IMPLEMENTING LOGIC BY SEMANTICS

The RISCAL Approach to Automating Program Reasoning over Finite Domains

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Formal Methods in Computer Science

- Specification:
  - Describe precisely a computational problem to be solved.
    - Precondition: what can be assumed about the inputs.
    - Postcondition: what has to be established for the outputs.
  - Formal specification: conditions are precisely formulated in the language of formal (first order predicate) logic.

- Verification:
  - Show that a program correctly implements the specification.
    - For all inputs that satisfy the precondition, the program must produce outputs that satisfy the postcondition.
  - Formal verification: given a formal program semantics, correctness can be established with mathematical rigor.

Formal logic is the fundament of precise program reasoning.
1. Formal Specification

2. Proof-Based Verification

3. Semantics-Based Checking

4. Conclusions
A Program Specification

The specification of a “conditional swap” problem.

- **Input:** an integer array $a$ of length $N$ and indices $i, j$ in $a$:
  
  requires $0 \leq i \land i < N \land 0 \leq j \land j < N$;

- **Output:** an array $result$ that is identical to $a$ except that the elements at $i$ and $j$ are in ascending order:

  ensures $a[i] \leq a[j] \Rightarrow result[i] = a[i] \land result[j] = a[j]$;
  
  ensures $a[i] > a[j] \Rightarrow result[i] = a[j] \land result[j] = a[i]$;
  
  ensures $\forall k:\text{index with } 0 \leq k \land k < N. $
  
  $k \neq i \land k \neq j \Rightarrow result[k] = a[k]$;
Conditional Swap

An implementation of the “conditional swap” problem.

val N: N; val M: N;
type index = Z[-N,N]; type elem = Z[-M,M]; type array = Array[N, elem];

proc cswap(a:array, i:index, j:index): array
    requires 0 ≤ i ∧ i < N ∧ 0 ≤ j ∧ j < N;
    ensures a[i] ≤ a[j] ⇒ result[i] = a[i] ∧ result[j] = a[j];
    ensures a[i] > a[j] ⇒ result[i] = a[j] ∧ result[j] = a[i];
    ensures ∀k:index with 0 ≤ k ∧ k < N. k ≠ i ∧ k ≠ j ⇒ result[k] = a[k];
{
    var b:array = a;
    if b[i] > b[j] then
        { var x:elem := b[i];
            b[i] := b[j];
            b[j] := x;
        }
    return b;
}
Formal Verification

How to rigorously demonstrate the correctness of the program with respect to its specification?

- **Generate verification conditions.**
  - Logic formulas whose validity implies the correctness.
  - Can be automatically generated by a formal calculus.
  - Requires the annotation of the program with sufficiently strong extra-information (loop invariants/termination terms).

- **Prove the conditions.**
  - Possibly performed/supported by an automated reasoner/interactive proof assistant.
  - First order logic is not decidable, thus fully automatic proofs can generally not be expected.
  - In general, (also) human effort is required.

The traditional (and only fully general) approach.
1. Formal Specification

2. Proof-Based Verification

3. Semantics-Based Checking

4. Conclusions
Verification Condition Generation

E.g., Dijkstra’s weakest precondition calculus.

- Verification condition: \( \text{pre} \Rightarrow \text{wp}(\text{prog}, \text{post}) \)

\[
\begin{align*}
\text{wp}(x := e, \text{post}) & := \text{post}[e/x] \\
\text{wp}(\text{skip}, \text{post}) & := \text{post} \\
\text{wp}(c_1; c_2, \text{post}) & := \text{wp}(c_1, \text{wp}(c_2, \text{post})) \\
\text{wp}(\text{if } b \text{ then } c_1 \text{ else } c_2, \text{post}) & := (b \Rightarrow \text{wp}(c_1, \text{post})) \\
& \quad \land (\neg b \Rightarrow \text{wp}(c_2, \text{post})) \\
& \quad \ldots
\end{align*}
\]

Fully automatic; without loops, no extra information is required.
Verification of the Program

```plaintext
var b:array = a;
if b[i] > b[j] then {
    var x:elem := b[i];
    b[i] := b[j];
    b[j] := x;
}
return b;

wp(cswap, post) :=
    (b[i] \neq b[j] \Rightarrow post[b/result][a/b]) \land
    (b[i] > b[j] \Rightarrow post[b/result][b[j] \mapsto x]/b[b[i] \mapsto b[j]]/b[b[i] \mapsto x][a/b])
```

We have to prove \( pre \Rightarrow wp(cswap, post); \)
for this we use some automation support.
The RISC ProgramExplorer

- An integrated program reasoning environment.
  - Programming language MiniJava.
  - Theory/specification language in the style of PVS/CVC.
  - Semi-automatic proving assistant RISC ProofNavigator.

- Semantics view.
  - Semantics of a method body.
  - Pre/post-condition reasoning.

- Analyze view (verification tasks).
  - Type checking conditions.
  - Statement preconditions.
  - Loop invariants.
  - Method frame preservation.
  - Method termination.
  - Method postcondition.

- Verify view.
  - Proof construction and management.
The RISC ProgramExplorer

https://www.risc.jku.at/research/formal/software/ProgramExplorer
BubbleSort

```plaintext
pred sorted(a:array, i:index)
    requires 0 ≤ i ∧ i < N;
⇔ ∀k:index. i ≤ k ∧ k < N-1 ⇒ a[k] ≤ a[k+1];

proc bubbleSort(a:array): array
    ensures sorted(result,0);
{
    var b:array = a;
    for var i:index := 0; i < N-1; i := i+1 do
        for var j:index := 0; j < N-i-1; j := j+1 do
            b := cswap(b,j,j+1);
    return b;
}
```

How to verify the correctness of this program?
Verification Condition Generation for Loop

- Weakest precondition of a loop annotated with an invariant:

\[
wp(\text{while } b \text{ do } \text{inv}(x, x') \ c^x \text{, post}) := \text{inv}(x, x) \\
\qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{\small \& (} \forall x'. \text{inv}(x, x') \Rightarrow \text{post}[x'/x]\text{)}
\]

- \(c^x\): a command that only changes variable \(x\).
- \(\text{inv}(x, x')\): a formula that relates a variable’s prestate value \(x\) to its poststate value \(x'\).
- Also have to prove that the loop body maintains the invariant:

\[
\text{inv}(x, x') \land b[x'/x] \Rightarrow wp(c^x, \text{inv}(x, x'))
\]

Only partial correctness: for termination, also a “termination measure” is required.
BubbleSort

pred sorted(a:array, i:index) requires 0 ≤ i ∧ i < N;
⇔ ∀k:index. i ≤ k ∧ k < N-1 ⇒ a[k] ≤ a[k+1];
pred lesseq(a:array, i:index) requires 0 ≤ i ∧ i < N;
⇔ ∀k:index. 0 ≤ k ∧ k < i ⇒ a[k] ≤ a[i];

proc bubbleSort(a:array): array
ensures sorted(result,0);
{
    var b:array = a;
    for var i:index := 0; i < N-1; i := i+1 do
        invariant 0 ≤ i ∧ (N > 0 ⇒ i < N);
        invariant sorted(b,N-1-i) ∧ (i > 0 ⇒ lesseq(b,N-i));
    {
        for var j:index := 0; j < N-i-1; j := j+1 do
            invariant 0 ≤ i ∧ (N > 0 ⇒ i < N) ∧ 0 ≤ j ∧ j < N-i;
            invariant sorted(b,N-1-i) ∧ (i > 0 ⇒ lesseq(b,N-i));
            invariant lesseq(b,j);
            b := cswap(b,j,j+1);
        }
    return b;
}
Verification of Loop-Based Programs

- Many potential sources of errors.
  - Errors in the program.
  - Errors in the specification.
  - Errors in the loop invariants.
  - Failure to find the adequate proof strategy.

- If a proof fails, it is hard to determine the reason.
  - Most time in proof-based verification is wasted by attempting to prove invalid verification conditions!

It would be good to have an easier way to find errors.
1. Formal Specification

2. Proof-Based Verification

3. Semantics-Based Checking

4. Conclusions
Formal Verification

Is there an alternative to proof-based verification?

- **Model Checking**
  - Check whether program runs satisfy the specification.
  - Runtime assertion checking: user selects certain runs.
  - Model checking: automatic consideration of all possible runs.

- **Problem**: only complete under restrictive assumptions.
  - Decidable conditions to be checked.
  - Original model checking: state space is finite (domains of all program variables are finite).
  - Abstraction-based model checking: program can be checked in a finite abstraction of the state space.
  - Bounded model checking: program runs with a bounded number of loop iterations.

Usually applied only to automatic detection of runtime errors.
The RISC Algorithm Language (RISCAL)

- Formal theory and algorithm specification language.
  - Static type system with parameterized types $T[n]$.
  - Functions (implicit, explicit, recursive).
  - Predicates (explicit, recursive).
  - Theorems (predicates claimed to be always true).
  - Procedures (functions defined by commands).
  - Pre-/post-conditions, loop invariants, termination measures.

- Non-deterministic semantics.
  - Implicit function definitions and non-deterministic choices in formulas and programs.

- Semantics-based implementation of programs/formulas.
  - All phrases are translated to their denotational semantics.
  - Model checker executes semantics for all possible inputs.
  - Parallel implementation allows to check large state spaces.

A semantics-based approach to checking and verification.
Denotational Semantics of Programs

\[ . \] : Command \times State \rightarrow State

\[
\begin{align*}
\llbracket x := e \rrbracket(s) &:= s[x \mapsto \llbracket e \rrbracket(s)] \\
\llbracket \text{skip} \rrbracket(s) &:= s \\
\llbracket c_1; c_2 \rrbracket(s) &:= \llbracket c_2 \rrbracket(\llbracket c_1 \rrbracket(s)) \\
\llbracket \text{if } b \text{ then } c_1 \text{ else } c_2 \rrbracket(s) &:= \text{if } \llbracket b \rrbracket(s) = \text{true} \text{ then } \llbracket c_1 \rrbracket(s) \text{ else } \llbracket c_2 \rrbracket(s) \\
\llbracket \text{while } b \text{ do } c \rrbracket(s) &:= w(b, c, s) \text{ where} \\
& \quad w(b, c, s) := \text{if } \llbracket b \rrbracket(s) = \text{false} \\
& \quad \quad \text{then } s \\
& \quad \quad \text{else } w(b, c, \llbracket c \rrbracket(s))
\end{align*}
\]

Executable by a direct implementation.
Denotational Semantics of Formulas

\[
[ . ] : \text{Formula} \times \text{State} \rightarrow \{ \text{true, false} \}
\]

\[
[p(t_1, \ldots, t_n)](s) := [p]([t_1](s), \ldots, [t_n](s))
\]

\[
[-F](s) := \begin{cases} 
\text{true} & \text{if } [F](s) = \text{false} \\
\text{false} & \text{else} 
\end{cases}
\]

\[
[F_1 \land F_2](s) := \begin{cases} 
\text{true} & \text{if } [F_1](s) = [F_2](s) = \text{true} \\
\text{false} & \text{else} 
\end{cases}
\]

\[
[\forall x : T. \, F](s) := \begin{cases} 
\text{true} & \text{if } \forall a \in [T]. \, [F](s[x \mapsto a]) = \text{true} \\
\text{false} & \text{else} 
\end{cases}
\]

Executable by a direct implementation (provided that the semantics \([T]\) of every type \(T\) is finite).
Relationship Semantics to Logic

The weakest precondition calculus is sound with respect to the denotational semantics of programs and formulas:

\[(\forall s \in \text{State}. [\text{pre} \Rightarrow \text{wp}(c, \text{post})](s) = \text{true}) \iff (\forall s \in \text{State}. [\text{pre}](s) \Rightarrow [\text{post}](\text{\{c\}}(s)))\]

- If the derived verification condition is valid, then every execution of the program that starts in a state that satisfies the precondition yields a result state that satisfies the postcondition . . .
- . . . and vice versa.

Proof-based verification and semantics-based checking yield the same result.
Semantics-based Implementation

\[ \text{ComSem} := \text{Single} + \text{Multiple} \]

\[ \text{Single} := \text{Command} \rightarrow (\text{Context} \rightarrow \text{Context}) \]

\[ \text{Multiple} := \text{Command} \rightarrow (\text{Context} \rightarrow \text{Seq}(\text{Context})) \]

\[ \text{Seq}(T) := \text{Unit} \rightarrow (\text{Null} + \text{Next}(T, \text{Seq}(T))) \]

```java
public static interface ComSem {
    public interface Single extends ComSem, Function<Context, Context> {}
    public interface Multiple extends ComSem, Function<Context, Seq<Context>> {}
}
public interface Seq<T> extends Supplier<Seq.Next<T>> {
    // public Seq.Next<T> get();
    public final static class Next<T> {
        public final T head; public final Seq<T> tail;
        ...
    }
}
```

Non-deterministic semantics based on lazy sequences.
The RISCAL Software

http://www.risc.jku.at/research/formal/software/RISCAL
Verifying via Checking Finite Instances

A step-wise approach to verification.

  - Parameter $n \in \mathbb{N}$ bounds size of every variable type $T[n]$.
    - May have different bounds for different types.
  - Value of parameter is arbitrarily large (not fixed in program).
- Program operates over a finite domain.
  - Can be executed for all inputs of the domain.
- Specification and annotations are decidable.
  - By evaluating their semantics over the domain.

Structure of program/specification can be used for the validation of correctness before its actual verification.
Verification via Checking Finite Instances


- **Testing**
  - Run $P[c]$ with some input $i \in D[c]$ and watch output.
  - Validate informal correctness of program for some inputs.

- **Runtime assertion checking**
  - Additionally evaluate $A[c]$ and $S[c]$ and report violations.
  - Validate formal correctness of program for some inputs.

- **Model checking**
  - Runtime assertion checking for every input $i \in D[c]$.
  - Validate formal correctness for all inputs in $D[c]$.
    - May detect that $A[c]$ respectively $S[c]$ is too strong.

- **Generate verification condition** $VC[c] = vc(P^A[c], S[c])$.
  - Decidable by evaluation.
    - May detect that $A[c]$ is too weak.


- **Prove** $\forall n \in \mathbb{N}. \ VC[n]$  
  - Computer-assisted reasoning again. 
  - But now, by previous validation, high chance of being valid.

1. Formal Specification

2. Proof-Based Verification

3. Semantics-Based Checking

4. Conclusions
Conclusions

  - Since then used in formal method courses
  - Contents developed by students (computer science, discrete mathematics, number theory, computer algebra, ...).
  - Very positive initial feedback due to “full automation”.
  - Visualization component recently added.
  - Current work on generation of verification conditions.
  - Further development within LOGTECHEDU and SEMTECH.

https://www.risc.jku.at/research/formal/software/RISCAL