Noname manuscript No.
(will be inserted by the editor)

Specifications and Extended Static Checking for LLVM

Joshua Robbins · David R. Cok

Received: date / Accepted: date

Abstract Current behavioral specification languages are integrated with their corresponding source programming language. Hence, each needs its own front-end and translation to an intermediate language, and from there to logical encodings. Using LLVM as a compiler intermediate language allows an architecture that unifies and saves effort: compile source to LLVM with a standard compiler; compile source specifications to an intermediate LLVM specification language; and then use a common LLVM + LLVM specifications tool to produce and check logical encodings. This paper describes an LLVM specification language and a tool to produce checkable logical encodings in WhyML format from LLVM.

Keywords Specification languages,LLVM,software verification,BISL

1 Introduction

One method of checking that the software implementation of a program is consistent with formally stated specifications translates both the software and the specifications into a common logical framework and then uses automated reasoning tools to check their consistency. Using automated decision procedures reduces the effort and knowledge required of the users. Checking such specifications is not a panacea—the formally written specifications may be incorrect or incomplete, specifications still need to be written, and tools may not be able to confirm the consistency of highly complex code or specifications. Nevertheless such consistency checking adds to the confidence that the implementation matches the user intent and satisfies stated correctness properties.

Several systems for this purpose share a common design:

- a BISL[27] (*Behavioral Interface Specification Language*) appropriate to the specific source programming language is used to express the formal specifications;
- a parsing and translation tool converts the source program and its specifications into a logical language such as Why3[14] or SMT-LIB[9];
- an automated SMT[9] solver or semi-automated tool such as Coq[3] or PVS[32] checks the validity of the *verification conditions* (VCs) generated by the translation tool;
- when all VCs are validated, the software and its specifications can be considered consistent, that is, either both correct or both wrong in the same way.

Specification languages exhibit a variety of styles. Some, such as Z [34], are intended to be programming-language independent and mirror the underlying mathematical structures. However, the specification languages most often used at present for consistency checking are designed to match the syntax and semantics of an associated programming language. This close tie makes it easier for a software engineer writing in a specific pro-

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1314674 and an accompanying NSF REU award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Joshua Robbins
GrammaTech, Inc., Ithaca NY USA
E-mail: iconmaster@csh.rit.edu

David R. Cok
GrammaTech, Inc., Ithaca NY USA
E-mail: dcok@grammatech.com

gramming language to understand, write, and review assertions in the associated specification language. Several examples of such pairings are listed in Table 1. In some cases the specification mechanisms are built into the language (e.g., Eiffel) or are very closely tied to the associated tools.

A drawback of this design is that each combination in Table 1 requires its own front-end language processor. Ideally a specification-supporting tool will extend an existing parser for the host programming language. Nevertheless, there is significant effort required after parsing and type checking to normalize the control flow, simplify statements, and convert code and specifications into an equivalent logical form. One attempt to unify some of this effort is the Boogie [6] language. Boogie provides an intermediate form that can be targeted by a front-end language processor. However, the work required to create a transformer from arbitrary source language to Boogie is still considerable.

We report here an alternative approach. The compiler community has in hand a now widely used intermediate representation for source programs: LLVM [29]. Many compilers target LLVM bitcode as an intermediate form, generated by the compiler front-end, and leaving it to LLVM tools to produce platform-specific executables from the bitcode or to interpret the bitcode directly. The design we describe here then has these components:

- we use existing compilers to produce LLVM bitcode from source programs;
- we write a specification language translation tool for the BISL associated with a given source language, but this translator need only produce expressions in the form of LLVM metadata;
- our LLVM-to-Why3 translator represents an LLVM program and the LLVM metadata holding the program's specifications in logical form, including assertions that check for runtime errors associated with misuse of the programming language (divide by zero, null pointer dereference, array index out of bounds, etc.);
- we use the existing Why3 tools and associated logical proof engines, such as Coq [3], alt-ergo [13], Z3 [22], CVC4 [8], and Yices2 [25], to discharge the generated verification conditions.

This architecture is also illustrated in Table 2.

We can use an existing compiler to transform source programming text to LLVM. The translator that converts specifications in a source specification language to LLVM metadata must be custom written for each BISL. Ideally, such a transformation would be part of a compiler, so that it could make use of all of the parsing, name resolution, type checking, AST infrastructure, and optimizations that are part of a typical compiler. It is not part of this project to create these source specification language transformations.

The contributions of this paper are the design articulated above, a description of our prototype tool implementing that design, and discussions of the problems solved and the challenges remaining in using this design for the project of automated checking of software specifications.

2 LLVM

From its initial goals as a virtual machine, the LLVM tool suite [29] has evolved into a collection of capabilities that includes a large number of compiler front-ends, various platform-specific back-ends, and tools for program manipulation and static checking. The aspect important for our purposes is LLVM's bitcode representation of software. This is now an intermediate representation used by many compilers and program analysis and transformation tools.

The bitcode for a program is modular: each subroutine is a separate unit. Furthermore the software is already normalized: the bitcode representation is already in single-assignment form, basic blocks and transitions among them are all identified, and each bitcode statement is a single operation. As expected, intermediate representations do lose information. For example, to identify the source code location of some improperly used operation, such as a divide-by-zero, one must translate back to source code using debug information. This point is discussed further in §8.

As an example of simple LLVM, the C code in Listing 1 is translated by the command

```
clang -O2 -emit-llvm -S -c add.c
```

(in the author's specific environment) to the LLVM as shown (excluding various metadata information). The LLVM toolset includes tools that convert (losslessly) between the human-readable .ll and the binary .bc formats; our tool can ingest either form.

Listing 1 Simple C program and its LLVM (.ll format)

```
int add(int a, int b) {
  return a+b;
}

define i32 @add(i32 %a, i32 %b) #0 {
  %1 = add nsw i32 %b, %a
  ret i32 %1
}
```

The flow and computation of the program is easy to read and it is easy to imagine translating the LLVM

Table 1 Examples of programming and specification languages

Programming Language	Specification Language	Tools
Java	Java Modeling Language (JML) [15]	OpenJML [20], Key [11], ESC/Java2 [21]
C	ACSL [10]	Frama-C [4]
C	VCC [17]	VCC [17]
SPARK Ada	SPARK [5]	AdaCore tool set [2]
C#, .NET	CodeContracts [26]	CodeContracts [26]
Scala	Leon [12]	Leon [12]
Eiffel	Eiffel [31]	Eiffel [31]
Dafny	Dafny [30]	Dafny [30]
C#	Spec# [7]	Spec# [7]
C and Java	Verifast's language	VeriFast

Table 2 Tool chain architecture. This paper presents the LLVM specification language and a prototype tool to convert LLVM to logical representation

Input	Source programming language	Source specification language
Tool	Standard compilers	compiler augmented to parse specification language text
Intermediate representation	LLVM	**LLVM specifications** in LLVM metadata
Tool	**LLVM to Why3/SMT-LIB tool**	
Logical form	Why3 or SMT-LIB	
Proof tools	Off-the-shelf SMT proof tools	

into a logical representation. However, this simplicity is quickly lost when inspecting the output from other languages and even from clang without optimization. The command `clang -emit-llvm -S -c add.c` produces the output in Listing 2.

Here the LLVM allocates 4-byte locations on the stack solely for the purpose of saving the values of the formal parameters. In this program they are not used, which is why the optimizer can elide them. However, when translating this bit of LLVM, the translation tool must now be concerned with models of memory, potential aliasing and the like.

Listing 2 Unoptimized LLVM for simple C program

```
define i32 @add(i32 %a, i32 %b) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = store i32 %a, i32* %1, align 4
  %4 = store i32 %b, i32* %2, align 4
  %5 = add nsw i32 %3, %4
  ret i32 \%5
}
```

As an additional example, the (optimized) LLVM assembly instructions corresponding to the C program in Listing 3 are shown in Listing 4. This program consists of four basic blocks, optimized to three blocks clearly identified in the LLVM assembler. Each basic block has an entry point with a label (except the entry block) and ends with some kind of terminating statement; in this example the terminating statements are either a conditional branch to one of two labels or a return statement. The optimizer moved the final return statement, which would be in a fourth basic block, to be a return in each of the branches that lead to it.

Listing 3 Program with branches

```
int branches(int x) {
  int r = 1;
  if (x == 0) {
    r = 0;
  } else if (x < 0) {
    r = -1;
  }
  return r;
}
```

Listing 4 LLVM for program with branches

```
define i32 @branches(i32 %x) #0 {
  %1 = icmp eq i32 %x, 0
  br i1 %1, label %5, label %2
; <label>:2
  %3 = ashr i32 %x, 31
  %4 = or i32 %3, 1
  ret i32 %4
; <label>:5
  ret i32 0
}
```

A complexity arises for programs in single-assignment form when basic blocks join together, such as at the end of an if or switch statement. It is typically the case that a variable will have different assignments in the branches that are joining and will therefore have different single-assignment names. A new name is required for the variable in the block following the join; that name must be initialized depending on the source block through which the flow of execution feeds into the join.

LLVM has a special operation, the phi operation, that is a special kind of if-then-else expression: it selects a value based on the block from which control flow came. An example in LLVM is shown in Listing 5: the phi operation appears at the beginning of the block labeled `:10:`, where it joins values from three different sources (one of which has an uninitalized value).

Listing 5 Program with phi expression

```
define i32 @mm(i32) local_unnamed_addr #0 {
  %2 = icmp sgt i32 %0, 0
  br i1 %2, label %3, label %5

; <label>:3:
  %4 = add nsw i32 %0, 1
  br label %10

; <label>:5:
  %6 = icmp slt i32 %0, 0
  br i1 %6, label %7, label %10

; <label>:7:
  %8 = mul nsw i32 %0, %0
  %9 = add nuw nsw i32 %8, 100
  br label %10

; <label>:10:
  %11 = phi i32 [ %4, %3 ],
                [ %9, %7 ], [ undef, %5 ]
  %12 = add nsw i32 %11, 100
  ret i32 %12
}
```

3 Why3

Why3 [14], a rewrite of the earlier Why platform, is intended as an intermediate language between software verification systems and the proof tools that are needed to check validity. By using the WhyML programming language, the platform can be used directly to represent algorithms and check their properties. Why3 is used by front-ends for a number of programming languages; it uses a number back-end proof tools, both interactive and automatic. The Why3 documentation at `http://why3.lri.fr` provides a current list.

We adopted Why3 for this project specifically because it supports conversions of its logical representation to a number of different proof tools, enabling us to experiment easily with those alternatives. Given our goal of as much automation as possible, the competing alternative is direct translation to verification conditions in standard SMT-LIB format, which would still enable experimentation with a number of SMT solvers. The jSMTLIB library [19] enables easy integration with a variety of SMT tools, correcting for any deficiencies in an individual tool's support for standard SMT-LIB.

The key reason why a direct translation might be worthwhile is discussed in §8, namely, the challenges in translating the results of SMT proof tools back to the source code context.

4 Specifications for LLVM

A source code program is converted to an LLVM intermediary, and then to machine code for the target architecture. Similarly, the source language specifications, in the relevant source specification language, are converted to an intermediate LLVM specification language and then to a target logical language. In this design, multiple front-ends can target the one LLVM specification language; all these front-ends can then share one tool that converts LLVM specifications into a target logical language.

Thus an important design issue is how to represent specifications in LLVM tools. Fortunately, LLVM has a metadata facility that permits attaching a metadata node to other nodes in an LLVM program. Thus we can attach function specifications, such as pre- and postconditions, to the function definition. Assert or other specification statements in the body of a program can be attached to the immediately following node, which would be a LLVM statement. The specifications are metadata in a format recognizable to our tool. Assert statements that appear at the end of a block in a C program still precede the LLVM statement that terminates the block.

In human readable assembler, this metadata appears as shown in Listing 6. We implemented two styles of metadata. The native form of metadata is S-expressions, that is, expressions are represented as abstract syntax trees. An example is shown in Listing 6. For this absolute-value procedure, the `requires` clause states that the argument (`%n`) is not equal to the minimum 32-bit negative integer and the `ensures` clause states that the returned result is greater than or equal to (signed comparison) zero. Since most of the time, the LLVM files are not read by humans, readability is not an issue. Rather, since the front-end represents expressions as ASTs and the back-end needs them as parsed ASTs, the intermediate form might as well be ASTs. In either case, the variable names used in the expressions are those used elsewhere in the LLVM representation and not necessarily the names used in source code.

We also implemented an alternative syntax, in which expressions in specifications appear much like the expressions would be written in C, using infix operations, parentheses and operator precedence to allow expressions to be written in familiar, source language form. We included a a small expression parser in the llvm2why tool to convert these expressions to AST form. This

Listing 6 LLVM assembler with metadata

```
define i32 @abs(i32 %n) !whyr.requires !{!{!"neq", !{!"arg", !"n"}, i32 -2147483648}}
                                           !whyr.ensures !{!{!"sge", !{!"result"},  i32 0}}  {
    %is_neg = icmp slt i32 %n, 0
    %neg = sub nsw i32 0, %n
    %val = select i1 %is_neg, i32 %neg, i32 %n
    ret i32 %val
}
```

Listing 7 LLVM assembler with metadata as expressions

```
define i32 @abs(i32 %n)  !whyr.requires !{!{!"war", !"%n␣!=␣(i32)-2147483648"}}
                                     !whyr.ensures !{!{!"war", !"result␣sge␣(i32)0"}}  {
    %is_neg = icmp slt i32 %n, 0
    %neg = sub nsw i32 0, %n
    %val = select i1 %is_neg, i32 %neg, i32 %n
    ret i32 %val
}
```

form of metadata is shown in Listing 7. Note that in both cases, LLVM expects the metadata to be on the same line as the node to which it is attached, without line breaks; this results in long lines, which are wrapped here for formatting and readability. Also, however, metadata declarations can be introduced so that lengthy content can be placed elsewhere in the LLVM file and a simple reference placed in the body of the function definition.

An alternative is for LLVM specifications to be written in a file associated with but not part of the LLVM bit-code files. The advantage is that the syntax can be more human readable and the specifications can be separately maintained. The disadvantages are maintaining the association of the various files and that it is difficult to reliably connect statement specifications such as assert statements and loop specifications that are in a separate file with their correct places in the bit-code file. We have implemented the metadata mechanism and are experimenting with this alternative.

5 The LLVM-SL specification language

Our design is to represent specifications and in-line assertions using LLVM metadata attached to LLVM bitcode. However, the source→LLVM conversion and the LLVM→Why3 conversions share a common LLVM specification language as an intermediary. That language is described in this section.

5.1 Basic LLVM-SL

The various BISLs described in §1 and Table 1 are quite similar, with characteristics that differ according to differences in their host programming languages. When compared to existing BISLs, LLVM-SL is most similar to ACSL, because both C and LLVM are languages that are close to machine level, with little information hiding, implicit run-time checking, or implicit object-oriented delegation. The equivalent of class or data structure invariants are expressed in LLVM-SL (as in ACSL) as global or type invariants. We are still experimenting with the best representation of these in LLVM. They are most easily expressed in text separate from the .ll or .bc files; if expressed as metadata we need to attach them to a dummy declaration.

The LLVM specification language intends to describe the functional behavior of a LLVM procedure in sufficient detail that the specification of behavior (what is computed) can be checked for consistency with the implementation (how it is computed). It may be that in some circumstances, analyses may wish to inline a procedure at its call-site to avoid requiring it to have explicit specifications. This design enables consistency checks to be performed modularly, even though, for soundness, it must be established that all procedures satisfy their respective specifications and all procedures terminate.

At this point in implementation, llvm2why has implemented the following:

– procedure preconditions, as *requires* clauses
– procedure postconditions, as *ensures* clauses
– procedure frame conditions, as *assigns* clauses
– statement assertions, as *assert* clauses
– statement assumptions, as *assume* clauses
– loop invariants and variants are in process

All these should be familiar to a user of ACSL, or of any of the BISLs described previously. We expect that future work will include some of the complications that are necessary in ACSL, such as the ability to describe memory regions, to express separation between memory regions, and more features to describe fields and array elements than are currently implemented.

Most of the clauses listed above take expressions as arguments. LLVM-SL's expression language is a straightforward logical encoding of LLVM's operations. They include

– literals: true, false, integers, reals, null, array constant
– identifiers (variables)
– unary operations: arithmetic negation, boolean negation, bit complement
– binary operations: add, subtract, multiply, divide, modulo, remainder, shift left, arithmetic and logical shift right, equality, inequality, arithmetic comparisons, boolean and, boolean or, logical implication, logical equivalence
– if-then-else (ternary) operation
– type casts
– quantified expressions: forall, exists, let
– pointer dereference, address of, pointer offset
– array element, functional array update
– vector, set and struct operations

Another measure of completeness is whether all LLVM instructions are implemented. Our tool does implement most instructions, with those related to exception handling, multi-threaded execution, variable argument lists, and indirect function calls remaining for future work.

One other unusual feature in LLVM is that there are vector operations. These do not affect the type system, but do need translation into a logical representation.

5.2 LLVM types

Type handling in LLVM-SL is a unique challenge. Most programming languages and corresponding specification languages treat values as having types, such as boolean, real, integer, short integer, and the like. In particular, in C and C++, the type system distinguishes integral values of different bit-widths and distinguishes signed and unsigned integral values. Operations on these values, such as equality or less than, apply appropriate conversions for both bit-width and signedness. LLVM in contrast, does have integral values of different bit-widths, but does not include signedness in the data type. Instead, there are different signed and unsigned operations. Thus there is both a signed-less-than and an unsigned-less-than operation; the arguments are n-bit bit-vectors and the operators interpret the bit-vectors as signed or unsigned integers respectively. Similarly there is a signed and unsigned version of integer divide. Operations such as add and subtract yield the same bit patterns whether their arguments are interpreted as signed or unsigned, so one operation will suffice. However, if the operation overflows, the result may be undefined, and the undefinedness differs depending on whether the operands are interpreted as signed or unsigned. Thus some operations, such as add, have a qualifier, `nsw` or `nuw`, which causes the operation to produce a *poison value* if signed or unsigned overflow, respectively occurs.

In contrast, logical representations sometimes prefer to work with mathematical integers, rather than bit-limited values. Some previous work [16] determined that persons writing specifications preferred to think in mathematical types (or simply did so without thinking about it). Consequently, the LLVM-SL type system includes both mathematical and bit-vector integers, but signedness of operations on bit-vectors is reflected in the operators rather than in the data types.

6 Why3 and WhyML

WhyML [14] is an intermediate language between software verification systems and the proof tools; it is part of the Why3 set of tools. Why3 is used by front-ends for several programming languages and proof systems, and integrates a number back-end proof tools, both interactive and automatic.

We adopted Why3 for this project specifically because it supports conversions of its logical representation to a number of different proof tools, enabling us to experiment easily with those alternatives. Given our goal of as much automation as possible, the competing alternative is direct translation to verification conditions in standard SMT-LIB format, which would still enable experimentation with a number of SMT solvers. The jSMTLIB library [19] enables easy integration with a variety of SMT tools, correcting for any deficiencies in individual tool's support for standard SMT-LIB.

There are three reasons why a direct translation that bypasses Why3 might be desirable:
(1) using Why3 makes it difficult to translate failed assertions back to source locations;
(2) counterexample information is essential to understanding failed proof attempts, but Why3 does not retrieve such information from the delegate solvers;
(3) Why3's encoding causes solvers to frequently timeout in cases where there are invalid assertions. We suspect this is because of overuse of quantified expressions to state model axioms and is still a matter of experimentation.

Accordingly, in future work we plan to evaluate a direct-to-SMTLIB alternative and report on the comparison.

7 Converting LLVM-SL to Why3: the llvm2why tool

7.1 General approach

We chose the Why language as the target logical encoding for llvm2why because it is an intermediate language that can delegate its verification conditions to a variety of automated decision procedure-based SMT tools, such as alt-ergo, Z3, and CVC4, or to interactive proof systems, such as Coq. Thus, llvm2why translates an LLVM program with its specifications into a collection of Why3 theories, such that all the proof obligations implicit in the LLVM+specifications are contained as proof obligations in the Why3 theories. The Why3 output is then submitted in a separate step to the Why3 tool to check all the proof obligations. The translation includes both implicit (e.g., undefined operations) and explicit requirements (those in the given specification) in the Why3 output.

There are various design decisions to be made about the translation, depending on the degree of precision desired. Each simplification reduces the soundness of the resulting static check, but may make the proof steps more tractable or quicker.

– Various of the implicit requirements can be omitted, if the user is not concerned about their presence or is otherwise convinced of the absence of undefined operations.
– Numerical values can be modeled as bit-precise bit vectors or as mathematical integers.
– Memory can be modeled in a variety of ways.

These points and other design challenges are discussed in the following subsections.

7.2 Design challenge: machine arithmetic and bit widths

An important design choice in mapping software to Why3 or SMT-LIB is whether integral values are modeled as bit-vectors or as mathematical integers. Using bit-vectors makes modeling bit-wise and bit-shift operations straightforward. Bit-vector theories include operations to perform arithmetic on 2's-complement bit-vectors, but these can be expensive for solvers to reason about. On the other hand, using mathematical integers to model integral values in the program makes arithmetic easy, but bit-operations hard. Converting between integers and bit-vectors is also expensive for SMT solvers.

When using mathematical integers, one can also choose whether to enforce overflow and underflow limits or to ignore them. Again, ignoring overflow and underflow reduces the soundness of the static checking but simplifies the problem statements and increases performance.

In llvm2why we have implemented both a bit-vector representation and a mathematical integer representation; checking of overflow and underflow can be enabled or disabled.

7.3 Design challenge: segmented memory

The memory in a typical processor is a flat array, indexed by integer values from start to finish. Programs obtain portions of memory dynamically by asking the operating system for a block of memory. Well-behaved programs do not rely on the absolute position of a block of memory, do not access outside of the block, and do not access one block from another. A program analysis that wishes to accurately model the effects of non-well-behaved programs, such as whether a frame-pointer-smashing attack is possible, needs to model the absolute positions of stack and dynamic memory. However, for most programs it is sufficient to simply warn about memory-unsafeness, without modeling its effects: the programs should be checked that all memory references are indeed safe, but otherwise can expect that a program does not depend on unsafe behavior.

Making this assumption permits memory to be modeled in a segmented fashion. Each staticly or dynamically allocated block of memory can be treated independently, with its size known at allocation time. There is no assumption about what is 'next to' a block of memory. It is much easier to establish separation—that modifications in one block of memory do not affect another block—in a segmented memory model.

Segmented memory models do not easily model the memory relationships in C structs. Structs are often copied or initialized to zero using byte operations that copy or zero the structure independent of the boundaries between structure fields. Nor do segmented models handle address offset computations among structure elements. For operations like these a hierarchical segmented model is needed, where structures are sometimes handled as one block of memory, despite the internal type boundaries and sometimes handled as a collection of individual blocks.

This first implementation of llvm2why uses segmented memory and does not handle all struct operations easily. It is a point of future work to investigate the conceptual and performance trade-offs among the varieties of models discussed in this section.

7.4 Design challenge: memory granularity

Typical processors index memory by the 8-bit byte. Blocks of memory may be allocated with other granularity: 32-bit integers or n-bit structs for example. Also typically, a block of memory is read and written in a uniform fashion, always with the same data type in mind. However, it is possible, for instance, to treat an array of 32-bit values as an array of 8-bit values; if one does, issues of which bytes are most and least significant or how a floating-point number is encoded may become relevant.

Reasoning about memory is far simpler if it is assumed that each block of memory has its own uniform type. Again, it is a research topic to carefully measure the conceptual and performance tradeoffs of different choices. The current implementation of llvm2why presumes that each block has its own uniform type.

7.5 Handling loops and exceptional control flow

The extended static analysis methodology pioneered by ESC/Modula and ESC/Java has some difficulty with loops and back-edges: control-flow back-edges must be cut and the corresponding join points must have inductive invariants. In languages such as Java that enforce structured control flow, only loops have particular difficulty, but C and LLVM can have arbitrarily tangled control-flow.

The framework used here for translating programs into logical equivalents follows the methodology of the tools and BISLs described in §1. In particular, the program must be translated into a loop-free over-approximation with loop invariants specified at the locations where edges are cut. We will not describe this process here except to make two points. First, simply, that loop invariants are required, whether manually constructed or inferred. Second, in most source programming languages (e.g., Java), control structures are limited such that it is obvious where loop invariants are to be placed and where back edges should be cut. This is not as obvious in C (with arbitrary goto edges) and in LLVM. In fact an LLVM program is simply a set of basic blocks with edges defined, with no other imposed sense of helpful order that might have come from the syntactic structure of the original source program. However, where the LLVM has been generated from a well-structured control flow, simple algorithms can identify the heads of loops where loop invariants are needed.

The following conditions are sufficient to be able to reason about a program without further induction, given a directed graph of basic blocks:

- chose a set of edges to cut that will render the directed graph acyclic; each edge e has a block at its head H_e and its tail T_e; if the edge is indeed cutting a cycle, there is a path from T_e to H_e.
- state (possibly infer) an invariant I_e for each edge being cut
- the invariant must be proved at the head of the cut edge
- define a set S_e of program locations that is a superset of the program locations assigned to at some point along the path from T_e to H_e
- at the beginning of basic block T_e, the following are inserted:
  - an assertion (that must be proved) that I_e holds
  - havocing (assigning arbitrary values to) the program locations in S_e
  - an assumption that I_e holds
- at the end of basic block H_e, the assertion I_e must be proved

It is generally that case that any program location in S_e must be mentioned in I_e, or nothing will be able to be proved about that variable.

The challenge for tools is that, while pre- and post-condition method specifications are conveniently associated with procedure declarations, loop specifications must be associated with locations within the LLVM, after translation.

8 Interpreting the results of proof attempts

Early static checkers, such as ESC/Java [23], encoded source program variable names and locations in the logical variable names used in the logical representation. If the tool failed to prove a given assertion, it generally was able to provide a counterexample in the form of assignments to each of the logical values that satisfy all assumptions but do not satisfy some assertion. Even with this name mangling these counterexamples were very difficult to map back into the source program and its specifications, in a way that allowed efficient debugging of the program and specifications.

Later tools (e.g., [18]) improved on this situation by explicitly building in mechanisms to map logical counterexample models to source program information. Such mechanisms are essential to usable proof systems.

Providing this mapping is a challenge in a system composed of multiple independent pieces: compiler, specification language translator, LLVM$\rightarrow$ logical representation, and SMT or other solver. A single unified tool can more easily maintain the translation information needed to map result information back to source information. This challenge remains a very necessary research and engineering task for the future

9 Related work

This paper defines a BISL for LLVM and describes the translation of LLVM+specifications to Why3 models, which can then be proved using the user's choice of a number of automatic or interactive back-end tools. Since humans do not generally write LLVM, LLVM can be viewed here as an intermediate representation for program source, generated by already-existing compilers. This section describes related work.

9.1 Boogie

The Boogie tool [6] is also an intermediate representation used for software verification. Front-end tools translate source code to BoogiePL; the Boogie tool translates BoogiePL to SMT-LIB. This latter step implements single-assignment and passification transformations and targets the SMT-LIB formats (though not Why3). Thus the front-end must parse the source programming language and the annotation language and transform both into BoogiePL programs, breaking up the control flow into basic blocks and rewriting to use the BoogiePL type system. In contrast, Why3 makes use of existing tools for the front-end and is an implementation of the back-end transformation to Why3 and indirectly to SMT. In our experience, it is the front-end that is the most work, and hence we believe that our approach saves the implementer the most work. LLVM is targeted by many compilers; fewer independently generated tools target BoogiePL.

9.2 SAW

Galois' Static Analysis Workbench (SAW)[24] translates LLVM bitcode files to SMT-LIB. That piece of functionality is replicated by llvm2why. However, our tool also encodes a rich language for specifications in a BISL style and targets Why3. By targeting Why3, we make use of the techniques and optimizations implemented by the Why3 team; using Why3 also enables use of interactive tools like Coq, PVS and Isabelle and wide variety of other, not necessarily SMT-based tools (see the list of Why3-supported tools at the bottom of `http://why3.lri.fr/`). SAW is able to compare a procedure against a Cryptol specification, but has not integrated specifications into the LLVM metadata and does not have the ability to consider annotations within a procedure, such as assert statements or loop invariants.

Note that both llvm2why and SAW benefit from compiler optimizations available in the LLVM framework. Simple programs translated to LLVM with different levels of optimization show considerably different local structure. In our informal observations, optimized LLVM is represented in much simpler logic and is an easier target of proof tools.

9.3 SMACK

The SMACK[1,33] tool performs bounded model checking on LLVM bitcode in the style of the SVCOMP competition. That is, the program under test is annotated with assertions that are checked by the tool. The tool performs symbolic execution rather than deductive verification. It does use the Boogie tool as part of its tool chain. It does not have specific support for loop invariants; in execution, one must state how many times loops should be unrolled. However, the tool does read and interpret LLVM programs translating them to an SMT encoding through Boogie, and as such has an overlap with the llvm2why tool. An alternative implementation of llvm2why, which we are investigating, is to build the encoding of specifications into SMACK, leveraging SMACK's translation of LLVM into Boogie.

9.4 LLVM static analysis

LLVM has developed a rich ecosystem of tools and research projects. For example, LLVM has an infrastructure for performing static analysis on programs translated into LLVM. This static analysis currently targets non-functional properties (does the program execute any runtime errors?), rather than functional properties (does this procedure fulfill its specification?). Thus the domain of static analysis is more or less a subset of that targeted by specification-based tools like llvm2why.

By targeting LLVM, we hope that tools such as llvm2why will be able to leverage other research in program analysis. For instance, one important need in specification-based checking in the paradigm described here is having loop specifications. These are internal to a procedure but are essential to sound verification. Tools that infer (most) loop specifications automatically from LLVM bitcode would be useful to integrate with tools such as llvm2why and SAW, above. Hence research projects such as PAGAI [28] are relevant.

9.5 Other specification languages

As described throughout this paper, the specification language we prototyped for LLVM is analogous to and derived from similar languages for other programming languages, such as JML, ACSL, Spec# and SPARK. Indeed, ACSL and Spec# both are built closely on JML, with modifications and innovations appropriate for their different source languages.

10 Observations, Conclusions, and Future Work

The LLVM2Why system is a working prototype. In building it we encountered a number of challenges in working with LLVM and translating it to a logical encoding. We summarize here the challenges that we presented earlier in the paper.

– **Types**. An important design challenge is the difference in handling signedness of types, discussed earlier in §5.2.
– **Verboseness**. The LLVM-SL language uses the LLVM metadata format. This is a verbose format, even when we use the more readable expression format. If we expected humans to read these specifications regularly, this property would be fatal to usability. However, since the primary goal is tool-to-tool communication, this aspect is manageable. One might think that LLVM's metadata naming feature would be helpful here. This feature allows one to give a name to a metadata expression and then use that name elsewhere. LLVM tools use this feature regularly, to include debug information, for example. Using metadata naming can reduce line length at the cost of non-locality: specifications a textually located separate from the entity they specify.
– **Modeling memory** Like C and C++ programs, LLVM interacts directly with memory addresses and allows reinterpreting memory with different types. This low-level nature creates challenges for the memory model being used.
– **Using solvers through Why3** The problem of obtaining sufficient useful information to debug failed proofs is discussed in §8. This problem is exacerbated by using a tool chain of separate components that have no backward communication channels (though using such a tool chain has the obvious benefit of avoid reimplementation of already well-engineered tools). In the case of this paper, the boundary between Why and SMT solvers is particularly inaccessible, as Why does not interpret solver output to provide counterexample information in the language of the input Why model. This remains a research and engineering challenge.
– **Usability** Working with low-level program representations, such as LLVM (or assembly code) always poses challenges to readability, usability, and consequently to debuggability — both for developers and users of the tool. Usability will be an ongoing research area as we further experiment and improve our tool chain through LLVM.

References

1. `https://smackers.github.io/#body`.
2. Adacore. `https://www.adacore.com/`, 1994.
3. Coq. `http://coq.inria.fr/`, 2012.
4. Frama-c. `http://frama-c.com/`, 2012.
5. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison Wesley, Boston, MA, 2003.
6. 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 LCSC Volume 4111*, pages 364–387, 2006.
7. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS*, volume 3362, pages 49–69. Springer-Verlag, 2004/// 2004.
8. C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, pages 171–177, 2011/// 2011.
9. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.
10. P. Baudin. ACSL: ANSI C Specification Language. `http://frama-c.com/download/acsl_1.4.pdf`, 2011.
11. B. Beckert, R. H?hnle, and P. H. Schmitt. *Verification of object-oriented software: The KeY approach.* 2007.
12. R. Blanc, V. Kuncak, E. Kneuss, and P. Suter. An overview of the Leon Verification System: Verification by Translation to Recursive Functions. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 1:1–1:10, New York, NY, USA, 2013. ACM.
13. F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The alt-ergo automated theorem prover, 2008, 2013.
14. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.
15. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.
16. P. Chalin. Jml support for primitive arbitrary precision numeric types: Definition and semantics. *Journal of Object Technology*, 3(6):57–79, 2004.

17. E. Cohen, M. Dahlweid, HillebrandmMark, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*, pages 23–42, 2009/// 2009. http://dx.doi.org/10.1007/978-3-642-03359-9_2.
18. D. Cok. Improved usability and performance of SMT solvers for debugging specifications. *STTT*, 12:467–481, 2010.
19. D. R. Cok. jSMTLIB: Tutorial, Validation and Adapter Tools for SMT-LIBv2. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 480–486. Springer Berlin Heidelberg, 2011.
20. D. R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *F-IDE*, pages 79–92, 2014.
21. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. volume 3362, pages 108–128, 2005.
22. L. De Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*, pages 337–340, 2008/// 2008.
23. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, Dec. 1998.
24. R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. Tomb. Constructing semantic models of programs with the software analysis workbench. In *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers*, pages 56–72. Springer, 2016.
25. B. Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.
26. M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110, New York, NY, USA, 2010. ACM.
27. J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Computing Surveys (CSUR)*, 44(3):16, 2012.
28. J. Henry, D. Monniaux, and M. Moy. Pagai: A path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25, 2012.
29. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, Washington, DC, USA, 2004. IEEE Computer Society.
30. K. R. M. Leino. Developing verified programs with Dafny. In *HILT '12*, pages 9–10, 2012/// 2012.
31. B. Meyer. *Eiffel: The Language.* Prentice-Hall, Inc., 1992/// 1992.
32. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, volume 607, page 752, 1992/// 1992.
33. Z. Rakamarić and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In A. Biere and R. Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014.
34. J. M. Spivey. *The Z notation: a reference manual.* Prentice Hall International (UK) Ltd., 1992.