary programs, race-free or not. Under the assumption of race-freedom, we will later obtain a stronger "consensus" result: not only is a consensus possible, but there is *exactly one* possible observable write, not more.

Lemma 12 (Consensus possible). *Weak configurations obey the following invariant*

$$\bigcap_{p \in P} W_P^{\circ}(z@p) \neq \emptyset\ . \tag{10}$$

6.1.2 Race-free reductions Next, we present invariants that hold specifically for race-free programs but not generally. They will be needed to define the relationship between the strong and weak semantics via a bisimulation relation. More concretely, the following properties are ultimately needed to establish that the relationship connecting the strong and the weak behavior of a program is well-defined.

Lemma 13 (No concurrent writes when it counts). *Assume $P_0 \rightarrow_w^* P$ where P_0 is the initial configuration derived from program P.*

1. *Assume P has no read-write race. If $P \xrightarrow{p(z?)}_w$, then $W_P^{|||}(z@p) = \emptyset$.*
2. *Assume P has no write-write race. If $P \xrightarrow{p(z!)}_w$, then $W_P^{|||}(z@p) = \emptyset$.*

Lemma 14 (Race-free consensus when it counts). *Assume $P_0 \rightarrow_w^* P$ with P_0 race-free. If $P \xrightarrow{p(z?)}_w$ or $P \xrightarrow{p(z!)}_w$, then*

$$\bigcap_{p_i} W_P^o(z@p_i) = \{n\} \ , \tag{11}$$

where the intersection ranges over an arbitrary set of processes which includes p.

Lemma 15 (Race-free consensus). *Weak configurations for race-free programs obey the following invariant*

$$\bigcap_{p_i \in P} W_P^o(z@p_i) = \{n\} \ . \tag{12}$$

Definition 16 (Well-formedness for race-free programs). *A weak configuration P is* well-formed *if*

1. *write-event references and channel references are unique, and*
2. *equation (12) from Lemma 15 holds.*

We write $\vdash_w^{rf} P : ok$ for well-formed configurations P.

We need to relate the weak and strong configurations via a simulation relation in order to establish the connection between the race-free behaviors of the weak and strong semantics. We will do so by the means of an erasure function from the weak to the strong semantics.

Definition 17 (Erasure). *The erasure of a well-formed weak configuration P, written $\lfloor P \rfloor$, is defined as $\lfloor P \rfloor^{\emptyset}$ where $\lfloor P \rfloor^R$ is given on Table 5 and R is a set of write event identifiers. On the queues q_1 and q_2 in the last case, the function simply jettisons the σ-component in the queue elements.*

Note that $\lfloor P \rfloor$ is not necessarily a well-formed strong configuration. In particular, $\lfloor P \rfloor$ may contain two different write events $(\!|z{:=}v_1|\!)$ and $(\!|z{:=}v_2|\!)$ for the same variable. Besides, it is not *a priori* clear whether $\lfloor P \rfloor$ could remove all write events for a given variable (thus leaving its value undefined) and the configuration ill-formed.

Lemma 18 (Erasure and congruence). *$P_1 \equiv P_2$ implies $\lfloor P_1 \rfloor \equiv \lfloor P_2 \rfloor$.*

$$\lfloor \bullet \rfloor^R = \bullet \tag{13}$$

$$\lfloor p\langle \sigma, t\rangle \rfloor^R = \langle t \rangle \tag{14}$$

$$\lfloor n(\!|z{:=}v|\!) \rfloor^R = \begin{cases} \bullet & \text{if } n \in R \\ (\!|z{:=}v|\!) & \text{otherwise} \end{cases} \tag{15}$$

$$\lfloor P_1 \parallel P_2 \rfloor^R = \lfloor P_1 \rfloor^R \parallel \lfloor P_2 \rfloor^R \tag{16}$$

$$\lfloor \nu n\ P \rfloor^R = \begin{cases} \lfloor P \rfloor^R & \text{if } \forall p \in P.\ n \in W^{\circ}_P(\_@p) \\ \lfloor P \rfloor^{R \cup \{n\}} & \text{otherwise} \end{cases} \tag{17}$$

$$\lfloor c[q_1, q_2] \rfloor^R = c[\lfloor q_1 \rfloor^R, \lfloor q_2 \rfloor^R] \tag{18}$$

Table 5: Definition of the erasure function $\lfloor P \rfloor^R$

Lemma 19 (Erasure preserves well-formedness). *Let P be a race-free reachable weak configuration. If $\vdash_w P : ok$ then $\vdash_s \lfloor P \rfloor : ok$.*

Theorem 20 (Race-free simulation). *Let S_0 and P_0 be a strong, resp. a weak initial configuration for the same thread t and representing the same values for the global variables. If S_0 is data-race free, then $S_0 \gtrsim P_0$.*

Proof. Assume two initial race-free configurations P_0 and S_0 from the same program and the same initial values for the shared variables. To prove the $\gtrsim$-relationship between the respective initial configurations we need to establish a simulation relation, say $\mathcal{R}$, between well-formed strong and weak configurations such that P_0 and S_0 are in that relation.

Let P and S be well-formed configurations reachable (race-free) from P_0 resp. S_0. Define $\mathcal{R}$ as relation between race-free reachable configurations as

$$P\ \mathcal{R}\ S \quad \text{if} \quad S \equiv \lfloor P \rfloor \tag{19}$$

using the erasure from Definition 17. Note that by Lemma 18, $P_1\ \mathcal{R}\ S$ and $P_1 \equiv P_2$ implies $P_2\ \mathcal{R}\ S$.

Case: R-WRITE$_w$: $p\langle \sigma, z := v; t\rangle \rightarrow_w \nu n\ (p\langle \sigma', t\rangle \parallel n(\!|z{:=}v|\!))$,
where $\sigma = (E_{hb}, E_s)$ and $\sigma' = (E'_{hb}, E'_s) = (E_{hb} + (n,z), E_s + E_{hb}(z))$. By the concurrent-writes Lemma 13(2), $W^{|||}_P(z@p) = \emptyset$, i.e., there are no concurrent write events from the perspective of p. This implies that for all write events $n'(\!|z{:=}v'|\!)$ in P, we have $n' \in E_{hb}$. If $n' \in E_s$, then $n \in E'_s$ as well. If $n' \in E_{hb} \setminus E_s$, then $n' \in E'_s$ as well. Either way, *all* write events to z contained in P prior to the step are shadowed in p after the step.

Now for the new write event n in P': clearly $n \in W^{\circ}_{P'}(z@p_i)$, i.e., the event is observable for all threads. By the race-free consensus Lemma 15, we have that this is the only event that is observable by all threads, i.e.

$$\bigcap_{p_i} W^{\circ}_{P'}(z@p_i) = \{n\}\ . \tag{20}$$

That means for the erasure of P' that $\lfloor P' \rfloor \equiv \ldots \parallel p\langle t \rangle \parallel (\!|z{:=}v|\!)$ where $(\!|z{:=}v|\!)$ is the result of applying $\lfloor \_ \rfloor$ to the write event $n(\!|z{:=}v|\!)$ of P. In particular, equation (20) shows that the write event n is not "filtered out" (cf. the cases of equation (15) and (17) in Definition 17) and furthermore that all other write events for z in P' *are* filtered out.[4] It is then easy to see that by R-WRITE$_s$, $\lfloor P \rfloor \to_s \lfloor P' \rfloor$.

The remaining cases are similar. □

7 Related work

There are numerous proposals for and investigations of weak and relaxed memory models [1, 28, 3]. One widely followed approach, called *axiomatic*, specifies allowed behavior by defining various ordering relations on memory accesses and synchronizing events. Go's memory model [18] gives an informal impression of that style of specification. Less frequent are *operational* formalizations.

Boudol and Petri [10] investigate a relaxed memory model for a calculus with locks relying on concepts of rewriting theory. Unlike the presentation here, writes are buffered in a hierarchy of fifo-buffers reflecting the syntactic tree structure of configurations: immediately neighboring processors share one write buffer, neighbors syntactically further apart share a write buffer closer to the shared global memory located at the root. The position of a redex in the configuration is used as thread identifier and determines which buffers are shared. Consequently, parallel composition cannot be commutative and, therefore, terms cannot be interpreted up-to congruence $\equiv$ as in our case.

Flanagan and Freund [16] give an operational semantics of a weak memory model ("adversarial" memory) used as the basis for a race checker. The model is not as weak as the official JMM but weaker than standard JVM implementations.

Zhang and Feng [37] use an abstract machine to operationally describe a happens-before memory model. Different from us, they make use of event *buffers*. Similar to us, they keep "older" write events to account for more than one observable variable value. The paper does not, however, deal with channel communication. Another operational semantics that uses histories of time-stamped, past read/write events is given by Kang et al. [23]. In this semantics, threads can promise future writes, and a reader acquires information on the writer's view of memory. Fences then synchronize global time-stamps on memory with thread-local information. Bi-simulation proofs mechanized in Coq show correctness of compilation to various architectures.

Demange et al. [12] formalize a weak semantics for Java using buffers. The semantics is quite less relaxed than the official JMM specification, the goal being to avoid the intricacies of the happens-before JMM and offer a firmer ground for reasoning. The model is defined axiomatically and operationally and the equivalence of the two formalizations is established.

Pichon-Pharabod and Sewell [33] investigate an operational representation of a weak memory model that avoids problems of the axiomatic candidate-execution approach in addressing out-of-thin-air behavior. The semantics is studied in a calculus featuring locks as well as relaxed atomic and non-atomic memory accesses.

[4] The latter is indirectly clear already as we have established that $\lfloor \rfloor$ preserves well-formedness under the assumption of race-freedom (Lemma 19).

Guerraoui et al. [19] introduce a "relaxed memory language" with an operational semantics to enable reasoning about various relaxed memory models. Their aim is to allow correctness arguments for software transactional memories implemented on weak-memory hardware.

Alrahman et al. [4] formalize a relaxed total-store order memory model with fence and wait operations. They provide an implementation in Maude, a rewriting-based executable framework that precedes $\mathbb{K}$, and explore ways to mitigate state-space explosion.

Lange et al. [26] define a small calculus, dubbed MiGo or mini-Go, featuring channels and thread creation. The formalization does not cover weak memory. Instead, the paper uses a behavioral effect type system to analyze channel communication.

8 Conclusion

We present an *operational* specification for a weak memory model with channel communication as the prime means of synchronization. We prove a central guarantee, namely that race-free programs behave sequentially consistently. The our semantics is accompanied by an implementation in the $\mathbb{K}$ framework and by several examples and test cases [14]. We plan to use the implementation towards the verification of program properties such as data-race freedom.

The current weak semantics remembers past write events as part of the run-time configuration, but does not remember read events. We are working on further relaxing the model by treating read events similar to the representation of writes. This will allow us to accommodate load buffering behavior common to relaxed memory models, including that of Go.

Bibliography

[1] Adve, S. V. and Gharachorloo, K. (1995). Shared memory consistency models: A tutorial. Research Report 95/7, Digital WRL.

[2] Adve, S. V. and Hill, M. D. (1990). Weak ordering — a new definition. *SIGARCH Computer Architecture News*, 18(3a).

[3] Alglave, J., Maranget, L., and Tautschnig, M. (2014). Herding cats: Modelling, simulation, testing, and data-mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 36(2).

[4] Alrahman, Y. A., Andric, M., Beggiato, A., and Lluch-Lafuente, A. (2014). Can we efficiently check concurrent programs under relaxed memory models in Maude? In Escobar, S., editor, *Rewriting Logic and Its Applications – 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, volume 8663 of *Lecture Notes in Computer Science*, pages 21–41. Springer Verlag.

[5] Aspinall, D. and Ševčík, J. (2007). Java memory model examples: Good, bad and ugly. *Proc. of VAMP*, 7.

[6] Batty, M., Mamarian, K., Nienhuis, K., Pinchion-Pharabod, J., and Sewell, P. (2015). The problem of programming language concurrency semantics. In Vitek, J., editor, *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 283–307. Springer Verlag.

[7] Becker (2011). Programming languages — C++. ISO/IEC 14882:2001.

[8] Boehm, H.-J. and Adve, S. V. (2008). Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[9] Boehm, H.-J. and Demsky, B. (2014). Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 7:1–7:6, New York, NY, USA. ACM.

[10] Boudol, G. and Petri, G. (2009). Relaxed memory models: An operational approach. In *Proceedings of POPL '09*, pages 392–403. ACM.

[11] Castagna, G. and Gordon, A. D., editors (2017). *44th Symposium on Principles of Programming Languages (POPL)*. ACM.

[12] Demange, D., Laporte, V., Zhao, L., Jagannathan, S., Pichardie, D., and Vitek, J. (2013). Plan B: A buffered memory model for Java. In *Proceedings of POPL '13*, pages 329–342. ACM.

[13] Donovan, A. A. A. and Kernighan, B. W. (2015). *The Go Programming Language*. Addison-Wesley.

[14] Fava, D. (2017). Operational semantics of a weak memory model with channel synchronization. `https://github.com/dfava/mmgo`.

[15] Fava, D., Steffen, M., and Stolz, V. (2018). Operational semantics of a weak memory model with channel synchronization: Proof of sequential consistency for race-free programs. Technical Report 477, University of Oslo, Faculty of Mathematics and Natural Sciences, Dept. of Informatics. Available at `http://www.ifi.uio.no/~msteffen/download/18/oswmm-chan-rep.pdf`.

[16] Flanagan, C. and Freund, S. N. (2010). Adversarial memory for detecting destructive races. In Zorn, B. and Aiken, A., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[17] Go language specification (2016). The Go programming language specification. `https://golang.org/ref/spec`.

[18] Go memory model (2014). The Go memory model. `https://golang.org/ref/mem`. Version of May 31, 2014, covering Go version 1.9.1.

[19] Guerraoui, R., Henzinger, T. A., and Singh, V. (2009). Software transactional memory on relaxed memory models. In Bouajjani, A. and Maler, O., editors, *Proceedings of CAV '09*, volume 5643 of *Lecture Notes in Computer Science*, pages 321–336. Springer Verlag.

[20] Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.

[21] Jones, G. and Goldsmith, M. (1988). *Programming in occam2*. Prentice-Hall International, Hemel Hampstead.

[22] K framework (2017). The K framework. available at `http://www.kframework.org/`.

[23] Kang, J., Hur, C., Lahav, O., Vafeiadis, V., and Dreyer, D. (2017). A promising semantics for relaxed-memory concurrency. In [11], pages 175–189.

[24] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

[25] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691.

[26] Lange, J., Ng, N., Toninho, B., and Yoshida, N. (2017). Fencing off Go: Liveness and safety for channel-based programming. In [11].

[27] Manson, J., Pugh, W., and Adve, S. V. (2005). The Java memory memory. In *Proceedings of POPL '05*. ACM.

[28] Maranget, L., Sarkar, S., and Sewell, P. (2012). A tutorial introduction to the ARM and POWER relaxed memory models (version 120).

[29] Milner, R. (1971). An algebraic definition of simulation between programs. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 481–489. William Kaufmann.

[30] Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77.

[31] Palamidessi, C. (1997). Comparing the expressive power of the synchronous and the asynchronous π-calculus. In *Proceedings of POPL '97*, pages 256–265. ACM.

[32] Peters, K. and Nestmann, U. (2012). Is it a "good" encoding of mixed choice? In *Proceedings of the International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '12)*, volume 7213 of *Lecture Notes in Computer Science*, pages 210–224. Springer Verlag.

[33] Pichon-Pharabod, J. and Sewell, P. (2016). A concurrency-semantics for relaxed atomics that permits optimisation and avoids out-of-thin-air executions. In *Proceedings of POPL '16*. ACM.

[34] Pugh, W. (1999). Fixing the Java memory model. In *Proceedings of the ACM Java Grande Conference*.

[35] Roşu, G. and Şerbănuţă, T. F. (2010). An overview of the K semantic framework. *Journal of Logic and Algebraic Methods in Programming*, 79(6):397–434.

[36] Steffen, M. (2016). A small-step semantics of a concurrent calculus with goroutines and deferred functions. In Ábrahám, E., Huisman, M., and Johnsen, E. B., editors, *Theory and Practice of Formal Methods. Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday (Festschrift)*, volume 9660 of *Lecture Notes in Computer Science*, pages 393–406. Springer Verlag.

[37] Zhang, Y. and Feng, X. (2016). An operational happens-before memory model. *Frontiers in Computer Science*, 10(1):54–81.