Towards Deductive Verification of C11 Programs with Event-B and ProB

Mohammadsadegh Dalvandi
Department of Computer Science
University of Surrey
United Kingdom
m.dalvandi@surrey.ac.uk

Brijesh Dongol
Department of Computer Science
University of Surrey
United Kingdom
b.dongol@surrey.ac.uk

Abstract

This paper introduces a technique for modelling and verifying weak memory C11 programs in the Event-B framework. We build on a recently developed operational semantics for the RAR fragment of C11, which we use as a top-level abstraction. In our technique, a concrete C11 program can be modelled by refining this abstract model of the semantics. Program structures and individual operations are then introduced in the refined machine and can be checked and verified using available Event-B provers and model checkers. The paper also discusses how ProB model checker can be used to validate the Event-B model of C11 programs. We applied our technique to the C11 implementation of Peterson's algorithm, where we discovered that the standard invariant used to characterise mutual exclusion is inadaquate. We therefore propose and verify new invariants necessary for characterising mutual exclusion in a weak memory setting.

CCS Concepts • **Theory of computation** → **Concurrency**; **Shared memory algorithms**; • **Software and its engineering** → **Correctness**; **Software verification**;

Keywords C11, Verification, Event-B, ProB, Model Checking, Peterson's Algorithm

ACM Reference Format:
Mohammadsadegh Dalvandi and Brijesh Dongol. 2019. Towards Deductive Verification of C11 Programs with Event-B and ProB. In *Formal Techniques for Java-like Programs (FTfJP'19), July 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3340672.3341117

1 Introduction

Modern languages such as Java [20] and C11 [6] (the 2011 C++ standard) have introduced language-level relaxed memory models in order to take advantage of the weak memory optimisations provided by multi-core processors (including Intel-TSO, Power and ARMv8). However, program development under weak memory is complex. One must not only consider conventional shared memory inter-thread synchronisation, but additionally also consider appropriate *relaxed memory annotations* that control visibility of writes between threads. There has been intense interest in the programming languages community in formalising the semantics of such relaxed memory models [4, 6, 17], and more recently, in developing verification techniques that build on these formalisations [1, 10, 11, 14, 16].

This paper describes a technique for modelling and checking weak memory C11 programs using Event-B [2], which is a framework based on set theory and predicate logic. Event-B is supported by Rodin [3], an extensible tool platform that facilitates modelling and verification of Event-B models.

We focus on the so-called RAR fragment [5, 6] of C11, which is a fragment that allows both *relaxed* and *release-acquire* memory accesses. If an acquiring read reads-from a releasing write, then this establishes a *happens-before* relation (see Section 2 for details). Moreover, the *reads-from* relation is assumed to be consistent with program order, which ensures thin-air reads do not arise. In particular, this assumption precludes the *load-buffering* litmus test (see [15]).

To enable deductive verification, we build on a recently developed operational semantics [10], which enables one to step through a program in thread order. As in standard (i.e., sequentially consistent) semantics, concurrency is modelled by an interleaving of threads. However, to model weak memory the states of the model are graphs representing C11 executions (as opposed to mappings from variables to values typically used under sequential consistency). This operational semantics has been shown to be both sound and complete [10] with respect to well-accepted axiomatic semantics [6, 17].

In this paper, we show how this operational semantics can be encoded as a generic Event-B machine that serves as an abstraction for C11 programs. In particular, we require that every C11 program be a refinement of this generic model.

FTfJP'19, July 15, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6864-3/19/07…$15.00
https://doi.org/10.1145/3340672.3341117

This enables us to verify that the program structures and individual operations are consistent with the C11 semantics. Any consistent concrete model can then be checked and verified within Event-B. Thus, the main contributions of this paper are **(a)** a technique for modelling C11 programs that is consistent with the operational semantics [10]; **(b)** the use of the ProB model checker for validating Event-B models of C11 programs; and **(c)** application of the technique to the C11 implementation of Peterson's algorithm. Using ProB, we show that the standard invariant for Peterson's algorithm (as used in [10]) is valid, but also insufficient for characterising mutual exclusion in a weak memory setting. We propose a strengthening and validate this stronger invariant.

The rest of this paper is organised as follows: Section 2 describes background to the C11 operational semantics and Section 3 formalises the semantics in Event-B. Section 4 illustrates that how a concrete algorithm can be modelled in Event-B by refining the abstract model of the operational semantics, and Section 5 describes model checking using ProB. In Section 6 we present the modelling and verification of a relaxed memory version of Peterson's algorithm [25] as a case study.

2 Background to the Operational Semantics

The operational semantics of Doherty et al. [10] covers the RAR fragment of the C11 memory model and has been proved to be both sound and complete with respect to the axiomatic description [6, 17, 24] of the C11 memory model. This section briefly introduces the background to this operational semantics.

We use two versions of the message passing litmus test in Figures 1 and 2 to illustrate the various definitions. The program in Figure 1 comprises two threads and two shared variables, d and f, both of which are instantiated to 0. Thread 1 writes to data variable d then sets the flag f using a releasing write (as depicted by the annotation "R"). Thread 2 waits for the flag to be set, reading from f using an acquiring read (as depicted by the annotation "A") in a spin loop. Once thread 2 detects that the flag has been set, it proceeds to read from d and assign the value read to a local register r. The program in Figure 2 is similar, but the write and read to f are both relaxed (i.e., do not contain any release or acquire annotations).

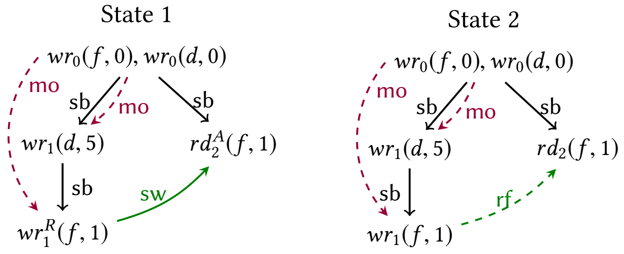
The semantics is an *operational event semantics* and is defined by events of type *ACT*. To avoid confusion between the events of the operational semantics and Event-B events, in this paper, we refer to the events from the semantics as actions. We use $\mathsf{Rd_X}$, $\mathsf{Rd_A}$, $\mathsf{Wr_X}$, $\mathsf{Wr_R}$ and U to denote read relaxed, read acquire, write relaxed, write release and (read-modify-write) update actions, respectively. We have $\mathsf{Wr_R} \supseteq \mathsf{U}$ and $\mathsf{Rd_A} \supseteq \mathsf{U}$ and define $\mathsf{Rd} = \mathsf{Rd_A} \cup \mathsf{Rd_X}$ and $\mathsf{Wr} = \mathsf{Wr_R} \cup \mathsf{Wr_X}$. For simplicity, we assume all update actions are both releasing and acquiring, although it is possible to define variations of these that leave out one or both of these annotations.

To capture weak memory effects, a *C11 state* is defined as a triple $\mathbb{D} = ((D, \mathsf{sb}), \mathsf{rf}, \mathsf{mo})$ where D is a set of *actions* paired with a *sequenced-before* relation $\mathsf{sb} \subseteq D \times D$, a *reads-from* relation $\mathsf{rf} \subseteq \mathsf{Wr} \times \mathsf{Rd}$, and a modification order $\mathsf{mo} \subseteq \mathsf{Wr} \times \mathsf{Wr}$. The sb relation records the program order within one thread and ensures initialising writes occur before any other actions. The rf relation provides justification for reads (i.e. there must have been a write action that writes the value read by the read action), and the mo relation describes the ordering of writes on variables.

The semantics formalises the synchronisation between release-acquiring actions by *synchronised-with* relation:

$$\mathsf{sw} = \mathsf{rf} \cap (\mathsf{Wr_R} \times \mathsf{Rd_A}) \ .$$

Example 2.1. Consider the programs in Figures 1 and 2 after execution of thread 1 followed by the guard evaluation in thread 2. These states are depicted as States 1 and 2 below, respectively. Note that in both states, the read of d in thread 2 has not yet occurred.



Due to the release-acquire annotations, the reads-from edge in State 1 is updgraded to a synchronised-with edge. We shall see how this effects the behaviour of the program below.

Weak memory models often include a *happens-before* relations which formalises a notion of causality. An action happening in a thread before another action in the same thread induces happens before order. Synchronising an acquiring read with a releasing write also induces happens before order. This is formally defined as

$$\mathsf{hb} = (\mathsf{sb} \cup \mathsf{sw})^+ \ .$$

In State 1 (Example 2.1), the action $rd_2^A(f,1)$ happens after each of the actions in thread 1. In State 2, no such relation exists between $rd_2(f,1)$ and the actions of thread 1.

The *from-read* relation $\mathsf{fr} = (\mathsf{rf}^{-1}; \mathsf{mo}) \backslash Id$ relates each read to all writes that are mo-after the write the read has read from. In addition, the semantics also uses the extended coherence order *eco* which fixes the order of reads and writes to each variable and is defined as

$$\mathsf{eco} = (\mathsf{fr} \cup \mathsf{mo} \cup \mathsf{rf})^+$$

Each step of the semantics is formalised by the transition relation $\rightsquigarrow_{RA} \subseteq \Sigma \times \mathsf{Wr}_{\perp} \times Evt \times \Sigma$, where Σ is the set of

```
              Init: $f = 0 \wedge d = 0$
thread 1                 thread 2
1 :  $d := 5$;           1 :  while $!f^{A}$ do skip;
2 :  $f :=^{R} 1$;       2 :  $r := d$;
```

Figure 1. Synchronised message passing

```
              Init: $f = 0 \wedge d = 0$
thread 1                 thread 2
1 :  $d := 5$;           1 :  while $!f$ do skip;
2 :  $f := 1$;           2 :  $r := d$;
```

Figure 2. Unynchronised message passing

all possible C11 states and we have $\mathsf{Wr}_{\perp} = \mathsf{Wr} \cup \{\perp\}$ and $\perp \notin \mathsf{Wr}$. We write $\sigma \overset{w,e}{\rightsquigarrow}_{RA} \sigma'$ for $(\sigma, w, e, \sigma') \in \rightsquigarrow_{RA}$. For each transition $\sigma \overset{w,e}{\rightsquigarrow}_{RA} \sigma'$, w is the *write being observed* by the action e. If e is a read action, then w is the write (or update) that e reads from. If e is a write action, then w is the write (or update) that will occur immediately before e in mo order in the post state. Finally, if e is an update, then w is the write that e reads from and also occurs immediately before e in mo order in the post state. We give details of this transition relation directly in terms of Event-B; interested readers may also consult [10].

Before we formally define the transitions, we need to introduce the notion of *observability* of writes. This is achieved using three different types of writes: *encountered writes*, which are writes that a thread is aware of; *observable writes*, which are writes that a thread is allowed to observe; and *covered writes*, which are writes that are read by an update action.

For a state $\sigma = ((D, sb), rf, mo)$, the set of encountered writes is given by:

$$EW_{\sigma}(t) = \{w \in \mathsf{Wr} \cap D \mid \exists e \in D.\ tid(e) = t \wedge (w, e) \in \mathsf{eco}^{?}; \mathsf{hb}^{?}\}$$

where $R^{?}$ is the reflexive closure of relation R and tid returns the thread id of the action. For example, in State 1, the encountered write set for both thread 1 and thread 2 is $E = \{wr_0(f, 0), wr_0(d, 0), wr_1(d, 5), wr_1^{R}(f, 1)\}$. In State 2, the encountered write set for thread 1 is also E, but for thread 2 is $\{wr_0(f, 0), wr_0(d, 0), wr_1^{R}(f, 1)\}$.

Observable writes are the writes that a thread can read from when executing its next read action. Observable writes are defined based on encountered writes as follows:

$$OW_{\sigma}(t) = \{w \in \mathsf{Wr} \cap D \mid \forall w' \in EW_{\sigma}(t).\ (w, w') \notin \mathsf{mo}\}$$

For example, in State 1, the observable write set for both thread 1 and thread 2 is $O = \{wr_1(d, 5), wr_1^{R}(f, 1)\}$, i.e., neither initialising write to d and f is observable. In State 2, the observable write set for thread 1 is also O, but for thread 2 is $\{wr_0(d, 0), wr_1(d, 5), wr_1^{R}(f, 1)\}$.

Covered writes are defined to preserve the *atomicity* of update actions: there cannot be a write in modification order between the write that the update action is reading from and the write of the update action itself. Covered writes are defined as follows:

$$CW_{\sigma} = \{w \in \mathsf{Wr} \cap D \mid \exists u \in \mathsf{U}.\ (w, u) \in \mathsf{rf}\}$$

3 C11 Operational Semantics in Event-B

Modelling a complex system in Event-B benefits from its built-in theories for abstraction and refinement. The abstract level models the general purpose of the system by specifying *what* the system is supposed to achieve. Each refinement level adds more details to the model to describe *how* the goal of the system can be achieved. The abstract and concrete levels are linked via a series of refinement proofs that ensures the concrete model "displays the same behaviour" as the abstract one [19].

We exploit refinement in our framework by developing an abstract model that formalises the operational semantics of C11 executions. In particular, abstract events are generic read, write and update events that manipulate the C11 state according to the operational weak memory rules [10]. In Section 4, we show how these can be refined into concrete read, write and update events corresponding to a program code. Interestingly, an implementor is not required to instantiate or modify a C11 state; they can rely on the refinement framework and the abstract model to ensure that their program exhibits behaviours consistent with the C11 memory model. This decoupling allows modelling of C11 programs to progress rapidly. Moreover, given a concrete implementation, it is possible to replace the abstract memory specification with an entirely different memory model; the concrete program would then execute as specified by this replaced memory model.

A model in Event-B has two main parts: a *context* and a *machine*. A context is the static part of a model which is specified using carrier sets, constants and axioms. A machine is the dynamic part, which is specified using variables, invariants and events.

The Event-B context for the operational semantics is given as follows:

CONTEXT *OpSemCtx*
SETS *ACT*, *T*, *VAR*
CONSTANTS $\mathsf{Rd_A}$, $\mathsf{Rd_X}$, $\mathsf{Wr_R}$, $\mathsf{Wr_X}$, U, *VAL*, $t0$
AXIOMS
axm1: $partition(ACT, \mathsf{Rd_A}, \mathsf{Rd_X}, \mathsf{Wr_R}, \mathsf{Wr_X}, \mathsf{U})$
axm2: $VAL = \mathbb{N}$
axm3: $t0 \in T$

where *ACT* is the set of all actions, *T* represents threads, and *VAR* represents program variables. Constants $\mathsf{Rd_A}$, $\mathsf{Rd_X}$, $\mathsf{Wr_R}$, $\mathsf{Wr_X}$, and U are sets of release-acquire and relaxed actions, respectively; $t0$ is the initialising thread; and *VAL* is the set of values. The first axiom states that set *ACT* is partitioned by the sets Rd_A, Rd_X, Wr_R, Wr_X, and U. The second axiom defines *VAL* to be the set of natural numbers and the third axiom declares $t0$ to be a member of T.

To model the semantics, we start by formalising the C11 state:

Machine *actions* **Sees** *OpSemCtx*
Variables *D*, *sb*, *mo*, *rf*
Invariants
inv1: $D \subseteq ACT$
inv2: $sb \in D \leftrightarrow D$
inv3: $mo \in (D \cap \mathsf{Wr}) \leftrightarrow (D \cap \mathsf{Wr})$
inv4: $rf \in (D \cap \mathsf{Wr}) \leftrightarrow (D \cap \mathsf{Rd})$

The machine above *sees* the context (*OpSemCtx*) that was introduced earlier. The four variables *D*, *sb*, *mo*, and *rf* together with their invariants define the C11 state. There are other variables and functions in the machine that, for presentational purposes, are not shown here. An interested reader can consult the full model in http://dalvandi.github.io/FTfJP2019/ to see how these properties are encoded.

A machine also comprises a set of *events*, which model the state change in the system. A general Event-B event has the following form:

$$\mathrm{E} \triangleq \textbf{any}\ t\ \textbf{where}\ P(t,v)\ \textbf{then}\ S(t,v)\ \textbf{end}$$

where *E* is the name of the event, *t* is a set of input parameters, v is the set of model variables (given by the context), $P(t, v)$ is a set of guards and $S(t, v)$ is a set of assignments.

The events modelling read, write and update actions of C11 are given in Figure 3. These events model different cases of the transition relation $\rightsquigarrow_{RA} \subseteq \Sigma \times \mathsf{Wr}_\perp \times Evt \times \Sigma$ given in Section 2.

The *read* event is parametrised by e, t, x, n, and w, whose types are formalised by the guards. We require that e is a new acquiring or relaxed read (*grd1*), t is a non-initialising thread (*grd2*), x is a program variable (*grd3*), n is the value being read (*grd4*), and w is an observable write (*grd6*) to the variable x (*grd7*) with write value n (*grd8*). w is the write that read e is reading from. The three actions of the event specify the way that the state is changed once a read action is executed. Function *var* returns the action variable, *wrval* returns the value written by a write action, *tid* returns the thread that executed the action, and *OW* returns the set of observable writes by the thread. Here we have omitted the details of how other properties like hb, sw, eco, *OW*, *EW* and *CW* are specified in the model to improve readability.

The *write* event modifies *D* (*act1*) and *sb* (*act2*) in the same way as a *read*. The difference is that it does not change *rf* but changes *mo* (*act3*). The notation $R[S]$ denotes the relational image of R with respect to set S. Thus, given that w is a write that is observable to the write (*grd5*), *mo* is updated so that e is inserted immediately after w in *mo*. To achieve this, we extend the *mo* relation with edges from all actions that are *mo*-prior to w (inclusive) to e, and from e to all edges that are *mo*-after w (exclusive). Further details are available in [10].

The *update* event is again similar. Since it comprises both a read and a write, its assignment modifies both *rf* and *mo* in addition to *D* and *sb*.

Our abstract model includes one more event called *init_t*, which initialises all the variables of the program; it initialises D to be the set of initialising writes, and updates *rf*, *mo* and *sb* to the empty set.

4 Modelling Concrete Programs in Event-B

The previous section introduced an abstract model of the operational semantics in Event-B. In this section, we discuss how this abstract model can be refined to model concrete algorithms. Refinement of an Event-B model may require refining existing events and/or introducing new events, variables and invariants. All abstract events may be refined by one or more concrete events.

Event-B does not impose any explicit ordering on the execution of events. An event is chosen for execution nondeterministicly from a set of enabled events (i.e. events whose guards evaluate to true). Absence of explicit ordering for modelling a concrete algorithm is not always desirable, e.g., when modelling control flow. In order to tackle this, we provide explicit program counters (*pc*) into the model.

To illustrate how a concrete algorithm can be modelled in Event-B, consider the following message-passing algorithm from Figure 1. First, we extend the abstract constant and instantiate sets *VAR* and T with their exact members:

CONTEXT $c1$ **extends** *OpSemCtx*
CONSTANTS $f, d, t1, t2$
AXIOMS
axm1: $partition(T, \{t0\}, \{t1\}, \{t2\})$
axm2: $partition(VAR, \{f\}, \{d\})$

We model thread 1 by refining the abstract model of the semantics as follows:

Event $t1\_wr\_d\_5$
refines *write*
where
grd7: $e \in \mathsf{Wr}_\mathsf{X}$
grd8: $x = d$
grd9: $n = 5$
grd10: $t = t1$
grd11: $pc(t) = 1$
then
act4: $pc(t) := pc(t) + 1$
End

Event $t1\_wrR\_f\_1$
refines *write*
where
grd7: $e \in \mathsf{Wr}_\mathsf{R}$
grd8: $x = f$
grd9: $n = 1$
grd10: $t = t1$
grd11: $pc(t) = 2$
then
act4: $pc(t) := pc(t) + 1$
End

The above events refine the abstract *write* event: *grd7* determines if the action is a relaxed ($e \in \mathsf{Wr}_\mathsf{X}$) or releasing write ($e \in \mathsf{Wr}_\mathsf{R}$), *grd8*, *grd9*, *grd10* specify the exact variable and value of the write and the thread that is performing it. The control flow is modelled by *grd11* and *act14.*

```
Event $read$
any $e, t, x, n, w$
where
    grd1: $e \in (\mathsf{Rd_A} \cup \mathsf{Rd_X}) \setminus D$
    grd2: $t \in T \setminus \{t0\}$
    grd3: $x \in VAR$
    grd4: $n \in VAL$
    grd5: $w \in D \cap \mathsf{Wr}$
    grd6: $w \in OW(t)$
    grd7: $var(w) = x$
    grd8: $wrval(w) = n$
then
    act1: $D := D \cup \{e\}$
    act2: $sb := sb \cup$
        $\{e1 \mapsto e \mid e1 \in D \land tid(e1) \in \{t, t0\}\}$
    act3: $rf := rf \cup \{w \mapsto e\}$
End
```

```
Event $write$
any $e, t, x, n, w$
where
    grd1: $e \in (\mathsf{Wr_X} \cup \mathsf{Wr_R}) \setminus D$
    grd2: $t \in T \setminus \{t0\}$
    grd3: $x \in VAR$
    grd4: $n \in VAL$
    grd5: $w \in OW(t) \setminus CW$
    grd6: $var(w) = x$
then
    act1: $D := D \cup \{e\}$
    act2: $sb := sb \cup$
        $\{e1 \mapsto e \mid e1 \in D \land tid(e1) \in \{t, t0\}\}$
    act3: $mo :=$
        $mo \cup ((\{w\} \cup mo^{-1}[\{w\}]) \times \{e\})$
        $\cup (\{e\} \times mo[\{w\}])$
End
```

```
Event $update$
any $e, t, x, n, m, w$
where
    grd1: $e \in \mathsf{U} \setminus D$
    grd2: $t \in T \setminus \{t0\}$
    grd3: $x \in VAR$
    grd4: $n \in VAL$
    grd5: $m \in VAL$
    grd6: $w \in OW(t) \setminus CW$
    grd7: $var(w) = x$
    grd8: $wrval(w) = m$
then
    act1: $D := D \cup \{e\}$
    act2: $rf := rf \cup \{w \mapsto e\}$
    act3: $sb := sb \cup$
        $\{e1 \mapsto e \mid e1 \in D \land tid(e1) \in \{t, t0\}\}$
    act4: $mo :=$
        $mo \cup ((\{w\} \cup mo^{-1}[\{w\}]) \times \{e\})$
        $\cup (\{e\} \times mo[\{w\}])$
End
```

Figure 3. Read, write and update actions

Line 1 of Thread 2 can be modelled by refining the abstract *read* event. There are two control flow possibilities, depending on the value returned by the read:

```
Event $t2\_rdA\_f\_not1$
refines $read$
where
        grd9: $e \in \mathsf{Rd_A}$
        grd10: $x = f$
        grd11: $n \neq 1$
        grd12: $t = t2$
        grd13: $pc(t) = 1$
End
```

```
Event $t2\_rdA\_f\_1$
refines $read$
where
        grd9: $e \in \mathsf{Rd_A}$
        grd10: $x = f$
        grd11: $n = 1$
        grd12: $t = t2$
        grd13: $pc(t) = 1$
then
        act4: $pc(t) := pc(t) + 1$
End
```

Event $t2\_rdA\_f\_not1$ models the "retry" branch of the busy loop, and hence, does not increase the program counter value, since it has not observed value 1 for *f*. Event $t2\_rdA\_f\_1$ models the "exit" branch of the loop, where the thread observes value 1 for *f*, and hence the event increments the program counter value to exit the loop.

The relaxed read action on line 2 of thread 2 can be modelled in a similar manner; we do not present its details here.

All of the concrete events presented above implicitly inherit the behaviours of their corresponding abstract events, and therefore, change the abstract state in the exact same way. This enables us to conclude that the model is indeed consistent with the behaviour of C11.

5 Model Checking with ProB

In this section we discuss that how we can use the model checking facilities of the ProB plug-in for Rodin to quickly validate a C11 program involving relaxed accesses and release-acquire annotations. To achieve this, first recall that we use a deferred (possibly infinite) set *ACT* to model actions (e.g., relaxed reads, writes, and updates). The model checker, however, can only work with finite sets. We limit the size of infinite sets by extending the context and specifying the cardinality of the deferred sets. For instance, we can extend the abstract context *c1* for the message passing algorithm as follows:

```
CONTEXT $c2$ extends $c1$
AXIOMS
axm1: $card(\mathsf{Rd_A}) = 5$     axm2: $card(\mathsf{Rd_X}) = 5$
axm3: $card(\mathsf{Wr_R}) = 5$     axm4: $card(\mathsf{Wr_X}) = 5$
```

The size of sets should be large enough so that all parts of the algorithm can be covered in model checking and small enough so that the model checker terminates in a reasonable time and does not run out of memory. Finding the right value sometimes requires a number of trials and may vary for different programs.

The desirable behaviour of the given message passing algorithm is that when thread 2 reads value 1 for variable *f*, then the only value that can be read for variable *d* is 5. We have stated this property in terms of observable writes as follows:

$$pc(t2) = 2 \Rightarrow (\forall w . w \in OW(t2) \land var(w) = d \Rightarrow wrval(w) = 5) \quad (1)$$

This states that whenever thread 2 reaches line 2 (i.e. the busy loop is terminated) then all observable writes to variable *d* have value 5.

Algorithm 1 Modified Peterson's algorithm

```
Init: $flag_1 = false \wedge flag_2 = false \wedge turn = 1 \wedge cs_1 = 0 \wedge cs_2 = 0$
1: thread 1
2:     $flag_1 := true$ ;
3:     $turn.\mathbf{swap}(2)^{\mathsf{RA}}$ ;
4:     while $(flag_2 = true)^{\mathsf{A}}$
               $\wedge\ turn = 2$
         do skip
5:     $cs_1 := 1$
6:     Critical section ;
7:     $cs_1 := 0$
8:     $flag_1 :=^{\mathsf{R}} false$;

1: thread 2
2:     $flag_2 := true$;
3:     $turn.\mathbf{swap}(1)^{\mathsf{RA}}$ ;
4:     while $(flag_1 = true)^{\mathsf{A}}$
               $\wedge\ turn = 1$
         do skip
5:     $cs_2 := 1$
6:     Critical section ;
7:     $cs_2 := 0$
8:     $flag_2 :=^{\mathsf{R}} false$;
```

We applied model checking to the Event-B model of both synchronised (Figure 1) and unsynchronised (Figure 2) versions of message passing example to check the preservation of Invariant (1). For the synchronised version, the model checker terminates in just few seconds finding no invariant violation. For the unsynchronised version, it quickly found an invariant violation and returned the trace of events leading to the violation for inspection. As one may expect, for the unsynchronised version, a weaker property can be proved:

$$pc(t2) = 2 \Rightarrow (\forall w. w \in OW(t2) \wedge var(w) = d \Rightarrow wrval(w) = 0 \vee wrval(w) = 5) \tag{2}$$

This states that upon termination of the loop, thread 2 may observe any of the writes to d: $wr_0(d, 0)$ or $wr_1(d, 5)$.

Validity of invariants (1) and (2), can respectively be can be mapped to States 1 and 2 in Example 2.1. From State 1, it is impossible for thread 2 to perform a transition into a state in which it reads 0 for d. However, from State 2, the reads to values 0 and 5 for d are both available to thread 2.

6 Case Study: Peterson's Algorithm

In this section, we apply our modelling approach to a more complex example: a release-acquire version of the classic Peterson's algorithm (see Algorithm 1) taken from [25]. The algorithm uses a variable $flag_i$ to indicate that thread i intends to enter its critical section and a shared variable *turn* to give way when both try to enter their critical sections simultaneously. The modification to *turn* is via a **swap** operation (line 3) that atomically updates *turn* to the given value. Crucially, the **swap** corresponds to a read-modify-write update event that induces both a release and acquire synchronisation on *turn*.

To prove the mutual exclusion property of the algorithm, the following invariant is given in [10]:

$$pc(t1) \neq 6 \vee pc(t2) \neq 6 \tag{3}$$

The model checker terminates in just over one minute with Symmetry Marker [18] reduction enabled, finding no violation of invariant 3. To further scrutinise the algorithm, we relaxed some of the release-acquire writes and reads, expecting that the model checker would find an invariant violation. To our surprise, after relaxing the writes/reads to *flag* in lines 4 and 8, model checking terminates without any issues. On the other hand, replacing the **swap** operation with a standard write event immediately generates a counterexample, showing that these annotations are critical to invariance of condition (3).[1]

Upon further investigation, we discovered that invariant (3) is in fact not strong enough to establish atomicity of the critical section. In particular, (3) does not take the observability of critical section by other threads into consideration. In particular, removal of the annotations on *flag* would mean that any writes performed in the critical section may not be visible to the other thread when this other thread enters its critical section.

To strengthen the invariant, for each thread i, we introduce an auxiliary variable cs_i, which is initialised to 0, set to 1 when i enters its critical section, and set back to 0 when i exits its critical section. If the release-acquire annotations are used correctly (i.e. threads are synchronised correctly) then whenever a thread enters its critical section the only observable write to variable cs of the other thread should be a write with value 0. This means that the other thread as either not entered its critical section, or has already left it. Formally, we strengthen the program invariant by introducing the following additional properties:

$$pc(t1) = 6 \Rightarrow (\forall w. w \in OW(t1) \wedge var(w) = cs2 \Rightarrow wrval(w) = 0) \tag{4}$$

$$pc(t2) = 6 \Rightarrow (\forall w. w \in OW(t2) \wedge var(w) = cs1 \Rightarrow wrval(w) = 0) \tag{5}$$

Invariant (4) states that if thread 1 is in its critical section, then the only value of cs_2 that it can observe is 0. Invariant (5) is symmetric. Now, relaxing any of the release-acquire reads/writes in Algorithm 1 results in the violation of at least one of these invariants.

7 Conclusion and Related Work

This paper takes a step towards mechanised deduction verification of weak memory C11 programs using Event-B and ProB. We modelled the C11 operational semantics developed by Doherty et al in Event-B. Our model of the semantics serves as an abstraction. Concrete C11 programs can be modelled consistently with respect to the semantics by refining this abstract model. This decoupling enables rapid modelling of C11 programs in Event-B, and also enables one to quickly experiment with different abstract memory models for the same concrete model. We show that the desired safety properties of an algorithm can be formally specified and quickly checked using the ProB model checker. The utility of this approach is demonstrated by the discovery of the inadequacy

[1] The full Event-B model archive of the Peterson's algorithm can be found at http://dalvandi.github.io/FTfJP2019/.

of the standard invariant used to characterise mutual exclusion. Moreover, model checking makes it straightforward to propose a fix and validate its adequacy.

Event-B has been widely used to verify and develop programs (e.g., [7, 8, 13, 21]). However none of them addresses the verification of concurrent programs under weak memory, partly due to the absence of a suitable operational characterisations of weak memory.

There have been several recent works on model checking for weak memory, including tools specifically for C11 [1, 16, 22, 23]. The focus of these works, however, is the development of the model checkers themselves, i.e., specialised techniques that reduce the search space and improve tractability of model checking for C11. Executions here are validated against established axiomatic semantics of C11. Our work differs in that we use a new operational description of a fragment of the language [10] against which checking is carried out. Our aim is to leverage this semantics to verify weak memory behaviour using existing tools and techniques developed for sequentially consistent memory.

An advantage of Event-B is that it includes a theorem prover for Event-B models. In a future extension of this work, we aim to use these facilities to fully verify C11 programs. We also aim to develop more robust refinement proofs so that invariants necessary for proving atomicity (e.g., via weak memory linearizability [9, 12]) are not missed.

Acknowledgments

This work was supported by EPSRC grant EP/R032556/1. The authors thank Lindsay Groves for his comments and suggestions.

References

[1] P. A. Abdulla, M. F. Atig, B. Jonsson, and T. P. Ngo. 2018. Optimal stateless model checking under the release-acquire semantics. *PACMPL* 2, OOPSLA (2018), 135:1–135:29.

[2] Jean-Raymond Abrial. 2010. *Modeling in Event-B: system and software engineering*. Cambridge University Press.

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, ThaiSon Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer* 12, 6 (2010), 447–466.

[4] S. V. Adve and H.-J. Boehm. 2011. Memory Models. In *Encyclopedia of Parallel Computing*. Springer, 1107–1110.

[5] M. Batty, A. F. Donaldson, and J. Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *POPL*. ACM, 634–648.

[6] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. 2011. Mathematizing C++ concurrency. In *POPL*, T. Ball and M. Sagiv (Eds.). ACM, 55–66.

[7] Pontus Boström, Fredrik Degerlund, Kaisa Sere, and Marina Waldén. 2014. Derivation of concurrent programs by stepwise scheduling of Event-B models. *Formal Aspects of Computing* 26, 2 (2014), 281–303.

[8] Mohammadsadegh Dalvandi, Michael Butler, Abdolbaghi Rezazadeh, and Asieh Salehi Fathabadi. 2018. Verifiable Code Generation from Scheduled Event-B Models. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 234–248.

[9] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2018. Making Linearizability Compositional for Partially Ordered Executions. In *IFM (Lecture Notes in Computer Science)*, Vol. 11023. Springer, 110–129.

[10] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2019. Verifying C11 programs operationally. In *PPoPP*. ACM, 355–365.

[11] M. Doko and V. Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *ESOP*. 448–475.

[12] B. Dongol, R. Jagadeesan, J. Riely, and A. Armstrong. 2018. On abstraction and compositionality for weak-memory linearisability. In *VMCAI (Lecture Notes in Computer Science)*, Vol. 10747. Springer, 183–204.

[13] Andrew Edmunds and Michael Butler. 2011. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. (February 2011). https://eprints.soton.ac.uk/272006/ Event Dates: 2nd April 2011.

[14] J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *ECOOP*. 17:1–17:29.

[15] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL*. ACM, 175–189.

[16] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *PACMPL* 2, POPL (2018), 17:1–17:32.

[17] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer. 2017. Repairing sequential consistency in C/C++11. In *PLDI*, A. Cohen and M. T. Vechev (Eds.). ACM, 618–632.

[18] Michael Leuschel and Thierry Massart. 2010. Efficient approximate verification of B and Z models via symmetry markers. *Annals of mathematics and artificial intelligence* 59, 1 (2010), 81–106.

[19] Grant Malcolm and Joseph A Goguen. 1994. *Proving correctness of refinement and implementation*. Oxford University. Computing Laboratory. Programming Research Group.

[20] J. Manson, W.Pugh, and S. V. Adve. 2005. The Java memory model. In *POPL*. ACM, 378–391.

[21] Dominique Mery and Rosemary Monahan. 2013. Transforming Event-B Models into Verified C# Implementations. In *VPT 2013. First International Workshop on Verification and Program Transformation*, Vol. 16.

[22] Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *OOPSLA*. ACM, 131–150.

[23] Brian Norris and Brian Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.* 38, 3 (2016), 10:1–10:51.

[24] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. 2017. Automatically comparing memory consistency models. In *POPL*. ACM, 190–204.

[25] A. Williams. 2018. https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html. Accessed: 2018-06-20.