OREGON PROGRAMMING LANGUAGES SUMMER SCHOOL

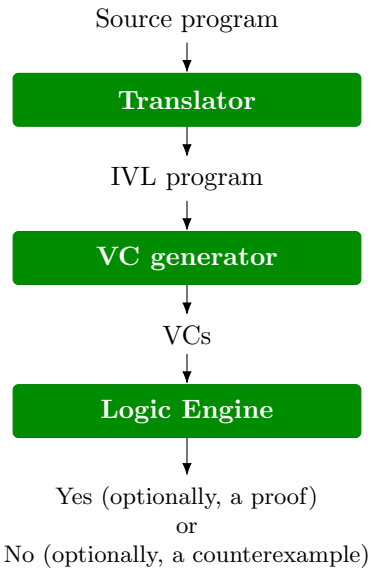
The Use and Anatomy of an Auto-Active Verifier

Rustan Leino

Lecture 3 - *July 2, 2026*

1 Architecture of Verifiers

Compilers usually take source code through some intermediate representation (IR) down to executable machine code. This delineation is logical, as the front end vs. the back end address different concerns. The front end might be concerned with variables renamings or consolidating different types of loops. The back end might be concerned with instruction selection from the particular architecture or register allocation. Analogously, verifiers go through an intermediate verification languages (IVL) before generating verification conditions, i.e. formulae, that are then discharged to a logic engine.



Compiled By:
Namratha Gangamreddypalli, Jasper Geer, Max Gross, Abigail Pribisova,
Hans Sardar, and Konami Shu

More specifically, as in the above diagram, a given source program is translated into an intermediate verification language program which generates certain verification conditions and is handed to a logic engine to solve (returning true or false, perhaps with proof or counterexample, respectively). Usually, the logic engine operates fully automatically.

1.1 Intermediate Verification Languages

Dafny, the auto-active verifier we have seen so far in this course, uses Microsoft's Boogie as an IVL but other examples include Why3 and Coma from France's LMF and Viper from ETH Zurich. A more modern example is B3, a continuation of the Boogie project which is itself written (and therefore verified) in Dafny. Coma is considered the next-generation Why3, and B3 is considered the next-generation Boogie.

At a high level, IVLs look as follows:

Listing 1: IVL in a nutshell

```
IVL   ::= Axiom* Proc*
Axiom ::= axiom Expr
Proc  ::= procedure Id
            requires Expr
            ensures Expr
          { Stmt }
```

They are comprised of axioms and procedures. The procedures have a precondition following the `requires` and a postcondition following the `ensures`.

1.2 Verification Condition Generation

We will examine the generation of verification conditions from a very high level. Assume you have axioms $A_0, A_1, \ldots$ and procedures $M_0, M_1, \ldots$ over preconditions $P_0, P_1, \ldots$ and postconditions $Q_0\{S_0\}, Q_1\{S_1\}$ as follows:

```
    axiom A0
    axiom A1
    ...
    procedure M0 requires P0 ensures Q0 { S0 }
    procedure M1 requires P1 ensures Q1 { S1 }
    ...
```

The verification conditions generator will produce (something akin to) the following formulae:

$$A0 \wedge A1 \wedge \cdots \wedge P0 \implies \mathrm{wp}[\![S0, Q0]\!]$$

$$A0 \wedge A1 \wedge \cdots \wedge P1 \implies \text{wp}[\![S1, Q1]\!]$$

Intuitively, this `wp` (weakest precondition) characterizes states in the precondition P_N where we can execute programs S_N and successfully get to the postcondition Q_N. This is another of stating that the Hoare triple $\{P_N\}S_N\{Q_N\}$ is valid in Hoare's partial incorrectness program logic. Dafny processes this output and relays verification information to the user.

1.3 Demonstration!

terminal

```
$ dafny verify BoolSort.dfy --bprint=BoolSort.bpl --solver-log=BoolSort.smt2
```

Try it out! The `bpl` file corresponds to the Boogie file, and the `smt2` file is the encoding that is sent to the SMT solver. Theoretically, if one has the SMT solver installed on their machine, they should be able to pass the `smt2` file directly to the solver and observe the same output.

1.4 Intermediate Verification Language in More Detail

Listing 2: IVL in more detail

```
Stmt ::=
  | var x := Rhs
  | x := Rhs
  | assert Expr
  | Stmt ; Stmt
  | Stmt [] Stmt
  | loop
  | procedure call
  | ...

Rhs ::= Expr | *
```

Listing 2 describes statements in our IVL. It is an imperative language with mutable variables, loops, and procedures. One possibly unfamiliar feature is the `[]` statement, which models demonic non-deterministic control flow. In other words, we imagine that an adversary chooses which branch to execute, with the intent of causing a failure; we must rule out failure for all branches. An `Rhs` is either an expression or the wildcard $\star$. Assigning a variable to $\star$ assigns it to an arbitrary value of the variable's type.

2 Weakest Preconditions

As explained above, weakest preconditions characterize the weakest possible restriction on the initial state of the program such that the postcondition will hold after a terminating execution. Then, verification can simply check whether the user-annotated precondition implies this weakest precondition. We provide some intuition for the following weakest preconditions.

Listing 3: Weakest preconditions

```
wp[[x := e, Q]]      = Q[x := e]
wp[[x := *, Q]]      = ∀x. Q
wp[[assert e, Q]]    = e ∧ Q
wp[[assume e, Q]]    = e ⟹ Q
wp[[S; T, Q]]        = wp[[S, wp[[T, Q]]]]
wp[[S [] T, Q]]      = wp[[S, Q]] ∧ wp[[T, Q]]
```

Variable assignment and sequencing are the standard weakest preconditions. Assigning a variable to the "wild card" symbol requires ensuring that the postcondition holds for *all* possible instantiations of `x`, hence the quantification over `x`. As a failed `assert` causes program failure, `e` needs to hold in addition to the postcondition. On the other hand, an `assume e` introduces information that can be assumed and used when proving the postcondition. Finally, demonic choice requires that the weakest precondition holds for both branches. One notes that this can lead to exponential number of formulas; however, given that the number of paths through such a program is exponential, this is not too surprising.

2.1 Well-Formedness

Not all source-language expressions we can write have well-defined meanings. For example, consider assigning a variable to the result of a division:

Source Code

```
a := b/c
```

If `c` turns out to be `0`, the result of `b/c` is not well-defined; chaos ensues! As a result, our encoding of our source must rule out this possibility. To do so, the encoding of our source program into the IVL inserts a *well-formedness* check prior to the assignment:

IVL Code

```
1 assert b != 0
2 x := a/b
```

If it is possible that the division is ill-defined, this assertion will fail to verify, and we can provide an appropriate error message to the user.

2.2 Modelling Source Language Operations

Suppose that our source language supports fixed-size integers; if our IVL supports only unbounded integers, we must do some extra work to faithfully model integer operations. Consider the following source language program:

Source Code

```
1 var a : u32, b : u32, x : u32
2     x := a + b
```

If the sum of `a` and `b` (as unbounded integers) is larger than the largest representable `u32`, we may have a problem! Depending on the semantics of our source language, we might model this in different ways.

If this operation would cause integer overflow, then our IVL program might perform addition modulo the maximum `u32` value:

IVL Code

```
1 x := a + b % 0x1_0000_0000
```

Alternatively, if this would cause a program to be unsafe or if we decide that our verifier should rule out integer overflow, we might choose to precede the assignment with an assertion, similar to our well-formedness checks:

IVL Code

```
1 assert a + b <  0x1_0000_0000
2     x := a + b
```

Another possibility is that integer overflow raises an exception (like in Java); we might choose to model the operation as follows:

IVL Code

```
if a + b <  0x1_0000_0000 {
        x := a + b
    } else {
        // model throwing an exception
    }
```

2.3 Definite Assignment

Accessing an uninitialised variable is often considered an error. Take for instance the following snippet of code:

Source Code : definite assignment

```
    var x
if B {
x := 100
}
...
if C { y := x
}
```

This code could fail when C is true but B is false, as x may be uninitialised. To avoid logical errors associated with the access these uninitialised variable, one adds a boolean variable tracking x's initialisation - `_defined_x` below.

IVL Code : after translation

```
    var x := *; var _defined_x := false
if B {
x := 100; _defined_x := true
}
...
if C {
assert _defined_x; y := x
}
```

2.4 Types

The source language might support high-level features such as ADTs and pattern matching that are not supported by the IVL. Consider the following source program:

Source Code : types

```
datatype Color = Red | Blue | Green
var c: Color

match c
case Red => S0
case Blue => S1
case Green => S2
```

In order to properly represent this `match` in the IVL, one can naively use `if ... else if ... else` to specify that a branch should only be taken in the corresponding case:

IVL Code : after translation

```
if c == Red {
S0
} else if !(c == Red) && c == Blue {
S1
} else if !(c == Red) && !(c == Blue) && c == Green {
S2
} else {
assert false;
}
```

While this translation is correct, it contains extra checks, notably the negations `!(c == Red)` and `!(c == Blue)`, that do not make use of the semantics of `if ... else if ... else` blocks in the IVL, which ensure that this check is trivially true. Instead, we can restructure as follows:

IVL Code : after translation

```
1 if c == Red {
2 S0
3 } else if c == Blue {
4 S1
5 } else if c == Green {
6 S2
7 } else {
8 assert false;
9 }
```

This is an improvement, but it still does not capitalize on the fact that the `else` branch, which is necessary to construct a proper `if ... else if ... else` block in the IVL, is never reached, as guaranteed by the source-language semantics. We would much rather assume that we will *never* reach the `else` branch, which is a construct in the IVL - `assume false`! We can make one more improvement utilizing demonic choice in the IVL *and* that S_N is executed only on the corresponding branch:

IVL Code : after translation

```
1 assume c == Red; S0
2 □
3 assume c == Blue; S1
4 □
5 assume c == Green; S2
6 □
7 assume false
```

Finally, we can make use of the fact that `assume false` is the unit element of $\square$, meaning that its addition to a logical formula does not affect the satisfiability of the formula, like `True` with $\wedge$ and `False` with $\vee$. This leads to our final IVL encoding:

IVL Code : after translation

```
1 assume c == Red; S0
2 □
3 assume c == Blue; S1
4 □
5 assume c == Green; S2
```

3 Desugaring

We discuss two forms of desugaring in the IVL: procedure calls and loops.

3.1 Procedure Calls

Consider the following procedure declaration and call in the IVL:

IVL Code : desugar call

```
1 procedure M(x) returns (y)
2 requires P
3 ensures Q
4
5 call a := M(b)
```

This corresponds to the following desugared IVL:

IVL Code : desugared IVL

```
1 {
2 var x := b
3 assert P
4 var y := *
5 assume Q
6 a := y
7 }
```

Since we call the procedure with `b`, we bind the function parameter `var x` to this in-parameter. Then, we have to confirm that the precondition to the procedure call is satisfied via `assert P`. Then, we assign the output `y` to the wild card operator, accounting for any operation that could happen within the procedure body. We assume that the postcondition holds and assign this output to the actual out-parameter.

3.2 Loops

We can encode source-language loops using more primitive operations in the IVL. Consider the following source-language program:

Source Code

```
while e
invariant J
{
S
}
```

We model this program in the IVL as follows:

IVL Code

```
assert J

x := *
assume J

if e {
S
assert J
assume false
}
```

Before we enter the loop, the loop invariant must be checked, so we first assert the loop invariant `J`. Next, we imagine that we have "fast-forwarded" to an arbitrary point immediately before the loop condition is checked. This generalizes both the end arbitrary loop iterations and the initial entry into the loop. Note that at both of these points, we can assume that the loop invariant has been checked. We model this by havocing any variables modified in the loop and assuming the loop invariant. Next, we must check the loop condition and enter the loop body if it succeeds. We model this with an `if`-statement that tests the loop condition. The body of the `if`-statement contains the encoding of the loop body, followed by a check that the loop invariant still holds, and finally an `assume false`. Without the `assume false`, the IVL program would model extraneous executions in which the loop body is executed and the loop condition continues to hold, but the loop body is not executed again. This would cause our verifier to reject many correct programs.

References

[1] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*, pages 82–87. ACM, 2005. doi: 10.1145/1108792.1108813. `https://leino.science/papers/krml157.pdf`.

[2] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*, pages 193–205. ACM, 2001. doi: 10.1145/360204.360220.

[3] K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6): 281–288, 2005. doi: 10.1016/j.ipl.2004.10.015. `https://leino.science/papers/krml114.pdf`.

[4] K. Rustan M. Leino. Specification and verification of object-oriented software. In *Engineering Methods and Tools for Software Safety and Security*, Marktoberdorf International Summer School, Lecture Notes. 2008. `https://leino.science/papers/krml190.pdf`.