Algebraic Semantics for C++11 Memory Model

***Abstract*—The C++11 standard introduced a language level weak memory model (i.e., the C++11 memory model) to improve the performance of the execution of C/C++ programs, and the linearizability is an important issue for this model. Algebra is well-suited for direct use by engineers in symbolic calculation of parameters. It is a challenge to investigate the algebraic semantics for C++11, aiming to support the linearizability of it.**

Inspired by the promising semantics, in this paper, we explore the algebraic laws for the C++11 memory model, including a set of sequential and parallel expansion laws, which are implemented in the rewriting engine Maude. We introduce the concept of guarded choice, and every program can be converted into the head normal form of guarded choice. Then the linearizability of the C++11 memory model is supported. In addition, we define a read mechanism for each variable in the generated configuration sequences, which has the ability to handle the relaxed and release/acquire accesses. Consequently, the valid execution results of any program under this memory model can be provided, based on the achieved algebraic laws.

***Index Terms*—Relaxed Memory Model, C++11 Memory Model, Algebraic Laws, Maude, Read Mechanism**

I. Introduction

Modern multi-processors and programming languages employ relaxed (*aka weak*) memory models for efficiency reasons. The TSO memory model [1], which is supported by x86 architecture and SPARC implementations, and the revised ARMv8 memory model [2] are both expressed as abstract machines. Their transitions can be realized by writing to the private store buffers and propagating to the shared memory, and so on. However, the C++11 memory model is always defined as an axiomatic memory model [3], which does not execute stepwise. It formalizes the valid results of any program as execution graphs, which need to conform to a set of coherence axioms.

The promising semantics (PS) by Kang et al. [4], [5] provides the SC-style operational semantics for the C++11 concurrency model, on the basis of the concept of promise and time stamp. A thread T may promise to write a value v to a memory location x at some point in the future. It enables other threads to read from it before the write is actually executed. However, for preventing out-of-thin-air (OOTA) behaviors, the promise must be fulfilled later. For each thread T, it owns a map from any location x to the largest time stamp of a write to x that T has observed or performed. This map can be regarded as T's view of memory. When T reads from x, it can only read from the message whose time stamp recorded for x is larger than or equal to that in T's view. If T wants to write to x, it must pick a time stamp that is strictly larger than that recorded for x in its view.

Unifying Theories of Programming (*UTP*) [8] was developed by Hoare and He in 1998. It targets at proposing a convincing unified framework to combine and link operational semantics, denotational semantics [9] and algebraic semantics [10]. Each of the semantics has distinctive advantages for theories of programming. For instance, the algebraic semantics is well suited in symbolic calculation of parameters and structures of an optimal design. The algebraic approach has been successfully applied in provably correct compilation [11].

In this paper, aiming to support the linearizability of the weak memory model, we explore the algebraic laws for the C++11 concurrency model, including a set of sequential and parallel expansion laws. Our investigation is inspired by the promising semantics [4], [5]. The concept of guarded choice and head normal form is introduced, and every program can be expressed in the form of guarded choice. Then the linearizability of this memory model is supported. In addition, we implement the algebraic laws in the rewriting engine Maude. Finally, we give the definition of the read mechanism for the variables read in the generated sequences. The read mechanism can get all the valid executions, and it does not rely on the introduction to time stamp and a variety of relations appearing in the traditional execution graphs of the C++11 memory model. The framework of this work is illustrated in Figure 1.

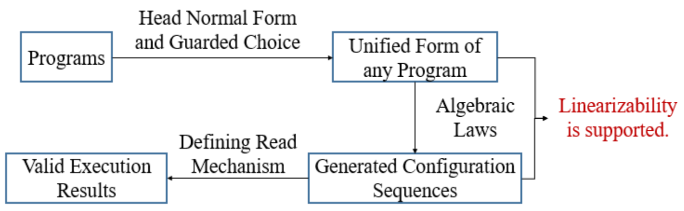

Programs
Head Normal Form and Guarded Choice
Unified Form of any Program
Algebraic Laws
Linearizability is supported.
Defining Read Mechanism
Valid Execution Results
Generated Configuration Sequences

Fig. 1. The framework of our work.

Further, Xiao et al. also investigated the trace semantics for the C++11 memory model, acting in the denotational semantics style [12]. The investigation of the trace semantics provides the mathematical foundation for algebraic semantics for C++11 in this paper.

The remainder of this paper is organized as follows. We introduce the promising semantics briefly in Section II. Section III presents the algebraic laws of the C++11 memory model. In Section IV, we implement the algebraic laws in the rewriting engine Maude. The read mechanism is defined in Section V. Section VI concludes the paper and discusses the future work. Some technical definitions are given in the appendix.

II. Background

In this section, we give the introduction to the promising semantics (PS) [4], [5], including relaxed reads and writes, release and acquire accesses, and the release fence and acquire fence instructions.

A. Relaxed Reads and Writes

In the C++11 memory model, the weakest memory ordering is relaxed atomics. It imposes no additional constraints on reordering and only guarantees coherence. The view of the thread T, which is formed by the time stamps of all the memory locations appearing in T, is used to determine the correct semantics upon memory accesses. Consider the simple program below, where x and y are global variables, while a and b are local. Assume the shared memory has the messages $\langle x : 0@0\rangle$ and $\langle y : 0@0\rangle$ initially. Here, the message is the triple in the form of $\langle x : v@t\rangle$, where x is a location, v is a value and t is a time stamp.

$$\begin{array}{c||l} T_1 & T_2 \\ x := 1; & y := 1; \\ a := y & b := x \end{array}$$

The thread T_1 picks the time stamp 1 to promise to write the value 1 to x. Then, the message $\langle x : 1@1\rangle$ is added into the memory and the time stamp recorded for the location x in T_1 is updated to 1. When T_1 performs the read from y, the initial value 0 is possible to be returned, because T_1's view of y is not changed. It is the same for the thread T_2. Therefore, when T_2 reads from the location x, it can get the value 0 too.

Relaxed atomic updates are a pair of accesses to the same memory location (i.e., a read followed by a write), such as read-modify-write (RMW) instructions including compare-and-swap (CAS) and fetch-and-add (FADD). These instructions only replace time stamps with time stamp intervals, and we do not analyze them in detail in this paper.

B. Release and Acquire Accesses

An interesting feature in the C++11 memory model is the ability for threads to synchronize using memory fences. When a read before an acquire fence reads from a write after a release fence, and the two fences synchronize, any write before the release fence must be visible to any read after the acquire fence. An acquire read in the form of x_{acq} can be regarded as a relaxed read followed by an acquire fence, and a release write expressed as x_{rel} is a release fence followed by a relaxed write. However, the fences here only induce synchronization on the location of the access.

In order to provide the appropriate semantics to release and acquire accesses, PS separates a thread view into three views, namely release view, current view and acquire view. The current view is explained as the thread view introduced previously. The acquire view records what the thread's current view will become if it performs an acquire fence. The release view of a thread is treated as one separate view per location instead of a single view. It is used to record the thread's current view reaching the latest release fence or release write to that location.

$$\begin{array}{c||l} T_1 & T_2 \\ x := 1; & a := y_{\text{acq}}; \\ y_{\text{rel}} := 1 & b := x \end{array}$$

In addition, the message view is also used in PS. It records the release view of the writing thread when the write happens, which is updated to include the write itself. Based on the introduction to the message view, a release access can inform another thread, which performs the acquire access to the same location, about the modification in the thread. In the example above, the release view of y in T_1 is $[x@t_x, y@t_y]$. Then, when T_2 reads from y, it updates its current view to $[x@t_x, y@t_y]$. Consequently, x can be 1 merely.

III. Algebraic Semantics

Program properties can be expressed as algebraic laws (equations usually). This section aims to explore a set of algebraic laws of the C++11 memory model, including sequential and parallel expansion laws. In our approach, every program can be expressed as the head normal form of the guarded choice. Then the linearizability of C++11 is supported.

A. Types of Guarded Choice

Now, we introduce the concept of guarded choice. A guarded choice is composed of a set of guarded components. The introduction to guarded choice is to support the sequential and parallel expansion laws.

$h\&(act, tid, idx)[q] \looparrowright P$ is a guarded component, where:

1) h is a Boolean condition. Except for the branching condition, it has the value of true, which can be ignored for simplicity.
2) The parameter act indicates the action extracted from one statement. For a relaxed write to the memory location x, it is always separated into two steps: (a) promising to write to x, and then (b) fulfilling the promise later. If act is a promise operation, it is expressed as $\langle x = e\rangle$, and q is a fulfill operation in the form of $h\&(act', tid, idx')$. For other types of statements, q is always ε.
3) We use tid to stand for the identity of the thread which performs the above action act.
4) The element idx is used to denote the location of act, which can be understood intuitively by Example 1 as below.

Example 1. Consider the five different statements in the following. Here, x is a global variable, while a is local. The notation e ranges over arithmetic expressions on real numbers.

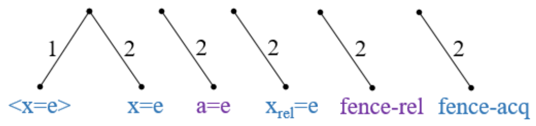


Fig. 2. The illustration of the location.

The actions $\langle x = e\rangle$ and $x = e$ are separated from the relaxed write $x := e$. $\langle x = e\rangle$ is promising to write the calculated value of the expression e to the memory location x, while $x = e$ is to fulfill the mentioned promise. And the indices of the two actions are $\langle 1\rangle$ and $\langle 2\rangle$, shown in Figure 2. The operation $a = e$ is corresponding to the local assignment $a := e$, and then its index is $\langle 2\rangle$. Release writes are not allowed to be promised. Therefore, the index of the operation $x_{\text{rel}} = e$ which is extracted from the release write $x_{\text{rel}} := e$ is $\langle 2\rangle$. The analysis of the release fence and acquire fence is similar to that of a release write. □

Next, we use Example 2 to describe the intuitive understanding of the generation of the thread id.

Example 2. Consider the parallel process $P =_{df} U||V$, where $U =_{df} A||B$. Below is the graph that illustrates the structure of the process P. Further, we assign a label for each edge. If it is the left edge, the label is 1, otherwise the label is 2.

Now, we consider the sequence that can index each subprocess of the parallel process P. We assume that every sequential process has the thread id λ. Initially, in U, the thread id of A is $\langle 1\rangle$, and that of B is $\langle 2\rangle$. When U would like to make parallel composition with another process V, the processes U and V are labeled by $\langle 1\rangle$ and $\langle 2\rangle$ respectively. Then, the indices of A and B are attached by a prefix $\langle 1\rangle$, and updated to $\langle 1\rangle^\wedge\langle 1\rangle$ and $\langle 1\rangle^\wedge\langle 2\rangle$.

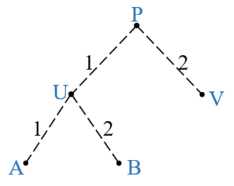


Fig. 3. The introduction to thread id.

For simplicity, we use $\langle 1, 1\rangle$ instead of $\langle 1\rangle^\wedge\langle 1\rangle$, and $\langle 1, 2\rangle$ for $\langle 1\rangle^\wedge\langle 2\rangle$. Further, for any tid, we have $tid^\wedge\lambda = tid$. □

Then, the guarded choice under C++11 can be divided into three types.

1) $[\!]_{i\in I}\{h_i\&(act_i, tid_i, idx_i)[(act'_i, tid_i, idx'_i)] \looparrowright P'_i\}$
The first type of guarded choice is only composed of a set of relaxed write actions. Any can be fired when the corresponding Boolean condition is satisfied.
2) $[\!]_{i\in I}\{h_i\&(act_i, tid_i, idx_i) \looparrowright P'_i\}$
The second type consists of a variety of atomic actions including register writes, release writes, branching conditions, acquire fence and release fence instructions.
3) $[\!]_{i\in I}\{h_i\&(act_i, tid_i, idx_i)[(act'_i, tid_i, idx'_i)] \looparrowright P'_i\}\ [\!]$ $[\!]_{j\in J}\{h_j\&(act_j, tid_j, idx_j) \looparrowright Q'_j\}$
The third type can be obtained by combining the first and second types of guarded choice.

B. Head Normal Form

Next, we assign every program P a normal form called *head normal form*, $HF(P)$. Our later introduction to the read mechanism is based on the head normal form.

(1) With regard to local assignment, the remaining part of the first step expansion is the empty process, which we use the notation E to denote.

$$HF(a := e) =_{df} [\!]\{\text{true}\&(a = e, \lambda, \langle 2\rangle) \looparrowright E\}$$

(2) For a relaxed write to the memory location x, it can be simulated by first promising to write to x, and then fulfilling the mentioned promise.

$$HF(x := e) =_{df} [\!]\{\text{true}\&(\langle x = e\rangle, \lambda, \langle 1\rangle)[(x = e, \lambda, \langle 2\rangle)] \looparrowright E\}$$

The order between the two operations cannot be broken.

(3) Below is the analysis of release accesses. Different from a relaxed write, a release write does not allow the promise.

$$HF(x_{\text{rel}} := e) =_{df} [\!]\{\text{true}\&(x_{\text{rel}} = e, \lambda, \langle 2\rangle) \looparrowright E\}$$

(4) For acquire fence and release fence, the definition of head normal form of them is similar to that of a release write.

$$HF(\text{fence–acq}) =_{df} [\!]\{\text{true}\&(\text{fence–acq}, \lambda, \langle 2\rangle) \looparrowright E\}$$
$$HF(\text{fence–rel}) =_{df} [\!]\{\text{true}\&(\text{fence–rel}, \lambda, \langle 2\rangle) \looparrowright E\}$$

(5) For $Conditional$, $h\&(\varepsilon, \lambda, \langle 2\rangle)$ and $\neg h\&(\varepsilon, \lambda, \langle 2\rangle)$ are used to produce the head normal form. That the action act being ε indicates that the evaluation h does not make any effect in the changes of any variable.

$$HF(\text{if } h \text{ then } P \text{ else } Q)$$
$$=_{df} [\!]\{h\&(\varepsilon, \lambda, \langle 2\rangle) \looparrowright P, \neg h\&(\varepsilon, \lambda, \langle 2\rangle) \looparrowright Q\}$$

(6) With regard to $Iteration$, its analysis is similar to that of $Conditional$.

$$HF(\text{while } h \text{ do } P)$$
$$=_{df} [\!]\{h\&(\varepsilon, \lambda, \langle 2\rangle) \looparrowright (P; \text{while } h \text{ do } P), \neg h\&(\varepsilon, \lambda, \langle 2\rangle) \looparrowright E\}$$

The definition of the head normal form for sequential and parallel composition is obtained by applying the corresponding sequential and parallel expansion laws.

C. Algebraic Laws

In this section, we study the algebraic laws for the C++11 memory model. Based on these laws, every program can be converted to a guarded choice.

Firstly, we focus on the sequential expansion laws. The program is transfered into a variety of configurations statement by statement. When the subsequent program Q comes to make sequential composition, it is only attached to the selected P_i.

(seq–1) Let $P = [\!]_{i\in I}\{h_i\&(act_i, tid_i, idx_i)[q_i] \looparrowright P'_i\}$
Then $P; Q = [\!]_{i\in I}\{h_i\&(act_i, tid_i, idx_i)[q_i] \looparrowright (P'_i; Q)\}$

So far, the configurations achieved above do not depend on each other. Now we need to make them form configuration sequences whose indices can reflect the program order. This operation is formalized by the law seq-2.

(seq–2) $h\&(act, tid, idx)[q] \looparrowright P'$
$=(h\&(act, tid, idx) \to q) \hookrightarrow (\langle 2\rangle^\wedge P')$

Except for the fetched configurations $h\&(act, tid, idx)[q]$, the following configurations are supposed to add a prefix $\langle 2\rangle$, which can be modeled by the function in the following.

$$\langle 2\rangle^\wedge P =_{df} \forall h\&(act, tid, idx) \in P \bullet$$
$$P[h\&(act, tid, \langle 2\rangle^\wedge idx)/h\&(act, tid, idx)]$$

Notation $P[u/v]$ denotes the replacement of v by u in P.

Example 3. Let $P =_{df} x := 1$ and $Q =_{df} y := 1$, where x and y are both global variables.

Using the laws seq-1 and seq-2, we can get the head normal form of the sequential program $P; Q$ in the following formalization. The transformation of the indices of the actions in P and Q is the same as that in Figure 4.

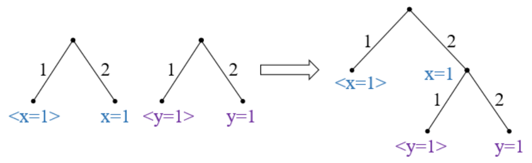


Fig. 4. The transformation of locations.

$$\begin{aligned}
HF(P) =& (\langle x=1\rangle, \lambda, \langle 1\rangle)[(x=1,\lambda,\langle 2\rangle)] \looparrowright E \\
HF(P;Q) =& (\langle x=1\rangle, \lambda, \langle 1\rangle)[(x=1,\lambda,\langle 2\rangle)] \looparrowright (E;Q) \\
=& (\langle x=1\rangle, \lambda, \langle 1\rangle)[(x=1,\lambda,\langle 2\rangle)] \looparrowright \\
& \qquad (\langle y=1\rangle, \lambda, \langle 1\rangle)[(y=1,\lambda,\langle 2\rangle)] \\
=& ((\langle x=1\rangle, \lambda, \langle 1\rangle) \to (x=1,\lambda,\langle 2\rangle)) \hookrightarrow \\
& \qquad \langle 2\rangle^\wedge((\langle y=1\rangle, \lambda, \langle 1\rangle) \to (y=1,\lambda,\langle 2\rangle)) \\
=& ((\langle x=1\rangle, \lambda, \langle 1\rangle) \to (x=1,\lambda,\langle 2\rangle)) \hookrightarrow \\
& \qquad ((\langle y=1\rangle, \lambda, \langle 2,1\rangle) \to (y=1,\lambda,\langle 2,2\rangle))
\end{aligned}$$

□

Law seq-3 is used to generate all the valid configuration sequences of any program. The first configuration c_{11} can always be scheduled. If we want to select the first c_{i1} after the operator $\hookrightarrow$, the corresponding condition d_i should be satisfied.

(seq–3)

$$\begin{aligned}
& (c_{11} \to c_{12} \to ...c_{1n_1}) \hookrightarrow ...(c_{m1} \to c_{m2} \to ...c_{mn_m}) \\
=& c_{11} \to \boxed{(c_{12} \to ...c_{1n_1})} \hookrightarrow ...(c_{m1} \to c_{m2} \to ...c_{mn_m}) \\
& [\!] \; ... \qquad \text{if } d_i \\
& [\!] \; c_{m1} \to (c_{11} \to c_{12} \to ...c_{1n_1}) \hookrightarrow ...(c_{m2} \to ...c_{mn_m}) \text{ if } d_m
\end{aligned}$$

The condition d_i indicates that the action in c_{i1} is a promise and all the configurations in front of c_{i1} are not of the release fence instructions and release writes. It is formalized as below.

$$d_i =_{df} \forall c \bullet \left(\begin{array}{l} \pi_1(c_{i1}) \text{ is in the form of } \langle x=e\rangle \wedge \\ \left(\begin{array}{l} \pi_3(c) < \pi_3(c_{i1}) \to \\ (\pi_1(c) \text{ is not in the form of fence-rel} \wedge \\ \pi_1(c) \text{ is not in the form of } x_{\text{rel}} = e) \end{array} \right) \end{array} \right)$$

The projection function $\pi_i (i \in \{1,2,3\})$ is defined to get the i-th element of the configuration, e.g., $\pi_1(act, tid, idx) = act$ and $\pi_3(act, tid, idx) = idx$.

Example 3: Continuation. Due to the constraint in d_i, the sequence $(\langle y=1\rangle, \lambda, \langle 2,1\rangle) \to ((\langle x=1\rangle, \lambda, \langle 1\rangle) \to (x=1,\lambda,\langle 2\rangle)) \hookrightarrow (y=1,\lambda,\langle 2,2\rangle)$ can only be converted into $(\langle y=1\rangle, \lambda, \langle 2,1\rangle) \to (\langle x=1\rangle, \lambda, \langle 1\rangle) \to (x=1,\lambda,\langle 2\rangle) \to (y=1,\lambda,\langle 2,2\rangle)$. All the configuration sequences of the sequential program $P;Q$ are exhibited as below.

$$\begin{aligned}
HF(P;Q) =& ((\langle x=1\rangle, \lambda, \langle 1\rangle) \to (x=1,\lambda,\langle 2\rangle)) \hookrightarrow \\
& \qquad ((\langle y=1\rangle, \lambda, \langle 2,1\rangle) \to (y=1,\lambda,\langle 2,2\rangle))
\end{aligned}$$

$$= \left(\begin{array}{l} (\langle x=1\rangle, \lambda, \langle 1\rangle) \to (x=1,\lambda,\langle 2\rangle) \hookrightarrow \\ \qquad ((\langle y=1\rangle, \lambda, \langle 2,1\rangle) \to (y=1,\lambda,\langle 2,2\rangle)) \\ [\!] \\ (\langle y=1\rangle, \lambda, \langle 2,1\rangle) \to ((\langle x=1\rangle, \lambda, \langle 1\rangle) \to \\ \qquad (x=1,\lambda,\langle 2\rangle)) \hookrightarrow (y=1,\lambda,\langle 2,2\rangle) \end{array} \right)$$

$$= \left(\begin{array}{l} (\langle x=1\rangle, \lambda, \langle 1\rangle) \to (x=1,\lambda,\langle 2\rangle) \to \\ \qquad (\langle y=1\rangle, \lambda, \langle 2,1\rangle) \to (y=1,\lambda,\langle 2,2\rangle) \\ [\!] \\ (\langle x=1\rangle, \lambda, \langle 1\rangle) \to (\langle y=1\rangle, \lambda, \langle 2,1\rangle) \to \\ \qquad (x=1,\lambda,\langle 2\rangle) \to (y=1,\lambda,\langle 2,2\rangle) \\ [\!] \\ (\langle y=1\rangle, \lambda, \langle 2,1\rangle) \to (\langle x=1\rangle, \lambda, \langle 1\rangle) \to \\ \qquad (x=1,\lambda,\langle 2\rangle) \to (y=1,\lambda,\langle 2,2\rangle) \end{array} \right)$$

□

Next, we continue to study the parallel expansion law. Our parallel model can be explained as the variant of the interleaving model, which is based on the configuration sequences produced by the sequential expansion laws above.

(par–1)

Let $P = [\!]_{i\in I}\{h_i \& (act_i, tid_i, idx_i) \to P'_i\}$

$\quad Q = [\!]_{j\in J}\{h_j \& (act_j, tid_j, idx_j) \to Q'_j\}$

Then $P||Q = [\!]_{i\in I}\{h_i \& (act_i, \langle 1\rangle^\wedge tid_i, idx_i) \to (P'_i||Q)\}$

$\qquad [\!][\!]_{j\in J}\{h_j \& (act_j, \langle 2\rangle^\wedge tid_j, idx_j) \to (P||Q'_j)\}$

If the configuration in the left branch is chosen, the prefix $\langle 1\rangle$ is attached to the corresponding thread id tid_i. Otherwise, $\langle 2\rangle$ is added to tid_j. This results in the difference between the traditional interleaving semantics and the one introduced here.

Example 4. Consider the parallel program $P||Q$, where $P =_{df} x := 1$ and $Q =_{df} y := 1$. Now, we calculate all the configuration sequences of it, with the application of the proposed laws.

$$HF(P||Q)$$

$$= \left(\begin{array}{l} (\langle x=1\rangle, \langle 1\rangle, \langle 1\rangle) \to \\ ((x=1,\lambda,\langle 2\rangle)||((\langle y=1\rangle, \lambda, \langle 1\rangle) \to (y=1,\lambda,\langle 2\rangle))) \\ [\!] \\ (\langle y=1\rangle, \langle 2\rangle, \langle 1\rangle) \to \\ (((\langle x=1\rangle, \lambda, \rangle 1\rangle) \to (x=1,\lambda,\langle 2\rangle))||(y=1,\lambda,\langle 2\rangle)) \end{array} \right)$$

$$= \left(\begin{array}{l} (\langle x=1\rangle, \langle 1\rangle, \langle 1\rangle) \to (x=1,\langle 1\rangle,\langle 2\rangle) \to \\ \qquad (\langle y=1\rangle, \langle 2\rangle, \langle 1\rangle) \to (y=1,\langle 2\rangle,\langle 2\rangle) \\ [\!] \\ (\langle x=1\rangle, \langle 1\rangle, \langle 1\rangle) \to (\langle y=1\rangle, \langle 2\rangle, \langle 1\rangle) \to \\ \qquad (x=1,\langle 1\rangle,\langle 2\rangle) \to (y=1,\langle 2\rangle,\langle 2\rangle) \\ [\!] \\ (\langle x=1\rangle, \langle 1\rangle, \langle 1\rangle) \to (\langle y=1\rangle, \langle 2\rangle, \langle 1\rangle) \to \\ \qquad (y=1,\langle 2\rangle,\langle 2\rangle) \to (x=1,\langle 1\rangle,\langle 2\rangle) \\ [\!] \\ (\langle y=1\rangle, \langle 2\rangle, \langle 1\rangle) \to (\langle x=1\rangle, \langle 1\rangle, \langle 1\rangle) \to \\ \qquad (x=1,\langle 1\rangle,\langle 2\rangle) \to (y=1,\langle 2\rangle,\langle 2\rangle) \\ [\!] \\ (\langle y=1\rangle, \langle 2\rangle, \langle 1\rangle) \to (\langle x=1\rangle, \langle 1\rangle, \langle 1\rangle) \to \\ \qquad (y=1,\langle 2\rangle,\langle 2\rangle) \to (x=1,\langle 1\rangle,\langle 2\rangle) \\ [\!] \\ (y=1,\langle 2\rangle,\langle 2\rangle) \to (y=1,\langle 2\rangle,\langle 2\rangle) \to \\ \qquad (\langle x=1\rangle, \langle 1\rangle, \langle 1\rangle) \to (x=1,\langle 1\rangle,\langle 2\rangle) \end{array} \right)$$

□

IV. Implementation of Algebraic Laws

Rewriting logic has been introduced as a general semantic and logical framework. Many applications are implemented in the rewrite engine Maude [13] and have revealed inspiring results.

In this section, we formalize the algebraic laws proposed before in the Maude system. We first list some notations used in our implementation in Table I.

TABLE I
NOTATIONS USED IN THE IMPLEMENTATION.

Notations	Descriptions
g_x	Global variables used in relaxed accesses
grel_x	Global variables in release writes
gacq_x	Global variables in acquire reads
l_x	Local variables
bl	Program statements
al	Configurations after law seq-1
cl	Configurations after law seq-2, configurations of the actions (apart from promises) after law seq-3
dl	Configurations of promises after law seq-3
pl	Configurations after law par-1

For the law seq-1 whose functionality is to transfer the programs into configurations, if the first statement is a relaxed write, two configurations will be produced in order. Otherwise, only one configuration whose third parameter (i.e., its location) is $\langle 2\rangle$ will be generated. Here, the parameter $NS1$ in *bl* is acting as a placeholder merely.

```
rl bl(g_x := e, T1, NS1) ; b =>
(al(g_x := e, T1, 1)
[al(g_x := e, T1, 2)]) +> b .
rl bl(l_x := e, T1, NS1) ; b =>
al(l_x := e, T1, 2) +> b .
rl bl(ra_x := e, T1, NS1) ; b =>
al(ra_x := e, T1, 2) +> b .
rl bl(fence-rel,T1,NS1) ; b =>
al(fence-rel,T1,2) +> b .
rl bl(fence-acq,T1,NS1) ; b =>
al(fence-acq,T1,2) +> b .
```

Note that, the operator ; is used to connect statements in programs, while the operator $+>$ corresponding to $\rightarrow$ is to link configurations. We will not bring in any other operator in configuration sequences in Maude, because the effect of $\looparrowright$ and $\hookrightarrow$ can be reflected by the notations *al* and *cl* respectively.

When implementing the law seq-2, we assume that the index of $cl(s1, T1, NS1)$ has been updated. Then the operation upon $(al(s2, T1, NS2\ 1)[al(s2, T1, NS2\ 2)])$ or $al(s2, T1, NS2\ 2)$ after it is to replace $NS2$ with the updated $NS1$.

```
rl cl(s1,T1,NS1) +>
(al(s2,T1,NS2 1)[al(s2,T1,NS2 2)]) +> b
=> cl(s1,T1,NS1) +> cl(s2,T1,NS1 1) +>
cl(s2,T1,NS1 2) +> b .
rl cl(s1,T1,NS1) +>
(al(s2,T1,NS2 1)[al(s2,T1,NS2 2)])
=> cl(s1,T1,NS1) +> cl(s2,T1,NS1 1) +>
cl(s2,T1,NS1 2) .
rl cl(s1,T1,NS1) +> al(s2,T1,NS2 2) +> b
=> cl(s1,T1,NS1) +> cl(s2,T1,NS1 2) +> b.
rl cl(s1,T1,NS1) +> al(s2,T1,NS2 2) =>
cl(s1,T1,NS1) +> cl(s2,T1,NS1 2) .
```

The reordering in PS is reflected by the appearance of promises, and they can be made anywhere. However, the release fence instructions and release writes bind the reordering to some extent. They can be formalized and explained by the law seq-3. In the rewriting engine Maude, for a configuration $cl(s1, T1, NS1\ 1)$ of a promise, there mainly are four cases:

1) If the configuration sequence $c1$ in front of it does not have any configuration which is abstracted from the release fence instruction and release write, the promise can be put in the head.
2) The promise can be put between the sequences $c1$ and $c2$. The constraint is that $c2$ does not have any configuration of the release fence and release write.
3) Provided that the promise is in the head originally, it does not need to move.
4) The promise can stay in its place regardless of the conditions above, shown by lines 9 and 10 as below.

```
1  crl c1 +> cl(s1,T1,NS1 1) +> c2 =>
2  dl(s1,T1,NS1 1) +> c1 +> c2
3  if has-no-rel(c1) .
4  crl c1 +> c2 +> cl(s1,T1,NS1 1) +> c3
5  => c1 +> dl(s1,T1,NS1 1) +> c2 +> c3
6  if has-no-rel(c2) .
7  rl cl(s1,T1,NS1 1) +> c1 =>
8  dl(s1,T1,NS1 1) +> c1 .
9  rl c1 +> cl(s1,T1,NS1 1) +> c2 =>
10 c1 +> dl(s1,T1,NS1 1) +> c2 .
```

The interleaving semantics requires that the operations from the same thread appear in the order specified by their own program order. Here, we implement the interleaving through putting the operations in the right branch T_2 into the left one T_1. Then, if the sequence $b1$ has some configurations with thread id being that of T_2, and $c1$ does not, the configuration $pl(s1, T1, N1)$ from T_2 can only be put after $b1$.

```
rl inter(b1 +> c1, pl(s1,T1,N1) +>
cl(s2,T2,N2) +> b3) =>
inter(b1 +> pl(s1,T1,N1) +>
c1, pl(s2,T2,N2) +> b3) .
rl inter(b1, pl(s1,T1,N1) +>
cl(s2,T2,N2) +> b3) =>
inter(b1 +> pl(s1,T1,N1),
pl(s2,T2,N2) +> b3) .
```

Example 5. We use the rewriting engine Maude to produce all the possible configuration sequences of the parallel program $(x := 1; y_{\text{rel}} := 1)|||(a := y_{\text{acq}}; b := x)$ in Figure 7 (page 9). □

V. Definition of Read Mechanism

In this section, we introduce the read mechanism, which can be applied in the configuration sequences of C++11 generated from the algebraic laws in previous section. Then all the valid executions of any program can be produced. The framework of the read mechanism is exhibited in Figure 5. Also, the method used here is able to throw away the concept of time stamp and various relations such as hb and rf relations in the traditional execution graphs of the C++11 memory model.

A. Overview of Read Function

For any variable x read in the configuration c, we know that the value should be provided by the sequence *seq* in front of c. The configuration of the initial write to x is always included in *seq*. Now, we present the read function r in detail. Above all, we need to check whether the variable read from is global or not. If yes, the function g is scheduled to execute. Otherwise, the function l is given to complete the following operations.

$$r(x, c, seq) =_{df} g(x, c, seq) \lhd x \in Globals \rhd l(x, c, seq)$$

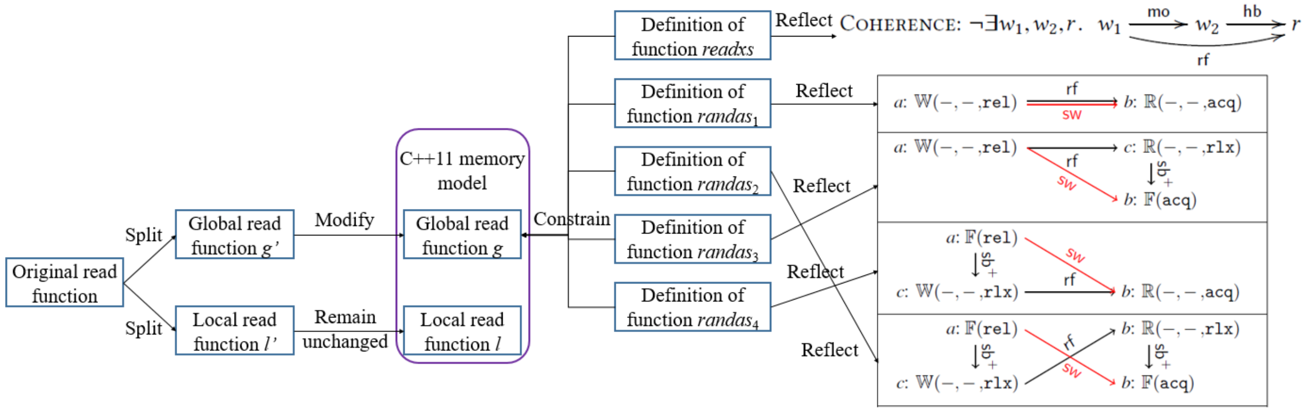

Original read function
Split
Split
Global read function g'
Local read function l'
Modify
Remain unchanged
C++11 memory model
Global read function g
Local read function l
Constrain
Definition of function readxs
Definition of function randas₁
Definition of function randas₂
Definition of function randas₃
Definition of function randas₄
Reflect
Reflect
Reflect
Reflect
Reflect
COHERENCE: ¬∃w₁, w₂, r. w₁ mo w₂ hb r rf
a: W(−,−,rel) rf sw b: R(−,−,acq)
a: W(−,−,rel) c: R(−,−,rlx) rf sw sb+ b: F(acq)
a: F(rel) sb+ c: W(−,−,rlx) sw rf b: R(−,−,acq)
a: F(rel) sb+ c: W(−,−,rlx) rf sw b: R(−,−,rlx) sb+ b: F(acq)

Fig. 5. Introduction to the read mechanism under C++11 (the right part is adapted from [6], [7]).

Here, $Globals$ is used to denote the set of all global variables. The notation $e \triangleleft h \triangleright f$ stands for e when the conditional judgment h is true, otherwise f.

B. Introduction to Function l

Traditionally, the read mechanism for any variable in the configuration sequence can be described: (1) the sequence is searched in reverse order, (2) and the private data can only be accessed by the one with the same thread id, (3) and the shared information are visible to all. The execution of function l still follows this principle.

When a thread T reads from a local variable x, it searches the sequence in front of it in reverse order. The local variables are not visible to other threads. Then, if such a configuration whose action is writing to x performed by T exists, the result of val acting on the action is returned.

$$l(x, c, seq) =_{df} \left(\begin{array}{c} val(\pi_1(lt(seq))) \\ \triangleleft \left(\begin{array}{c} \pi_2(c) = \pi_2(lt(seq)) \wedge \\ x \in var(\pi_1(lt(seq))) \end{array} \right) \triangleright \\ l(x, c, ft(seq)) \end{array} \right)$$

$$l(x, c, \varepsilon) =_{df} 0$$

Here, $lt(seq)$ stands for the last configuration of seq, and $ft(seq)$ is applied to denote the result of removing the last configuration in the sequence seq. Further, the notations $var(act)$ and $val(act)$ are used to denote the assigned variable and calculated value of the action act.

C. Introduction to Function g

Compared with the analysis of function l, the interesting features in the C++11 memory model will bring in some difficulties to the read operations upon global variables. Inspired by the promising semantics, the principle of function g is modified to be: we search the sequence in reverse order until we find a write to x contributed by the thread itself. Then, between the interval, the writes to x produced by the environment can also be read.

However, we find that the five constraints originating from the coherence axiom and the four ways to form synchronization (i.e., the occurrence of release/acquire accesses or release/acquire fence instructions) will narrow the scope of the function g's execution, which is presented in Figure 5.

For facilitating formalizing function g and these constraints, we firstly extend the configuration with another element $rinfo$. This parameter is used to record the information of the read operations, and composed of some quintuples in the form of $(rvar, arflag, tid, idx, rwflag)$, where:

- We use $rvar$ to stand for the variable read from.
- $arflag$ is equal to 1 when the action on $rvar$ is an acquire read, and it is 0 if a relaxed read.
- We use the parameter tid to denote the thread id of the write which provides the value of $rvar$.
- idx records the location of the mentioned write.
- The parameter $rwflag$ indicates whether the write is a release write or relaxed write. If it is a release write, $rwflag$ is 1, otherwise 0.

Example 6. Consider the parallel program $(x := 1; y_{\text{rel}} := 1) || (a := y_{\text{acq}}; b := x)$. Assume that $(x := 1; y_{\text{rel}} := 1)$ is running in T_1, while $(a := y_{\text{acq}}; b := x)$ is executing in T_2.

One configuration sequence of this program is presented in Figure 6, and we use the configuration of $a := y_{\text{acq}}$ to help to give the intuitive understanding of the parameter $rinfo$. As shown in Figure 6, if the acquire read from y gets the value 1 from the configuration before it, which is described by the dotted line in Figure 6, the quintuple $(y, 1, \langle 1 \rangle, \langle 2, 2 \rangle, 1)$ is added in $rinfo$, and the configuration of $a := y_{\text{acq}}$ is transferred into $(a = y_{\text{acq}}, \langle 2 \rangle, \langle 2 \rangle, \{(y, 1, \langle 1 \rangle, \langle 2, 2 \rangle, 1)\})$. Here, in order to exhibit the example clearly, the fourth element of each configuration is not added in Figure 6.

$\square$

Before presenting the execution of function g, we give the formal description of the constraints shown in Figure 5. Due to the space limitation, here we only provide the formalization and explanation of the first two functions $readxs$ and $randas_1$, discussed in the subsections Relaxed Atomics and Release/Acquire Accesses respectively. And the others are given in the appendix.

*1) **Relaxed Atomics:*** For two reads from the same memory location x in the same thread, when the former read gets a

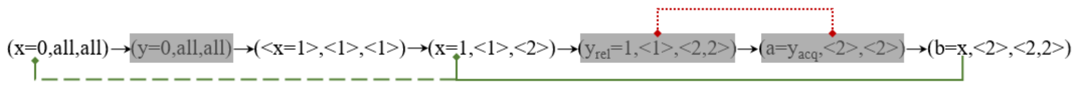


Fig. 6. The new mechanism applied in $(x := 1; y_{\text{rel}} := 1)||(a := y_{\text{acq}}; b := x)$.

value from a write w_1, the value of the latter one cannot be provided by another write w_2 whose appearance is earlier than that of w_1 in the sequence. This constraint can be achieved by the functions $readxs$ and $readx$ in the following.

$$readxs(x, c, seq') =_{df} \left(\begin{array}{l} \left(\begin{array}{l} readx(\pi_4(lt(seq')), \text{null}) \\ \lhd(x, \_, \_, \_, \_) \in \pi_4(lt(seq'))\rhd \\ readxs(x, c, ft(seq')) \end{array} \right) \\ \lhd\pi_2(c) = \pi_2(lt(seq'))\rhd \\ readxs(x, c, ft(seq')) \end{array} \right)$$

$$readxs(x, c, \varepsilon) =_{df} \{\}$$

For convenience, we say that the action in c is performed in T. If there exists another read from x contributed by T in seq', the thread id and location of the write which provides the value will be collected by $readxs$, through calling the function $readx$ as below. Note that, the sequence seq' which is composed of the sequence seq in front of c and the configuration c, instead of the single sequence seq, is used here. The purpose is to handle the special situation that there are more than one reads from the location x in one statement and the values of them are provided by different writes.

$$\begin{array}{l} readx(set, tuple) \\ =_{df} \left(\begin{array}{l} \left(\begin{array}{l} (\pi_3(tuple), \pi_4(tuple)) \cup \\ \quad readx(set \backslash tuple, rand(set \backslash tuple)) \\ \lhd\pi_1(tuple) = x\rhd \\ readx(set \backslash tuple, rand(set \backslash tuple)) \end{array} \right) \\ \lhd tuple \in set\rhd \\ readx(set, rand(set)) \end{array} \right) \\ readx(\{\}, tuple) =_{df} \{\} \end{array}$$

The notation $rand(set)$ is applied to select one element from the set set randomly. We also define the projection function $\pi_i (i \in \{1, 2, 3, 4\})$ to get the i-th element of the configuration, e.g., $\pi_1(act, tid, idx, rinfo) = act$.

*2) **Release/Acquire Accesses:*** Before we meet such a configuration of the write to x contributed by T, we find another configuration which is related to an acquire read produced by T, and the read gets the value from a release write by the parallel component. Consequently, a barrier may be established. Then the finishing point of the search process may be reduced to the write to x of the parallel component.

Based on the analysis above, we firstly study how to collect the ids of the threads which construct the synchronization with the one that produces the configuration c. The collection is realized by the three steps as below (we do not show the formal description of the first two steps, and it is similar to that of functions $readxs$ and $readx$):

- During the process of seeking the configuration of a write to x by T, the function $randas_1$ may catch some other configurations contributed by the process T.
- For any configuration caught by $randas_1$, the function $randa_1$ will check whether this configuration contains an acquire read and has established the synchronization with a release write produced by another thread, through the fourth element in this configuration. If yes, the elements tid and idx of the release write are collected.
- With regard to a certain thread, only the most recent barrier is required for the read operation. Then we filter the collected information in the last step. And it is formalized by the function $filter$ as below.

$$\begin{array}{l} filter(set) \\ =_{df} \left(\begin{array}{l} filter(set \backslash p) \\ \lhd\exists p \in set \bullet (\exists p' \in set \bullet (\pi_1(p) = \pi_1(p') \wedge \\ \qquad\qquad len(\pi_2(p)) < len(\pi_2(p'))))\rhd \\ set \end{array} \right) \end{array}$$

For two pairs in the set, if they have the same thread id, the pair whose length of index is smaller is deleted. Here, $len(idx)$ records the length of the parameter idx. The notation $set \backslash p$ is applied to remove p from set.

*3) **Execution of Function** g:* After knowing the constraints, the search process needs to be redefined. The search process will terminate if:

1) We find a configuration contributed by T, and it is of the write to x.
2) The configuration of a write to x by another thread, has been checked, and the pair composed of its thread id and index is included in the set generated by $readxs$.
3) The configuration we discover is produced by the environment of T, but its thread id tid belongs to the set we collect using the functions $randas_i$ $(i \in \{1, 2, 3, 4\})$ and $filter$. In addition, it is the configuration of a write to x and its index must be equal to or smaller than that recorded for tid in the mentioned set.

The configuration is of one fulfill operation in the first case, and of a promise in the left two cases. Then, the sequence we actually get values from is computed by the function $calseq$, whose detailed definition is not presented in this paper for the space limitation.

Based on the discussion about the scope of the search process, now, we collect the configurations which have the possibility to be read by x in c.

$$gs(x, c, seq) =_{df} \left(\begin{array}{l} lt(seq) \cup gs(x, c, ft(seq)) \\ \lhd \left(\begin{array}{l} \left(\begin{array}{l} (\pi_2(c) = \pi_2(lt(seq)) \wedge \\ lt(\pi_3(lt(seq))) = 2) \\ \vee \\ (\pi_2(c) != \pi_2(lt(seq)) \wedge \\ lt(\pi_3(lt(seq))) = 1) \end{array} \right) \\ \wedge var(\pi_1(lt(seq))) = x \end{array} \right) \rhd \\ gs(x, c, ft(seq)) \end{array} \right)$$

$$gs(x, c, \varepsilon) =_{df} \{\}$$

The configuration c' that makes the search process terminate is possible to be read. If it is performed by the thread T, it should be of one fulfill operation, while it must be of a promise if not by T. Moreover, in the interval between c' and c, if the configurations are contributed by the threads except for T, and their actions are promising to write to x, they are all possible. The possible configurations are all collected by the function gs. The sequence seq in gs is calculated by $calseq$.

Function gv selects one from gs randomly. val acting on $\pi_1(gv(x, c, seq))$ is the return value of g. After the read from x finishes, the fourth element in c should be updated. If the action on x is an acquire read, the second element of the added quintuple in $rinfo$ is 1, otherwise 0. Further, the fifth element is 1 if the value is gotten from a release write, otherwise 0.

$$gv(x, c, seq) = rand(gs(x, c, seq))$$
$$g(x, c, seq) = val(\pi_1(gv(x, c, seq)))$$
$$chg(x, c)$$
$$=_{df} \left(\begin{array}{l} \left(\begin{array}{l} c[\pi_4(c) \cup (x, 1, \pi_2(gv(x)), \pi_3(gv(x)), 1)/\pi_4(c)] \\ \lhd \pi_1(gv(x, c, seq)) \text{ is in the form of } x_{\text{rel}} = e \rhd \\ c[\pi_4(c) \cup (x, 1, \pi_2(gv(x)), \pi_3(gv(x)), 0)/\pi_4(c)] \end{array} \right) \\ \lhd \text{the action on } x \text{ is an acquire read } \rhd \\ \left(\begin{array}{l} c[\pi_4(c) \cup (x, 0, \pi_2(gv(x)), \pi_3(gv(x)), 1)/\pi_4(c)] \\ \lhd \pi_1(gv(x, c, seq)) \text{ is in the form of } x_{\text{rel}} = e \rhd \\ c[\pi_4(c) \cup (x, 0, \pi_2(gv(x)), \pi_3(gv(x)), 0)/\pi_4(c)] \end{array} \right) \end{array} \right)$$

Note that, there are some additional constraints for the read mechanism. One is that the read in the promise operation is always delayed, until the read in its corresponding fulfill operation finishes. And then the read in the mentioned promise uses the gotten value directly. Another is that the reads from different locations in the same sequence are still independent. When conducting the read from x, the configurations that are not related to x are all in shadow area in Figure 6.

D. Mechanization

The sequences of any program under C++11 are written to the .xml file with the code as below. Based on the .xml file, we can get all the possible executions, through the mechanization of the read mechanism with C++ language. Note that the gotten values may make a judgment become false, and the sequences containing such a judgment should be deleted.

```
maude −xml−log=relacq.xml
```

Example 7. Consider the parallel program $(x := 1; y_{\text{rel}} := 1)||(a := y_{\text{acq}}; b := x)$. When analyzing the sequence in Figure 6, there are mainly three situations when performing a relaxed read for the memory location x.

1) If y gets the value from the release write to y, which is described by the red line, x can be 1 provided by the configuration $(x = 1, \langle 1 \rangle, \langle 2 \rangle)$.
2) Otherwise, when reading from x, the return value can be 0 or 1.

Figure 7 (page 10) shows the three executions of the sequence in Figure 6, by implementing the read mechanism. □

VI. Conclusion and Future Work

The C and C++ languages introduced the relaxed-memory concurrency into the language specification for efficiency reasons in 2011. In this paper, a set of algebraic laws including sequential and parallel expansion laws has been investigated with the concept of the guarded choice. Therefore, the linearizability of the C++11 memory model is supported in our algebraic model. We have also implemented the algebraic laws in the rewriting engine Maude. Further, in order to present all the possible execution results of any program under C++11, we introduced a read mechanism, which can handle the relaxed and release/acquire accesses in C++11.

In the future, we will continue our work on the C++11 memory model. We would like to investigate the semantics linking theories of this memory model.

References

[1] Hóu, Z., Sanan, D., Tiu, A., Liu, Y., Hoa, K. C., Dong, J. S., An Isabelle/HOL Formalisation of the SPARC Instruction Set Architecture and the TSO Memory Model, Journal of Automated Reasoning, 2021, 65(4): 569-598.
[2] Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P., Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8, Proceedings of the ACM on Programming Languages, 2017, 2(POPL): 1-29.
[3] Batty, M. J., The C11 and C++ 11 concurrency model, University of Cambridge, 2015.
[4] Kang, J., Hur, C.K., Lahav, O., Vafeiadis, V., Dreyer, D., A promising semantics for relaxed-memory concurrency, ACM SIGPLAN Notices, 2017, 52(1): 175-189.
[5] Lee, S. H., Cho, M., Podkopaev, A., Chakraborty, S., Hur, C. K., Lahav, O., Vafeiadis, V., Promising 2.0: global optimizations in relaxed memory concurrency, Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020: 362-376.
[6] Mike, D., Mark B., Alexey G., Compositional Verification of Compiler Optimisations on Relaxed Memory, Programming Languages and Systems - 27th European Symposium on Programming, 2018: 14-20.
[7] He, M., Reasoning about C11 programs with fences and relaxed atomics, Teesside University, Middlesbrough, UK, 2018.
[8] Hoare, C. A. R., He, J., Unifying theories of programming, Englewood Cliffs: Prentice Hall, 1998.
[9] Stoy, J. E., Denotational semantics: the Scott-Strachey approach to programming language theory, MIT press, 1981.
[10] Hoare, C. A. R., Hayes, I. J., He, J., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sorensen, I. H., Spivey, J. M., Sufrin, B. A., Laws of programming, Communications of the ACM, 1987, 30(8): 672-686.
[11] Hoare, C. A. R., He, J., Sampaio, A., Normal Form Approach to Compiler Design, Acta Informatica, 1993, 30(8): 701-739.
[12] Xiao, L., Zhu, H., He, M., Qin, S., Traces Semantics for C++11 Memory Model, submitted to TASE 2022.
[13] Clavel, M., Dur´an, F., Eker, S., Lincoln, P., Mart´ı-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude, Lecture Notes in Computer Science, vol. 4350. Springer (2007).

Appendix

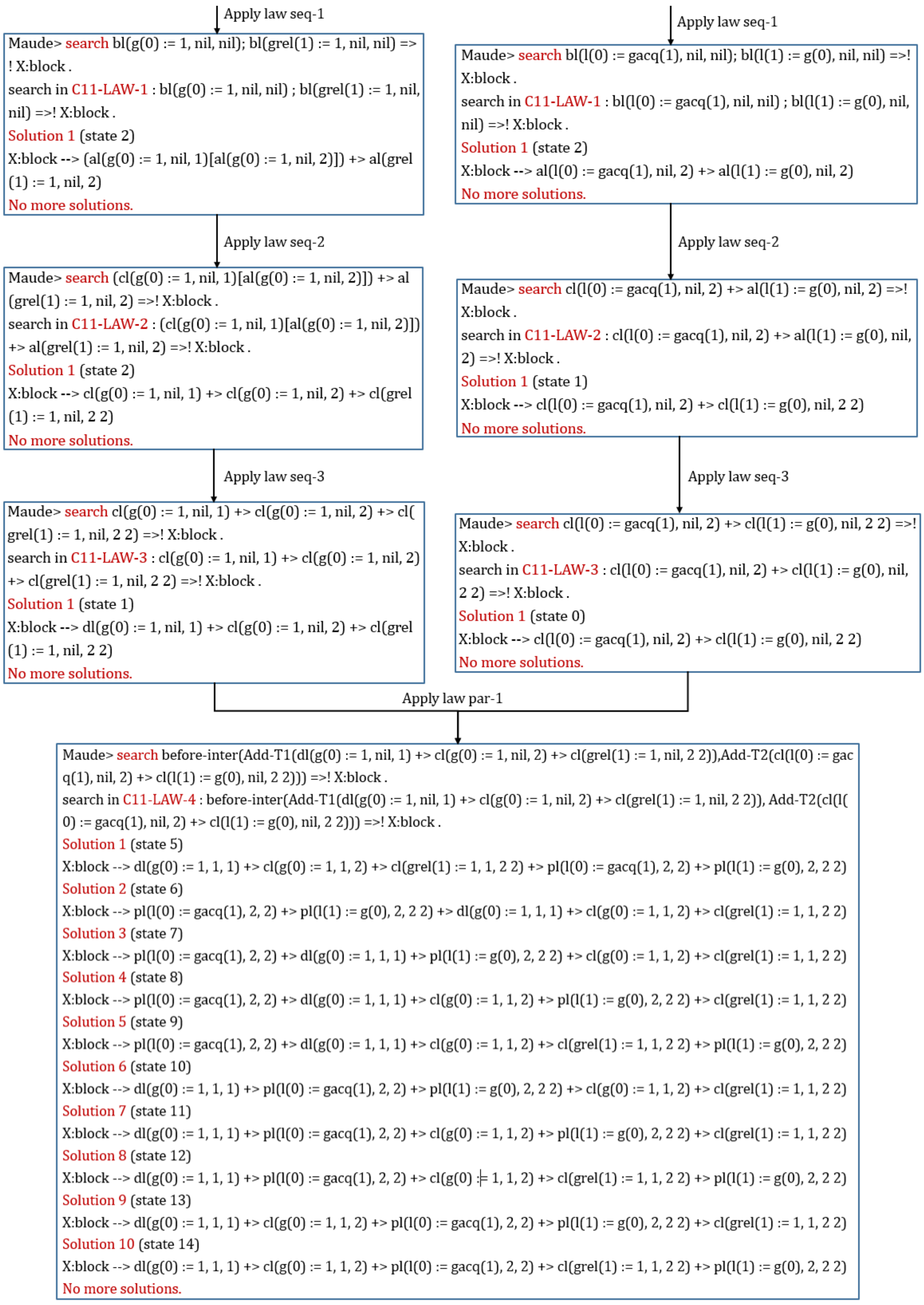
Here, we will introduce the last three constraints as below.

A. Introduction to $randas_2$

The barriers can be constructed in another way. If a read before an acquire fence reads from a write after a release fence, and the two fences synchronize, any write before the release fence must be visible to any read after the acquire fence.

Before giving the detailed formalization and explanation, we use Example 8 to provide the intuitive understanding of it.

Example 8. Consider the parallel program $(x := 1; \text{fence-rel}; y := 1)||(a := y; \text{fence-acq}; b := x)$. Consider one configuration sequence of the program above in Figure 8. If the read from y has the value 1, which is offered by the configuration of a promise made by the thread with thread id



```
Maude> search bl(g(0) := 1, nil, nil); bl(grel(1) := 1, nil, nil) =>
! X:block .
search in C11-LAW-1 : bl(g(0) := 1, nil, nil) ; bl(grel(1) := 1, nil,
nil) =>! X:block .
Solution 1 (state 2)
X:block --> (al(g(0) := 1, nil, 1)[al(g(0) := 1, nil, 2)]) +> al(grel
(1) := 1, nil, 2)
No more solutions.
```

```
Maude> search (cl(g(0) := 1, nil, 1)[al(g(0) := 1, nil, 2)]) +> al
(grel(1) := 1, nil, 2) =>! X:block .
search in C11-LAW-2 : (cl(g(0) := 1, nil, 1)[al(g(0) := 1, nil, 2)])
+> al(grel(1) := 1, nil, 2) =>! X:block .
Solution 1 (state 2)
X:block --> cl(g(0) := 1, nil, 1) +> cl(g(0) := 1, nil, 2) +> cl(grel
(1) := 1, nil, 2 2)
No more solutions.
```

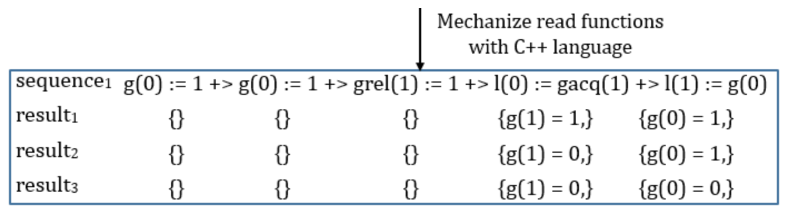
```
Maude> search cl(g(0) := 1, nil, 1) +> cl(g(0) := 1, nil, 2) +> cl(
grel(1) := 1, nil, 2 2) =>! X:block .
search in C11-LAW-3 : cl(g(0) := 1, nil, 1) +> cl(g(0) := 1, nil, 2)
+> cl(grel(1) := 1, nil, 2 2) =>! X:block .
Solution 1 (state 1)
X:block --> dl(g(0) := 1, nil, 1) +> cl(g(0) := 1, nil, 2) +> cl(grel
(1) := 1, nil, 2 2)
No more solutions.
```

```
Maude> search bl(l(0) := gacq(1), nil, nil); bl(l(1) := g(0), nil, nil) =>!
X:block .
search in C11-LAW-1 : bl(l(0) := gacq(1), nil, nil) ; bl(l(1) := g(0), nil,
nil) =>! X:block .
Solution 1 (state 2)
X:block --> al(l(0) := gacq(1), nil, 2) +> al(l(1) := g(0), nil, 2)
No more solutions.
```

```
Maude> search cl(l(0) := gacq(1), nil, 2) +> al(l(1) := g(0), nil, 2) =>!
X:block .
search in C11-LAW-2 : cl(l(0) := gacq(1), nil, 2) +> al(l(1) := g(0), nil,
2) =>! X:block .
Solution 1 (state 1)
X:block --> cl(l(0) := gacq(1), nil, 2) +> cl(l(1) := g(0), nil, 2 2)
No more solutions.
```

```
Maude> search cl(l(0) := gacq(1), nil, 2) +> cl(l(1) := g(0), nil, 2 2) =>!
X:block .
search in C11-LAW-3 : cl(l(0) := gacq(1), nil, 2) +> cl(l(1) := g(0), nil,
2 2) =>! X:block .
Solution 1 (state 0)
X:block --> cl(l(0) := gacq(1), nil, 2) +> cl(l(1) := g(0), nil, 2 2)
No more solutions.
```

```
Maude> search before-inter(Add-T1(dl(g(0) := 1, nil, 1) +> cl(g(0) := 1, nil, 2) +> cl(grel(1) := 1, nil, 2 2)),Add-T2(cl(l(0) := gac
q(1), nil, 2) +> cl(l(1) := g(0), nil, 2 2))) =>! X:block .
search in C11-LAW-4 : before-inter(Add-T1(dl(g(0) := 1, nil, 1) +> cl(g(0) := 1, nil, 2) +> cl(grel(1) := 1, nil, 2 2)), Add-T2(cl(l(
0) := gacq(1), nil, 2) +> cl(l(1) := g(0), nil, 2 2))) =>! X:block .
Solution 1 (state 5)
X:block --> dl(g(0) := 1, 1, 1) +> cl(g(0) := 1, 1, 2) +> cl(grel(1) := 1, 1, 2 2) +> pl(l(0) := gacq(1), 2, 2) +> pl(l(1) := g(0), 2, 2 2)
Solution 2 (state 6)
X:block --> pl(l(0) := gacq(1), 2, 2) +> pl(l(1) := g(0), 2, 2 2) +> dl(g(0) := 1, 1, 1) +> cl(g(0) := 1, 1, 2) +> cl(grel(1) := 1, 1, 2 2)
Solution 3 (state 7)
X:block --> pl(l(0) := gacq(1), 2, 2) +> dl(g(0) := 1, 1, 1) +> pl(l(1) := g(0), 2, 2 2) +> cl(g(0) := 1, 1, 2) +> cl(grel(1) := 1, 1, 2 2)
Solution 4 (state 8)
X:block --> pl(l(0) := gacq(1), 2, 2) +> dl(g(0) := 1, 1, 1) +> cl(g(0) := 1, 1, 2) +> pl(l(1) := g(0), 2, 2 2) +> cl(grel(1) := 1, 1, 2 2)
Solution 5 (state 9)
X:block --> pl(l(0) := gacq(1), 2, 2) +> dl(g(0) := 1, 1, 1) +> cl(g(0) := 1, 1, 2) +> cl(grel(1) := 1, 1, 2 2) +> pl(l(1) := g(0), 2, 2 2)
Solution 6 (state 10)
X:block --> dl(g(0) := 1, 1, 1) +> pl(l(0) := gacq(1), 2, 2) +> pl(l(1) := g(0), 2, 2 2) +> cl(g(0) := 1, 1, 2) +> cl(grel(1) := 1, 1, 2 2)
Solution 7 (state 11)
X:block --> dl(g(0) := 1, 1, 1) +> pl(l(0) := gacq(1), 2, 2) +> cl(g(0) := 1, 1, 2) +> pl(l(1) := g(0), 2, 2 2) +> cl(grel(1) := 1, 1, 2 2)
Solution 8 (state 12)
X:block --> dl(g(0) := 1, 1, 1) +> pl(l(0) := gacq(1), 2, 2) +> cl(g(0) := 1, 1, 2) +> cl(grel(1) := 1, 1, 2 2) +> pl(l(1) := g(0), 2, 2 2)
Solution 9 (state 13)
X:block --> dl(g(0) := 1, 1, 1) +> cl(g(0) := 1, 1, 2) +> pl(l(0) := gacq(1), 2, 2) +> pl(l(1) := g(0), 2, 2 2) +> cl(grel(1) := 1, 1, 2 2)
Solution 10 (state 14)
X:block --> dl(g(0) := 1, 1, 1) +> cl(g(0) := 1, 1, 2) +> pl(l(0) := gacq(1), 2, 2) +> cl(grel(1) := 1, 1, 2 2) +> pl(l(1) := g(0), 2, 2 2)
No more solutions.
```


Mechanize read functions
with C++ language

sequence₁	g(0) := 1 +>	g(0) := 1 +>	grel(1) := 1 +>	l(0) := gacq(1) +>	l(1) := g(0)
result₁	{}	{}	{}	{g(1) = 1,}	{g(0) = 1,}
result₂	{}	{}	{}	{g(1) = 0,}	{g(0) = 1,}
result₃	{}	{}	{}	{g(1) = 0,}	{g(0) = 0,}

Fig. 7. Execution results of algebraic laws and read mechanism.

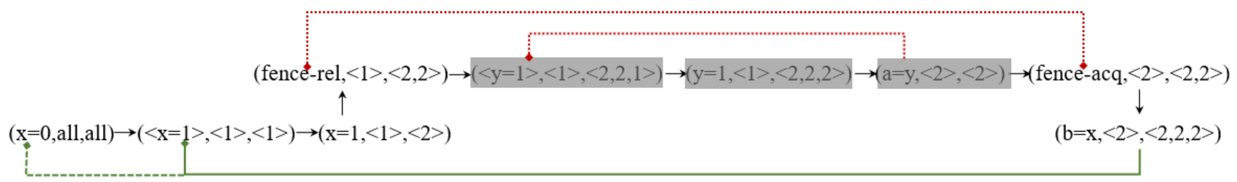


Fig. 8. The new mechanism applied in $(x := 1; \text{fence-rel}; y := 1)||(a := y; \text{fence-acq}; b := x)$.

being $\langle 1\rangle$, the read from x can only be provided by the solid line. It means that the previous write to x by the thread whose id is $\langle 2\rangle$ cannot be accessed. The notation "all" denotes any thread id and we do not care about the index of the initial write.

□

Then, we explore how to collect the information of these barriers in the search procedure. Different from the release/acquire accesses, here, at first, we need to obtain the sequence from the configuration of the write to x to the configuration c, which are both contributed by the thread T, through the function itv as below.

$$itv(x,c,seq,seq') =_{df} \left(\begin{array}{l} seq' - ft(seq) \\ \lhd \pi_2(c) = \pi_2(lt(seq)) \wedge var(\pi_1(lt(seq))) = x \rhd \\ itv(x,c,ft(seq),seq') \end{array}\right)$$

$$itv(x,c,\varepsilon,seq') =_{df} \varepsilon$$

The reason for this operation is that if the two fences synchronize (specially, the acquire fence is issued by T) in the sequence achieved by the function itv, the scope of the execution of function g will be reduced.

The judgment of whether such a synchronization is established, will be separated into two steps, illustrated by functions $randas_2$ and $randa_2$.

$$randas_2(x,c,seq,seq') =_{df} \left(\begin{array}{l} \left(\begin{array}{l} randa_2(\pi_4(lt(seq)), \text{null}, seq') \\ \qquad \cup randas_2(x,c,ft(seq),seq') \\ \lhd \left(\begin{array}{l} apd(lt(seq), (\text{fence-acq}, \pi_2(c), \\ \qquad idx, rinfo)) \preccurlyeq seq' \\ \wedge len(idx) < len(\pi_3(c)) \end{array}\right) \rhd \\ randas_2(x,c,ft(seq),seq') \end{array}\right) \\ \lhd \pi_2(c) = \pi_2(lt(seq)) \rhd \\ randas_2(x,c,ft(seq),seq') \end{array}\right)$$

$$randas_2(x,c,\varepsilon,seq') =_{df} \{\}$$

In function $randas_2$, seq and seq' both indicate the sequence calculated by the function itv. When seeking the sequence seq, function $randas_2$ may find such a sub-sequence whose start point is a common configuration and the end point is the configuration of an acquire fence, which is denoted by the operator "$\preccurlyeq$" and the function apd. In addition, the mentioned two configurations are both produced by T and in front of c.

Then, function $randa_2$ is required to find another sub-sequence starting from the configuration of a release fence, and ending at the configuration of a write to an arbitrary global variable x. The two configurations are both contributed by another thread T'. The other two requirements should also be satisfied: 1) the sub-sequence in $randa_2$ should be before that in $randas_2$, 2) the information of the write to x is included in the fourth element of the common configuration mentioned in $randas_2$.

$$randa_2(set,tuple,seq) =_{df} \left(\begin{array}{l} \left(\begin{array}{l} (\pi_3(tuple), \pi_4(tuple)) \cup \\ \qquad randa_2(set \backslash tuple, rand(set \backslash tuple), seq) \\ \lhd \left(\begin{array}{l} apd((\text{fence-rel}, tid, idx, rinfo), \\ \qquad (act, tid, idx', rinfo)) \preccurlyeq seq \\ \wedge len(idx) < len(idx') \wedge \\ tid = \pi_3(tuple) \wedge idx' = \pi_4(tuple) \end{array}\right) \rhd \\ randa_2(set \backslash tuple, rand(set \backslash tuple), seq) \end{array}\right) \\ \lhd tuple \in set \rhd \\ randa_2(set, rand(set), seq) \end{array}\right)$$

$$randa_2(\{\}, tuple, seq) =_{df} \{\}$$

When passing all the checks, the pair consisting of the thread id and index of the write will be collected. The operation filtering the redundant pairs is the same to the one introduced in release/acquire accesses.

B. Introduction to $randas_3$ and $randas_4$

The formalization and explanation of the other two ways to form synchronization are similar to $randas_2$. The only difference is that the former requires the configurations of a release fence by T', a write to x by T' and an acquire read from x by T (a release write to x by T', a read from x by T and an acquire fence by T) to occur in the sub-sequence in order.