Leveraging Compiler Intermediate Representation for Multi- and Cross-Language Verification[⋆]

Jack J. Garzella, Marek Baranowski, Shaobo He, and Zvonimir Rakamarić

School of Computing, University of Utah
Salt Lake City, UT, USA
`jjgarzella@gmail.com`,
{`baranows,shaobo,zvonimir`}`@cs.utah.edu`

Abstract. Developers nowadays regularly use numerous programming languages with different characteristics and trade-offs. Unfortunately, implementing a software verifier for a new language from scratch is a large and tedious undertaking, requiring expert knowledge in multiple domains, such as compilers, verification, and constraint solving. Hence, only a tiny fraction of the used languages has readily available software verifiers to aid in the development of correct programs. In the past decade, there has been a trend of leveraging popular compiler intermediate representations (IRs), such as LLVM IR, when implementing software verifiers. Processing IR promises out-of-the-box multi- and cross-language verification since, at least in theory, a verifier ought to be able to handle a program in any programming language (and their combination) that can be compiled into the IR. In practice though, to the best of our knowledge, nobody has explored the feasibility and ease of such integration of new languages. In this paper, we provide a procedure for adding support for a new language into an IR-based verification toolflow. Using our procedure, we extend the SMACK verifier with prototypical support for 6 additional languages. We assess the quality of our extensions through several case studies, and we describe our experience in detail to guide future efforts in this area.

Keywords: Verification · Multi-Language · Cross-Language · Compiler Intermediate Representation.

1 Introduction

The evolution of software systems motivates the need for new programming languages with novel features to better adapt to new programming goals, such as improving program safety or easing programming. For example, Rust [45]

⋆ This work was supported by funding from the Undergraduate Research Opportunities Program at the University of Utah awarded to Jack J. Garzella, the National Science Foundation awards CNS 1527526 and CCF 1837051, and a gift from the VMware's University Research Fund.

is a novel performant systems programming language with guaranteed memory safety and safer parallel programming. The D programming language also aims to provide memory safety and high-level programming primitives, while maintaining performance and low-level programming capabilities. Swift and Kotlin employ modern programming language concepts to reduce language verbosity and allow for easier programming. On the other hand, there are legacy languages that are still widely used in certain domains. For example, Fortran dominates as a programming language of choice among domain scientists, such as physicists and chemists. To further complicate matters, developers typically build real-world software systems using a combination of several programming languages by implementing various components in different languages depending on the requirements and trade-offs.

Software verification is integral to improving software quality. Among software verification techniques, the ones based on *satisfiability modulo theories* (SMT) have become increasingly popular due to its rigor, automation, and tremendous scalability improvements of the past two decades. There are numerous SMT-based tools available with various capabilities, features, and trade-offs (e.g., [1,2,3,5,6,7,8,10,11,12,13,14,15,23,30,37,39,41,47,51]). However, the traditional way to prototype a program verifier, by implementing all of its components (e.g., front-end, SMT formula generator) from scratch, is extremely time-consuming and heavily coupled with target language details. Hence, despite widespread usage of many programming languages and their combinations, automatic software verifiers still predominantly target the C programming language, thereby denying many developers a valuable tool for ensuring safety and reliability of their programs. It would be ideal if program verifiers can keep pace with the development of emerging programming languages such that users can benefit from this rigorous software analysis technique.

LLVM [35,34] is a popular, open source compiler infrastructure, which features an assembly-like intermediate compiler representation, known as the LLVM intermediate representation (IR). LLVM IR has been leveraged to build program verifiers [41,26,37], since LLVM IR frees the verifier designer from the error prone tasks of modeling the semantics and parsing of the source language [41]. In theory, a well-designed verifier targeting LLVM IR should be able to support any programming language that has a compiler front-end capable of emitting LLVM IR, as well as their combinations. However, to the best of our knowledge, verifiers built upon LLVM IR only support C/C++, and there has been no systematic study exploring how well such verifiers extend to support other programming languages. This is despite the fact that compilers for other programming languages can produce LLVM IR, such as the Rust compiler and the Flang compiler [24] for Fortran.

The goal of this paper is to investigate the feasibility of multi- and cross-language verification that leverages an intermediate compiler representation (e.g., LLVM IR). We chose to use SMACK [41,48] as an exemplar mature verification toolchain based on LLVM IR. As our first step, we prescribe a procedure for adding a new language to such a tool chain, consisting of interoperating with language models, compiling into IR, and adding models for missing language features. Then, we evaluate our procedure by prototyping in SMACK support for 6 additional programming languages with compilers capable of emitting LLVM IR. Since SMACK is an LLVM IR-based verifier with extensive, preexisting support for the C programming language, it is a good basis for building a verifier for a new language. Additionally, the modular design of SMACK is a desirable feature due to its decoupling of source language details from the verification task through LLVM IR.

We performed several empirical case studies based around multi- and cross-language verification. To this end, we created a microbenchmark suite that tests support for key language features such as dynamic dispatch. We also explore cross-language verification using an example that exercises the interaction between Rust, Fortran, and C code. This is an important task as many new programming languages include a facility to invoke C functions natively to support legacy code. We summarize our experience and lessons learned. We observe that depending on features present in a programming language, SMACK may not always work out-of-the-box. This is due to either SMACK not supporting certain LLVM IR patterns or lack of suitable models for the standard libraries and runtime. We discuss the process of improving SMACK's support for various LLVM IR patterns, which involves modeling additional IR instructions. We also describe how we provide basic models for standard libraries and runtimes for several languages we added.

To summarize, our main contributions are as follows:

- We prescribe a procedure by which support for new programming languages can be added to an IR-based software verifier.
- By following our procedure, we added basic multi-language support to the SMACK software verifier for 6 additional programming languages: C++, Objective-C, D, Fortran, Swift, and Kotlin. We also made the preexisting support for Rust more robust, and hence we include it in our evaluation.
- We developed a suite of microbenchmarks for testing the robustness of multi-language verification, which implements key language features across all of the additional languages.
- We performed several multi- and cross-language case studies using SMACK, and we report on our experience and lessons learned in the process to guide future efforts in this area.

2 Related Work

In the past decade, numerous software verifiers have been developed on top of the LLVM compiler infrastructure (e.g., [5,6,20,26,37,41]), while others leverage GCC in a similar fashion (e.g., [21,27]). The authors of these tools have realized the benefits LLVM offers for the development of verifiers, such as a canonical intermediate representation and readily available analysis and optimization passes. In particular, verifiers such as SAW [20], LLBMC [37], SeaHorn [26], and SMACK [41] all take as input LLVM bitcode produced by the clang compiler, which is then handled differently by each verifier. SAW (Software Analysis Workbench) uses symbolic execution with path merging to produce formal models from LLVM IR in a dependently-typed intermediate verification language; it reasons about the resulting models using rewriting or external satisfiability solvers. LLBMC generates its own intermediate logical representation (ILR) based on the input LLVM IR program, and leverages SMT solvers to check the formula derived from ILR. SeaHorn encodes an input LLVM IR program into Horn clauses, which are further solved using different techniques. SMACK translates LLVM IR into an intermediate verification language called Boogie, which is then verified using different back-end verification engines. Both LLBMC and SeaHorn support both C and C++ (to some extent), while SMACK has mature support only for C. Unlike the aforementioned tools, ESBMC [14] leverages clang just as a parser to obtain ASTs, and it does not use LLVM IR; it supports both C and C++. Despite the popularity of LLVM IR in building software verifiers, to the best of our knowledge, we are the first to study the feasibility of leveraging an intermediate representation to perform multi- and cross-language verification.

Some of the languages we considered in this paper have standalone verifiers. For instance, CRUST [50] verifies unsafe Rust code by translating a Rust program into a C program, and then using an off-the-shelf C verifier. Rust2Viper [28] and its successor Prusti [4] are modular verifiers for Rust programs that include a design-by-contract specification language. As input they take an annotated program in the Rust's high-level intermediate representation, which simplifies and canonicalizes complex language constructs. Then, such a program is encoded into the Viper intermediate verification language [38] for verification. These approaches would require substantial effort to support verification of other programming languages. To the best of our knowledge, there are no verifiers available targeting Swift, Kotlin, D, or Fortran.

There are software verifiers that process the input languages directly as opposed to delegating to a compiler IR. For example, CPAchecker [7], Ultimate Automizer [30], CBMC [12], SAW [20], and CIVL [47] all leverage off-the-shelf or custom parsers to generate abstract syntax trees (ASTs), and then process these ASTs in various ways to carry out verification. (Note that CPAchecker, CBMC, and SAW support LLVM IR as well.) The verifiers in this category can potentially perform multi-language verification, but often at the expense of having to perform some language-specific work. For example, SAW's work-in-progress support for Rust involves implementing a designated symbolic simulator. While taking compiler IR as input has its drawbacks over directly handling input languages, such as losing type information and precise debugging data, it also demonstrates an advantage in the context of multi-language verification — only the details of one language (namely compiler IR) need to be addressed. This implies that supporting a new programming language does not require supporting all the new constructs that it brings to the table. For example, C++ templates are completely compiled away at the LLVM IR level. Instead, only the new program constructs in the IR that are not yet supported, but are used by the new language, need to be modeled.

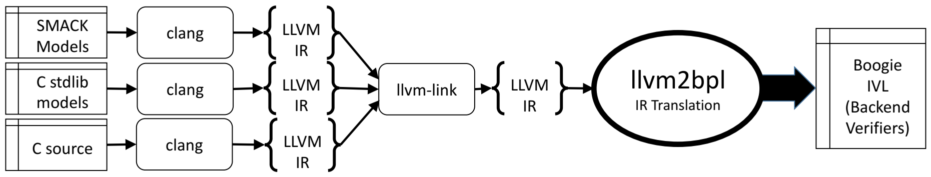

Fig. 1. Toolflow of SMACK.

3 SMACK Software Verification Toolchain

SMACK [9,41,48] is an open source, modular software verification toolchain. The core component of SMACK converts LLVM IR code into Boogie intermediate verification language [18]. The remainder of the SMACK toolchain handles details such as compiling the source program into LLVM IR and invoking a Boogie verifier. Its modular nature decouples source language details from verification by leveraging compiler front-ends to translate programs into the Boogie intermediate verification language through LLVM IR. Before we implemented the multi-language extensions described in this paper, SMACK had been predominantly used to verify LLVM IR programs produced by the clang C compiler.

Fig. 1 shows the current toolflow of SMACK, which proceeds as follows:

1. SMACK first invokes clang, the LLVM's C compiler, to compile the input program, SMACK models, and C standard library models (e.g., strings, pthreads, math). SMACK models contain various verification primitives (e.g., for generating nondeterministic values) and the encoding of the memory model for handling of dynamically allocated memory. All of the models are written in C since SMACK provides a convenient mechanism for interoperating with the underlying Boogie code, which we describe below.
2. SMACK links together all of the generated LLVM IR files into one LLVM IR program.
3. The core LLVM2BPL component of SMACK transforms an LLVM IR program produced by the previous step into a semantically equivalent Boogie program.
4. Finally, a back-end verifier, such as Corral [33], verifies the generated Boogie program using an SMT solver, such as Z3 [17].

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.

SMACK models verifier primitives and memory models through the use of `__SMACK_code` function. This C routine takes a formatted string as a parameter, and is declared in the SMACK header files, but not implemented in any models. When LLVM2BPL comes across a call to this function, instead of translating the function call, it simply inserts the parameters into the Boogie code snippet passed as string; this functionality is akin to C's inline assembly. This allows for Boogie code or ghost variables to be injected into the translation, giving an easy

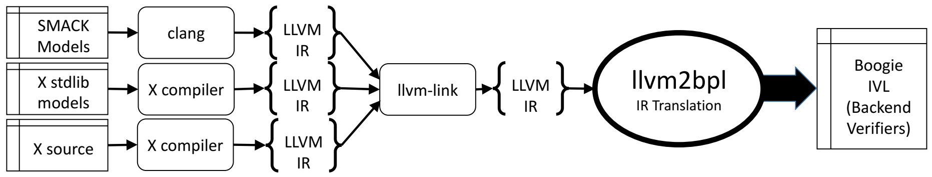

Fig. 2. Toolflow for adding support for programming language X to SMACK.

way to encapsulate routines like `assume` which are not normally available in C. It also provides an abstraction that can be used for any primitive or model.

4 Procedure for Adding a Language

In this section, we introduce our prescribed procedure for adding support for a new programming language into an LLVM-based software verifier. The procedure is based on our study of adding languages to SMACK, but the lessons we learned generalize to other verifiers that have similar architecture. By showing how the procedure of adding a new language to SMACK is relatively straightforward, we also motivate the adoption of a SMACK-like verifier architecture. Note that while our procedure is focused on LLVM, we expect that a similar process could be adopted for any IR-based verifier. We structure our procedure into three tasks: interoperating with language models, compiling into compiler intermediate representation (IR), and adding models for missing language features.

4.1 Interoperating with Language Models

A software verifier has to encode the desired semantics of an input programming language in order to perform verification. In the case of an LLVM-based software verifier, that typically amounts to providing a memory model in addition to models for LLVM IR statements generated by the chosen compiler. A memory model encodes dynamic memory allocation, pointer dereferencing, and memory accesses. Adding a new programming language necessitates for the verification to be able to interoperate with the mentioned models.

The architecture of SMACK allows for a new programming language to easily interoperate with SMACK models, as Fig. 2 shows. First, SMACK's models for LLVM IR instructions are general and internal to SMACK, and hence they can be shared across all languages that are compiled into LLVM IR. Second, SMACK's memory model [42] is encoded as a regular C language header and its accompanying implementation. This is achieved using the convenient `__SMACK_code` mechanism described in the previous section, which allows for the low-level model encoding to be done at the level of C. We must be able to link an input program with this header in order to interoperate with the memory model. According to our experience, most languages have interoperability with the C language as a feature. Hence, linking against the SMACK's memory model in a new programming language is an easy task. It is worth emphasizing that since the code in the new language is linking with the C code of the memory model, every verification in a new language is already a cross-language verification.

4.2 Compiling and Linking into Compiler IR

As opposed to verifiers that operate directly on program source, an IR-based verifier needs for the input program to be first compiled into the chosen IR. In the case of the LLVM compiler infrastructure, there are many popular programming languages with front-ends that output LLVM IR. Hence, producing LLVM IR for the programming language of choice is typically straightforward. Once the input program is compiled into IR, it gets linked with the SMACK models, which are written in either C (common ones) or the target language (language-specific ones) and automatically compiled by SMACK. The resulting linked IR file is in turn handed over to the SMACK verifier for processing (see Fig. 2). A verifier needs a program entry point, such as function `main` in C, to know from where to start the verification process. The LLVM IR specification does not prescribe a well-defined entry point, and hence language developers are free to choose how to define the entry point for a program in their language — most languages define entry points other than `main`. Thus, we either implement a simple post-processing step to mark the program's entry point, or manually specify it in SMACK's command line, which is in turn passed to the SMACK verifier.

4.3 Adding Models for Missing Language Features

When adding a new language, we typically observe three categories of models that might be missing in a verifier: unsupported LLVM IR instructions, runtime features, and standard libraries.

LLVM IR is an extensive format comprised of more than one hundred instructions and intrinsics [36], many of which are not commonly used. Hence, when adding a new language, its compiler can potentially generate IR instructions or intrinsics that a verifier has not encountered before, and hence are potentially not supported. This necessitates updating the verifier to account for the semantics of such instructions. Our experience shows that in the case of a mature LLVM-based verifier such as SMACK, we rarely encountered a new compiler generating instructions/intrinsics that it did not already support.

Most languages require the use of a standard library to achieve almost anything of practical value. SMACK provides extensive models for the C standard library, such as pthreads, strings, and math. However, every programming language comes with its own standard libraries that it relies on, with different specifications. A language may rely heavily on its standard libraries, even if it has little or no runtime. For example, unlike C, D, and Fortran, languages such as Rust and Swift implement arrays as a compound type in the standard library. Hence, models for the standard libraries of a new programming language have to be written manually mostly from scratch. This is the most tedious and time consuming aspect of adding support for a new language. To somewhat alleviate the burden of developing models, SMACK architecture enables for a user to write models for standard library functions as header files that are linked with input programs. This is a convenient mechanism for writing such models since it requires no updates to be made to the actual SMACK verifier source code.

Some languages are also heavily dependent on runtime functionality, such as Objective-C, Swift, and Kotlin. For example, Objective-C relies heavily on its runtime for method dispatch, memory management, and other basic features. Code from the runtime is not included in the LLVM IR which is generated by the compiler. Therefore, runtime functions must be modeled before any nontrivial verification. Languages like C, Fortran, and D have very few runtime models, and as a result these are much easier languages to verify out-of-the-box.

5 Case Studies

We perform three case studies to assess the feasibility and ease of adding support for an additional input programming language into an IR-based software verification toolchain such as SMACK. Before we describe each case study in detail, we provide our strategy for selecting input programming languages we attempted to support.

5.1 Choice of Input Programming Languages

There are numerous programming languages in existence today, and clearly it would be infeasible for us to handle all of them. Hence, for our case studies, we used the following criteria for choosing which languages to add support for in SMACK. First, we selected popular languages from the Stack Overflow Developer Survey [19] that can be compiled down into LLVM IR. Second, we performed a thorough search for other languages that can be compiled into LLVM IR, and are important in certain domains but less popular overall (i.e., domain specific). Then, we prune this list based on our requirements on the front-end, which are as follows:

1. Compile input programs into LLVM IR *Ahead-of-Time*
2. Target the same version of LLVM as SMACK
3. Be stable and under active development

Table 1 lists the languages we considered and their relevant properties.

As SMACK directly translates an entire program from LLVM to Boogie, it requires all related definitions to be available at translation time. A *Just-in-Time* compiler does not have a whole program readily available in the LLVM IR format for SMACK to process. Therefore, *Ahead-of-Time* compilers are the only ones that can currently be used with SMACK. LLVM does not preserve backwards compatibility of the LLVM IR format. Hence, the LLVM version supported by SMACK and the chosen language front-end have to match. The used version of SMACK supports LLVM 3.9, and hence our requirement is for a language

Table 1. List of programming languages we considered and their properties. Column **Ahead-of-Time** shows whether an Ahead-of-Time compiler is available for the language; column **LLVM 3.9** indicates whether the needed LLVM version is supported; column **Active** indicates whether the compiler is under active development, while column **Stable** indicates if there is a stable release; finally, column **Used it?** shows whether we used the language in our evaluation.

Language	**Compiler**	**Ahead-of-Time**	**LLVM 3.9**	**Active**	**Stable**	**Used it?**
C	clang	✓	✓	✓	✓	✓
C++	clang	✓	✓	✓	✓	✓
Fortran [24]	flang	✓	✓	✓	✓	✓
D [16]	ldc	✓	✓	✓	✓	✓
Rust [45]	rustc	✓	✓	✓	✓	✓
Objective-C	clang	✓	✓	✓	✓	✓
Swift [49]	swiftc	✓	✓	✓	✓	✓
Kotlin [32]	kotlinc	✓	✓	✓	✓	✓
Scala [46]	scala-native	✓	✓	✓	✓	✗
C# [52]	llilc	✓	?	✓	✓	✗
Haskell [29]	ghc	✓	✓	✓	✓	✗
Julia [31]	julia	✗	✓	✓	✗	✗
Go [25]	llgo	✓	✓	✗	✗	✗
Python [40]	pyston	✓	?	✗	✗	✗
Ruby [43]	ruby-llvm	✓	✗	✗	✗	✗
Java [22]	falcon (Azul)	✗	✗	✓	✗	✗

front-end to support the same LLVM version. We sometimes had to revert to an older front-end version to satisfy this requirement. For example, Swift 4.2 does not target the required LLVM 3.9, but Swift 3.0 does. Of course, as SMACK gets updated to newer LLVM versions, this requirement will change as well. In order to limit our focus to compilers of practical value, we ignore the ones that are not stable and under active development.

Of the LLVM-IR-based languages in the developer survey, there are 4 that satisfy our criteria: C, C++, Objective-C, and Swift. Kotlin, Scala [46], and C# [52]) have compilers that are not yet fully mature, but are stable and under active development. We chose Kotlin as the representative of this “managed language into LLVM IR” category. In addition to the popular languages listed on Stack Overflow, there are other notable, stable languages that target LLVM. Most of these are tailored for domain-specific coding. The Rust programming language [45] is a performant systems language with an emphasis on safety and concurrency. The D programming language [16] is a mature language which offers low-level control combined with high-level abstractions. Both Rust and D target the systems programming community. Fortran is primarily used in the scientific

```c
int cap(int x) {
  int y = x;
  if (10 < x) { y = 10; }
  return y;
}

int main(void) {
  assert(cap(2) == 2);
  assert(cap(15) == 10);
  int x = nondet_int();
  assert(cap(x) <= 10);
}
```

```swift
func cap(_ x: Int) -> Int {
  var y = x
  if 10 < x {
    y = 10
  }
  return y
}

assert(cap(2) == 2)
assert(cap(15) == 10)
let x = Int(nondet_int())
assert(cap(x) <= 10)
```

```rust
fn cap(x: usize) -> usize {
  let mut y = x;
  if 10 < x {
    y = 10;
  }
  return y;
}


fn main() {
  let two = cap(2);
  let ten = cap(15);
  assert!(two == 2);
  assert!(ten == 10);
  let x = nondet_int();
  assert!(x <= 10);
}
```

```fortran
pure function cap(x)
  integer, intent(in) :: x
  integer :: cap, y
  y = x
  if (10 < y) then
    y = 10
  end if
  cap = y
end function

program main
  integer :: cap, x
  call assert(cap(2) == 2)
  call assert(cap(15) == 10)
  x = nondet_int()
  call assert(cap(x) <= 10)
end program main
```

Fig. 3. Microbenchmark with program versions in C, Swift, Rust, and Fortran.

programming community, since it provides support for parallel processing and compatibility with legacy code for projects that span multiple decades. The only language we do not use which satisfies our criteria is Haskell. Its LLVM back-end is not compatible with SMACK, mainly because the entry point for the code is not included in a standalone bitcode file.

5.2 Case Study 1: Microbenchmarks

We developed a microbenchmark suite to evaluate the quality of the support for different languages we implemented in SMACK. We crafted each benchmark to exercise across all languages (8 languages total, see Table 1) a specific language feature we deem important, meaning that a benchmark consists of a number of programs, each implementing the chosen feature in a different language. In addition, we injected a property to be verified into each benchmark using assertions.

Table 2. Characteristics of our microbenchmark suite. Column **LOC** is the average number of lines of code per benchmark across supporting languages; column **#Lang** is the number of languages supporting the features tested.

Benchmark	**Features Tested**	**LOC**	**#Lang**
basic	basic assertions	8	8
compute	integer arithmetic	12	8
function	functions, if-then-else, nondet values	19	8
forloop	for loops	14	8
fib	recursion	20	8
compound	objects and structures, fields	18	8
array	array creation, array access	10	8
pointer	dynamic memory allocation, references	14	6
inout	updates via side effects	17	7
method	single type dispatch	26	6
dynamic	polymorphic dispatch	29	6

Hence, there are several (at least two) program versions per each benchmark-language pair: a passing version (i.e., no failing assertions) and a failing version (i.e., a failing assertion) for each assertion. Figure 3 shows several variations of one of our microbenchmarks. Table 2 gives basic characteristics of our microbenchmark suite.[1]

We designed the microbenchmarks to be as small as possible, and yet still test a particular language feature. Hence, a failing benchmark is a good indicator of which feature is not properly supported by a verifier. While our microbenchmarks are not based on real-world programs, since they focus on common and widely-used language features, being able to handle them is a prerequisite to verifying real-world code. One can think of our microbenchmarks as being *litmus tests* for various key language features.

Not all benchmarks have a program version for every language since not all language features are supported across the board. For example, languages without support for object-oriented programming (e.g., C, Fortran) do not have versions of the corresponding benchmark (i.e., method). Then, Swift and Kotlin do not have syntactic support for pointers, and so we could not implement versions of the pointer benchmark for these languages. We also sometimes had to implement benchmark versions differently across languages. For example, we implemented the dynamic dispatch benchmark in Rust using *traits* instead of inheritance. As another example, we implemented the inout benchmark in Swift and Fortran using a specific mutable-parameter syntax, while in most other languages we replicate this feature using pointers. We did not implement this bench-

[1] We made our microbenchmark suite publicly available at `https://github.com/soarlab/gandalv`.

Table 3. Results of running SMACK on our microbenchmarks. Symbol ✓ marks passing, symbol ✗ failing, and N/A marks benchmarks that do not have a version for the corresponding language.

Benchmark	**C**	**C++**	**Objective-C**	**Rust**	**Fortran**	**D**	**Swift**	**Kotlin**
basic	✓	✓	✓	✓	✓	✓	✓	✓
compute	✓	✓	✓	✓	✓	✓	✓	✓
function	✓	✓	✓	✓	✓	✓	✓	✓
forloop	✓	✓	✓	✓	✓	✓	✗	✓
fib	✓	✓	✓	✓	✓	✓	✓	✓
compound	✓	✓	✗	✓	✓	✓	✓	✗
array	✓	✓	✗	✓	✓	✓	✗	✗
pointer	✓	✓	✓	✓	✓	✓	N/A	N/A
inout	✓	✓	✓	✓	✓	✓	✓	N/A
method	N/A	✓	✗	✓	N/A	✓	✗	✓
dynamic	N/A	✗	✗	✓	N/A	✗	✗	✗

mark in Kotlin since it has no support for pointers, nor for mutable-parameter syntax.

Table 3 summarizes the results of running SMACK with our extensions on the microbenchmarks. Overall, SMACK successfully discharges all available program versions of benchmarks in C, Fortran, and Rust. For C++ and D, the main missing language feature that we still have to add support for is dynamic dispatch. Swift, Objective-C, and Kotlin need more work before SMACK could support language features beyond just the very basic ones. The primary cause for the failing benchmarks is SMACK lacking models of standard libraries and runtime.

Swift, Objective-C, Kotlin, and Rust are all very library- and runtime-dependent. Hence, there are many basic language features that SMACK does not capture precisely (i.e., that are not modeled in SMACK), which causes even some small benchmarks to fail. As we note in Sec. 6.2, developing such models for a verifier is typically a tedious manual process, and is an exercise we could not perform for all languages in the limited amount of time we had for our case study. However, the version of SMACK we used already contained models of several popular Rust standard library functions. Hence, in our experiments, the other three languages have more failing benchmarks than Rust, which are caused by the following unmodeled functionality:

Swift range structures (forloop), array subscripts (array), dispatching functions via function pointers (method)

Obj-C objc-msg-send for dispatching methods via function pointers (compound, method), NSArray class (array)

Kotlin dynamic object instantiation (compound, array)

Table 4. Time commitment summary for adding a new language.

Procedure Step	Person-Hours
Write code to interoperate with SMACK models	8
Compile and link at the LLVM IR level, test	3
Add and model missing functionality	4
Total:	15

5.3 Case Study 2: Adding a Language

In order to get a rough estimate of the time commitment required to add support for a new language to an IR-based verifier, we conducted an informal timed exercise where an undergraduate student working on this project (Jack J. Garzella, one of the coauthors) added support for one additional language, the D programming language, to SMACK. During the exercise, the student followed the steps we prescribe in our procedure from Sec. 4, and we measured elapsed time (in hours) it took him to accomplish each of the steps. Table 4 summarizes our measurements.

The student had no experience with D beyond implementing the microbenchmarks; he was also not familiar with the SMACK internals, which ended up not being important for this exercise since no changes to SMACK were needed. However, as D was the sixth programming language the student added, he had ample experience adding support for new languages, which contributed to this exercise proceeding smoothly. Furthermore, D was an easy language to add since the LLVM IR it generates is close to the one generated by the C clang compiler, and hence heavily tested with SMACK. In addition, basic code in D does not heavily depend on its standard library and runtime. Hence, the student spent very little time modeling missing functions for D. For languages with extensive usage of standard libraries and runtime (e.g., Swift, Kotlin), we expect that modeling the runtime and standard library functionality to dominate the total time.

5.4 Case Study 3: Cross-Language Verification

One of the major advantages to the IR-based approach to verification is the ease of cross-language verification. In fact, with the approach that SMACK takes, every verification (of a non-C language) is a cross-language verification, as SMACK's models that have to be linked against the input program are written in C. With this in mind, non-trivial cross-language verification efforts are typically as simple as any regular single-language verification. As a proof of this concept, we took a simple algorithm, namely a classic triangle classifier, and implemented it in C, Rust, and Fortran. Our triangle classifier takes 3 integers as input, which represent the sides of a triangle, and it determines and returns the type of the triangle defined by the input sides. We wrote a harness program that invokes triangle classifiers from each language in turn, feeds equal nondeterministic inputs to all of them, and asserts that they return the same result. Hence,

Table 5. Equivalence checking of half-precision floating-point implementations in C and Rust. Column **Equal?** shows whether the two implementations are equivalent; column **Time** gives the verification runtime in seconds; column **LOC** gives the number of lines of code in the checked Rust function.

Function	Equal?	Time(s)	LOC
eq	✓	13	8
lt	✓	22	13
le	✓	18	13
gt	✓	17	13
ge	✓	18	13
to_f32	✓	8	41
to_f64	✓	8	41
from_f32	✗	5	68
from_f64	✗	4	70
is_nan	✓	4	3

we performed cross-language verification to verify the equivalence of our implementations in all three languages. SMACK was able to verify the equivalence (i.e., the harness program) in around 19 seconds. We expect such cross-language equivalence checking to be a valuable tool for developers when rewriting legacy applications in, for example Fortran or C, into more modern languages, such as C++ or Rust.

We further push our cross-language verification case study to a real-world Rust application — the *half* crate [44] that implements the half-precision floating-point type `f16`. We chose the *half* crate because its implementation is compact in terms of code size (functions range from only a few to around 70 LOC, see Table 5), but difficult to reason about because it frequently performs low-level bit manipulations. Furthermore, the equivalence of functions implementing the half-precision floating-point type can be easily expressed. This makes the *half* crate a suitable target for our cross-language verification case study.

For the purpose of this case study, we developed a simple C reference implementation of the half-precision floating-point type that leverages the available `__f16` type. Then, we verify that several important representative methods of the *half* crate, such as `lt`, `gt`, and `to_f32`, are equivalent to the respective C implementations. We leverage the Rust's *Foreign Function Interface* to write harness programs that assert the equivalence between Rust and C functions. (Note that if such a mechanism for interoperating between languages does not exist, we could implement the equivalence check at the LLVM IR level; however, working directly with the low-level LLVM IR would be more tedious.) Thanks to Rust's high interoperability with C, we are able to trivially express equivalence using the equality operator. For example, relational operators in C evaluate to 1 if the relation is true and otherwise they evaluate to 0. In Rust, casting a value of type `bool` into an integer has the same behavior. Therefore, comparing a predicate function such as `eq` in C and Rust reduces to checking if the return value of the C version is equal to the return value of the Rust version cast to type `u8`.

Table 5 summarizes the results of this case study. SMACK is able to verify that most of the chosen functions of the `f16` type are equivalent to their reference C implementations. The only exceptions are functions from_f32 and from_f64, for which SMACK discovered inconsistencies between the two implementations: conversions from larger bit-width floating-point types to `f16` are rounded differently. We reported this issue to the *half* crate developers, and they confirmed and fixed it. The verification runtimes range between 4–22 seconds on a 3.5GHz Intel 3770k machine.

6 Experience

In this section, we describe our experience of applying the procedure introduced in Section 4 to add support for new languages into SMACK as well as perform cross-language verification. First, we discuss why our approach and procedure allow us to trivially support many language constructs of the added programming languages (Section 6.1). Second, we describe key challenges that we encountered in the process of adding support for new languages and propose solutions for them (Section 6.2). Third, we present our experience with leveraging the cross-language verification capability to perform equivalence checking (Section 6.3).

6.1 Trivially Supported Features

SMACK is a mature C verifier that has been successfully applied on numerous C programs, including large-scale real-world C projects such as OpenSSH, SQLite, and Linux device drivers. Hence, SMACK already fully supports an extensive subset of LLVM IR that gets generated by the clang C compiler. For example, the key language constructs of LLVM IR such as functions, control flow, arithmetic, and derived types are completely modeled. As a result, SMACK readily supports new languages of which compilers emit LLVM IR code that is akin to what clang generates. We find that these languages are typically also procedural C-like languages, such as Fortran and D.

As it turns out, to our surprise, SMACK was often able to out-of-the-box support even language features that are not found in C. For example, without any modifications SMACK could handle the vectorized addition of arrays in Fortran, which we show in Figure 4. After inspecting the IR code generated by the Fortran compiler, we observe that the vectorized addition operation compiles into an element-wise array addition, which is a common IR operation and hence was already supported by SMACK.

Having an extensive subset of LLVM IR supported also saved us from modeling a lot of key program constructs in non-C-like languages such as Rust and Swift. For example, even though function calls and control flow constructs are different from those found in C (e.g., closures and match expressions), they are

```
program main
  use smack
  implicit none
  integer, dimension(2) :: A = (/ 2, 3 /)
  integer, dimension(2) :: B = (/ 3, 4 /)
  integer, dimension(2) :: S
  S = A + B
  call assert(S(1) == 5)
  call assert(S(2) == 7)
end program main
```

Fig. 4. Fortran program that utilizes vectorized addition.

compiled to the subset of LLVM IR that was already understood by SMACK. Therefore, our approach enables us to evade cumbersome modeling of advanced language features such as closures. In this regard, our experience demonstrates the advantages of our IR-based approach for multi-language verification.

6.2 Adding New Languages: Challenges and Solutions

As expected, supporting even a small subset of a new language in a verifier is often challenging. For example, a major challenge is the need to model previously unsupported LLVM IR constructs. Our experience shows that the compilers of non-C-like languages, such as Rust and Swift, indeed produce LLVM IR that was not supported by SMACK. Moreover, another important challenge is that we have to model a language runtime and its standard libraries to enable for practically usable verification. In the rest of this section, we describe in detail these challenges as well as our efforts to solve them.

Unsupported LLVM Constructs SMACK is a mature verifier that has been thoroughly tested on C programs, including thousands of SVCOMP benchmarks as well as large real-world applications such as OpenSSH. Despite SMACK's maturity, we found that compilers for the emerging languages, such as Rust and Swift, readily generate LLVM IR constructs we do not observe in LLVM IR generated from C code by clang. Hence, we had to extend SMACK with support for such constructs, and we describe some of these next.

Both the Rust and Swift compilers heavily rely on the use of LLVM structure types, often emitting different instructions involving structures than what clang would generate. We solved this problem by modeling LLVM IR structure types using uninterpreted functions that recursively constrain each field. For example, we represent value `{v,1}` of structure type `{T,i1}` 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. This encoding allows us to model two basic LLVM IR 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.

Another previously unsupported but frequently used LLVM IR construct is intrinsics. For example, both the Rust and Swift compilers default to using LLVM IR's overflow arithmetic intrinsics, such as `llvm.add.with.overflow.i32`. The `leading_zeros` methods of unsigned integer types in Rust are compiled to `llvm.ctlz.*` intrinsics. Such intrinsics can be easily modeled. For example, we model these intrinsics in SMACK by first performing the requested operation in the double-bit-width precision, to avoid potential overflows. Then, we inspect the result to detect if it overflowed, in which case we either report an overflow error or we block the overflowing path.

In addition to supporting more LLVM IR instructions, we also extended SMACK to support instruction sequences that are not regularly generated by clang. For example, the Rust compiler performs a *packing* optimization where structures with a size less than 8 bytes are packed into 8 byte integers (e.g., a load of a structure of type `{i32,i32}` gets encoded as a load of `i64`). This breaks the completeness of SMACK's memory model [42], which is not precise enough to capture such low-level operations, thereby leading to false alarms. We added an analysis pass to SMACK that detects load/store instruction patterns with pointer operands of integer element type that refer to structures. We translate such patterns to load from or store into structure fields (following the encoding described earlier), thereby avoiding packing.

Although we had to model these additional constructs, our approach still demonstrates the advantages discussed in Section 6.1: modeling one LLVM IR construct benefits the support of multiple languages, and this process becomes progressively easier as adding a new language benefits from previous modeling efforts.

Languages with Large Runtimes Getting a verifier to translate LLVM IR generated from a language with a large runtime is not any more difficult than for languages with smaller runtimes. However, performing a nontrivial verification task for such a language is much harder, because even rudimentary language features are sometimes under the hood implemented using complex runtime constructs and standard libraries. Moreover, the source code implementing such features is not readily accessible to the verifier as IR code linked with the program source. We found this to be the most challenging problem when adding a new language to a verifier. Note that this problem persists even if the verification is done directly on program source (as opposed to IR) since the source code of the underlying runtime is typically not available, written in a different programming language, or too large to be efficiently handled by a verifier.

As an example of such a language feature, consider the for-in loop over an iterable structure. All of the languages with substantial runtime we considered provide such a feature. In fact, in Swift, Kotlin, and Rust, the C-style for loop is not even supported, and range structures are used to emulate the same behavior. Consider this simple example in Swift: `for i in 0..<10 {x += 5}`. The compiler translates the code `0..<10` using a `Range<Int>` structure/class, whose member methods are then called in the compiled loop code. The code of such member methods is not readily available to the verifier, but is a part of the runtime. On the other hand, both Kotlin and Rust compile such loops into basic LLVM IR instructions that do not contain method calls into runtime, despite the high-level concept of a range being similar to Swift. Many features of large-runtime languages are implemented like this, and they vary wildly between languages. Examples of other basic language features the heavily depend on runtime include method dispatch (Swift, Objective-C), arrays (Swift, Objective-C, Rust, Kotlin), and object instantiation (Kotlin). As a more extreme example, even basic arithmetic in Kotlin is abstracted into invoking methods belonging to its runtime, instead of generating the appropriate LLVM IR instructions directly. We relied on two solutions to overcome such problems, with different trade-offs, as we describe next.

We compile and link an existing implementation of the runtime/standard library with the input program. For example, to support basic integer operations in Kotlin, we used the existing implementation of these operations from the Kotlin runtime and linked it with the input program. The main advantage of this approach is that it requires no manual effort. It also avoids the user potentially introducing errors while modeling the runtime. The main drawback is that the standard libraries and runtime are generally very large, and this may cause verification to blow up even on small input programs. For example, the implementation of the array structure in Swift is thousands of lines of code. Such code is also heavily optimized, and often relies on low-level bit vector operations and compiler builtins, which further complicate its verification.

We model the standard libraries and runtime by writing stubs for the relevant methods. Table 6 gives the sizes of the models we developed for each language we support. SMACK already came with extensive models for the C standard library and a part of the Rust standard library, which is why these two models are by far the largest. The main advantage of this approach is that the manually written models make the verification much more tractable, and hence most verifiers, no matter whether they are IR-based or not, require it to achieve scalable verification. The main drawback is that writing them is a tedious manual effort that requires detailed understanding of the language specifications. Hence, we did not do

Table 6. Sizes of models we developed for each language. Column **Model LOC** gives the size of each model in terms of lines of code.

Language	**Model LOC**
C	2566
C++	13
Fortran	38
D	0
Rust	480
Objective-C	0
Swift	2
Kotlin	17

that for other languages. In principle, the standard libraries and runtime of Kotlin, Objective-C, and Swift could each be modeled in a similar way to Rust. Note that this solution is contradictory to the general principle of our approach since it requires per-language modeling.

6.3 Cross-Language Verification

Although our experience with cross-language verification is limited to equivalence checking of programs written in different languages, it captures an important pattern in cross-language development: a program written in one language uses external libraries written in another language. Furthermore, equivalence checking is a useful application of cross-language verification, giving confidence to developers that a new, native implementation of a library retains the behaviors of the previous non-native implementation. This is especially true when large rewriting efforts are under way, such as replacing legacy libraries implemented using Fortran with C/C++ implementations in the context of high-performance computing, or libraries implemented using C with their Rust counterparts.

We find that once the languages involved in the cross-language verification process are well-supported and there are available mechanisms for these languages to interoperate, cross-language verification is feasible, highly automated, and comes almost for free. This is expected since our approach casts the problem of cross-language verification into the problem of verifying a single language, namely LLVM IR. Therefore, the main impediments we encountered while verifying cross-language programs were related to SMACK's incomplete support for LLVM IR, similarly to our efforts to add support for new languages. For example, while performing the case study, the only issue we encountered was that SMACK did not model the LLVM count-leading-zeros intrinsics. We quickly added support for this instruction and were able to complete the verification process smoothly.

7 Conclusions

In this paper, we proposed a procedure for extending an IR-based verifier with multi- and cross-language verification capabilities. By relying on the proposed procedure, we extended the LLVM-IR-based SMACK software verification toolchain with basic prototypical support for 6 additional languages. We performed several case studies to assess the quality of our extensions and the feasibility of leveraging the IR-based verifier architecture in the context of multi- and cross-language verification. Our evaluation is encouraging and indicates that the IR-based architecture indeed lowers the bar for adding support for a new language into an existing verifier — languages with small runtimes could be reliably added with only a modest effort. It also allows for straightforward cross-language verification. As we anticipated, supporting languages with large runtimes that heavily rely on standard libraries is possible, but mature support would require a large manual effort to model the runtime and libraries.

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: A framework for abstraction- and interpolation-based software verification. In: International Conference on Computer Aided Verification (CAV). pp. 672–678 (2012). https://doi.org/10.1007/978-3-642-31424-7_48
2. Arlt, S., Rubio-González, C., Rümmer, P., Schäf, M., Shankar, N.: The gradual verifier. In: NASA Formal Methods Symposium (NFM). pp. 313–327 (2014). https://doi.org/10.1007/978-3-319-06200-6_27
3. Arlt, S., Rümmer, P., Schäf, M.: Joogie: From Java through Jimple to Boogie. In: ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis (SOAP). pp. 3–8 (2013). https://doi.org/10.1145/2487568.2487570
4. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. Proc. ACM Program. Lang. **3**(OOPSLA), 147:1–147:30 (2019). https://doi.org/10.1145/3360573
5. Babić, D., Hu, A.J.: Calysto: Scalable and precise extended static checking. In: International Conference on Software Engineering (ICSE). pp. 211–220 (2008). https://doi.org/10.1145/1368088.1368118
6. Baranová, Z., Barnat, J., Kejstová, K., Kucera, T., Lauko, H., Mrázek, J., Rockai, P., Still, V.: Model checking of C and C++ with DIVINE 4. In: International Symposium on Automated Technology for Verification and Analysis (ATVA). pp. 201–207 (2017). https://doi.org/10.1007/978-3-319-68167-2_14
7. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: International Conference on Computer Aided Verification (CAV). pp. 184–190 (2011). https://doi.org/10.1007/978-3-642-22110-1_16
8. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX Conference on Operating Systems Design and Implementation (OSDI). pp. 209–224 (2008)
9. Carter, M., He, S., Whitaker, J., Rakamarić, Z., Emmi, M.: SMACK software verification toolchain. In: International Conference on Software Engineering (ICSE). pp. 589–592 (2016). https://doi.org/10.1145/2889160.2889163
10. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 19–33 (2007). https://doi.org/10.1007/978-3-540-71209-1_4
11. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 570–574 (2005). https://doi.org/10.1007/978-3-540-31980-1_40
12. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 168–176 (2004). https://doi.org/10.1007/978-3-540-24730-2_15
13. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: International Conference on Theorem Proving in Higher Order Logics (TPHOLs). pp. 23–42 (2009). https://doi.org/10.1007/978-3-642-03359-9_2
14. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 137–148 (2009). https://doi.org/10.1109/TSE.2011.59

15. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. In: International Conference on Software Engineering and Formal Methods (SEFM). pp. 233–247 (2012). https://doi.org/10.1007/s00165-014-0326-7
16. The D programming language. `https://dlang.org/`
17. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 337–340 (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005). https://doi.org/10.1.1.212.7449
19. Stack overflow developer survey. `https://insights.stackoverflow.com/survey/2018` (2018)
20. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE). pp. 56–72 (2016). https://doi.org/10.1007/978-3-319-48869-1_5
21. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: International Conference on Computer Aided Verification (CAV). pp. 372–378 (2011). https://doi.org/10.1007/978-3-642-22110-1_29
22. Azul Falcon. `https://www.azul.com/called-new-jit-compiler-falcon/`
23. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In: International Conference on Formal Engineering Methods (ICFEM). pp. 15–29 (2004). https://doi.org/10.1007/978-3-540-30482-1_10
24. The Flang Fortran compiler. `https://github.com/flang-compiler/flang`
25. The Go programming language. `https://golang.org/`
26. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: International Conference on Computer Aided Verification (CAV). pp. 343–361 (2015). https://doi.org/10.1007/978-3-319-21690-4_20
27. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. In: International Conference on Computer Aided Verification (CAV). pp. 424–440 (2011). https://doi.org/10.1007/978-3-642-22110-1_34
28. Hahn, F.: Rust2Viper: Building a Static Verifier for Rust. Master's thesis, ETH (2016)
29. The Haskell programming language. `https://www.haskell.org/`
30. Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., Podelski, A.: Ultimate Automizer with SMTInterpol. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 641–643 (2013). https://doi.org/10.1007/978-3-642-36742-7_53
31. Jeff Bezanson, Alan Edelman, S.K., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review **59**, 65–98 (2017). https://doi.org/10.1137/141000671
32. Kotlin/Native for native. `https://kotlinlang.org/docs/reference/native-overview.html`
33. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: International Conference on Computer Aided Verification (CAV). pp. 427–443 (2012). https://doi.org/10.1007/978-3-642-31424-7_32

34. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (CGO). pp. 75–86 (2004)
35. The LLVM compiler infrastructure. http://llvm.org
36. LLVM language reference manual. https://llvm.org/docs/LangRef.html
37. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: International Conference on Verified Software: Theories, Tools, Experiments (VSTTE). pp. 146–161 (2012). https://doi.org/10.1007/978-3-642-27705-4_12
38. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 41–62 (2016). https://doi.org/10.1007/978-3-662-49122-5_2
39. Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 15–26 (2008). https://doi.org/10.1145/1390630.1390635
40. Pyston. https://blog.pyston.org/about/
41. Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementations. In: International Conference on Computer Aided Verification (CAV). pp. 106–113 (2014). https://doi.org/10.1007/978-3-319-08867-9_7
42. Rakamarić, Z., Hu, A.J.: A scalable memory model for low-level code. In: International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI). pp. 290–304 (2009). https://doi.org/10.1007/978-3-540-93900-9_24
43. Ruby-LLVM. https://github.com/ruby-llvm/ruby-llvm
44. half: f16 type for Rust. https://github.com/starkat99/half-rs
45. The Rust programming language. https://www.rust-lang.org
46. Scala Native. http://www.scala-native.org/en/v0.3.8/
47. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: CIVL: The concurrency intermediate verification language. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC). pp. 61:1–61:12 (2015). https://doi.org/10.1145/2807591.2807635
48. SMACK software verifier and verification toolchain. http://smackers.github.io
49. The Swift programming language. https://swift.org/
50. Toman, J., Pernsteiner, S., Torlak, E.: CRUST: A bounded verifier for Rust. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 75–80 (2015). https://doi.org/10.1109/ASE.2015.77
51. Wang, W., Barrett, C., Wies, T.: Cascade 2.0. In: International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). pp. 142–160 (2014). https://doi.org/10.1007/978-3-642-54013-4_9
52. Woodward, M.: Announcing LLILC — a new LLVM-based compiler for .NET. https://www.dotnetfoundation.org/blog/2015/04/14/announcing-llilc-llvm-for-dotnet (2015)