Verifying Rust Programs with SMACK*

Marek Baranowski, Shaobo He, and Zvonimir Rakamarić

School of Computing, University of Utah, USA
{baranows,shaobo,zvonimir}@cs.utah.edu

Abstract. Rust is an emerging systems programming language with guaranteed memory safety and modern language features that has been extensively adopted to build safety-critical software. However, there is currently a lack of automated software verifiers for Rust. In this work, we present our experience extending the SMACK verifier to enable its usage on Rust programs. We evaluate SMACK on a set of Rust programs to demonstrate a wide spectrum of language features it supports.

1 Introduction

Rust [12] is a new programming language that aims to enable safe systems programming by means of an elaborate type system, while providing advanced language features such as traits, smart pointers, and closures. It avoids memory safety issues prevalent in programs written in other low-level programming languages such as C/C++ without adding performance overhead often imposed by runtime systems or garbage collectors. Because of these merits, Rust has received a lot of attention from both academia and industry, and it has already been used to implement industrial-strength safety-critical applications, such as web browsers, cloud storage, and embedded software.

Although memory safety is enforced through type checking of Rust programs at compile time, functional correctness (e.g., no violations of user-specified assertions) is not guaranteed. Automated software verifiers based on satisfiability modulo theories (SMT) solvers [3] are a popular choice for assuring the absence of assertion violations. However, building a verifier, or extending an existing one, for a new language is often tedious and time-consuming (e.g., implement a frontend, understand and encode the language semantics). This was done in Rust2Viper [6], which translates Rust programs from the high-level intermediate representation (syntactically similar to Rust) into an intermediate verification language in order to check program correctness. CRUST [14] transforms Rust into C to verify memory safety of unsafe Rust code. As both tools use custom translators, changes to Rust necessitate these to be updated, which is a large undertaking; neither tool appears to be maintained. To the best of our knowledge, currently there are no readily available SMT-based verifiers for Rust.

In this paper, we describe how we enable the verification of Rust programs in the SMACK verifier [11, 13]. An advantage of SMACK is that it is mostly

* Supported in part by the National Science Foundation (NSF) award CNS 1527526.

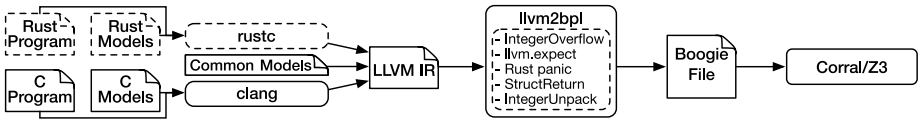

Fig. 1: Toolflow of SMACK.

input-language agnostic as it works by verifying a simple intermediate representation, specifically LLVM IR [10]. Since the official Rust compiler, rustc, can produce LLVM IR code corresponding to Rust programs, a large frontend development effort was not needed as a rich set of LLVM IR features is already supported by SMACK. Rust is an advanced, low-level programming language that controls heap sharing and aliasing using an elaborate type system. Hence, Rust's compiler emits LLVM IR code patterns that are often significantly different from code generated by the Clang compiler, which is the primary target for SMACK. In particular, it emits aliasing patterns that SMACK could not handle well. Nevertheless, we managed to extend SMACK to support the verification of a modern programming language such as Rust at a relatively small cost, and our evaluation shows that it can already handle a variety of key language features.

2 SMACK Software Verification Toolchain

SMACK [11,13] is a software verification toolchain that translates LLVM IR code into Boogie intermediate verification language [2], which is in turn verified using back-end Boogie verifiers such as Corral [9]. Before our Rust effort, SMACK had been predominantly used to verify LLVM IR programs produced by the Clang C compiler. Fig. 1 shows the toolflow of SMACK, which works as follows:

1. The SMACK top-level script automates the entire toolflow. It determines which compiler to invoke and flags to use for program compilation. In the case of C programs, it invokes Clang to generate LLVM IR code, while including SMACK's C language models. The models specify the semantics of common C library functions such as malloc, free, and string operations.
2. The common models file is then linked with the generated LLVM IR file to provide basic verification capabilities. This includes modeling dynamic memory, and support for assertions, assumptions, and nondeterministic values.
3. The core LLVM2BPL component takes an LLVM IR file as input, and produces Boogie code that captures the semantics of LLVM IR instructions; it outputs a Boogie file for verification.
4. Finally, the Corral back-end verifier is invoked on the generated Boogie file, and it uses Z3 [5] as its SMT solver. (Note that SMACK supports other back-end verifiers, which we omitted here.)

In this work, we use Corral in its bounded verification mode, meaning that it unrolls loops and recursion up to a certain user-provided bound.

3 Rust-Driven Extensions to SMACK

Fig. 2 gives a Rust program illustrating the language features that our SMACK extensions leverage or support. Rust's foreign function interface (FFI) allows

```
1  #[macro_use] mod smack; use smack::*;
2  extern{fn fib_c(n:u64)->u64;}
3  fn fib(x: usize, cache:&mut Vec<u64>) {
4    for i in 2..x+1 as usize
5      { cache[i]=cache[i-1]+cache[i-2]; }
6  }
7  fn main() {
8    let n=5u64.nondet();
9    assume!(n > 2);
10   let mut cache=vec![0; n+1];
11   cache[0]=0; cache[1]=1;
12   fib(n, &mut cache);
13   let c_result=unsafe{fib_c(n)};
14   assert!(cache[n]==c_result);
15 }
```

```
1  typedef unsigned long ul;
2  ul fib_c(ul x) {
3    ul a = 0, b = 1;
4    for (ul i=0; i<x-1; i++) {
5      ul tmp = a;
6      a = b;
7      b = a + tmp;
8    }
9    return b;
10 }
```

Fig. 2: Rust program that checks the equivalence between the Rust (`fib`) and C (`fib_c`) implementations of the Fibonacci function.

zero-cost interaction with C code, verification of which had already been extensively supported by SMACK. As a result, we are able to reuse SMACK's C models as well as perform cross-language verification of Rust programs containing calls to external C functions (line 13). For example, we implemented macros `assume` (line 9) and `assert` (line 14) to expand into calls to SMACK's built-in C functions. Line 8 invokes the `nondet` function that introduces nondeterministic unconstrained values. Note that we implemented these so that programs can be easily compiled into executables even with SMACK annotations present — in that case `nondet` is replaced with value `5` in the example.

Instead of being undefined or triggering wrap-around behaviors as in C, integer overflows in Rust are checked and can lead to *program panic*. For example, while not visible at the source level, the signed integer addition operation at line 5 may optionally be checked for integer overflows via the Rust compiler emitting LLVM arithmetic with overflow intrinsics; we had to extend SMACK to support such intrinsics. Finally, unlike C, standard libraries and modern language constructs such as the `Vec` library (line 10) and iterators (line 4) are abundant in Rust code. Modeling these libraries and language constructs is challenging yet essential to build a practical Rust verifier; SMACK's modeling mechanism allowed us to implement models for common Rust libraries. We describe some of these extensions in more detail next.

3.1 Supporting Rust-Generated LLVM IR Constructs

The LLVM IR code that rustc emits contains several key constructs that are not used in IR code produced by Clang. Hence, we had to extend SMACK to add support for such constructs.

Types. The Rust compiler generates load/store instructions of the LLVM `i1` data type, which is almost never emitted by Clang. We added support for such instructions by zero-extending their operands to `i8` when a store operation occurs, and casting them back when they are loaded.

Instructions operating on LLVM structure types occur frequently in rustc-generated IR code, while Clang-generated IR almost always uses only primitive

types. For example, it is a common practice for Rust programmers to use the `Option` type as the return type of functions. It is generic over type `T` and represented in LLVM IR as structure type `{T,i1}`, where setting `i1` is used to indicate a valid return value. Moreover, load/store instructions over structures are frequently generated by rustc, but not by Clang. Hence, SMACK did not have elaborate support for such instructions.

We support such instructions by modeling LLVM structure types using uninterpreted functions that constrain each field. For example, value `{v,1}` of type `{T,i1}` is represented using an integer `s` with constraint `f(s,0)==v && f(s,1)==1`, where `f` is an uninterpreted function with the second argument being the index of a structure field. Such encoding allows us to model two basic LLVM structure instructions `extractvalue` and `insertvalue` that read and write structure fields, respectively. Loads and stores of structures into memory are recursively translated into a sequence of instructions that generate load/store for each field of primitive type, in conjunction with the two aforementioned instructions. This extension enables SMACK to handle structure constructs without us having to introduce extensive modifications to its underlying memory model.

Integer Packing. The Rust compiler frequently packs smaller structures into 8-byte integers. For example, rustc optimizes loading of a structure of type `{i32,i32}` into loading of `i64`. This requires less scalable bit-precise reasoning to be selected in SMACK to avoid false bugs [7]. Hence, we added an analysis pass to SMACK that detects load/store instructions with pointer operands of integer element type that refer to structures. We translate such instructions to load/store directly from/into structure fields (following the encoding described earlier), thereby essentially avoiding packing. This approach helps to scale the verification of Rust programs by avoiding the need for bit-precise reasoning.

Intrinsics. We added support for two types of LLVM intrinsics heavily used by rustc: `llvm.expect` and arithmetic with overflow. The Rust compiler emits the LLVM intrinsic `llvm.expect` as an optimization hint. We modified SMACK to transform a call to this intrinsic into essentially a no-op. As future work, we will explore leveraging such hints to speed up verification.

```
1  $a2 := $zext.i8.i16($a);
2  $b2 := $zext.i8.i16($b);
3  $x2 := $add.i16($a2, $b2);
4  $x  := $trunc.i16.i8($x2);
5  $flag  := $ugt.i16($x2, 255);
6  assert !$flag;
```

Fig. 3: Translation of an unsigned 8-bit checked-addition intrinsic, where `$a` and `$b` are the operands and `$x` is the sum.

The Rust compiler typically emits instructions for checking all integer operations for overflow through the use of LLVM arithmetic with overflow intrinsics, such as `llvm.uadd.with.overflow.i32`. The intrinsics indicate the sign and bitwidth in which to perform the given operation. We extended SMACK with an integer overflow checking pass that replaces the intrinsics with instruction sequences implementing the corresponding overflow checking. Fig. 3 shows an example translation. Lines 1 and 2 extend the precision of the arguments to double the original bitwidth, thereby avoiding potential overflow. Line 3 computes the result of the addition, while line 4 converts the result back to the original bitwidth. Line 5 determines whether the

Table 1: Summary of the benchmark suite we developed.

Benchmark category	#Files	LOC	Features demonstrated
functions	8	153	Function calls, closures, recursion
generics	6	55	Generic functions, structures, traits
ifc	4	214	Information flow control example
loops	4	35	Range-based for loops
ops	12	171	Basic operations, overflows
structures	4	76	Creation, passing, returning of structures
vector	6	88	Dynamic memory management
memory-safety	4	58	Memory safety verification
cross-language	4	48	Combining Rust and C

operation overflowed, while line 6 checks it. Note that the translation shown in Fig. 3 is not optimal for dynamic checking since we optimized it for SMT-based verification with SMACK. Furthermore, while the conversion of the intrinsic is always performed, checking is made optional following the convention that it is disabled in the release mode.

3.2 Modeling Rust Libraries

Standard Rust libraries define most of the language's containers as generic over the contained type, and generate the corresponding code for the container when the program is compiled. However, the generated code is heavily optimized for performance, and contains constructs and functions that are difficult for SMACK to analyze, such as custom allocators. Hence, we leveraged SMACK's existing modeling capabilities to write models for popular Rust data structures, such as vector (`Vec`). Vector is a dynamically-sized array used in many Rust programs as well as for implementing other data structures such as stacks and queues. Currently, our vector model supports dynamic resizing, push, pop, get and mutable get, and indexing among other features. The model resides in a separate file, which SMACK automatically links as a Rust module.

4 Experiments

4.1 Microbenchmarks

We developed a benchmark suite containing various Rust language features to test the SMACK extensions we developed.[1] Table 1 summarizes our benchmark suite. Every category includes both correct and buggy benchmark versions. Some notable included features are:

- The functions category tests recursion and passing closures as arguments.
- The generics category implements a generic trait for two generic structures. A statically dispatched function is then invoked on the structures.
- The vector category tests dynamic resizing and indexing of the Rust vector.
- The cross-language category contains Rust programs that invoke C functions, including the Fig. 2 benchmark.

[1] For our tool and benchmarks see `https://github.com/smackers/smack`

Table 2: Summary of the real-world programs we verified using SMACK. Column **Time** shows the runtime of applying SMACK to verify a property.

Program	Checked property	LOC	Time
`uptime`	General assertion	81	2s
`expr`	Signed integer overflow	137	5min
`factor`	Unsigned integer overflow	100	50s
	Functional correctness		17min

- In the memory safety category, we verify the absence of buffer overflows and memory leaks arising from C-allocated arrays in unsafe Rust programs.
- The ifc category contains the *information flow control* (IFC) example from related work [1]. IFC models an access control method where access authority can only be increased. Using nondeterministic access levels, we verify that the IFC Rust implementation only allows access to the appropriate authority.

Currently, SMACK verifies most benchmarks in under 20 minutes. The only exception is the full-blown IFC benchmark version that takes several hours to complete. The development of the benchmark suite helped us to identify key language features that SMACK struggled with, and hence it guided our efforts.

4.2 Real-World Programs

To better judge the quality of our implementation, we tested SMACK on three real-world programs, *uptime*, *expr*, and *factor*, from the *uutils* project [4]. The project is a popular repository on GitHub (starred more than 4000 times) containing Rust reimplementations of the GNU core utilities. Table 2 shows the properties we verified for each program, their size, and the runtime of SMACK. We slightly modify all the programs to simplify the verification processes. Most notably, we replace the return values of external library calls with nondeterministic values, and we ignore string literals by redefining macros that accept string arguments, such as `println!`, to empty expressions.

In the *uptime* utility, which prints the uptime of a machine, we verify that the reported uptime is 0 only when the system calls related to reporting the uptime also return 0. SMACK generates an error trace through the Rust program where an uptime of 0 is erroneously reported when certain resources are unavailable; GNU's version of *uptime* reports an error in this scenario. We reported this problem to the developers, who issued a fix.[2] The *expr* utility evaluates a string argument as an arithmetic expression. We check this program for signed integer overflows using SMACK. Our input to *expr* is the addition of two nondeterministic 64-bit integers, and SMACK discovers input values that trigger signed integer overflow. GNU's version of *expr* either reports an error, or uses unlimited precision, rather than reporting an overflowed result. We again reported the outcome to the developers, who issued a fix.[3] In the *factor* utility, we focused on verifying individual functions in its numeric library, namely `sm_mul` and `big_mul`.

[2] `https://github.com/uutils/coreutils/issues/1195`
[3] `https://github.com/uutils/coreutils/issues/1194`

Both of these functions take 3 arguments a, b, and m, and compute $(a \cdot b)\%m$. We verify several properties related to integer overflows, and that `sm_mul` indeed performs the specified computation. Note that we reduced the integer bit-width to 8 bits to speed up verification.

5 Limitations and Future Work

While the described extensions we made to SMACK enable its usage on many Rust programs, some work remains. Rust programs extensively rely on Rust's standard libraries. While we implemented models for the most common ones, such as `Vec`, we plan to model a more substantial subset in the future. An additional feature we plan to add is checking of unsafe pointers to ensure they obey the semantics of the Rust's borrow system. In particular, we want to check pointers from external functions. The Rustbelt [8] project gives the conditions for which pointers generated from unsafe Rust code can be verified to be safely used. Since Rust enables legacy code to be used within a project, this feature will enable developers to verify their wrappers adhere to Rust's aliasing semantics. Finally, concurrent programming is an important feature of Rust, and we plan to support it in SMACK in the near future.

References

1. A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk. System programming in Rust: Beyond safety. In *HotOS*, 2017.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.
3. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. In *SMT*, 2010.
4. Cross-platform Rust rewrite of the GNU coreutils. `https://github.com/uutils/coreutils`.
5. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
6. F. Hahn. Rust2Viper: Building a static verifier for Rust. Master's thesis, ETH, 2016.
7. S. He and Z. Rakamarić. Counterexample-guided bit-precision selection. In *APLAS*, 2017.
8. R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. RustBelt: Securing the foundations of the Rust programming language. In *POPL*, 2017.
9. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV*, 2012.
10. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.
11. Z. Rakamarić and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In *CAV*, 2014.
12. The Rust programming language. `https://www.rust-lang.org`.
13. SMACK software verifier and verification toolchain. `http://smackers.github.io`.
14. J. Toman, S. Pernsteiner, and E. Torlak. CRUST: A bounded verifier for Rust. In *ASE*, 2015.