Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 304 (2014) 167–182

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Rewriting Semantics and Analysis of Concurrency Features for a C-like Language

Traian Florin Șerbănuță[1,2]

Department of Computer Science
University of Bucharest

Abstract

This paper shows how one can easily transform $\mathbb{K}$ definitions of programming languages into runtime verification tools. To increase the confidence that these runtime verification tools can be used for testing real-world programs, the paper uses KernelC, a subset of the C programming language containing functions, memory allocation, pointer arithmetic, and input/output, which can be used to execute and test real C programs. KernelC is extended with threads and synchronization constructs, and two concurrent semantics are derived from its sequential semantics. The first semantics, defining a sequentially consistent memory model, can be easily transformed into a runtime verification tool for checking datarace and deadlock freeness. The second semantics defines in a relatively minimal fashion a relaxed memory model based on the x86-TSO memory model. By exploring the executions of an implementation of Peterson's mutual exclusion algorithm for both definitions, it is shown that the algorithm guarantees mutual exclusion for the sequentially consistent model, but cannot guarantee it for the relaxed model, but also that by allowing fence operations in the language, the algorithm can be fixed and proven correct for the TSO model, too.

Keywords: Runtime verification, tools, datarace, deadlock, Peterson's mutual exclusion algoriithm

1 Introduction

This paper offers a glimpse on the process of using formal definitions of programming languages as testing and analysis tools. We argue here that $\mathbb{K}$ [21,22] definitions can be used to test and analyze executions of programs written in real-life languages either directly or by extending them to become runtime analysis tools.

The rewriting logic representation of $\mathbb{K}$ definitions gives them access to the arsenal of generic tools for rewriting logic available through the Maude rewrite engine [5]: state space exploration, LTL model checking, inductive theorem proving, and so on. This collection of analysis tools is by itself enough to provide more information about the behaviors of a program than one would get by simply testing the program using an interpreter or a compiler for that language. Nevertheless, the effort of defining

[1] This work was supported by Contract 161/15.06.2010, SMISCSNR 602-12516 (DAK).

[2] Email: traian.serbanuta@fmi.unibuc.ro

http://dx.doi.org/10.1016/j.entcs.2014.05.009

1571-0661/© 2014 Elsevier B.V. All rights reserved.

the semantics pays back in more than just one way: by relatively few alterations to the definition, one can use the same generic tools to obtain type checkers and type inferencers [9], static policy checking tools [13,14], runtime verification tools [20], and even Hoare-like program verification tools [19].

Contributions

In this paper we focus on analyzing concurrency aspects of programming languages using the 𝕂 framework and show that runtime verification tools for dataraces and deadlocks can easily be obtained by slightly adjusting the definition of a language. To stress the "real-life" aspect, we choose as our running example a subset of the C programming language, named KernelC, and use these extensions to find and fix concurrent problems in KernelC programs.

Moreover, we show how one can obtain an x86-TSO-like [15] relaxed memory model for KernelC with minimal effort and use it to test the differences between various memory consistency models by exploring the possible behaviors of program executions under both models. Being able to analyze the behavior of concurrent programs under various memory consistency models is very important both in the early stages of language design, when one can decide how to enforce memory consistency, but also once a language is already in use, at it allows detecting and solving problems in programs written in that language.

This paper only provides a proof of concept that one can derive such tools directly from the language definition; however, we see no major impediment for building more efficient tools based on the same ideas.

To contain the size of the paper, we assume the user is already familiar with the 𝕂 framework [22,21], including writing, executing, and exploring 𝕂 definitions using the 𝕂 tool [23].

The remainder of this paper is organized as follows. Section 2 presents the syntax of KernelC. Section 3 presents the complete semantics for its purely sequential constructs and a simple sequentially consistent semantics for the concurrent part. Section 4 shows how KernelC can be used to explore behaviors of programs and can be adjusted for checking whether the executions of a program are datarace and deadlock free. Section 5 defines an alternative semantics for memory accesses and concurrency constructs based on a relaxed memory model inspired from the x86-TSO memory model [15] and shows that the differences between this model and the sequential consistent one can be tested using the available tools. Section 6 concludes.

Related Work

In addition to the ideas described in this paper, KernelC has already been used to show how one can easily obtain a runtime verification tool for *strong* memory safety [20] directly from the semantics, or integrated into a rewriting-based program verification tool [18] based on matching logic, a new verification logic based on 𝕂 [19]. While in this paper we use KernelC, the results that we present have been adapted and applied to a comprehensive 𝕂 definition of the C language [8,7].

Due to the complexity of reasoning under relaxed memory models and the assumption most programmers do about the executions being sequentially consistent, there have been many recent research efforts of detecting non-sequentially-consistent executions [16,1,2,3,11,12,4]. Our approach is different from the others in the sense that it is based on the formal definition of the language being analyzed and directly derived from it. Although still in an incipient phase, it shows that rewriting-based definitions can be used to obtain tools for programming languages.

2 KernelC Syntax

```
Exp ::= #Int | #Id | DeclId | PointerId
      | Exp + Exp [strict] | Exp - Exp [strict]
      | Exp == Exp [strict] | Exp <= Exp [strict]
      | ! Exp | Exp && Exp | Exp ? Exp : Exp
      | (int*)malloc(Exp *sizeof(int)) [strict] | * Exp
      | Exp [ Exp ]
      | Exp = Exp [strict(2)]
      | #Id ( Exps ) [strict(2)]
      | printf("%d;", Exp ) [strict]
Stmt ::= {} | { StmtList }
      | Exp ; [strict]
      | if( Exp ) Stmt
      | if( Exp ) Stmt else Stmt [strict(1)]
      | while( Exp ) Stmt
      | DeclId DeclIds { StmtList }
      | return Exp ; [strict]

StmtList ::= Stmt | StmtList StmtList
Pgm ::= StmtList
DeclId ::= int Exp | void PointerId
PointerId ::= #Id | * PointerId [strict]
#Id ::= main
```

Fig. 1. Syntax of KernelC— the sequential fragment

Fig. 1 describes the syntax of the sequential fragment of KernelC using a BNF-like notation. The syntax has been kept as close to the C syntax as possible to allow a reasonably large class of C programs to be parsed and executed with the KernelC definition. Nevertheless, the syntax is quite small, covering only 33 constructs of the C language. Function declarations consist of a *DeclId*, that is, a typed identifier, followed by a list of *DeclId*s (which should be surrounded by parentheses, although not required), and then by the body of the function. The statements allowed by KernelC are pretty simple, from the expression statement, to the block, conditional, and loop statements.

2.1 Extending the syntax with threads

The basic syntax of KernelC is extended with a couple of multi-threading primitives like thread creation, lock-synchronization, and thread-join. To keep things simple, we adopt a very restricted set of synchronization primitives and a syntax which, while not resembling the syntax proposed in the new C standard, it is much easier to work with in our model.

Exp ::= `spawn` Exp | `join(` Exp `)` [strict]
| `acquire(` Exp `)` [strict] | `release(` Exp `)` [strict]

We chose for the thread creation statement "spawn" to take as its only argument an expression which is supposed to be the call of a function. The intuition is that the arguments of the function are to be evaluated in the current thread, while the function call is executed in a newly spawned thread. "spawn" returns an identifier for the newly created thread which can be used by join to force the calling thread to wait for the specified thread to end before continuing. "acquire" and "release" can acquire and release any value; however, in our examples only memory locations are used as locks.

3 Basic KernelC Semantics

A $\mathbb{K}$ definition describes the execution model of a programming language by making explicit the structure of the configuration of the execution state and by providing a set of rules specifying how the execution state will be altered during the execution of a program.

The structure of the configuration for the sequential fragment of KernelC is pretty plain. At the top level we have two cells, T for the state of a running program, and result for a completed program. The T cell contains a computation cell k, an environment cell env mapping (local) variables to values, a funs cell mapping names of functions to their definitions, a function stack cell fstack for saving the control context upon calling a function, an output cell out, a memory cell mem mapping location (represented as naturals) to values (integers), a ptr cell for maintaining information about memory blocks allocation sizes, and a counter cell next for generating fresh locations and integers. A result cell, parallel to the T cell is used to allow a more concise result to be output to the user once the computation completes successfully. Hence, the T and result cells are meant to be mutually exclusive; the "?" symbol attached to their names specifies that any of them can miss from a running configuration.

CONFIGURATION:

$$\left\langle \begin{array}{l} \langle \cdot K \rangle_{\mathsf{k}} \; \langle \cdot Map \rangle_{\mathsf{env}} \; \langle \cdot Map \rangle_{\mathsf{funs}} \\ \langle \cdot List \rangle_{\mathsf{fstack}} \; \langle \text{""} \rangle_{\mathsf{out}} \; \langle \cdot Map \rangle_{\mathsf{mem}} \; \langle \cdot Map \rangle_{\mathsf{ptr}} \; \langle 0 \rangle_{\mathsf{next}} \end{array} \right\rangle_{\mathsf{T?}} \; \langle \text{""} \rangle_{\mathsf{result?}}$$

One important design choice is that we have decided to clearly distinguish between the heap allocated memory which is kept in the $\langle\rangle_{\mathsf{mem}}$ cell and the local variables memory which is kept in the $\langle\rangle_{\mathsf{env}}$ cell as a direct map from variables to values. Due to this choice, it is impossible to obtain the address of a variable, and this is enforced by the non-existence of the reference operator in KernelC. Another simplifying design decision was to not deal with scoping. The semantic rules presented below assume that once a variable is declared, it is visible for the remainder of the enclosing function execution, and therefore there should not be duplicated declarations of the same variable during the execution of the function.

Instead of presenting the entire semantics, which is pretty standard for the sequential fragment, we will only focus on rules related to memory access and function call.

Local Variables. Newly declared variables are mapped in the environment to the special **undef** computation constant which is not a value and thus it cannot be read.

CONTEXT
$$\texttt{int}\ \_\ \texttt{=}\ \square$$

RULE
$$\langle \frac{\texttt{int}\ X}{\texttt{void}} \cdots\rangle_{\mathsf{k}}\ \ \langle \frac{Env}{Env\ \texttt{[undef / }X\texttt{]}} \rangle_{\mathsf{env}}$$

𝕂 contexts specify evaluation strategies, ensuring that certain arguments of a construct are evaluated (by pushing them on the execution stack) before giving the execution rule for the construct itself. The 𝕂 rule for declaring a variable is almost self-explanatory: if the declaration of X is at the top of the execution stack represented by the $\langle\rangle_{\mathsf{k}}$ cell, then replace it by void and set the value of X to undefined in the environment. The ellipses to the right of the $\langle\rangle_{\mathsf{k}}$ cell show that there could be more things to execute after this declaration; however, this is the first. This rule also shows that transformations in 𝕂 are specified locally, by underlining the part to be replaced and writing its replacement underneath the line.

Local variables in KernelC are restricted. They cannot be shared, cannot be addressed, and therefore reside in a separate space called the environment.

RULE
$$\langle \frac{X}{V} \cdots\rangle_{\mathsf{k}}\ \ \langle \cdots X \mapsto V \cdots\rangle_{\mathsf{env}}$$

RULE
$$\langle \frac{X\ \texttt{=}\ V}{V} \cdots\rangle_{\mathsf{k}}\ \ \langle \cdots X \mapsto \frac{\_}{V} \cdots\rangle_{\mathsf{env}}$$

Heap allocation and dereferencing. The rules below define a very simple memory allocation mechanism, which basically allocates memory in order, starting with the location following the last allocated location.

RULE
$$\left(\begin{array}{ll} \langle \frac{\texttt{(int*)malloc(}\ N\ \texttt{*sizeof(int))}}{N'} \cdots\rangle_{\mathsf{k}} & \langle \cdots \frac{\cdot_{Map}}{N' \mapsto N} \cdots\rangle_{\mathsf{ptr}} \\ \langle \cdots \frac{\cdot_{Map}}{N'\ \texttt{..}\ N +_{Nat} N' \mapsto \texttt{undef}} \cdots\rangle_{\mathsf{mem}} & \langle \frac{N'}{N +_{Int} N'} \rangle_{\mathsf{next}} \end{array}\right)$$

The rules below specify atomic access to heap memory (ensuring a sequentially consistent semantics), for both read and write operations.

CONTEXT
$$\texttt{*}\ \square\ \texttt{=}\ \_$$

RULE DEREF
$$\langle \frac{\texttt{*}\ N}{V} \cdots\rangle_{\mathsf{k}}\ \ \langle \cdots N \mapsto V \cdots\rangle_{\mathsf{mem}}$$
[transition]

RULE UPDATE
$$\langle \frac{\texttt{*}\ N\ \texttt{=}\ V}{V} \cdots\rangle_{\mathsf{k}}\ \ \langle \cdots N \mapsto \frac{\_}{V} \cdots\rangle_{\mathsf{mem}}$$
[transition]

We use the transition tag for the rules expressing the semantics for the memory operations to instruct the 𝕂 tool that the order of their interaction should be considered in the transition graph of an execution.

User-declared Functions. Upon meeting a function declaration, the function is simply saved in the map of functions.

RULE
$$\langle \frac{\texttt{int}\ F\ \ Xl\ \texttt{\{}\ Sts\ \texttt{\}}}{\cdot_{K}} \cdots\rangle_{\mathsf{k}}\ \ \langle \cdots \frac{\cdot_{Map}}{F \mapsto \texttt{int}\ F\ \ Xl\ \texttt{\{}\ Sts\ \texttt{\}}} \cdots\rangle_{\mathsf{funs}}$$

Moreover, we desugar void functions into integer functions returning the special value `void` to avoid special casing the latter.

RULE
$\underset{\mathtt{int}\ F}{\underline{\mathtt{void}\ F}}\ \ Xl\ \{\ \underset{Sts\ \ \mathtt{return\ void\ ;}}{\underline{Sts}}\ \}$

When calling a function, a triple is pushed on the function stack, consisting of the name of the function, the current environment and the remainder of the computation. Then the current computation is replaced by the body of the function, the environment by the mapping of the arguments to their passed values. When returning, the environment and computation are restored with the function call being replaced by the return value.

The reason for pushing the name of the function on the function stack is that this effectively exposes the call stack for analysis purposes.

RULE
$$\left(\begin{array}{c} \langle \underset{\mathtt{bindTo(}\ Xl\ ,\ Vl\ \mathtt{)} \curvearrowright Sts}{\underline{F\ (\ Vl\ ) \curvearrowright K}} \rangle_{\mathsf{k}}\ \langle Env\rangle_{\mathsf{env}} \\ \langle \cdots\ F \mapsto \mathtt{int}\ F\ \ Xl\ \{\ Sts\ \}\ \cdots\rangle_{\mathsf{funs}}\ \langle \underset{F\ \#\ Env\ \#\ K}{\underline{\cdot_{List}}}\ \cdots\rangle_{\mathsf{fstack}} \end{array}\right)$$

RULE
$\langle \underset{V \curvearrowright K}{\underline{\mathtt{return}\ V\ ; \curvearrowright \_}}\rangle_{\mathsf{k}}\ \langle \underset{Env}{\underline{\_}}\rangle_{\mathsf{env}}\ \langle \underset{\cdot_K}{\underline{\_\ \#\ Env\ \#\ K}}\ \cdots\rangle_{\mathsf{fstack}}$

RULE
$\langle \underset{\cdot_K}{\underline{\mathtt{bindTo(,)}}}\ \cdots\rangle_{\mathsf{k}}$

RULE
$\langle \underset{\mathtt{bindTo(}\ Xl\ ,\ Vl\ \mathtt{)}}{\underline{\mathtt{bindTo(}\ X\ ,\ Xl\ ,\ V\ ,\ Vl\ \mathtt{)}}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{Env\ [\ V\ /\ X\ ]}{\underline{Env}}\rangle_{\mathsf{env}}$

Output. The output is simply appended to the $\langle\rangle_{\mathsf{out}}$ cell.

RULE PRINT
$\langle \underset{\mathtt{void}}{\underline{\mathtt{printf("\%d;",}\ I\ \mathtt{)}}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \underset{I\ \text{“;”}}{\underline{\cdot}}\rangle_{\mathsf{out}}$
[transition]

3.1 *A Sequentially Consistent Semantics for* KernelC *threads*

For executing multithreaded programs, the configuration must be updated to group computation, local variables and call stack in a thread cell, which is identified by an id. Multiple threads are grouped in a threads cell. Additionally, the ids of all completed threads are gathered in the cthreads cell.

CONFIGURATION:
$$\left\langle \begin{array}{l} \left\langle \left\langle \langle \cdot_K\rangle_{\mathsf{k}}\ \langle\cdot_{Map}\rangle_{\mathsf{env}}\ \langle\cdot_{List}\rangle_{\mathsf{fstack}}\ \langle 0\rangle_{\mathsf{id}}\right\rangle_{\mathsf{thread*}}\right\rangle_{\mathsf{threads}}\ \langle\cdot_{Map}\rangle_{\mathsf{locks}} \\ \langle\cdot_{Map}\rangle_{\mathsf{funs}}\ \langle \text{“”}\rangle_{\mathsf{out}}\ \langle\cdot_{Map}\rangle_{\mathsf{mem}}\ \langle\cdot_{Map}\rangle_{\mathsf{ptr}}\ \langle 1\rangle_{\mathsf{next}}\ \langle\cdot_{Set}\rangle_{\mathsf{cthreads}} \end{array}\right\rangle_{\mathsf{T?}}\ \langle \text{“”}\rangle_{\mathsf{result?}}$$

Note that, although the configuration changed, existing rules do not need to be changed, as the change was only a structural one. Given a configuration, the $\mathbb{K}$ tool uses its structure to concretize the rules and make them executable.

The semantics of spawn is the one mentioned in the thread syntax. We first have a context for evaluating the arguments of the function call (without calling the function), then we delegate the function call to a new thread.

CONTEXT
$\mathtt{spawn}\ \_\ (\ \square\ )$

RULE
$$\langle \frac{\texttt{spawn}\ X\ \texttt{(}\ Vl\ \texttt{)}}{T} \cdots\rangle_{\mathsf{k}}\ \ \langle \frac{T}{T +_{Int} 1}\rangle_{\mathsf{next}}\ \ \frac{\cdot_{Bag}}{\langle\cdots\ \langle X\ \texttt{(}\ Vl\ \texttt{)}\rangle_{\mathsf{k}}\ \langle T\rangle_{\mathsf{id}} \cdots\rangle_{\mathsf{thread}}}$$
[transition]

A lock can be acquired if not already acquired by any thread. Note that we don't model here re-entrant locks.

RULE
$$\left(\begin{array}{l}\langle \frac{\texttt{acquire(}\ N\ \texttt{)}}{\texttt{void}} \cdots\rangle_{\mathsf{k}}\ \ \langle T\rangle_{\mathsf{id}}\ \ \langle Locks\ \ \frac{\cdot_{Map}}{N \mapsto T}\rangle_{\mathsf{locks}} \\ \qquad\qquad\qquad \textit{when}\ \neg_{Bool}\ N\ \texttt{in}\ \texttt{keys}\ Locks\end{array}\right)$$
[transition]

RULE
$$\langle \frac{\texttt{release(}\ N\ \texttt{)}}{\texttt{void}} \cdots\rangle_{\mathsf{k}}\ \ \langle T\rangle_{\mathsf{id}}\ \ \langle\cdots\ \frac{N \mapsto T}{\cdot_{Map}} \cdots\rangle_{\mathsf{locks}}$$
[transition]

Upon completion, a thread registers its id in the set of completed threads, which is used as a signal to `join`.

RULE
$$\frac{\langle\cdots\ \langle V\rangle_{\mathsf{k}}\ \ \langle\cdot_{List}\rangle_{\mathsf{fstack}}\ \ \langle T\rangle_{\mathsf{id}} \cdots\rangle_{\mathsf{thread}}}{\cdot_{Bag}}\ \ \langle\cdots\ \frac{\cdot_{Set}}{T} \cdots\rangle_{\mathsf{cthreads}}$$

RULE
$$\langle \frac{\texttt{join(}\ T\ \texttt{)}}{0} \cdots\rangle_{\mathsf{k}}\ \ \langle\cdots\ T \cdots\rangle_{\mathsf{cthreads}}$$

4 Runtime Verification of Concurrent Programs

The definition above proposes a simple sequentially consistent semantics for our KernelC concurrent constructs. We will show here how this definition can be directly used as a tool for analyzing and observing program executions, but also how it can be developed further by defining an simple extension which allows checking program executions for datarace freeness.

The following *Banking* example is a C implementation of a Java class exhibiting a concurrent bug pattern [10]. The class attempts to define an account and some basic operations on it: creation, deposit, balance, withdraw, and transfer to another account. Figure 2 presents our C implementation of it, which encodes the objects as locations holding the amount of money available and the methods of the class as functions taking the receiver object's location as their first argument. Additionally, similarly to Java, we model the synchronized attribute of the methods by locking on the location of the receiver object at the beginning of the function and unlocking it before the return.

The simplest way to check a program's behavior is to run it and observe the result of one possible execution. We can do so using the **krun** command:

```
$ krun pAccount.c
100;20;300;220;
```

The output is the expected one, with both balances increased by 200, as it would probably happen in a normal execution. However, if we search for all possible outcomes of the execution using **krun**,

```
$ krun pAccount.c −−search
Search results:
```

```
int *newAccount(int m) {
  int *a=(int *)malloc(1*sizeof(int));
  *a=m;
  return a;
}

void deposit(int *a, int m) {
  acquire(a);
  *a=*a+m;
  release(a);
}

int balance(int *a) {
  acquire(a);
  int b=*a;
  release(a);
  return b;
}
```

```
void withdraw(int *a, int m) {
  acquire(a);
  if (m <= *a) {
    *a=*a-m;
  }
  release(a);
}

void transfer(int *a, int *b, int m) {
  acquire(a);
  if (m <= *a) {
    *a=*a-m;
    *b=*b+m;
  }
  release(a);
}
```

```
void run(int *a, int *b) {
  deposit(a,300);
  withdraw(a,100);
  transfer(a,b,100);
}
```

```
int main() {
  int *a = newAccount(100);
  int *b = newAccount(20);
  printf("%d;", balance(a));
  printf("%d;", balance(b));
  int t1 = spawn(run(a, b));
  int t2 = spawn(run(b, a));
  join(t1); join(t2);
  printf("%d;", balance(a));
  printf("%d;", balance(b));
  return 0;
}
```

Fig. 2. `pAccount.c`: The Account "class" and a concurrent test driver for it.

```
Solution 1, state 626:
<result>
  "100;20;200;20;"
</result>
```

```
Solution 2, state 665:
<result>
  "100;20;300;220;"
</result>
```

```
 .....
Solution 11, state 674:
<result>
  "100;20;100;220;"
</result>
```

we notice 10 additional, perhaps unexpected, solutions. We can guess it must be due to a datarace, but to locate it we need more powerful tools.

Searching for Dataraces

Let us illustrate below how one can detect dataraces, attempt to fix them, and then re-check the program for dataraces.

We adopt a straight-forward definition of dataraces: a datarace can be observed iff during the execution of the program there is a moment in which two threads can take as the next execution step transitions which access the same memory location, and at least one of the two accesses is attempting to update the location.

Although we could express this property as an assertion on states and then use Maude's model checker to check whether every possible execution of the program (for a given input) is datarace free, we here take a simpler approach by defining the race condition within $\mathbb{K}$ and then directly using the exploration capabilities provided through **krun**, the $\mathbb{K}$ tool for exploring program executions [23], to search for a datarace. If one is found, a configuration exhibiting the race is produced; if not, as the **krun** tool explores all possible interleavings due to scheduling, then the program is effectively proven datarace free (for the given input). To do that, we add two new cells as alternatives to existing cells, $\langle\rangle_{\mathsf{race}}$ as an alternative to the $\langle\rangle_{\mathsf{k}}$ cell and $\langle\rangle_{\mathsf{raceDetected}}$ as an alternative to the top cell $\langle\rangle_{\top}$, together with two rules capturing the write-write, and write-read dataraces, respectively:

RULE
$$< \underset{\mathsf{race}}{\underline{\mathsf{k}}} > *\, N = \_ \;\ldots</ \underset{\mathsf{race}}{\underline{\mathsf{k}}} > \;< \underset{\mathsf{race}}{\underline{\mathsf{k}}} > *\, N = \_ \;\ldots</ \underset{\mathsf{race}}{\underline{\mathsf{k}}} >$$

RULE
$$< \underset{\mathsf{race}}{\underline{\mathsf{k}}} > *\, N = \_ \;\ldots</ \underset{\mathsf{race}}{\underline{\mathsf{k}}} > \;< \underset{\mathsf{race}}{\underline{\mathsf{k}}} > *\, N \;\ldots</ \underset{\mathsf{race}}{\underline{\mathsf{k}}} >$$

These two rules ensure that any further computation is stopped for the two threads identified to be in a race, and ease their recognition in the configuration exhibiting the race. Note that these rules are actually renaming the computation cell, by rewriting its name (k) into race, without altering its contents.

In addition to that, we add another rule which changes the top computation itself once a race is detected, so we can easily identify an entire configuration exhibiting a race.

RULE
$$< \underset{\mathtt{raceDetected}}{\underline{\quad\top\quad}} >\ldots \langle K\rangle_{\mathsf{race}} \ldots</ \underset{\mathtt{raceDetected}}{\underline{\quad\top\quad}} >$$

A simple execution of the program under the new definition gives the same output as before. However, when searching for all configurations having $\langle\rangle_{\mathsf{raceDetected}}$ at the top, we obtain 16 race candidates, the first being the following:

```
$ krun pAccount.c --search --xsearch-pattern=' =>* ⟨raceDetected ⟩B:Bag ⟨/ raceDetected ⟩'
Search results:

Solution 1, state 299:
  ...
  ⟨threads⟩
    ...

                                                  ⟨thread⟩
  ⟨thread⟩                                          ...
    ...                                             ⟨race⟩
    ⟨race⟩                                            * 2 ↝  ...
      * 2 = 120 ↝  ...                              ⟨/race⟩
    ⟨/race⟩                                         ⟨fstack⟩
    ⟨fstack⟩                                          ListItem(deposit # a |−> 2
      ListItem(transfer # a |−> 1                     b |−> 1 # HOLE ; ↝
      b |−> 2 # HOLE ; ↝return void ;)                withdraw ( a , 100 ) ; ↝  ...
      ListItem(run # . # .)                           return void ;)
    ⟨/fstack⟩                                         ListItem(run # . # .)
    ...                                             ⟨/fstack⟩
  ⟨/thread⟩                                         ...
                                                  ⟨/thread⟩

  ⟨/threads⟩
  ...
```

Upon analyzing the counter-example configuration (including the fstack cell), one can notice that the race occurs because the access to account `b` in the `transfer` function is not synchronized. A simple-minded fix for this problem is to additionally lock on the `b` account in the `transfer` function. Upon applying this fix we can verify that the test driver became indeed datarace free using the previous command.

However, when searching for all possible results after fixing the datarace, we also obtain an unfinished computation in addition to the desired result:

```
$ krun pAccount.c --search
Search results:

Solution 1, state 184:
⟨T⟩
  ...
  ⟨threads⟩

                                                    ⟨thread⟩
    ⟨thread⟩                                          ⟨k⟩
      ⟨k⟩                                               acquire( 2 ) ~> ...
        acquire( 1 ) ~> ...                           ⟨/k⟩
      ⟨/k⟩                                            ⟨id⟩
      ⟨id⟩                                              3
        4                                             ⟨/id⟩
      ⟨/id⟩
        ...                                             ...
    ⟨/thread⟩                                       ⟨/thread⟩


    ...
  ⟨/threads⟩
  ⟨locks⟩
    1 |-> 3
    2 |-> 4
  ⟨/locks⟩

  ...
⟨/T⟩

Solution 2, state 253:
⟨result⟩
  "100;20;300;220;"
⟨/result⟩
```

Analyzing this configuration we can detect a deadlock between the two calls to `transfer`. By following Dijkstra's [6] solution to deadlock avoidance, we can ensure datarace freeness while avoiding deadlocks. The way to achieve that is by always acquiring resources in the same order in any thread, like:

```
if (!(a <= b)) {
  acquire(a); acquire(b);
} else {
  acquire(b); acquire(a);
}
```

Using this new implementation of the `transfer` function we can now effectively check that our test driver is datarace and deadlock free:

```
$ krun pAccount.c --search
Search results:

Solution 1, state 244:
⟨result⟩
  "100;20;300;220;"
```

⟨/result⟩

5 A Relaxed Memory Model for KernelC

Let us show how one can give another, more realistic, memory model semantics for the concurrent version of KernelC, and use the available analysis tools to analyze its behaviors and compare it against the sequentially consistent version of KernelC defined in Section 4. We base this semantics on the x86-TSO memory model [15], regarding threads as processors, and local variables as registers.

Relaxing the traditional sequential consistent semantics for memory access, which requires that reads and writes to the memory are perceived as atomic, the relaxed memory consistency models allow processors to basically have their own views of memory and only synchronize at specified points in the execution. These models allow more parallelism and thus more efficient executions of multithreaded programs; however, these models are harder to reason about, as they are less intuitive and yield a higher number of possible behaviors.

The x86-TSO memory model used in this section associates a write buffer to each process (or thread, in our case), which collects the local updates of memory variables, and defines the semantics of memory access and synchronization by taking into account these buffers. Therefore, the rules for all involved language constructs need to be changed in our $\mathbb{K}$ definition; nevertheless, nothing else except them and the configuration needs to be altered.

Two more cells need to be added to the $\langle\rangle_{\mathsf{thread}}$ cell: a $\langle\rangle_{\mathsf{buffer}}$ cell holding the queue of buffered writes, and a $\langle\rangle_{\mathsf{blocked}}$ cell containing a flag signaling whether the thread is blocked in waiting for a lock. Moreover, we add a list item constructor **`bwrite`** to represent a buffered write, that takes as parameters a location and a value; and we define a **`locations`** function which retrieves the set of locations from a list of buffered writes:

SYNTAX $K ::= \texttt{bwrite(}\ \#Nat\ \texttt{,}\ Val\ \texttt{)}$

SYNTAX $Set ::= \texttt{locations}\ List$

RULE
$\texttt{locations}\ \cdot_{List} \Rightarrow \cdot_{Set}$

RULE
$\texttt{locations}\ \texttt{bwrite(}\ A\ \texttt{,}\ V\ \texttt{)}\ \ Mem \Rightarrow A\ \ \texttt{locations}\ Mem$

In what follows we present the $\mathbb{K}$ rules specifying the new relaxed memory model semantics for concurrent KernelC preceded by their natural language description taken verbatim from the original TSO article [15]:

(i) p can read v from memory at address a if p is not blocked, has no buffered writes to a, and the memory does contain v at a;

RULE GLOBAL-DEREF
$\langle \underset{V}{\underline{\texttt{*}\ A}}\ \cdots\rangle_{\mathsf{k}}\ \ \langle Mem\rangle_{\mathsf{buffer}}\ \ \langle\cdots\ A \mapsto V\ \cdots\rangle_{\mathsf{mem}}\ \ \langle\texttt{false}\rangle_{\mathsf{blocked}}$

when $\neg_{Bool}\ A\ \texttt{in}\ \texttt{locations}\ Mem$
[transition]

Note that the fact that the thread has no buffered writes is modeled in the rule by the side condition, which requires that the current write buffer (represented by variable Mem) does not have any writes scheduled for the address A we want to read from.

(ii) p can read v from its write buffer for address a if p is not blocked and has v as the newest write to a in its buffer;

RULE LOCAL-DEREF

$$\langle \underset{V}{\underline{\texttt{*}\ A}}\ \cdots\rangle_{\mathsf{k}}\ \ \langle\cdots\ \texttt{bwrite(}\ A\ \texttt{,}\ V\ \texttt{)}\ \ \mathit{Mem}\rangle_{\mathsf{buffer}}\ \ \langle\texttt{false}\rangle_{\mathsf{blocked}}$$

when $\neg_{\mathit{Bool}}$ A `in locations` Mem
[transition]

V being the latest write to location A in the buffer is ensured by matching the entire contents of the buffer after the write of V to A and by checking (in the side condition) that it does not contain any other write to A.

(iii) p can read the stored value v from its register r at any time;

Since we view local variables as our registers, and since the rule is unconstrained, the existing rule for reading / writing local variables stays unchanged.

(iv) p can write v to its write buffer for address a at any time;

RULE BUFFER-WRITE

$$\langle \underset{V}{\underline{\texttt{*}\ A\ \texttt{=}\ V}}\ \cdots\rangle_{\mathsf{k}}\ \ \langle\cdots\ \underset{\texttt{bwrite(}\ A\ \texttt{,}\ V\ \texttt{)}}{\underline{\quad\cdot_{\mathit{List}}\quad}}\rangle_{\mathsf{buffer}}$$

Additionally, in KernelC we need to define the rules for incrementing values at memory locations, which, similarly to the regular reads rules (i) and (ii), have two flavors: depending on whether the location is or is not in the appropriate write buffer:

RULE LOCAL-INC

$$\langle \underset{I}{\underline{\texttt{*}\ A\ \texttt{++}}}\ \cdots\rangle_{\mathsf{k}}\ \ \langle\cdots\ \texttt{bwrite(}\ A\ \texttt{,}\ I\ \texttt{)}\ \ \mathit{Mem}\ \ \underset{\texttt{bwrite(}\ A\ \texttt{,}\ I\ +_{\mathit{Int}}\ 1\ \texttt{)}}{\underline{\quad\cdot_{\mathit{List}}\quad}}\rangle_{\mathsf{buffer}}$$

when $\neg_{\mathit{Bool}}$ A `in locations` Mem
[transition]

RULE GLOBAL-INC

$$\langle \underset{I}{\underline{\texttt{*}\ A\ \texttt{++}}}\ \cdots\rangle_{\mathsf{k}}\ \ \langle\mathit{Mem}\ \ \underset{\texttt{bwrite(}\ A\ \texttt{,}\ I\ +_{\mathit{Int}}\ 1\ \texttt{)}}{\underline{\quad\cdot_{\mathit{List}}\quad}}\rangle_{\mathsf{buffer}}\ \ \langle\cdots\ A \mapsto I\ \cdots\rangle_{\mathsf{mem}}$$

when $\neg_{\mathit{Bool}}$ A `in locations` Mem
[transition]

(v) If p is not blocked, it can silently dequeue the oldest write from its write buffer to memory;

RULE COMMIT-WRITE

$$\langle\mathsf{false}\rangle_{\mathsf{blocked}}\ \ \langle\underset{\cdot_{\mathit{List}}}{\underline{\texttt{bwrite(}\ A\ \texttt{,}\ V\ \texttt{)}}}\ \cdots\rangle_{\mathsf{buffer}}\ \ \langle\cdots\ A \mapsto \underset{V}{\underline{\ \_\ }}\ \cdots\rangle_{\mathsf{mem}}$$

[transition]

(vi) p can write value v to one of its registers r at any time;

Same as for item (iii), the existing rule needs not be changed.

(vii) If p's write buffer is empty, it can execute an MFENCE (so an MFENCE cannot proceed until all writes have been dequeued, modelling buffer flushing);[3]

[3] For the x86 processor, a memory fence (MFENCE) operation ensures that all load and store operations prior to the fence command will have been committed prior to any loads and stores issued following the fence.

We here assume that thread synchronization constructs, such as creation, termination, and join are all generating MFENCE operations:

CONTEXT
$\texttt{spawn \_ (} \square \texttt{)}$

RULE SPAWN

$$\langle \frac{\texttt{spawn } X \texttt{ (} Vl \texttt{)}}{T} \cdots\rangle_{\mathsf{k}} \; \langle \frac{T}{T +_{Int} 1} \rangle_{\mathsf{next}} \; \langle \cdot List \rangle_{\mathsf{buffer}} \; \frac{\cdot Bag}{\langle \cdots \langle X \texttt{ (} Vl \texttt{)} \rangle_{\mathsf{k}} \; \langle T \rangle_{\mathsf{id}} \cdots \rangle_{\mathsf{thread}}}$$

[transition]

RULE

$$\frac{\langle \cdots \langle V \rangle_{\mathsf{k}} \; \langle T \rangle_{\mathsf{id}} \; \langle \cdot List \rangle_{\mathsf{buffer}} \cdots \rangle_{\mathsf{thread}}}{\cdot Bag} \; \langle \cdots \frac{\cdot Set}{T} \cdots \rangle_{\mathsf{cthreads}}$$

RULE

$$\langle \frac{\texttt{join(} N \texttt{)}}{0} \cdots \rangle_{\mathsf{k}} \; \langle \cdot List \rangle_{\mathsf{buffer}} \; \langle \cdots N \cdots \rangle_{\mathsf{cthreads}}$$

(viii) If the lock is not held, and p's write buffer is empty, it can begin a LOCK'd instruction;

RULE ACQUIRE

$$\langle \frac{\texttt{acquire(} N \texttt{)}}{\texttt{void}} \cdots \rangle_{\mathsf{k}} \; \langle T \rangle_{\mathsf{id}} \; \langle \cdot List \rangle_{\mathsf{buffer}} \; \langle Locks \; \frac{\cdot Map}{N \mapsto T} \rangle_{\mathsf{locks}}$$

when $\neg_{Bool}$ N in keys $Locks$
[transition]

(ix) If p holds the lock, and its write buffer is empty, it can end a LOCK'd instruction.

RULE RELEASE

$$\langle \frac{\texttt{release(} N \texttt{)}}{\texttt{void}} \cdots \rangle_{\mathsf{k}} \; \langle T \rangle_{\mathsf{id}} \; \langle \cdot List \rangle_{\mathsf{buffer}} \; \langle \cdots \frac{N \mapsto T}{\cdot Map} \cdots \rangle_{\mathsf{locks}}$$

[transition]

Two additional rules are used to update the flag of the $\langle \rangle_{\mathsf{blocked}}$ cell:

RULE

$$\langle \texttt{acquire(} N \texttt{)} \cdots \rangle_{\mathsf{k}} \; \langle \frac{\mathsf{false}}{\mathsf{true}} \rangle_{\mathsf{blocked}} \; \langle \cdots N \mapsto T \cdots \rangle_{\mathsf{locks}}$$

RULE

$$\langle \texttt{acquire(} N \texttt{)} \cdots \rangle_{\mathsf{k}} \; \langle \frac{\mathsf{true}}{\mathsf{false}} \rangle_{\mathsf{blocked}} \; \langle \cdots Locks \cdots \rangle_{\mathsf{locks}}$$

when $\neg_{Bool}$ N in keys $Locks$

The first rule says that the thread becomes blocked if it tries to acquire a lock which is already held, while the second rule unblocks the thread once the lock is released.

Thus, with precisely one rule for each concurrency construct and without altering unrelated language constructs, we have defined a concurrent semantics for KernelC with a relaxed memory model.

Using this semantics, we can test, for example, that programs relying on busy-waiting synchronization are not portable from sequentially consistent memory models to relaxed memory models. Consider the KernelC specification of Peterson's software solution for mutual exclusion [17] presented in Figure 3. The presented implementation uses a function with three parameters, `flag`, `turn`, and `t`. `flag` is a (dynamically allocated) array, `turn` points to an integer in memory, and `t` is used as a thread identifier. To mark the critical sections, we are printing -1 and -2 for the beginning of critical section and 1 and 2 for the end of critical section for the threads identified by 0 and 1, respectively.

```
#include <stdio.h>
#include <stdlib.h>

void peterson(int *flag, int *turn, int t) {
  flag[t] = 1;
  *turn = 1−t;
  while (flag[1−t] && *turn == 1−t) {}
  printf("%d;",−1 − t);
  printf("%d;", 1 + t);
  flag[t] = 0;
}
```

```
int main() {
  int* flag= (int *)malloc(2*sizeof(int));
  flag[0]= 0; flag[1]= 0 ;
  int *turn= (int *)malloc(1*sizeof(int));
  int t1= spawn(peterson(flag, turn, 0));
  int t2= spawn(peterson(flag, turn, 1));
  join(t1); join(t2);
  return 0;
}
```

Fig. 3. An implementation of Peterson's algorithm in KernelC.

Using the previous (sequentially consistent) definition of concurrency for KernelC, one can verify that mutual exclusion is ensured asking **krun** to search for all final states obtainable upon running the program.

```
$ krun pPeterson.c −−search
Search results:

Solution 1, state 66:                 Solution 2, state 67:
⟨result⟩                              ⟨result⟩
  "−1;1;−2;2;"                          "−2;2;−1;1;"
⟨/result⟩                             ⟨/result⟩
```

The obtained results effectively show that the statements in the two critical sections cannot be interleaved.

However, when exploring the executions of the same program in the relaxed memory model definition of concurrent KernelC, mutual exclusion is not ensured: indeed **krun** finds 6 solutions to the same task, showing that the sequences -1,1 and -2,2 can be interleaved in every possible way:

```
$ krun pPeterson.c −−search
Search results:

Solution 1, state 433:      Solution 2, state 434:      Solution 3, state 435:
⟨result⟩                    ⟨result⟩                    ⟨result⟩
  "−1;1;−2;2;"                "−1;−2;1;2"                 "−1;−2;2;1"
⟨/result⟩                   ⟨/result⟩                   ⟨/result⟩

Solution 4, state 436:      Solution 5, state 437:      Solution 6, state 437:
⟨result⟩                    ⟨result⟩                    ⟨result⟩
  "−2;2;−1;1;"                "−2;−1;1;2;"                "−2;−1;2;1;"
⟨/result⟩                   ⟨/result⟩                   ⟨/result⟩
```

Thus, by applying a simple, generic, and already available rewriting logic tool on our $\mathbb{K}$ definitions we have shown that the relaxed memory model for KernelC defined in this section cannot be relied on for achieving mutual exclusion for programs which achieve that under the sequential consistency assumptions of the definition in Section 3.1.

We could even go one step further. Assume we decide to implement an additional library function call to a function mfence, whose semantics is to enforce the memory fence (i.e., that the buffer is emptied) before continuing:

SYNTAX $Exp ::= \texttt{mfence()}$

RULE $\langle \underset{\texttt{void}}{\underline{\texttt{mfence()}}} \cdots\rangle_{\mathsf{k}}\ \langle \cdot_{List}\rangle_{\mathsf{buffer}}$

With this simple, but powerful, library call, we can adjust the Peterson program by inserting a mfence() call right before the while loop. This is enough to guarantee mutual exclusion, as shown when exploring all the interleavings with the **krun** tool:

```
$ krun pPeterson.c --search
Search results:

Solution 1, state 66:                       Solution 2, state 67:
⟨result⟩                                    ⟨result⟩
  "-1;1;-2;2;"                                "-2;2;-1;1;"
⟨/result⟩                                   ⟨/result⟩
```

6 Conclusions

We have shown how 𝕂 definitions of programming languages can be turned (with negligible effort) into runtime analysis tools for testing and analyzing executions of concurrent programs.

Moreover, having different variants for the semantics of the same language features (e.g., different memory models) formalized in the same (executable) framework opens the door for testing and analyzing the relationship between different possible semantics of a language. This could be a very useful tool for language designers, allowing them to experiment by testing different possible semantics of the same feature against a suite of benchmark programs before deciding which semantics to implement.

We do not claim here that the tools one obtains almost for free within the 𝕂 framework completely eliminate the need of writing dedicated analysis tools in "real" programming languages. Nevertheless, we strongly believe that the 𝕂 framework can be viewed like a workbench for rapidly prototyping and experimenting with such analysis tools. Moreover, we believe that compilation techniques could be used to generate (more) competitive analysis tools directly from 𝕂 definitions.

References

[1] Atig, M. F., A. Bouajjani, S. Burckhardt and M. Musuvathi, *On the verification problem for weak memory models*, in: *POPL*, 2010, pp. 7–18.

[2] Burckhardt, S., R. Alur and M. M. K. Martin, *Checkfence: checking consistency of concurrent data types on relaxed memory models*, in: *PLDI*, 2007, pp. 12–21.

[3] Burckhardt, S. and M. Musuvathi, *Effective program verification for relaxed memory models*, in: *CAV*, LNCS **5123**, 2008, pp. 107–120.

[4] Burnim, J., K. Sen and C. Stergiou, *Sound and complete monitoring of sequential consistency for relaxed memory models*, in: *TACAS*, LNCS **6605**, 2011, pp. 11–25.

[5] Clavel, M., F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet and C. Talcott, "All About Maude, A High-Performance Logical Framework," LNCS **4350**, Springer, 2007.

[6] Dijkstra, E. W., *Solution of a problem in concurrent programming control*, Commun. ACM **8** (1965), p. 569.

[7] Ellison, C., "A Formal Semantics of C with Applications," Ph.D. thesis, University of Illinois (2012).

[8] Ellison, C. and G. Roşu, *An executable formal semantics of C with applications*, in: *POPL'12* (2012), pp. 533–544.

[9] Ellison, C., T. F. Șerbănuță and G. Roșu, *A rewriting logic approach to type inference*, in: *WADT'08*, LNCS **5486** (2009), pp. 135–151.

[10] Farchi, E., Y. Nir and S. Ur, *Concurrent bug patterns and how to test them*, in: *IPDPS* (2003), p. 286.

[11] Flanagan, C. and S. N. Freund, *Adversarial memory for detecting destructive races*, in: *PLDI*, 2010, pp. 244–254.

[12] Gopalakrishnan, G., Y. Yang and H. Sivaraj, *Qb or not qb: An efficient execution verification tool for memory orderings*, in: *CAV*, LNCS **3114**, 2004, pp. 401–413.

[13] Hills, M., F. Chen and G. Roșu, *A rewriting logic approach to static checking of units of measurement in C*, in: *RULE'08*, 2008, pp. 76–91, Tech. Rep. IAI-TR-08-02, Institut für Informatik III, Rheinische Friedrich-Wilhelm-Universität Bonn.

[14] Hills, M. and G. Rosu, *A rewriting logic semantics approach to modular program analysis*, in: *RTA'10*, LIPIcs **6** (2010), pp. 151–160.

[15] Owens, S., S. Sarkar and P. Sewell, *A better x86 memory model: x86-TSO*, in: *TPHOLs'09*, LNCS, 2009, pp. 391–407.

[16] Park, S. and D. L. Dill, *An executable specification, analyzer and verifier for rmo (relaxed memory order)*, in: *SPAA*, 1995, pp. 34–41.

[17] Peterson, G. L., *Myths about the mutual exclusion problem*, Information Processing Letters **12** (1981), pp. 115–116.

[18] Roşu, G. and A. Ştefănescu, *Matching logic: A new program verification approach (NIER track)*, in: *ICSE'11: Proceedings of the 30th International Conference on Software Engineering* (2011), pp. 868–871.

[19] Roşu, G., C. Ellison and W. Schulte, *Matching logic: An alternative to Hoare/Floyd logic*, in: *AMAST '10*, LNCS **6486**, 2010, pp. 142–162.

[20] Roșu, G., W. Schulte and T. F. Șerbănuță, *Runtime verification of C memory safety*, in: *RV'09*, LNCS **5779**, 2009, pp. 132–152.

[21] Roșu, G. and T. F. Șerbănuță, $\mathbb{K}$ *overview and SIMPLE case study*, this volume.

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

[23] Șerbănuță, T. F., A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu and G. Roșu, *The* $\mathbb{K}$ *primer (version 2.5)*, this volume.