Deductive Verification

Ondřej Lengál

SAV'24, FIT VUT v Brně

14 October 2024

Verified Programming

How to write software that is correct?

- First approach
  1. First, write the software.
  2. Then, whack it with whatever you can find (verify & test it, burn it) until no bugs.
- Second approach
  - **Verified Programming**: programming + **deductive verification**
    - i.e., writing codes with annotations

cautionary tale: binary search

- algorithm first published in 1946, but first correct version didn't appear until 1962
- in 1988, a survey of 20 textbooks on algorithms found that at least 15 of them had errors
- Bentley reports giving it as a programming problem to over 100 professional programmers from Bell Labs and IBM, with 2 hours to produce a correct program. At least 90% of the solutions were wrong. Dijkstra reported similar statistics in experiments he performed at many institutions.
- Bentley published a CACM "programming pearl" on binary search and proving it correct, expanded to 14 pages in "Programming Pearls" (1986).
- Joshua Bloch used Bentley's code as a basis for the binary search implementation in the JDK, in 1997.
- in 2006, a bug was found in the JDK code, the same bug that was in Bentley's code, which nobody had noticed for 20 years. The same bug was in the C code Bentley published for the second edition of his book in 2000.
- these are not exactly your average programmers

[from slides of Ernie Cohen]

Deductive Verification

- the system is accompanied by specification
- these are converted into proof obligations (program invariant—a big formula)
- the truth of proof obligations imply correctness of the system
  - this is discharged by different methods:
    - SMT solvers (Z3, STP, cvc5, ...)
    - automatic theorem provers (Vampire, Prover9, E, ...)
    - interactive theorem provers (Coq, Isabelle, Lean, ...)
- **Pros**:
  - **strong correctness guarantees** (e.g., program correct *"up to bugs in the solver"*)
  - modularity; can be quite general
- **Cons**:
  - quite manual $\rightsquigarrow$ expensive, high user expertise needed
  - garbage in, garbage out
  - not always easy to get counterexamples
  - not so strong tool support

A Bit of History . . .

- 1949: Alan Turing: Checking a Large Routine.
- 1969: Tony Hoare: An Axiomatic Basis for Computer Programming.
  - a formal system for rigorous reasoning about programs
  - Floyd-Hoare triples $\{pre\}\ stmt\ \{post\}$
    - 1967: Robert Floyd: Assigning Meaning to Programs
- 1971: Tony Hoare: Proof of a Program: FIND
- 1976: E. Dijkstra: A Discipline of Programming.
  - weakest-precondition calculus
- 2000: efficient tool support starts

Floyd-Hoare Logic

Let us consider the following imperative programming language:

- Expression: $E ::= n \mid x \mid E_1 + E_2 \mid E_1 \cdot E_2$ for $n \in \mathbb{Z}$ and $x \in \mathbb{X}$ (set of program variables)
- Conditional: $C ::= \texttt{true} \mid \texttt{false} \mid E_1 = E_2 \mid E_1 \leq E_2 \mid E_1 < E_2$
- Statement:

$$
\begin{array}{rll}
S ::= & x := E & \text{(assignment)} \\
\mid & S_1; S_2 & \text{(sequence)} \\
\mid & \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2 & \text{(if)} \\
\mid & \texttt{while}\ C\ \texttt{do}\ S & \text{(while)}
\end{array}
$$

A program is a statement.

Partial Correctness

Partial correctness of programs in Hoare logic is specified using Hoare triples:

$$\{P\}\ S\ \{Q\}$$

where

Partial Correctness

Partial correctness of programs in Hoare logic is specified using Hoare triples:

$$\{P\}\ S\ \{Q\}$$

where

- S is a statement of the programming language
- P and Q are formulae in a suitable fragment of logic (usually first-order logic or SMT)
  - P is called precondition
  - Q is called postcondition

Meaning:

- if S is executed from a state (program configuration) satisfying formula P
- and the execution of S terminates,
- then the program state after S terminates satisfies formula Q.

Example

1. Is $\{x = 0\}$ $\mathtt{x := x + 1}$ $\{x = 1\}$ a valid Hoare triple?

Partial Correctness

Partial correctness of programs in Hoare logic is specified using Hoare triples:

$$\{P\}\ S\ \{Q\}$$

where

- S is a statement of the programming language
- P and Q are formulae in a suitable fragment of logic (usually first-order logic or SMT)
  - P is called precondition
  - Q is called postcondition

Meaning:

- if S is executed from a state (program configuration) satisfying formula P
- and the execution of S terminates,
- then the program state after S terminates satisfies formula Q.

Example

1. Is $\{x = 0\}\ \mathrm{x} := \mathrm{x} + 1\ \{x = 1\}$ a valid Hoare triple?
2. $\{x = 0 \wedge y = 1\}\ \mathrm{x} := \mathrm{x} + 1\ \{x = 1 \wedge y = 2\}$?

Partial Correctness

Partial correctness of programs in Hoare logic is specified using Hoare triples:

$$\{P\}\ S\ \{Q\}$$

where

- S is a statement of the programming language
- P and Q are formulae in a suitable fragment of logic (usually first-order logic or SMT)
  - P is called precondition
  - Q is called postcondition

Meaning:

- if S is executed from a state (program configuration) satisfying formula P
- and the execution of S terminates,
- then the program state after S terminates satisfies formula Q.

Example

1. Is $\{x = 0\}$ x := x + 1 $\{x = 1\}$ a valid Hoare triple?
2. $\{x = 0 \land y = 1\}$ x := x + 1 $\{x = 1 \land y = 2\}$?
3. $\{x = 0\}$ x := x + 1 $\{x = 1 \lor y = 2\}$?

Partial Correctness

Partial correctness of programs in Hoare logic is specified using Hoare triples:

$$\{P\}\ S\ \{Q\}$$

where

- S is a statement of the programming language
- P and Q are formulae in a suitable fragment of logic (usually first-order logic or SMT)
  - P is called precondition
  - Q is called postcondition

Meaning:

- if S is executed from a state (program configuration) satisfying formula P
- and the execution of S terminates,
- then the program state after S terminates satisfies formula Q.

Example

1. Is $\{x = 0\}$ x := x + 1 $\{x = 1\}$ a valid Hoare triple?
2. $\{x = 0 \wedge y = 1\}$ x := x + 1 $\{x = 1 \wedge y = 2\}$?
3. $\{x = 0\}$ x := x + 1 $\{x = 1 \vee y = 2\}$?
4. $\{x = 0\}$ `while true do` x := 0 $\{x = 1\}$?

Total Correctness

- $\{P\}\ S\ \{Q\}$ does not require S to terminate (*partial correctness*).
- Hoare triples for total correctness:

$$[P]\ S\ [Q]$$

  Meaning:
  - if S is executed from a state (program configuration) satisfying formula P,
  - **then** the execution of S terminates and
  - the program state after S terminates satisfies formula Q.

Example

Is $[x = 0]$ `while true do x` $:= 0$ $[x = 1]$ valid?

In the following we focus only on *partial correctness*.

Examples

Example

What are the meanings of the following Hoare triples?

1. $\{true\}\ S\ \{Q\}$

Examples

Example

What are the meanings of the following Hoare triples?

1. $\{true\}\ S\ \{Q\}$
2. $\{P\}\ S\ \{true\}$

Examples

Example

What are the meanings of the following Hoare triples?

1. $\{true\}\ S\ \{Q\}$
2. $\{P\}\ S\ \{true\}$
3. $[P]\ S\ [true]$

Examples

Example

What are the meanings of the following Hoare triples?

1. $\{true\}\ S\ \{Q\}$
2. $\{P\}\ S\ \{true\}$
3. $[P]\ S\ [true]$
4. $\{true\}\ S\ \{false\}$

Examples

Example

What are the meanings of the following Hoare triples?

1. $\{true\}\ S\ \{Q\}$
2. $\{P\}\ S\ \{true\}$
3. $[P]\ S\ [true]$
4. $\{true\}\ S\ \{false\}$
5. $\{false\}\ S\ \{Q\}$

Examples

Example

What are the meanings of the following Hoare triples?

1. $\{true\}\ S\ \{Q\}$
2. $\{P\}\ S\ \{true\}$
3. $[P]\ S\ [true]$
4. $\{true\}\ S\ \{false\}$
5. $\{false\}\ S\ \{Q\}$

Example

Are the following Hoare triples valid or invalid?

1. $\{i = 0 \land n \geq 0\}$ `while i<n do i++` $\{i = n\}$

Examples

Example

What are the meanings of the following Hoare triples?

1. $\{true\}\ S\ \{Q\}$
2. $\{P\}\ S\ \{true\}$
3. $[P]\ S\ [true]$
4. $\{true\}\ S\ \{false\}$
5. $\{false\}\ S\ \{Q\}$

Example

Are the following Hoare triples valid or invalid?

1. $\{i = 0 \land n \geq 0\}$ `while i<n do i++` $\{i = n\}$
2. $\{i = 0 \land n \geq 0\}$ `while i<n do i++` $\{i \geq n\}$

Examples

Example

What are the meanings of the following Hoare triples?

1. $\{true\}\ S\ \{Q\}$
2. $\{P\}\ S\ \{true\}$
3. $[P]\ S\ [true]$
4. $\{true\}\ S\ \{false\}$
5. $\{false\}\ S\ \{Q\}$

Example

Are the following Hoare triples valid or invalid?

1. $\{i = 0 \land n \geq 0\}$ `while i<n do i++` $\{i = n\}$
2. $\{i = 0 \land n \geq 0\}$ `while i<n do i++` $\{i \geq n\}$
3. $\{i = 0 \land j = 0 \land n \geq 0\}$ `while i<n do {i++; j+=i}` $\{2j = n(1 + n)\}$

Inference Rules

We write proof rules in Hoare logic as inference rules:

$$\frac{\vdash \{P_1\}\ S_1\ \{Q_1\} \qquad \ldots \qquad \vdash \{P_n\}\ S_n\ \{Q_n\}}{\vdash \{P\}\ S\ \{Q\}}$$

Meaning:

- If all Hoare triples $\{P_1\}\ S_1\ \{Q_1\}, \ldots, \{P_n\}\ S_n\ \{Q_n\}$ are provable, then $\{P\}\ S\ \{Q\}$ is also provable.

In general, inference rules have the format $\dfrac{premises}{deductions}\ \textsc{Name}$. A rule with no premises is an axiom.

The proof system will have one rule for every statement of our language:

- an axiom for *atomic* statements: assignments,
- inference rules for *composite* statements: sequence, if, while
- auxiliary "*helper*" rules

Proof Rule (Assignment)

For assignment $x := E$, we have the following proof rule:

$$\frac{}{\vdash \{Q[E/x]\}\ x := E\ \{Q\}}\ \textsc{Assgn}$$

where $Q[E/x]$ denotes the formula obtained from Q by substituting all free occurrences of x by E

Example

Which of the following Hoare triples can we prove using this rule?

1. $\{y = 4\}$ `x := 4` $\{y = x\}$

Proof Rule (Assignment)

For assignment $x := E$, we have the following proof rule:

$$\frac{}{\vdash \{Q[E/x]\}\ x := E\ \{Q\}}\ \textsc{Assgn}$$

where $Q[E/x]$ denotes the formula obtained from Q by substituting all free occurrences of x by E

Example

Which of the following Hoare triples can we prove using this rule?

1. $\{y = 4\}$ `x := 4` $\{y = x\}$
2. $\{x = n - 1\}$ `x := x+1` $\{x = n\}$

Proof Rule (Assignment)

For assignment $x := E$, we have the following proof rule:

$$\frac{}{\vdash \{Q[E/x]\}\ x := E\ \{Q\}}\ \textsc{Assgn}$$

where $Q[E/x]$ denotes the formula obtained from Q by substituting all free occurrences of x by E

Example

Which of the following Hoare triples can we prove using this rule?

1. $\{y = 4\}$ `x := 4` $\{y = x\}$
2. $\{x = n - 1\}$ `x := x+1` $\{x = n\}$
3. $\{y = x\}$ `y := 2` $\{y = x\}$

Proof Rule (Assignment)

For assignment $x := E$, we have the following proof rule:

$$\frac{}{\vdash \{Q[E/x]\}\ x := E\ \{Q\}}\ \textsc{Assgn}$$

where $Q[E/x]$ denotes the formula obtained from Q by substituting all free occurrences of x by E

Example

Which of the following Hoare triples can we prove using this rule?

1. $\{y = 4\}$ `x := 4` $\{y = x\}$
2. $\{x = n - 1\}$ `x := x+1` $\{x = n\}$
3. $\{y = x\}$ `y := 2` $\{y = x\}$
4. $\{z = 3\}$ `y := x` $\{z = 3\}$

Proof Rule (Assignment)

For assignment $x := E$, we have the following proof rule:

$$\frac{}{\vdash \{Q[E/x]\}\ x := E\ \{Q\}}\ \textsc{Assgn}$$

where $Q[E/x]$ denotes the formula obtained from Q by substituting all free occurrences of x by E

Example

Which of the following Hoare triples can we prove using this rule?

1. $\{y = 4\}$ `x := 4` $\{y = x\}$
2. $\{x = n - 1\}$ `x := x+1` $\{x = n\}$
3. $\{y = x\}$ `y := 2` $\{y = x\}$
4. $\{z = 3\}$ `y := x` $\{z = 3\}$
5. $\{z = 3\}$ `y := x` $\{x = y\}$

Strengthening/Weakening

Strengthening/weakening might be necessary in order to be able to apply some rules

Precondition Strengthening

$$\frac{\vdash \{P'\}\ S\ \{Q\} \qquad P \Rightarrow P'}{\vdash \{P\}\ S\ \{Q\}}\ \textsc{Strength}$$

Precondition can be always tightened to something stronger.

Postcondition Weakening

$$\frac{\vdash \{P\}\ S\ \{Q'\} \qquad Q' \Rightarrow Q}{\vdash \{P\}\ S\ \{Q\}}\ \textsc{Weak}$$

Postcondition can be always relaxed to something weaker.

Conclusion (generalisation of the two above rules)

$$\frac{P \Rightarrow P' \qquad \vdash \{P'\}\ S\ \{Q'\} \qquad Q' \Rightarrow Q}{\vdash \{P\}\ S\ \{Q\}}\ \textsc{Concl}$$

Strengthening/Weakening (contd.)

Example

We can now prove the following: $\{z = 3\}$ `y := x` $\{x = y\}$

Strengthening/Weakening (contd.)

Example

We can now prove the following: $\{z = 3\}$ `y := x` $\{x = y\}$

$$\dfrac{\dfrac{}{\vdash \{(x = y)[x/y]\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Assgn}}{\dfrac{\vdash \{true\}\ \texttt{y := x}\ \{x = y\} \qquad\qquad z = 3 \Rightarrow true}{\vdash \{z = 3\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Strength}}$$

Strengthening/Weakening (contd.)

Example

We can now prove the following: $\{z = 3\}\ \texttt{y := x}\ \{x = y\}$

$$\dfrac{\dfrac{\dfrac{}{\vdash \{(x = y)[x/y]\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Assgn}}{\vdash \{\mathit{true}\}\ \texttt{y := x}\ \{x = y\}} \qquad z = 3 \Rightarrow \mathit{true}}{\vdash \{z = 3\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Strength}$$

Example

Assume $\vdash \{\mathit{true}\}\ S\ \{x = y \wedge z = 2\}$. Which of the following can we prove from it?

1. $\{\mathit{true}\}\ S\ \{x = y\}$

Strengthening/Weakening (contd.)

Example

We can now prove the following: $\{z = 3\}\ \texttt{y := x}\ \{x = y\}$

$$\dfrac{\dfrac{}{\vdash \{(x = y)[x/y]\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Assgn}}{\dfrac{\vdash \{true\}\ \texttt{y := x}\ \{x = y\} \qquad z = 3 \Rightarrow true}{\vdash \{z = 3\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Strength}}$$

Example

Assume $\vdash \{true\}\ S\ \{x = y \land z = 2\}$. Which of the following can we prove from it?

1. $\{true\}\ S\ \{x = y\}$
2. $\{true\}\ S\ \{z = 2\}$

Strengthening/Weakening (contd.)

Example

We can now prove the following: $\{z = 3\}\ \texttt{y := x}\ \{x = y\}$

$$\dfrac{\dfrac{}{\vdash \{(x = y)[x/y]\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Assgn}}{\vdash \{true\}\ \texttt{y := x}\ \{x = y\}} \quad z = 3 \Rightarrow true$$
$$\overline{\vdash \{z = 3\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Strength}$$

Example

Assume $\vdash \{true\}\ S\ \{x = y \wedge z = 2\}$. Which of the following can we prove from it?

1. $\{true\}\ S\ \{x = y\}$
2. $\{true\}\ S\ \{z = 2\}$
3. $\{true\}\ S\ \{z > 0\}$

Strengthening/Weakening (contd.)

Example

We can now prove the following: $\{z = 3\}$ `y := x` $\{x = y\}$

$$\cfrac{\cfrac{\cfrac{}{\vdash \{(x = y)[x/y]\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Assgn}}{\vdash \{\mathit{true}\}\ \texttt{y := x}\ \{x = y\}} \qquad z = 3 \Rightarrow \mathit{true}}{\vdash \{z = 3\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Strength}$$

Example

Assume $\vdash \{\mathit{true}\}\ S\ \{x = y \land z = 2\}$. Which of the following can we prove from it?

1. $\{\mathit{true}\}\ S\ \{x = y\}$
2. $\{\mathit{true}\}\ S\ \{z = 2\}$
3. $\{\mathit{true}\}\ S\ \{z > 0\}$
4. $\{\mathit{true}\}\ S\ \{\forall u(x = u)\}$

Strengthening/Weakening (contd.)

Example

We can now prove the following: $\{z = 3\}$ `y := x` $\{x = y\}$

$$\dfrac{\dfrac{\dfrac{}{\vdash \{(x = y)[x/y]\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Assgn}}{\vdash \{true\}\ \texttt{y := x}\ \{x = y\}} \qquad z = 3 \Rightarrow true}{\vdash \{z = 3\}\ \texttt{y := x}\ \{x = y\}}\ \textsc{Strength}$$

Example

Assume $\vdash \{true\}\ S\ \{x = y \wedge z = 2\}$. Which of the following can we prove from it?

1. $\{true\}\ S\ \{x = y\}$
2. $\{true\}\ S\ \{z = 2\}$
3. $\{true\}\ S\ \{z > 0\}$
4. $\{true\}\ S\ \{\forall u(x = u)\}$
5. $\{true\}\ S\ \{\exists u(x = u)\}$

Proof Rule (Sequence)

For a sequence of two statements $S_1; S_2$, we have the following proof rule:

$$\frac{\vdash \{P\}\ S_1\ \{R\} \qquad \vdash \{R\}\ S_2\ \{Q\}}{\vdash \{P\}\ S_1; S_2\ \{Q\}}\ \textsc{Seq}$$

Often, we need to find an appropriate R.

Example

Prove the correctness of $\{true\}$ `x := 2; y := x` $\{x = 2 \land y = 2\}$:

Proof Rule (Sequence)

For a sequence of two statements $S_1; S_2$, we have the following proof rule:

$$\frac{\vdash \{P\}\ S_1\ \{R\} \qquad \vdash \{R\}\ S_2\ \{Q\}}{\vdash \{P\}\ S_1; S_2\ \{Q\}}\ \textsc{Seq}$$

Often, we need to find an appropriate R.

Example

Prove the correctness of $\{true\}$ `x := 2; y := x` $\{x = 2 \wedge y = 2\}$:

$$\frac{\dfrac{}{\vdash \{true\}\ \texttt{x := 2}\ \{x = 2\}}\ \textsc{Assgn} \qquad \dfrac{}{\vdash \{x = 2\}\ \texttt{y := x}\ \{x = 2 \wedge y = 2\}}\ \textsc{Assgn}}{\vdash \{true\}\ \texttt{x := 2; y := x}\ \{x = 2 \wedge y = 2\}}\ \textsc{Seq}$$

Proof Rule (If)

For `if` C `then` S_1 `else` S_2 we have the following proof rule:

$$\frac{\vdash \{P \land C\}\ S_1\ \{Q\} \qquad \vdash \{P \land \neg C\}\ S_2\ \{Q\}}{\vdash \{P\}\ \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}}\ \textsc{If}$$

Example

Prove the correctness of $\{true\}$ `if x > 0 then y := x else y := -x` $\{y \geq 0\}$.

Proof Rule (If)

For `if` C `then` S_1 `else` S_2 we have the following proof rule:

$$\frac{\vdash \{P \land C\}\ S_1\ \{Q\} \qquad \vdash \{P \land \neg C\}\ S_2\ \{Q\}}{\vdash \{P\}\ \texttt{if}\ C\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{Q\}}\ \textsc{If}$$

Example

Prove the correctness of $\{true\}$ `if x > 0 then y := x else y := -x` $\{y \geq 0\}$.

$$\dfrac{\dfrac{\dfrac{}{\vdash \{x \geq 0\}\ \texttt{y := x}\ \{y \geq 0\}}\ \textsc{Assgn}}{\vdash \{x > 0\}\ \texttt{y := x}\ \{y \geq 0\}}\ \textsc{Strength} \qquad \dfrac{\dfrac{}{\vdash \{-x \geq 0\}\ \texttt{y := -x}\ \{y \geq 0\}}\ \textsc{Assgn}}{\vdash \{x \leq 0\}\ \texttt{y := -x}\ \{y \geq 0\}}}{\vdash \{true\}\ \texttt{if x > 0 then y := x else y := -x}\ \{y \geq 0\}}\ \textsc{If}$$

Proof Rule (While)

Consider the following code:

```
i := 0; j := 0; n := 10;
while i < n do {
  i := i + 1;
  j := i + j;
}
```

Which of the following formulae are loop invariants?

$$i \leq n \qquad i < n \qquad j \geq 0$$

For `while` C `do` S we have the following proof rule:

Proof Rule (While)

Consider the following code:

```
i := 0; j := 0; n := 10;
while i < n do {
  i := i + 1;
  j := i + j;
}
```

Which of the following formulae are loop invariants?

$$i \leq n \qquad i < n \qquad j \geq 0$$

For `while` C `do` S we have the following proof rule:

$$\frac{\vdash \{P \land C\}\ S\ \{P\}}{\vdash \{P\}\ \texttt{while}\ C\ \texttt{do}\ S\ \{P \land \neg C\}}\ \textsc{While}$$

"If P is a loop invariant, then $P \land \neg C$ must hold after the loop terminates."

Proof Rule (While)

Example

Prove the correctness of $\{x \leq n\}$ `while x < n do x := x+1` $\{x \geq n\}$.

Proof Rule (While)

Example

Prove the correctness of $\{x \leq n\}$ `while x < n do x := x+1` $\{x \geq n\}$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \{x+1 \leq n\}\ \texttt{x := x+1}\ \{x \leq n\}}\ \textsc{Assgn}}{\vdash \{x < n\}\ \texttt{x := x+1}\ \{x \leq n\}}\ \textsc{Strength}}{\vdash \{x \leq n \land x < n\}\ \texttt{x := x+1}\ \{x \leq n\}}\ \textsc{Strength}}{\vdash \{x \leq n\}\ \texttt{while x < n do x := x+1}\ \{x \leq n \land \neg(x < n)\}}\ \textsc{While} \qquad x \leq n \land \neg(x < n) \Rightarrow x \geq n
$$

$$
\dfrac{\vdash \{x \leq n\}\ \texttt{while x < n do x := x+1}\ \{x \leq n \land \neg(x < n)\} \qquad x \leq n \land \neg(x < n) \Rightarrow x \geq n}{\vdash \{x \leq n\}\ \texttt{while x < n do x := x+1}\ \{x \geq n\}}\ \textsc{Weak}
$$

Exercise

Prove partial correctness of the program below

```
/* { y = 12 } */
x := y;
while (x < 30) {
  x := x * 2;
  x := x - 2;
}
/* { x = 42 } */
```

Hint: a suitable candidate for the loop invariant might be the formula $(\exists n \in \mathbb{N} \colon x = 2^n(y-2)+2) \land (x \leq 42)$.

How does it work in practice?

In the following, we will be using **VCC** (A **V**erifier for **C**oncurrent **C**):

- available at https://github.com/microsoft/vcc
- can run as a MS Visual Studio plugin (needs older VS)
- currently somewhat orphaned and not industrial-strong
- but used to verify MS Hyper-V hypervisor
  - 60 KLOC of operating system-level concurrent C and x64 assembly code
- interactive web interface: https://rise4fun.com/Vcc
- other systems exist (Frama-C, OpenJML, KeY, . . .)

Example 1

Let's start with something simple

```
#include <vcc.h>

unsigned add(unsigned x, unsigned y)
{
  unsigned w = x + y;
  return w;
}
```

Example 1

VCC

Microsoft Research

Does this C program always work?

```
1 #include <vcc.h>
2
3 unsigned add(unsigned x, unsigned y)
4 {
5   unsigned w = x + y;
6   return w;
7 }
```

		Description	Line	Column
⊗	1	x + y might overflow.	5	16

```
Verification of add failed. [1.83]
snip(5,16) : error VC8004: x + y might overflow.
Verification errors in 1 function(s)
Exiting with 3 (1 error(s).)
```

Example 1

Fix attempt #1:

```
#include <vcc.h>

unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)        // <-- added precondition
{
  unsigned w = x + y;
  return w;
}
```

Example 1

Microsoft Research

VCC

Does this C program always work?

```c
#include <vcc.h>

unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
{
  unsigned w = x + y;
  return w;
}
```

```
Verification of add succeeded. [1.83]
```

Example 1

Microsoft Research

VCC

Does this C program always work?

```
#include <vcc.h>

unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
{
  unsigned w = x + y;
  return w;
}
```

```
Verification of add succeeded. [1.83]
```

- verifies, but what?

Example 1

Fix attempt #2:

```
#include <vcc.h>

unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y)          // <-- added postcondition
{
  unsigned w = x + y;
  return w;
}
```

Example 1

VCC

Microsoft Research

Does this C program always work?

```c
#include <vcc.h>

unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y)
{
  unsigned w = x + y;
  return w;
}
```

```
Verification of add succeeded. [1.88]
```

- verifies wrt the specification \o/

Example 1 — post mortem

What did we do?

1. First, we tried to verify a code with no annotations
   - VCC has a set of default correctness properties
     - e.g. no `NULL` pointer dereference, (over/under)-flows, 0-division, ...
   - one property was violated
2. We fixed the violation using a `_(requires` φ`)` annotation
   - **precondition**: formula φ holds on entry to to the function (extended C syntax)
3. We provided an `_(ensures` ψ`)` annotation to define what we expect as a result
   - **postcondition**: formula ψ holds on return from the function (`\result` is the output)

preconditions + postconditions = function contract

Example 1 — post mortem

What happened behind the scenes?

- the function and its specification were converted into a formula of the form

$$(pre \ \wedge \ \varphi_P) \rightarrow (post \ \wedge \ safe_P)$$

  - pre is the precondition
  - $post$ is the postcondition
  - φ_P is a formula representing the function
  - $safe_P$ represents implicit safety conditions on P
    - no overflows, no out-of-bounds array accesses, ...

$$\big(x_0{+}y_0 \leq \texttt{UINT\_MAX} \wedge w_1 = x_0{+}y_0 \wedge res = w_1\big) \rightarrow \big(res = x_0{+}y_0 \wedge x_0{+}y_0 \leq \texttt{UINT\_MAX}\big)$$

  - the formula is tested for validity with an SMT solver (Z3) that supports the theories

->

```
#include <vcc.h>

unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y)
{
  unsigned w = x + y;
  return w;
}
```

Example 2

Suppose we don't believe our compiler's implementation of "+": let's write our own!

```
#include <vcc.h>

unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y)
{
  unsigned i = x;        // ORIGINAL CODE:
  unsigned j = y;        // unsigned w = x + y;
                         // return w;
  while (i > 0)
  {
    --i;
    ++j;
  }

  return j;
}
```

Example 2

VCC

Does this C program always work?

```c
#include <vcc.h>

unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y)
{
  unsigned i = x;
  unsigned j = y;

  while (i > 0)
  {
    --i;
    ++j;
  }

  return j;
}
```

Example 2

VCC

Does this C program always work?

```
1  #include <vcc.h>
2
3  unsigned add(unsigned x, unsigned y)
4    _(requires x + y <= UINT_MAX)
5    _(ensures \result == x + y)
6  {
7    unsigned i = x;
8    unsigned j = y;
9
10   while (i > 0)
11   {
12     --i;
13     ++j;
14   }
15
16   return j;
17 }
```

		Description	Line	Column
	1	++j might overflow.	13	5
	2	Post condition '\result == x + y' did not verify.	16	3
	3	(related information) Location of post condition.	5	13

Example 2

Microsoft Research

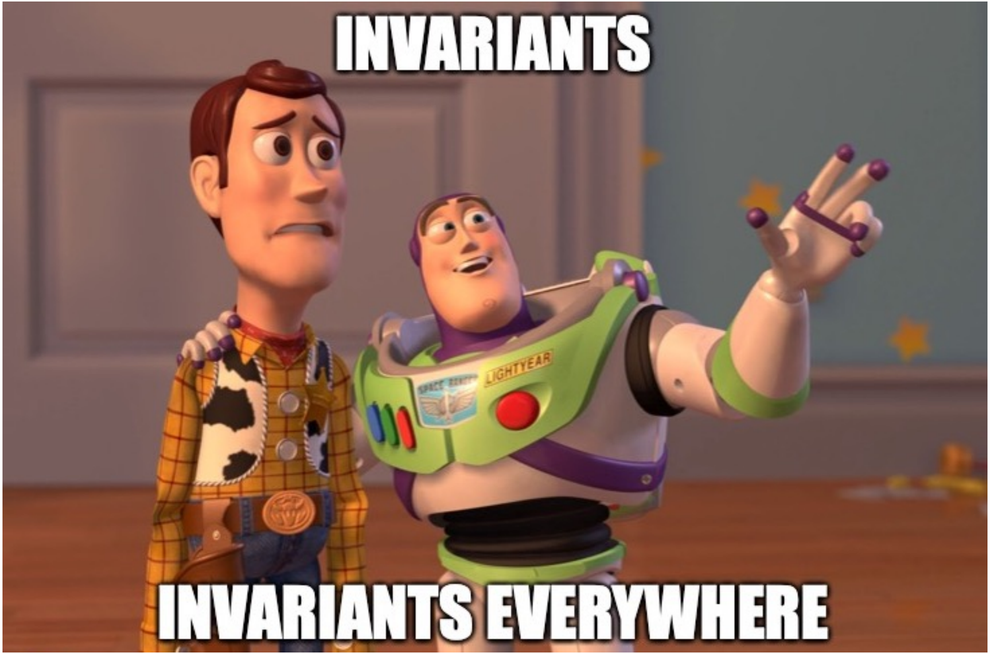
VCC

Does this C program always work?

```
 1 #include <vcc.h>
 2
 3 unsigned add(unsigned x, unsigned y)
 4   _(requires x + y <= UINT_MAX)
 5   _(ensures \result == x + y)
 6 {
 7   unsigned i = x;
 8   unsigned j = y;
 9
10   while (i > 0)
11   {
12     --i;
13     ++j;
14   }
15
16   return j;
17 }
```

		Description	Line	Column
✖	1	++j might overflow.	13	5
✖	2	Post condition '\result == x + y' did not verify.	16	3
✖	3	(related information) Location of post condition.	5	13

- doesn't verify, but the violation `++j might overflow.` is spurious. How to get rid of it?


INVARIANTS
INVARIANTS EVERYWHERE

Example 2

Fix #1:

```
unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y)
{
  unsigned i = x;        // ORIGINAL CODE:
  unsigned j = y;        // unsigned w = x + y;
                         // return w;
  while (i > 0)
    _(invariant i + j == x + y)     // <-- added invariant
  {
    --i;
    ++j;
  }

  return j;
}
```

Example 2

VCC

Does this C program always work?

```
#include <vcc.h>

unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y)
{
  unsigned i = x;
  unsigned j = y;

  while (i > 0)
    _(invariant i + j == x + y)
  {
    --i;
    ++j;
  }

  return j;
}
```

```
Verification of add succeeded. [0.78]
```

- verifies wrt the specification \o/

Example 2 — post mortem

What did we do?

1. We substituted implementation of a function with a different one
   - the contract is still the same
2. The new implementation cannot be verified as is
   - **unbounded loops** cannot be easily transformed into a static formula
3. We needed to provide a **loop invariant**: `_(invariant` I`)` where I is a formula s.t.
   - I holds every time the loop head is reached (before evaluating the loop test)

Example 2 — post mortem

```
while (C)
 _(invariant I)
{
  // Body
}
```

We can then substitute the loop by

```
_(assert I)
_(assume I && !C)
```

but we also need to check validity of the formula

$$(I \;\wedge\; \varphi_B) \to (I \;\wedge\; safe_B)$$

- φ_B is a formula representing the loop body
- $safe_B$ represents implicit safety conditions on the loop body

->

Example 3

```
unsigned lsearch(int elt, int *ar, unsigned sz)
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
{
  unsigned i;
  for (i = 0; i < sz; i = 1)
  {
    if (ar[i] == elt) return i;
  }

  return UINT_MAX;
}
```

Example 3

Microsoft Research

VCC

Does this C program always work?

```
1  unsigned lsearch(int elt, int *ar, unsigned sz)
3    _(ensures \result != UINT_MAX ==> ar[\result] == elt)
4    _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
5  {
6    unsigned i;
7    for (i = 0; i < sz; i = 1)
8    {
9      if (ar[i] == elt) return i;
10   }
11
12   return UINT_MAX;
13   }
```

		Description	Line	Column
⊗	1	Assertion 'ar[i] is thread local' did not verify.	9	9
⊗	2	Post condition '\forall unsigned i; i < sz && i < \result ==> ar[i] != elt)' did not verify.	9	23
⊗	3	(related information) Location of post condition.	4	13

Example 3

Fix #1:

```
unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \thread_local_array(ar, sz))         // <-- added precondition
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
{
  unsigned i;
  for (i = 0; i < sz; i = 1)
  {
    if (ar[i] == elt) return i;
  }

  return UINT_MAX;
}
```

Example 3

VCC Microsoft Research

Does this C program always work?

```
1  unsigned lsearch(int elt, int *ar, unsigned sz)
2    _(requires \thread_local_array(ar, sz))
3    _(ensures \result != UINT_MAX ==> ar[\result] == elt)
4    _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
5  {
6    unsigned i;
7    for (i = 0; i < sz; i = 1)
8    {
9      if (ar[i] == elt) return i;
10   }
11
12   return UINT_MAX;
13 }
```

		Description	Line	Column
✖	1	Post condition '\forall unsigned i; i < sz && i < \result ==> ar[i] != elt)' did not verify.	9	23
✖	2	(related information) Location of post condition.	4	13

Example 3

Microsoft Research

VCC

Does this C program always work?

```
1  unsigned lsearch(int elt, int *ar, unsigned sz)
2    _(requires \thread_local_array(ar, sz))
3    _(ensures \result != UINT_MAX ==> ar[\result] == elt)
4    _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
5  {
6    unsigned i;
7    for (i = 0; i < sz; i = 1)
8    {
9      if (ar[i] == elt) return i;
10   }
11
12   return UINT_MAX;
13 }
```

		Description	Line	Column
⊗	1	Post condition '\forall unsigned i; i < sz && i < \result ==> ar[i] != elt)' did not verify.	9	23
⊗	2	(related information) Location of post condition.	4	13

- still doesn't verify

Example 3

Fix #2: Let's provide a loop invariant!

```
unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \thread_local_array(ar, sz))
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
{
  unsigned i;
  for (i = 0; i < sz; i = 1)
    _(invariant \forall unsigned j; j < i ==> ar[j] != elt)  // <-- added invariant
  {
    if (ar[i] == elt) return i;
  }

  return UINT_MAX;
}
```

Example 3

VCC

Does this C program always work?

```
unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \thread_local_array(ar, sz))
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
{
  unsigned i;
  for (i = 0; i < sz; i = 1)
    _(invariant \forall unsigned j; j < i ==> ar[j] != elt)
  {
    if (ar[i] == elt) return i;
  }

  return UINT_MAX;
}
```

```
Verification of lsearch succeeded. [2.19]
```

Example 3

Microsoft Research

VCC

Does this C program always work?

```
unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \thread_local_array(ar, sz))
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
{
  unsigned i;
  for (i = 0; i < sz; i = 1)
    _(invariant \forall unsigned j; j < i ==> ar[j] != elt)
  {
    if (ar[i] == elt) return i;
  }

  return UINT_MAX;
}
```

```
Verification of lsearch succeeded. [2.19]
```

- Verifies! Great!!!! . . . or is it?

Example 3

Fix #3: provide a termination requirement

```
unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \thread_local_array(ar, sz))
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
  _(decreases 0)                  // <-- added termination requirement
{
  unsigned i;
  for (i = 0; i < sz; i = 1)
    _(invariant \forall unsigned j; j < i ==> ar[j] != elt)
  {
    if (ar[i] == elt) return i;
  }

  return UINT_MAX;
}
```

Example 3

Microsoft Research

VCC

Does this C program always work?

```
1  unsigned lsearch(int elt, int *ar, unsigned sz)
2    _(requires \thread_local_array(ar, sz))
3    _(ensures \result != UINT_MAX ==> ar[\result] == elt)
4    _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
5    _(decreases 0)
6  {
7    unsigned i;
8    for (i = 0; i < sz; i = 1)
9      _(invariant \forall unsigned j; j < i ==> ar[j] != elt)
10   {
11     if (ar[i] == elt) return i;
12   }
13
14   return UINT_MAX;
15 }
```

		Description	Line	Column
⊗	1	the loop fails to decrease termination measure.	8	3

- Ooops: `the loop fails to decrease termination measure.`

Example 3

Fix #4: fix the code

```
unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \thread_local_array(ar, sz))
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
  _(decreases 0)
{
  unsigned i;
  for (i = 0; i < sz; i += 1)      // <-- code fix
    _(invariant \forall unsigned j; j < i ==> ar[j] != elt)
  {
    if (ar[i] == elt) return i;
  }

  return UINT_MAX;
}
```

Example 3

VCC

Does this C program always work?

```
unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \thread_local_array(ar, sz))
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
  _(decreases 0)
{
  unsigned i;
  for (i = 0; i < sz; i += 1)
    _(invariant \forall unsigned j; j < i ==> ar[j] != elt)
  {
    if (ar[i] == elt) return i;
  }

  return UINT_MAX;
}
```

```
Verification of lsearch succeeded. [3.41]
```

- Verifies!

Example 3 — post mortem

What did we do?

- our annotations got more complex:

```
unsigned lsearch(int elt, int *ar, unsigned sz)
  _(requires \thread_local_array(ar, sz))
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
  _(decreases 0)
```

- `==>`, `<==`: implication, `<==>`: equivalence, `\forall`: $\forall$, `\exists`: $\exists$ — quantifiers (typed)
- `thread_local_array(ar, sz)`: `ar` points to (at least) `sz` items of the type of `*a`, which are "*owned*" by this thread
- `_(decreases 0)`: simply states that `lsearch` terminates
  - for more complex code, **termination measure** needs to be provided on loops
  - the measure should decrease in every iteration of the loop
  - for recursive procedures, termination measure should decrease in every call

Example 3 — post mortem

- **partial correctness**: every answer returned by a program is correct
- **total correctness**: above + the algorithm also **terminates**

Example 4

```
unsigned bsearch(int elt, int *ar, unsigned sz)
  _(requires \thread_local_array(ar, sz))
  _(ensures \result != UINT_MAX ==> ar[\result] == elt)
  _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
  _(decreases 0)
{
  if (sz == 0) return UINT_MAX;
  unsigned left = 0;
  unsigned right = sz - 1;

  while (left < right) {
    unsigned mid = (left + right) / 2;
    if (ar[mid] < elt) {
      left = mid + 1;
    } else if (ar[mid] > elt) {
      right = mid - 1;
    } else {
      return mid;
    }
  }

  return UINT_MAX;
}
```

Example 4

Microsoft Research

VCC

Does this C program always work?

```
 1 unsigned bsearch(int elt, int *ar, unsigned sz)
 2   _(requires \thread_local_array(ar, sz))
 3   _(ensures \result != UINT_MAX ==> ar[\result] == elt)
 4   _(ensures \forall unsigned i; i < sz && i < \result ==> ar[i] != elt)
 5   _(decreases 0)
 6 {
 7   if (sz == 0) return UINT_MAX;
 8   unsigned left = 0;
 9   unsigned right = sz - 1;
10
11   while (left < right) {
12     unsigned mid = (left + right) / 2;
13     if (ar[mid] < elt) {
14       left = mid + 1;
15     } else if (ar[mid] > elt) {
16       right = mid - 1;
17     } else {
18       return mid;
19     }
20   }
21   return UINT_MAX;
22 }
```

		Description	Line	Column
	1	left + right might overflow.	12	21
	2	Assertion 'ar[mid] is thread local' did not verify.	13	9
	3	mid − 1 might overflow.	16	15
	4	the loop fails to decrease termination measure.	11	3
	5	Post condition '\forall unsigned i; i < sz && i < \result ==> ar[i] != elt)' did not verify.	18	7
	6	(related information) Location of post condition.	4	13

Example 5

```c
unsigned add(unsigned x, unsigned y)
  _(ensures \result == x + y);

unsigned super_add(unsigned x, unsigned y, unsigned z)
  _(ensures \result == x + y + z)
{
  unsigned w = add(x, y);
  w = add(w, z);
  return w;
}
```

Example 5

VCC Microsoft Research

Does this C program always work?

```
unsigned add(unsigned x, unsigned y)
  _(ensures \result == x + y);

unsigned super_add(unsigned x, unsigned y, unsigned z)
  _(ensures \result == x + y + z)
{
  unsigned w = add(x, y);
  w = add(w, z);
  return w;
}
```

```
Verification of super_add succeeded. [2.88]
```

- Verifies!

Example 5

How about when we add an implementation of add?

```
unsigned add(unsigned x, unsigned y)
  _(ensures \result == x + y);

unsigned super_add(unsigned x, unsigned y, unsigned z)
  _(ensures \result == x + y + z)
{
  unsigned w = add(x, y);
  w = add(w, z);
  return w;
}

unsigned add(unsigned x, unsigned y)    // <-- added implementation
{
  return x + y;
}
```

Example 5

Microsoft Research

VCC

Does this C program always work?

```
1  unsigned add(unsigned x, unsigned y)
2    _(ensures \result == x + y);
3
4  unsigned super_add(unsigned x, unsigned y, unsigned z)
5    _(ensures \result == x + y + z)
6  {
7    unsigned w = add(x, y);
8    w = add(w, z);
9    return w;
10 }
11
12 unsigned add(unsigned x, unsigned y)
13 {
14   return x + y;
15 }
```

		Description	Line	Column
⊗	1	x + y might overflow.	14	10

Example 5

VCC

Does this C program always work?

```
1  unsigned add(unsigned x, unsigned y)
2    _(ensures \result == x + y);
3
4  unsigned super_add(unsigned x, unsigned y, unsigned z)
5    _(ensures \result == x + y + z)
6  {
7    unsigned w = add(x, y);
8    w = add(w, z);
9    return w;
10 }
11
12 unsigned add(unsigned x, unsigned y)
13 {
14   return x + y;
15 }
```

		Description	Line	Column
	1	x + y might overflow.	14	10

- Ouch!

Example 5

```
unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)        // <-- added precondition
  _(ensures \result == x + y);

unsigned super_add(unsigned x, unsigned y, unsigned z)
  _(ensures \result == x + y + z)
{
  unsigned w = add(x, y);
  w = add(w, z);
  return w;
}

unsigned add(unsigned x, unsigned y)
{
  return x + y;
}
```

Example 5

Microsoft Research

VCC

Does this C program always work?

```
1  unsigned add(unsigned x, unsigned y)
2    _(requires x + y <= UINT_MAX)
3    _(ensures \result == x + y);
4
5  unsigned super_add(unsigned x, unsigned y, unsigned z)
6    _(ensures \result == x + y + z)
7  {
8    unsigned w = add(x, y);
9    w = add(w, z);
10   return w;
11 }
12
13 unsigned add(unsigned x, unsigned y)
14 {
15   return x + y;
16 }
```

		Description	Line	Column
⊗	1	Call 'add(x, y)' did not verify.	8	16
⊗	2	(related information) Precondition: 'x + y <= 0xffffffff'.	2	14
⊗	3	Call 'add(w, z)' did not verify.	9	7
⊗	4	(related information) Precondition: 'x + y <= 0xffffffff'.	2	14

Example 5

Microsoft Research

VCC

Does this C program always work?

```
 1 unsigned add(unsigned x, unsigned y)
 2   _(requires x + y <= UINT_MAX)
 3   _(ensures \result == x + y);
 4
 5 unsigned super_add(unsigned x, unsigned y, unsigned z)
 6   _(ensures \result == x + y + z)
 7 {
 8   unsigned w = add(x, y);
 9   w = add(w, z);
10   return w;
11 }
12
13 unsigned add(unsigned x, unsigned y)
14 {
15   return x + y;
16 }
```

	Description	Line	Column
1	Call 'add(x, y)' did not verify.	8	16
2	(related information) Precondition: 'x + y <= 0xffffffff'.	2	14
3	Call 'add(w, z)' did not verify.	9	7
4	(related information) Precondition: 'x + y <= 0xffffffff'.	2	14

- Not enough. . .

Example 5

```
unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y);

unsigned super_add(unsigned x, unsigned y, unsigned z)
  _(requires x + y + z <= UINT_MAX)      // <-- added precondition
  _(ensures \result == x + y + z)
{
  unsigned w = add(x, y);
  w = add(w, z);
  return w;
}

unsigned add(unsigned x, unsigned y)
{
  return x + y;
}
```

Example 5

Microsoft Research

VCC

Does this C program always work?

```
unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y);

unsigned super_add(unsigned x, unsigned y, unsigned z)
  _(requires x + y + z <= UINT_MAX)
  _(ensures \result == x + y + z)
{
  unsigned w = add(x, y);
  w = add(w, z);
  return w;
}

unsigned add(unsigned x, unsigned y)
{
  return x + y;
}
```

```
Verification of add succeeded. [1.81]
Verification of super_add succeeded. [0.00]
```

Example 5

VCC

Does this C program always work?

```
unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y);

unsigned super_add(unsigned x, unsigned y, unsigned z)
  _(requires x + y + z <= UINT_MAX)
  _(ensures \result == x + y + z)
{
  unsigned w = add(x, y);
  w = add(w, z);
  return w;
}

unsigned add(unsigned x, unsigned y)
{
  return x + y;
}
```

```
Verification of add succeeded. [1.81]
Verification of super_add succeeded. [0.00]
```

- Verifies!

Example 5 — post mortem

What happened?

- super_add was using add in its body
- during verification of super_add, the call to add was substituted by its contract:

  ```
  _(assert add_requires)    // precondition
  _(assume add_ensures)     // postcondition
  ```
- validity of all asserts and super_add's postcondition needed to be checked:

1. for add(x, y):

$$(x + y + z \leq \texttt{UINT\_MAX}) \rightarrow (x + y \leq \texttt{UINT\_MAX})$$

2. for add(w, z):

$$(x+y+z \leq \texttt{UINT\_MAX} \wedge w_1 = x+y) \rightarrow (w_1+z \leq \texttt{UINT\_MAX})$$

3. super_add's postcondition:

$$(x+y+z \leq \texttt{UINT\_MAX} \wedge w_1 = x+y \wedge w_2 = w_1+z) \rightarrow (w_2 = x+y+z)$$

```
unsigned add(unsigned x, unsigned y)
  _(requires x + y <= UINT_MAX)
  _(ensures \result == x + y);

unsigned super_add(unsigned x, unsigned y,
  _(requires x + y + z <= UINT_MAX)
  _(ensures \result == x + y + z)
{
  unsigned w = add(x, y);
  w = add(w, z);
  return w;
}
```

Example 6

```
void swap(int* x, int* y)
  _(ensures *x == \old(*y) && *y == \old(*x))
{
  int z = *x;
  *x = *y;
  *y = z;
}
```

Example 6

VCC

Does this C program always work?

```
1 void swap(int* x, int* y)
2   _(ensures *x == \old(*y) && *y == \old(*x))
3 {
4   int z = *x;
5   *x = *y;
6   *y = z;
7 }
```

		Description	Line	Column
	1	Assertion 'x is thread local' did not verify.	4	12
	2	Assertion 'x is writable' did not verify.	5	4
	3	Assertion 'y is thread local' did not verify.	5	9
	4	Assertion 'y is writable' did not verify.	6	4

Example 6

Microsoft Research

VCC

Does this C program always work?

```
1 void swap(int* x, int* y)
2   _(ensures *x == \old(*y) && *y == \old(*x))
3 {
4   int z = *x;
5   *x = *y;
6   *y = z;
7 }
```

		Description	Line	Column
	1	Assertion 'x is thread local' did not verify.	4	12
	2	Assertion 'x is writable' did not verify.	5	4
	3	Assertion 'y is thread local' did not verify.	5	9
	4	Assertion 'y is writable' did not verify.	6	4

- side effect

Example 6

```
void swap(int* x, int* y)
  _(writes x)
  _(writes y)
  _(ensures *x == \old(*y) && *y == \old(*x))
{
  int z = *x;
  *x = *y;
  *y = z;
}
```

Example 6

Microsoft Research

VCC

Does this C program always work?

```
void swap(int* x, int* y)
  _(writes x)
  _(writes y)
  _(ensures *x == \old(*y) && *y == \old(*x))
{
  int z = *x;
  *x = *y;
  *y = z;
}
```

```
Verification of swap succeeded. [2.64]
```

Example 6

Microsoft Research

VCC

Does this C program always work?

```
void swap(int* x, int* y)
  _(writes x)
  _(writes y)
  _(ensures *x == \old(*y) && *y == \old(*x))
{
  int z = *x;
  *x = *y;
  *y = z;
}
```

```
Verification of swap succeeded. [2.64]
```

- `_(writes x)` talks about a side-effect

Example 7

```
#define RADIX ((unsigned)(-1) + ((\natural)1))
#define LUINT_MAX ((unsigned)(-1) + (unsigned)(-1) * ((unsigned)(-1) + ((\natural)1)))
typedef struct LongUint {
  _(ghost \natural val)
  unsigned low, high;
  _(invariant val == low + high * RADIX) // coupling invariant
} LongUint;

void luint_inc(LongUint* x)
  _(maintains \wrapped(x))
  _(writes x)
  _(requires x->val + 1 < LUINT_MAX)
  _(ensures x->val == \old(x->val) + 1)
{
  _(unwrapping x) {
    if (x->low == UINT_MAX) {
      ++(x->high);
      x->low = 0;
    } else {
      ++(x->low);
    }
    _(ghost x->val = x->val + 1)
  }
}
```

Example 7

Microsoft Research

VCC

Does this C program always work?

```
#define RADIX ((unsigned)(-1) + ((\natural)1))
#define LUINT_MAX ((unsigned)(-1) + (unsigned)(-1) * ((unsigned)(-1) + ((\natural)1)))
typedef struct LongUint {
  _(ghost \natural val)
  unsigned low, high;
  _(invariant val == low + high * RADIX) // coupling invariant
} LongUint;

void luint_inc(LongUint* x)
  _(maintains \wrapped(x))
  _(writes x)
  _(requires x->val + 1 < LUINT_MAX)
  _(ensures x->val == \old(x->val) + 1)
{
  _(unwrapping x) {
    if (x->low == UINT_MAX) {
      ++(x->high);
      x->low = 0;
    } else {
      ++(x->low);
    }
    _(ghost x->val = x->val + 1)
```

```
Verification of LongUint#adm succeeded. [2.39]
Verification of luint_inc succeeded. [0.03]
```

Example 7 — post mortem

What did we do?

- we needed to provide a data structure invariant via `_(invariant Inv)`
  - it describes what need to hold about the data structure in a *consistent state*
  - the invariant talks about a ghost variable
    - helps with verification but is not part of the compiled program
    - can have an *"ideal"* type (e.g., `\natural`, `\integer`, ...)
    - or can also be an inductive (functional-style) data type, e.g.
      `_(datatype List { case nil(); case cons(int v, List l); })`
  - we needed to use `_(unwrapping x) { ... }` for the block of code where the invariant is temporarily broken

Further issues

- concurrency (atomic actions, shared state)
- hardware
- assembly code (need to model instructions using function contract)
- talking about memory (possible aliasings)

Other Tools

- **Dafny**: a full programming language with support for specifications
- **Why3**: a programming language (WhyML) + specifications
- **Frama-C** (Jessie plug-in): deductive verification of C + ACSL annotations
- **KeY**: Java + JML annotations
- **Prusti**: Rust
- **IVy**: specification and implementation of protocols
- **Ada**, **Eiffel**, . . . : programming languages with in-built support for specifications

Used materials from

- Ernie Cohen, Amazon (former Microsoft)
- Işıl Dillig, University of Texas, Austin