Wuhan University
Journal of Natural Sciences

2020, Vol.25 No.2, 129-138

Article ID 1007-1202(2020)02-0129-10
DOI 10.19823/j.cnki.1007-1202.2020.0014

RSMC: A Safety Model Checker for Concurrency and Memory Safety of Rust

□ YAN Fei[1], WANG Qizhong[1],
ZHANG Liqiang[1†], CHEN Yasha[2]

1. Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, Hubei, China;

2. Institute of Systems Engineering, Academy of Military Science of the Chinese People's Liberation Army, Beijing 100082, China

Abstract: Rust is a system-level programming language that provides thread and memory safety guarantee through a suite of static compiler checking rules and prevents segmentation errors. However, since compiler checking is too strict to confine Rust's programmability, the developers prefer to use the keyword "unsafe" to bypass compiler checking, through which the caller could interact with OS directly. Unfortunately, the code block with "unsafe" would easily lead to some serious bugs such as memory safety violation, race condition and so on. In this paper, to verify memory and concurrency safety of Rust programs, we present RSMC (Safety Model Checker for Rust), a tool based on Smack to detect concurrency bugs and memory safety errors in Rust programs, in which we combine concurrency primitives model checking and memory boundary model checking. RSMC, with an assertion generator, can automatically insert assertions and requires no programmer annotations to verify Rust programs. We evaluate RSMC on two categories of Rust programs, and the result shows that RSMC can effectively find concurrency bugs and memory safety errors in vulnerable Rust programs, which include unsafe code.

Key words: Rust; memory safety; concurrency safety; model checking

CLC number: TP 309.2

Received date: 2019-08-20

Foundation item: Supported by the National Basic Research Program of China (973 Program) (2014CB340601)

Biography: YAN Fei, male, Associate professor, research direction: information security and trusted computing. E-mail: yanfei @ whu.edu.cn

† To whom correspondence should be addressed. E-mail: zhanglq@whu.edu.cn

0 Introduction

In recent years, Rust [1] has become a more and more popular system-level programming language in building a highly secure and concurrent software system. Aiming at avoiding memory disasters in C, Rust supports several unique mechanisms such as ownership, borrowing and lifetime rules etc, and then imposes the above rules for checking during compilation.

The concurrency and memory safety of Rust relies on the powerful type system and the compile-time checking, but it does not fit all scenarios. Firstly, all programming languages need to interact with external "unsafe" interfaces and call external libraries, but the above operations cannot be achieved under the "safe" Rust. Secondly, the "safe" Rust cannot efficiently represent complex data structures, in particular, when various pointers in the data structure are referenced to each other. Finally, there are actually some operations that are safe, but they cannot pass the checking of Rust compiler.

To solve the above problems, Rust defines keyword "unsafe" to mark a code block without checking the properties of concurrency and memory safety. With the help of unsafe block, the developer has the ability to bypass some compiler checking. As a result, calling unsafe code blocks (including external C code and Rust library code) would affect the concurrency and memory safety of Rust program themselves. To make up for some lack of compile-time checking in unsafe code blocks, it is urgent to apply effective verification methods to check Rust programs for the concurrency and memory safety.

For the formal verification Rust program, in recent years, more and more researchers have conducted many studies[2-4] on the formal semantics of the Rust. RustBelt[5] provided the first formal safety proof for the language that represents the actual subset of Rust, as well as, and this proof is scalable. Ref. [6] verified the Rust program using symbolic execution, and has resolved the performance issues caused by runtime verification. The proposing of formal semantics of Rust greatly promotes the progress of Rust program formal verification research and development of related verification tools.

For the verification of concurrent Rust program, there is little amount of research work[7] which focus on understanding of Rust concurrent programs and do not provide methods of concurrency safety verification for Rust program.

Rust2Viper [8] transforms Rust programs into an intermediate verification language, and then uses the verification tool Viper to verify Rust program correctness. CRUST [9] transforms Rust programs into C code, and uses C program verification tool CBMC[10] to verify the memory safety of transformed code. Both methods of translation verification are often complex and time-consuming. As well as to verify concurrent Rust programs, the verification language must handle concurrency in a manner similar to Rust. Due to the particularity of the use of Rust concurrent primitives, the method of translation verification is not suitable for verifying concurrent Rust programs. Smack[11] has been extended to support the functional correctness verification of Rust programs, and its input language can directly be Rust. As a result, we are able to extend Smack to support the verification of concurrent Rust programs.

In this paper, in order to effectively verifying the concurrency and memory safety of Rust programs, we introduce RSMC (Safety Model Checker for Rust), a safety model checker for Rust, which is targeted to detect the concurrency and memory safety problems created by the unsafe code. RSMC is extended by Smack[11], a very popular verification toolchain that has been extended to support Rust programs and concurrency and memory safety verification of Rust programs. RSMC works by a combination of concurrency primitives model checking and memory boundary model checking, and requires no programmer annotations to verify Rust program automatically with an assertion generator. RSMC takes Rust source program(.rs file) as input, and automatically inserts program properties assertions into the Rust program to verify the concurrency and memory safety of the Rust program with an assertion generator, then analyzes program assertions by verifier to report the result. The result of verification is either a memory safety and concurrency safety guarantee or a report of memory safety errors and concurrency safety bugs on this program.

We evaluated RSMC on two categories of Rust programs. For concurrency safety verification of Rust programs, we tested RSMC on a part of codes of three popular open sources Rust applications: Servo, Rand and TiKV. For memory safety verification of Rust programs, we tested RSMC on two types of Rust programs: Rust programs including C code and Rust programs including Rust standard library code. The test result shows that RSMC can effectively find concurrency bugs and memory safety errors in vulnerable Rust programs, which include unsafe code.

Our main contributions as follows:

1) Introduction of RSMC, combining concurrency primitives model checking and memory boundary model checking, which requires no programmer annotations to verify the concurrency and memory safety of Rust programs automatically with an assertion generator;

2) Evaluation of RSMC on two categories of Rust programs. The result shows that it can effectively detect concurrency bugs and memory safety errors in Rust programs.

We organize the rest of this paper as follows. Section 1 elaborates some security mechanisms of Rust and the method formal verification. Section 2 briefly presents the general understanding of RSMC. Section 3 presents the implement of concurrency and memory safety verification of Rust program. Section 4 evaluates RSMC on two categories of Rust programs, and we conclude this paper in Section 5.

1 Background Knowledge

1.1 The Security Mechanism of Rust

Rust [1] is a system-level programming language, and the design goal of Rust provides safe concurrency and memory with similar performance to C. It emphasizes safety, concurrency and memory control, and does not allow null pointers and dangling pointers, which are the source of system crashes, memory leaks, and unsafe code in C and C++. There are many security mechanisms in Rust, which can eliminate a lot of concurrency safety and memory safety issues. Those mechanisms include:

Ownership: Each variable has a domain of ownership. Function calls or returns will transfer variables' ownership to a new domain. When a domain comes to end, all of variables and memory resource binding to these variables would be destroyed by Rust compiler automatically. Because of the compiler checking, these variables could not be accessed any more.

Borrowing: In order to achieve temporary references to a variable, the owner of this variable can borrow it to the calling function. During the variable borrowing lifetime, the borrower can read and write the memory belongs to the variable, while the original owner is prohibited from reading and writing the memory. In addition, this rule guarantees that the memory is not released or transferred in the presence of borrowing. There are two types of borrowing in Rust: "immutable borrowing" (by default &*T*) and "mutable borrowing" (& mut *T*). There is at most one mutable borrowing at the same time, whereas there can be any immutable borrowing simultaneously. Rust can ensure that the lifetime of the borrower does not exceed the lifetime of the owner itself by compiler checking.

Lifetime: Whoever it is the owner of the variable or the borrowing/reference of the variable, there is an effective survival time or interval, which is called the life-time. Moreover, compiler would check each lifetime during compiling.

Rust manages memory almost perfectly in an efficient and secure way through ownership, borrowing, and lifetime. There is no load caused by manual memory management and also no program pause caused by Garbage Collection (GC).

Thread Safety: Rust provides two important features to help us achieve concurrent programming. One is Send. When type *T* implements Send, it will tell the compiler that this type can be safely passed between multiple threads (that is, the ownership of this type can be safely passed between multiple threads). The other is Sync. When type *T* implements Sync, it will tell the compiler that this type can be safely accessed between multiple threads (that is, the reference to this type can be safely passed between multiple threads).

Message Passing: The message passing is the simplest concurrent programming model, and the communication between threads is achieved by sending messages to each other. In Rust, we achieve message passing by channel. The message type of a channel must implement marker trait Send. If a type is Send, it means that its ownership can safely passed between threads. When the ownership is transferred from one thread to another and only one thread can access this type at the same time, thus avoiding data races and obtaining thread safety.

Shared Memory: In Rust, there are two ways to implement shared memory. One is using static variable, whose lifetime runs through the entire application program and there is only one instance at a fixed address in memory. Therefore, all threads have access to it. The other is saving the variable on the heap. Rust provides std::boxed::Box to allocate space on the heap. When the resource, allocated on the heap, is used in multiple threads, Rust needs to solve two problems: 1) When would the resource be released? 2) How would the variables be accessed safely and concurrently? Rust solves the first problem by reference counting and the second one by synchronous means.

Thread Synchronization: Rust achieves the collaboration between threads by synchronization primitives, such as atomic operations and lock. There is some difference between synchronization primitives in Rust and in other languages. Firstly, for the Mutex, Rust does not provide unlock function explicitly. Secondly, Rust supports the synchronous channel to achieve message passing.

Combining these mechanisms, Rust developers can safely use some types of thread-unsafe in other languages in a multi-threaded environment, without worrying about accidentally sending them from the current thread to other threads. The compiler will notify if there is no necessary synchronization mechanisms.

1.2 Formal Verification

Formal verification is a mathematically complete proof of whether the system has achieved the designer's intent. The basic method of formal verification is mainly divided into two categories: Theorem Proving and Model Checking.

The theorem proving technique uses the axiom system to formalize the description of the target system and describes the goal as a theorem that needs to be proved. It verifies that the target system meets the security objectives by proving that the theorem in the axiom system is true.

The basic idea of model checking is to use explicit state search or implicit fixed-point calculation to verify the modal/propositional properties (F) of the finite state concurrent system (S). When the system fails to pass the verification, this method can also determine where the problem exists. The detailed process is show in Fig. 1.

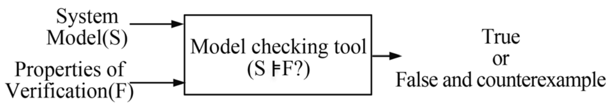

System Model(S)
Properties of Verification(F)
Model checking tool (S ⊧F?)
True or False and counterexample

Fig. 1 The process of model checking

1.3 The Analysis of Unsafe Code

Rust defines keyword "unsafe" to mark a code block. The unsafe block allows the programmer to do extra things including:

1) Dereferencing a raw pointer *const *T* and *mut *T*;

2) Reading and writing a mutable static variable static mut;

3) Calling an unsafe function.

It is important to note that the unsafe does not turn off the borrowing checker or disable any other Rust safety checks: if a reference used in an unsafe code, it will still be checked. The unsafe keyword simply provides the above three features that are not checked by the compiler. As a result, a certain degree of safety is still available in the unsafe block.

The raw pointer has the following characteristics: 1) allows compiler to ignore borrowing rules, so Rust can have both immutable and mutable pointers, or multiple mutable pointers pointing to the same location; 2) does not guarantee to point to valid memory; 3) is allowed to be empty; 4) cannot implement any automatic cleanup function. In unsafe block, dereferencing a raw pointer may lead to data race and reading arbitrary memory data. Reading and writing a mutable static variable may also lead to data race.

In unsafe block, in order to achieve the interacting with code written in other languages (in this paper, we only discuss C), Rust can call external code by keyword extern, which is helpful to build the Foreign Function Interface (FFI). Because external programming languages do not enforce Rust's rules and Rust cannot check them, the concurrency and memory safety of which cannot be guaranteed.

The verification work we do next in Rust is to make up for the lack of the above compiler check rules in unsafe block. In order to achieve this goal, we mainly complete the missing work of the compiler check through the verification of the concurrency and memory safety in unsafe Rust.

2 The General Understanding of RSMC

This section introduces the overall structure and verification process of RSMC, as shown in Fig. 2.

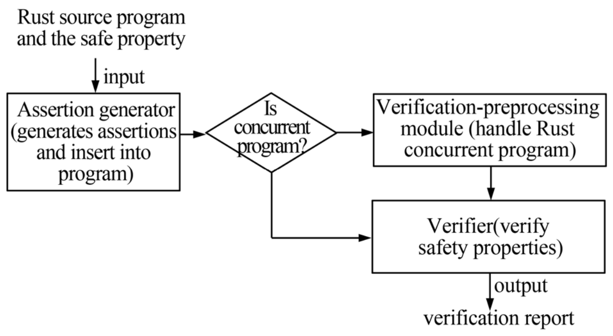

Rust source program and the safe property
input
Assertion generator (generates assertions and insert into program)
Is concurrent program?
Verification-preprocessing module (handle Rust concurrent program)
Verifier(verify safety properties)
output
verification report

Fig. 2 The overall structure of RSMC

The RSMC mainly consists of three modules: an assertion generator, a verification-preprocessing module, and a verifier.

The assertion generator takes the Rust source program (.rs file) as the input, generates corresponding assertions based on the property that need to be verified, and inserts the above assertions into corresponding locations in Rust programs. The output of the assertion generator is modeled Rust program. If the Rust program is concurrent program, we firstly use the verification-preprocessing module to handle the modeled Rust program, thus generating different program statement paths. Next, we apply the verifier to verify all program statement paths for the property, and generating the verification report. If the Rust program is non-concurrent program, we can directly use the verifier to verify the modeled Rust program.

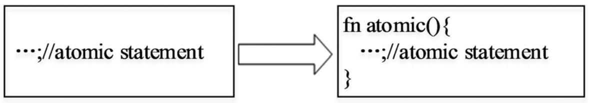
3 The Implement of Concurrency and Memory Safety Verification of Rust Program

3.1 Modeling Concurrency Primitives

Due to the particularity of the use of Rust concurrent primitives, it is necessary to add unique features of Rust concurrent program to RSMC. Specifically, we need to extend the support of modeling concurrent primitives of Rust to RSMC. The modeling work of Rust includes atomic type, mutex, condition and channel.

Modeling Atomic Type. The atomic type is the simplest mechanism to control access to shared resources. Atomic types do not require developers to deal with locking and releasing the lock, and they support modification and reading, and have high concurrency performance.

We define the range in which the atomic type works as the atomic section, and add constraints that no context switching is allowed in this section. We model an atomic type by introducing an atomic function (fn atomic(){}) which could be defined and identified in our RSMC tool (as shown in Fig. 3). The atomic section is useful in modeling other primitives and the verification of concurrent program.



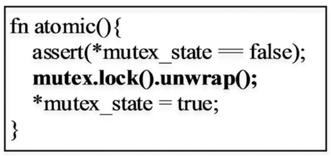
```
···;//atomic statement
```

```
fn atomic(){
  ···;//atomic statement
}
```

Fig. 3 Modeling of atomic type in Rust

Modeling Mutex. In addition to atomic types, the sharing resources in multiple threads can also be considered by using the lock. Before the operation, the lock must be obtained at first. Only one lock can be given to one thread at the same time, which ensures that only one thread can operate the shared resource at the same time. After the operation is completed, the lock is released to other waiting threads. In order to improve security, Rust will automatically release the lock when the object is destroyed. In Rust, the *std*::*sync*::*Mutex* is a lock.

The initial state of a *mutex* is defined as *false* (unlocked). When a *mutex* performs the lock operation, it will wait until the state of the *mutex* is *false* (unlocked) and then set its state to *true* (locked). When a *mutex* performs the unlock operation, it will set its state to *false* (unlocked). We model a *mutex* by add a *mutex* state flag *mutex_state* for it, this flag represents the state of the *mutex*(true for locked and false for unlocked). There are no unlock functions explicitly in Rust, and we can use the assertion generator to set the *mutex_state* to *false* at the end of lock operations scope. Figure 4 is the modeling of lock function in Rust.



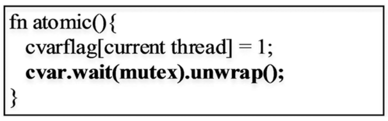
```
fn atomic(){
  assert(*mutex_state == false);
  mutex.lock().unwrap();
  *mutex_state = true;
}
```

Fig. 4 Modeling of lock (mutex.lock().unwrap()) in Rust

In Rust, the *mutex* is an exclusive lock, and the lock can only be held by one thread at a time. This type of lock causes all threads to be serialized, which is safe but not efficient. In order to resolving this program, Rust provides a mechanism called read-write lock, *std*::*sync*::*RwLock*. It is similar to the *mutex* in use.

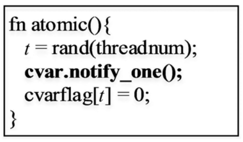
Modeling Condition. The *Condvar*(*cvar*) is a Rust's conditional variable that provides a *wait* method that actively lets the current thread to wait, while providing a *notify_one* method that lets other threads to wake up the waiting thread. And a condition in Rust has three primitives: *wait*, *notify_one*, and *notify_all*. The *cvar*.*wait*(*mutex*) stops the thread from running until it is woken up by another thread calling the *notify* function. The *cvar.notify_one*() wakes up one of the thread waiting for this condition, and there is no guarantee that which waiting threads will be woken up. The *cvar.notify_all*() wakes up all threads waiting for this condition.

Here, we model the *Condvar* for each condition by the flag vector *cvarflag*, one for each thread. To model the *wait* in thread *t*, we set the *cvarflag*[*t*] flag to 1, allowing context switching, and unlock the *mutex* to allow other threads to change the *Condvar*. After the *wait* function, we assume that *cvarflag*[*t*] has been set to 0, and lock the *mutex* to protect the *Condvar*, as shown in Fig. 5. To model the *notify_one*, we indefinitely choose a condition flag of a waiting thread will be woken up and set it to 0, as shown in Fig. 6. The *notify_all* is modeled by setting all the flags to 0.



```
fn atomic(){
  cvarflag[current thread] = 1;
  cvar.wait(mutex).unwrap();
}
```

Fig. 5 Modeling of wait (cvar.wait(mutex).unwrap()) in Rust



```
fn atomic(){
  t = rand(threadnum);
  cvar.notify_one();
  cvarflag[t] = 0;
}
```

Fig. 6 Modeling of notify_one (cvar.notify_one()) in Rust

Modeling Channel. Rust's channel can communicate the message (data) of one thread to another thread, allowing information to flow in different threads for collaboration. The two ends of the channel are the sender and the receiver. The sender is responsible for sending the message from one thread, and the receiver receives the message in another thread. Since asynchronous channels can support multiple senders at the same time and have the function of message buffering, there are few concurrency problems. The channel discussed in this paper mainly refers to the synchronization channel.

We can model the channel by adding to each channel a new bit variable *is_channel_empty*, indicating whether the channel cache is empty. In addition, on every sending message to channel we assert that its *is_channel_empty* is *true*; on every receiving message from channel we assert that its *is_channel_empty* is *false* (as shown in Fig. 7).

```
use std::sync::mpsc;
let (tx, rx) = mpsc::sync_channel(1);
fn atomic{
   assert(is_channel_empty == 1);
   tx.send(message) .unwrap();
   is_channel_empty = 0;
}
```

(a) tx.send(msg)

```
use std::sync::mpsc;
let (tx, rx) = mpsc::sync_channel(1);
fn atomic{
   assert(is_channel_empty == 0);
   rx.receive().unwrap();
   is_channel_empty = 1;
}
```

(b) rx.receive()

Fig. 7 Modeling of channel in Rust

3.2 Verifying Concurrency Safety Properties of Rust

For the verification of Rust concurrent program, it is significant to find data races and deadlocks. In order to achieve this goal effectively, in this section, we make some expansions in the model of concurrency primitives for Rust concurrent programs verification.

Detecting Data Races. The Data Race refers to that multiple threads access the same piece of data at the same time without proper locking, and at least one thread is a write operation, which competes with the reading and modification of data, resulting in various unpredictable problems.

In order to effectively detect data races, we add a global flag *is_m_writable* for each global variable *m* in Rust programs. The global flag indicates whether the global variable *m* is writable. We set the *is_m_writable* to true when we define the global variable *m* and set the *is_m_writable* to false in the next instruction. We model every access as an atomic type. Before every access of global variable *m*, we will assert whether its *is_m_writable* is set to *false*. Figure 8 presents the translation model of the access to global variable *m*.

```
fn atomic{
   assert(is_  _writable == 0);
   let m = 23;
   is_m_writable = 1;
}
is_m_writable = 0;
```

(a) translation of m=23

```
assert(is_m_writable == 0);
let n=m ;
```

(b) translation of $n=m$

Fig. 8 The translation for detecting data races

Detecting Deadlocks. In Rust concurrent program, there are two types of common deadlocks: a *mutex* deadlock indicates that all the threads in the concurrent program are waiting for a *mutex*, and a channel deadlock indicates that all the threads in the concurrent program are waiting for other party's response in a channel. We are able to the detect channel deadlock by just using channel model without modification. In this section, we mainly focus on the *mutex* deadlock.

We present a thread counter for each Rust program that represents the number of waiting threads. This counter is called *waiting_threads*. When modeling *mutex.lock*(), assuming that the value of the *mutex* is *false*, we increase *waiting_threads*, and then assert that *waiting_threads* < *T* (the total number of threads), as shown in Fig. 9. We can detect a *mutex* deadlock by verifying whether the assertion fails. When modeling *cvar.wait(mutex)*, we increase *waiting_threads* after setting the *cvarflag*[current thread] flag to 1, and then assert that *waiting_threads* < *T*, as shown in Fig. 10.

```
if (!is_dd){
  fn atomic{
    assert(*mutex_state == false);
    let unlocked = (*mutex_state == false);
    if (unlocked)
      { mutex.lock().unwrap();
      *mutex_state = true;}
    else waiting_threads + +;
  }
  fn atomic{
    if (!unlocked){
      is_dd = (waiting_threads ==T );
      assert(!is_dd);
    }
  }
}
```

Fig. 9 Modeling of lock (mutex.lock().unwrap()) when verifying deadlocks

```
if (!is_dd){
  fn atomic{
    cvarflag[current thread] = 1;
    waiting_threads ++;
    cvar.wait(mutex).unwrap();
  }
  fn atomic{
    is_dd = (waiting_threads == T);
    assert(!is_dd);
  }
}
```

Fig. 10 Modeling of wait (cvar.wait(mutex).unwrap()) when verifying deadlocks

Similarly, we can detect a mutex deadlock by verifying whether the assertion fails.

We also present a deadlock detection flag named *is_dd*. This flag indicates whether a deadlock is detected.

3.3 Modeling and Verifying Memory Safety Properties of Rust

For the verification memory safety of Rust program, it is significant to find memory safety errors. The memory safety properties of Rust program include:

1) Buffer overflow: Check whether the boundary of buffer is violated;

2) Pointer safety: Search for dangling raw pointer dereferences or null raw pointer dereferences;

3) Memory leak: Check whether the uninitialized memory is read.

For modeling the buffer overflow, we search for each array access, build assertions to assert that the boundary is not violated, and use the RSMC to verify buffer overflow property.

For modeling the pointer safety, we search for all dangling raw pointers and null raw pointers, build assertions to assert that dereferences of above pointers are valid, and use the RSMC to verify pointer safety property.

For modeling the memory leak, we search for each memory access, build assertions to assert that the memory is initialized, and use the RSMC to verify memory leak property.

3.4 Automatic Assertion Generation Algorithm

To verify Rust programs by RSMC effectively, we add an assertion generator module to RSMC. The assertion in Rust program specifies the concurrency and memory safety properties of Rust. The assertion generator can perform program static analysis to find the location of memory operations and concurrent primitive operations, and automatically insert program properties assertions. It makes RSMC automatically verify Rust program properties.

Definition 1 The model of Rust concurrent program can be defined as a multi-tuple RM1=⟨*P*, *S*, *n*⟩, where the *P* represents the Rust source program; the *S* represents a single statement; the *n* is the number of statements. The *i* is an arbitrary integer, $1 \leqslant i \leqslant n$.

The Algorithm 1 is for generating corresponding assertions based on the property that need to be verified.

Algorithm 1 Automatic assertion generation algorithm

```
Input:
Rust program and the property;
Output:
Modeled Rust program.
1:  RM1=<P, S, n>
2:  property=GetProperty(flag)//get the property that need to be verified
3:  if property== memory_safety_property then
4:      for i=1; i <n; i ++ do
5:          if $S_i$ is related to the property then
6:              generating corresponding assertions and
                inserting them into program (the location
                of the previous statement of $S_i$ )
7:          end if
8:      end for
9:  end if
10: if property== concurrency _safety_property then
11:     for i =1; i <n; i ++ do
12:         if $S_i$ is related to the property then
13:             generating corresponding assertions and
                inserting them into program (the location of
                the previous statement of $S_i$ )
14:         end if
15:     end for
16: end if
17: return the modeled Rust program
```

3.5 Verification Algorithm of Rust Concurrent Program

Definition 2 The checking model of Rust concurrent program can be defined as a multituple RM2=⟨*P*, *T*, *S*, Path, ^, Dep, Ato⟩. The *n* is the number of threads, the $m_1, m_2, \cdots, m_n$ is the number of statements per thread. The *i*, *j* and *k* is an arbitrary integer, $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant \max(m_i)$, $1 \leqslant k \leqslant \max(m_i)$.

The *P* represents the Rust source program, the *T* represents the thread in the program, the *S* represents a single statement, then $P = \{T_1, T_2, \cdots, T_n\}$, $T_n = \{S_{n1}, S_{n2}, \cdots, S_{nm_n}\}$, with $S_{ij} \in T_i, T_i \subseteq P$.

The symbol ^ denotes an ordered sequence that can be exchanged in sequence, $S_{ij} \wedge S_{ik} = \{(S_{ij}, S_{ik})(S_{ik}, S_{ij})\}$.

The Path represents all possible execution sequences, Path(*P*)={Path(T_1)^Path(T_2)^Path(T_3)^ ⋯ ^ Path(T_n)}.

Similarly, the Path(T_i) represents all possible linear combinations of $\{S_{i1}, S_{i2}, S_{i3}, \cdots, S_{im_i}\}$, Path ($T_i$) =

$\{S_{i1} \wedge S_{i2} \wedge S_{i3} \wedge \cdots \wedge S_{im_i}\}$.

The Dep and Ato in RM respectively indicate that there are several causalities in the program, which will impose constraints on the execution of the Rust program. The CPath represents all possible execution sequences which include constraints, CPath(P)=Path(P) ⋂ Dep ⋂ Ato.

The Dep(dependency) is a dependency in the program, including control dependency and data dependency), $\forall d \in$ Dep, $d = \{(S_{ij}, S_{ik})\}$ indicates that S_{ik} is dependent on S_{ij}.

If $d = \{(S_{ij}, S_{ik})\}$, Path(T_i)= $S_{ij} \wedge S_{ik}$ → CPath(T_i)= { (S_{ij} , S_{ik})}.

The Ato(atomic code) is a serialized block, $\forall s \in$ Ato, $s = \{(S_{ij}, S_{i(j+1)}, \cdots, S_{ik})\}$ indicates that these (k–j) statements are executed with a lock. The remaining threads cannot perform operations that conflict with $(S_{ij}, S_{i(j+1)}, \cdots, S_{ik})$ because of the lock. So we abstract $s = \{(S_{ij}, S_{i(j+1)}, \cdots, S_{ik})\}$ into $(S_{ij}, S_{i(j+1)}, \cdots, S_{ik})$, which is an atomic operation that is executed sequentially.

If $s = \{(S_{ij}, S_{i(j+1)}, \cdots, S_{ik})\}$, Path(T_i)= $S_{i1} \wedge S_{i2} \wedge S_{i3} \wedge \cdots \wedge S_{im_i}$ →

CPath (T_i) = $\{S_{i1} \wedge S_{i2} \wedge \cdots \wedge S_{i(j-1)} \wedge S_{i(k+1)} \wedge S_{i(k+2)} \wedge \cdots \wedge S_{im_i}\}$.

The Algorithm 2 is for building a model and applying the verifier to verify the model Rust Concurrent Program.

Algorithm 2 Verification algorithm of concurrent Rust

Input:
Modeled Rust Concurrent program;
Output:
Verification report.

```
1:  RM2=<P, T, S, Path, ^, Dep, Ato >
2:  repeat
3:      for i=1; i <n; i ++ do
4:          Path( $T_i$ )={ $S_{i1} \wedge S_{i2} \wedge S_{i3} \wedge \cdots \wedge S_{im_i}$ }
5:          for j=1; j < $m_i$ ; j ++ do
6:              for k=j ; k< $m_i$ ; k++ do
7:                  if $S_{ik}$ is dependent on $S_{ij}$ then
8:                      Dep=Dep ∪ ( $S_{ij}$ , $S_{ik}$ )
9:                  end if
10:             if ( $S_{ij}$ , $S_{i(j+1)}$ , $\cdots$ , $S_{ik}$ ) is an atomic block then
11:                 Ato = Ato ∪ {( $S_{ij}$ , $S_{i(j+1)}$ , $\cdots$ , $S_{ik}$ )}
12:             end if
13:         end for
14:         end for
15:     end for
16:     Path(P)={Path( $T_1$ ) ∧ Path( $T_2$ ) ∧ Path( $T_3$ ) ∧ Path( $T_n$ )}
17:     CPath(P)=Path(P)⋂Dep⋂Ato
18:     Applying the verifier on CPath(P)
19: until (There are errors in the model)
20: return the verification report
```

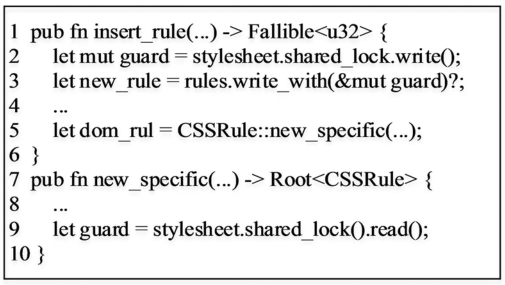
4 Experiment

4.1 Concurrency Safety Verification of Rust Program

In order to evaluate the effectiveness of RSMC to detect concurrency safety bugs, we tested RSMC on part of code of three Rust applications: Servo(a web browser), Rand(a random number generation library) and TiKV(a key-value storage system).

In Servo module, RSMC reports two deadlocks caused by double locks and circularly waiting. As we can see from the Rust code, although there are lock operations, Rust does not provide unlock functions explicitly. This is because Rust's compiler can automatically release the lock when the object is destroyed at the end of lock operations. Because of this, before the acquired lock is released, acquiring the lock again may result in a double-lock deadlock. In Rust programs, we achieve the message passing by the channel. When the buffer of a channel is empty, the thread that receives data from a channel will be block. Sometimes incorrect use of channel operations may cause all threads to circular wait for messages and deadlocks.

Figure 11 is part code of the Servo. We define a write lock at line 2, the scope of the lock is within the function *insert_rule*() from line1 to line 6. Before the lock is released at line 5, the program calls *new_specific*() and acquires the lock again at line 9, resulting in a double-lock deadlock.



```
1  pub fn insert_rule(...) -> Fallible<u32> {
2      let mut guard = stylesheet.shared_lock.write();
3      let new_rule = rules.write_with(&mut guard)?;
4      ...
5      let dom_rul = CSSRule::new_specific(...);
6  }
7  pub fn new_specific(...) -> Root<CSSRule> {
8      ...
9      let guard = stylesheet.shared_lock().read();
10 }
```

Fig. 11 The part code (including a double-lock deadlock) of Servo

Figure 12 is also part code of the Servo. There are two smooth paths. In Fig. 12(b), the worker thread (referred to as WT in the following) creates a channel at line 11, sends a message (*Func1*(*sender*2)) to the master thread(referred to as MT in the following) at line 12, and blocks itself at line 13. In Fig.12(a), the MT receives a message at line 7, sends an empty message to the WT and unblocks the WT at line 8. In Fig. 12(a), the MT receives a message (*ExitFunc*(···)) at line 3 and leaves the loop. The MT creates a channel at line 13, sends a message (*Func*2(*sender*3)) to the WT at line 14, and blocks itself at line 15. In Fig.12(b), the WT receives the message at line 2, sends an empty message to the MT and unblocks the MT at line 6.

When the MT receives a message (*ExitFunc*(...)) just before the WT sends a message(*Func1*(*sender*2)), the MT blocks itself waiting for the reply of WT at line 15. However, the WT also blocks itself waiting for the reply of MT at line 13, leading to a circular waiting deadlock.

In Rand and TiKV module, RSMC also reported similar concurrency bugs.

```
1  while !done {
2      match r1.recv(){
3          Some(Exit(...)) => {
4              done = true;
5              ...
6          }
7          Some(SetID(s2)) => {
8              s2.send(());
9          }
10     ...
11     }
12 }
13 let (s3, r3) = channel();
14 s2.send(ExitMsg(s3));
15 r3.recv(); //blocks
```

(a) master thread

```
1  loop {
2      let request = r2.recv();
3      match request {
4          ExitMsg(s3) => {
5              ...
6              s3.send(());
7              break;
8          }
9          ReadyMsg(...) => {
10             ...
11             let (s2, r2) = channel();
12             s1.send(SetID(s2))
13             r2.recv() //blocks
14         }
15     }
16 }
```

(b) worker thread

Fig. 12 The part code (including a circular waiting deadlock) of Servo

4.2 Memory Safety Verification of Rust Program

In order to evaluate the effectiveness of RSMC to detect memory safety errors, we tested RSMC on two types of Rust programs. For Rust program including C code, we use RSMC to check three Rust programs including C code, two of the three programs have apparent memory safety errors including buffer overflow and memory leak. The result shows that RSMC can accurately detect memory safety errors in vulnerable Rust programs including C code. Table 1 shows the verification result of Rust program including C code using RSMC.

Table 1 Verification result of the Rust program including C code using RSMC

Module name	Verified property	Lines of code	Time/s	Memory safety errors find
buffer overflow	Memory safety	42	380	Yes
memory leak	Memory safety	30	200	Yes
correct	Memory safety	54	540	No

For Rust program including Rust standard library, we use RSMC to check some Rust programs, which are called Rust standard libraries code including the ring buffer, array, vector, slice, and cell data structures. The size of the array type is fixed in Rust, which cannot add or remove elements from array once defined. The ring buffer type is a ring buffer, whose space is allocated by heap and its size is not fixed. The vector type is an array, similarly, whose space is allocated by heap and its size is not fixed. The cell type is a container. Everything contained inside can be modified in the state of shared reference, and it enables "interior mutability". The slice type is of unknown size, and it is a double word object. The first word is the data in a pointer, and the second word is the length of the slice.

Table 2 shows the verification result of Rust program including Rust standard library using RSMC. We

Table 2 Verification result of the Rust program including Rust standard library using RSMC

Module name	Verified property	Lines of code	Time/s	Memory safety errors find
Ringbuf(v1.0.0)	Memory safety	75	780	No
Ringbuf(v1.35.0)	Memory safety	69	720	No
Array(v1.0.0)	Memory safety	73	760	Yes
Array(v1.35.0)	Memory safety	70	730	No
Vector(v1.0.0)	Memory safety	86	880	Yes
Vector(v1.35.0)	Memory safety	81	850	No
Cell(v1.0.0)	Memory safety	78	810	No
Cell(v1.35.0)	Memory safety	72	760	No
Slice(v1.0.0)	Memory safety	84	880	Yes
Slice(v1.35.0)	Memory safety	79	820	No

examined the latest version and history version of Rust standard libraries code including the above data structures. By comparison testing, the result RSMC reported shows that no memory safety errors are found in the latest version of Rust library and some memory safety errors are found on the history version, and these memory safety errors are in array, vector and slice module.

The analysis of the experimental result of the above three modules shows that there are many unsafe blocks in the above module library code, in which we perform some improper operations. It may lead to memory safety errors in Rust programs. As a result, RSMC can find some memory safety errors on the history version of Rust standard library including the above three modules.

5 Conclusion

In this paper, we presented RSMC, a model checker for memory and concurrent safety verification of Rust program. RSMC works by a combination of concurrency primitives model checking and memory boundary model checking, and it can automatically insert safety properties assertions into each Rust program with an assertion generator to efficiently detect concurrent safety bugs and memory safety violations. We evaluated the effectiveness of RSMC by testing it on different categories of Rust programs, and the experiment result shows that it can efficiently detect memory safety errors and concurrent safety bugs in vulnerable Rust programs, which include unsafe code.

In this paper, we just focus on memory access concurrency problems. In the future, we will extend the verification of concurrent Rust to network concurrency and file concurrency.

References

[1] The Rust Project Developers. The Rust programming language [EB/OL]. [2015-05-24]. *http://www. rust-lang. org*.

[2] Wang F, Song F, Zhang M, *et al*. KRust: A formal executable semantics of Rust [C]//2018 *International Symposium on Theoretical Aspects of Software Engineering* (*TASE*). Washington D C: IEEE, 2018: 44-51.

[3] Kan S, Sanán D, Lin S W, *et al*. K-Rust: An executable formal semantics for Rust [EB/OL]. [2018-11-25]. *http*: *ArXiv Preprint ArXiv*:1804. 07608.

[4] Reed E. *Patina*: *A Formalization of the Rust Programming Language* [R]. Seatle: University of Washington, 2015.

[5] Jung R, Jourdan J H, Krebbers R, *et al*. RustBelt: Securing the foundations of the Rust programming language [C]// *Proceedings of the ACM on Programming Languages*. New York: ACM, 2017: 66.

[6] Lindner M, Aparicius J, Lindgren P. No panic! Verification of Rust programs by symbolic execution [C]//2018 *IEEE 16th International Conference on Industrial Informatics* (*INDIN*). Washington D C: IEEE, 2018: 108-114.

[7] Saligrama A, Shen A, Gjengset J. A practical analysis of Rust's concurrency story [J]. *ArXiv Preprint ArXiv*, 2019, **1904**: 12210.

[8] Hahn F. *Rust2Viper*: *Building a Static Verifier for Rust* [D]. Zurich: Swiss Federal Institute of Technology, 2016.

[9] Toman J, Pernsteiner S, Torlak E. Crust: A bounded verifier for Rust (N) [C]//2015 30*th IEEE/ACM International Conference on Automated Software Engineering* (*ASE*). Washington D C: IEEE, 2015: 75-80.

[10] Clarke E, Kroening D, Lerda F. A tool for checking ANSI-C programs [C]//*International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin: Springer-Verlag, 2004: 168-176.

[11] Baranowski M, He S, Rakamarić Z. Verifying Rust programs with SMACK [C]//*International Symposium on Automated Technology for Verification and Analysis*. Berlin: Springer-Verlag, 2018: 528-535.

□