Program semantics, analysis and verification

December 8, 2005

Program semantics

[Nielsen & Nielsen, Semantics with Applications, Wiley 1992, 1999]
Semantics = describing the meaning (behavior of programs).
formally: express the meaning in terms of a mathematical model

operational semantics: describes *how* the computation is executed (effects of a statement on the program *state*)
– *natural* (big-step) operational semantics: *overall* execution
– *structural* (small-step) operational semantics: effect composed from individual statements

denotational semantics: describes *effect* of program construct typically as function; not *how* execution is done
– direct style: meaning of standalone construct
– continuation style: meaning if followed by a given *continuation*

axiomatic semantics: assertions about effect of executing the program (can focus on some properties of interest)
– *partial correctness*: what is true *if* program terminates
– *total correctness*: also expresses *when* program terminates

History of program verification

– first practical successes of formal verification were for hardware
– but started by formalizing programming language semantics

Robert W. Floyd. Assigning Meanings to Programs (1967)
"*an adequate basis for formal definitions of the meanings of programs [...] in such a way that a rigorous standard is established for proofs*"
"*If the initial values of the program variables satisfy the relation* R_1*, the final values on completion will satisfy the relation* R_2*.*"
– method: annotating a program (or flow graph) with assertions
– introduces the notion of *verification condition*: a formula $V_c(P;Q)$ such that if P is true before executing c, then Q is true upon exit and *strongest verifiable consequent* for a program + initial condition
– very general approach: assertions expressed in first order logic
– develops *general* rules for combining verification conditions and *specific* rules for different instruction types
– explicitly introduces *invariants* for reasoning about loops
– handles *termination* using a positive decreasing measure

The works of Hoare

C.A.R. Hoare. An Axiomatic Basis for Computer Programming (1969)
– like Floyd, handles preconditions and postconditions for executing an instruction, but the notion of *Hoare triple* better displays the relation between the statement and the two assertions
– works with source programs, not flow graphs

– Notation: *partial correctness* $\{P\}\ S\ \{Q\}$
If S is executed in a state that satisfies P and it terminates, the resulting state satisfies Q.

– Later: similar reasoning for *total correctness* $[P]\ S\ [Q]$
If S is executed in a state that satisfies P, then it terminates and the resulting state satisfies Q.

Rigorous application: C.A.R. Hoare. Proof of a Program: FIND (1971)

Hoare's axioms (rules)

– defined for each statement type individually
by combining them, we can reason about entire programs

Assignment: $$\frac{}{\{Q[x/E]\}\ x := E\ \{Q\}}$$
where $Q[x/E]$ is the substitution of x with E in Q

Example: $\{x = y - 2\}\ x := x + 2\ \{x = y\}$ (in $x = y$, substitute x with the assigned expression, $x + 2$ and obtain $x + 2 = y$, thus $x = y - 2$)
Writing the rule "backward" (P as function of Q) simplifies it.

Sequencing: $$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

Decision: $$\frac{\{P \wedge E\}\ S_1\ \{Q\} \qquad \{P \wedge \neg E\}\ S_2\ \{Q\}}{\{P\}\ \text{if } E \text{ then } S_1 \text{ else } S_2\ \{Q\}}$$

Hoare rules

While loop: is key in reasoning about programs
– must find an *invariant* I = a property which stays true before/after each loop iteration
– if loop is entered (E), the invariant is maintained after loop body S
– if loop not entered ($\neg E$), imvariant implies postcondition Q

$$\frac{\{I \wedge E\}\ S\ \{I\} \qquad I \wedge \neg E \Rightarrow Q}{\{I\}\ \text{while } E \text{ do } S\ \{Q\}}$$

```
while (lo < hi) {   /* binary search; I: lo <= n && n <= hi */
  m = (lo + hi) / 2;
  if (n > m)        /* both cases maintain lo<=n && n<=hi */
    lo = m+1;       /* n > m => n >= m+1 => n >= lo */
  else hi = m;      /* !(n < m) => n <= m => n <= hi */
}                   /* I stays true */
n = lo;         /* lo<=n && n<=hi && !(lo<hi) => lo==n && n==hi */
```

Hoare rules with pointers/aliasing

Consider $\{P\}\ *x = 2\ \{v + *x = 4\}$
What is the precondition P ? Correct answer: $v = 2 \lor x = \&v$.
But using the simple rule $(v + *x = 4)[*x/2]$ misses second case.

$\Rightarrow$ we must model memory. m = memory, a = address, d = data.
Consider functions $rd(m, a)$ return d and $wr(m, a, d)$ return m'
We have the rule: $rd(wr(m, a_1, d), a_2) = \begin{cases} d & \text{if } a_2 = a_1 \\ rd(m, a_2) & \text{if } a_2 \neq a_1 \end{cases}$
We must deduce a property of memory m from the relation:
$rd(wr(m, x, 2), \&v) + rd(wr(m, x, 2), x) = 4$
$rd(wr(m, x, 2), \&v) + 2 = 4$
$rd(wr(m, x, 2), \&v) = 2$
$x = \&v \land 2 = 2 \lor x \neq \&v \land rd(m, \&v) = 2$
$x = \&v \lor v = 2$

Dijkstra's *weakest precondition* operator

E.W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs (1975)
– for a given statement S and postcondition Q there can be several preconditions P such that $\{P\}\ S\ \{Q\}$ or $[P]\ S\ [Q]$.
– Dijkstra calculates a *necessary and sufficient* precondition $wp(S, Q)$ for successful termination of S with postcondition Q.
– necessary (*weakest*): if $[P]\ S\ [Q]$ then $P \Rightarrow\ wp(S, Q)$
– wp is a *predicate transformer* (transforms post- into precondition)
– allows the definition of a *calculus* with such transformers

Dijkstra's preconditions (cont.)

Assignment: $wp(x := E, Q) = Q[x/E]$ (see Hoare's rule)
Sequencing: $wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$
Conditional:
$wp(\text{if } E \text{ then } S_1 \text{ else } S_2, Q) = (E \Rightarrow wp(S_1, Q)) \land (\neg E \Rightarrow wp(S_2, Q))$

For iteration, we need a recursive computation
Define wp_k, assuming loop terminates in at most k iterations:
$wp_0(\text{while } E \text{ do } S, Q) = \neg E \Rightarrow Q$ (loop not entered)
$wp_{k+1}(\text{while } E \text{ do } S, Q)) = (E \Rightarrow wp(S, wp_k(\text{while } E \text{ do } S, Q))) \land (\neg E \Rightarrow Q)$
($\leq k + 1$ iterations $\Leftrightarrow$ 1 iteration followed by $\leq k$, or 0 iterations;
equivalent with decomposing the first `while` iteration into an `if`)
$\Rightarrow$ can be written as a fixpoint formula

Finding loop invariants

We know P before the loop, we wish to find Q after execution.
How do we establish an invariant I of the loop in order to prove Q?

I must satisfy the following conditions:
– $P \Rightarrow I$ (I sufficiently weak to hold initially)
– $\{I \land E\}\ S\ \{I\}$ (I is an invariant)
– $I \land \neg E \Rightarrow Q$ (I sufficiently strong to be useful)

Determining loop invariants is difficult:
Trivial example:
$\{x \leq 0\}$ while $x < 9$ do $x \leftarrow x + 1$ $\{x < 10\}$
$x < 10$ is an invariant that can be successfully established
(also $x < n$ with $n \geq 10$, but not as useful)

Usually: iterative calculation (fixpoint); sometimes needs invariant strengthening

Predicate abstraction

Using Floyd-Hoare-style reasoning, we can express properties as *predicates* over the state variables of the program
– same as e.g., atomic prepositions were defined in Spin
– sample predicates: $x > 0$, $lock = 1$, $x + 1 < y$

Predicate abstraction [Graf & Saïdi '97]: method for constructing an *abstract state space* depending *only* on the values of given predicates.

We seek: well chosen predicates so the specification can be verified over abstract state space, without exploring concrete state space.

Operations needed to explore the *abstract* state space of the program:
– *concretisation*: calculating the concrete states (in the initial model) represented by the predicates in the abstract state
– computing successors / predecessors of these states
(using the *concrete* semantics of program statements
– *abstration* of concrete states, expressed using predicates
$\Rightarrow$ We need a (possibly approximate) method for backward/forward exploration in the abstract state space.

Exploring the abstract state space

General framework:
– *symbolic* approach, with *state sets* represented as formulas
$post(r, t) = \{s' \mid \exists s \in r \,.\, s \xrightarrow{t} s'\}$: *successor* of *region* (state set) r
– we seek the abstract operator $post^a(r, t) = \alpha(post^c(\gamma(r), t))$
– in general, this computation is infeasible/expensive in practice
(particularly the abstractisation operation α)
$\Rightarrow$ abstractions with different kinds of approximation

Variant 1: approximation with monomials [Graf-Saïdi]

– each predicate represented in disjunctive normal form
(as disjunctions of *monomials* ϕ)
monomial = conjunction (product) of predicates p_i or their negation $\neg p_i$
– successor $post^a(\psi, t)$ for the transition (statement) t also approximated by a monomial
$\Rightarrow$ we determine for each predicate if the monomial contains p_i or $\neg p_i$ (or none)
$\Rightarrow$ we determine for each predicate p_i if $post^a(\psi, t)$ implies p_i or $\neg p_i$,
i.e., if $\psi \Rightarrow wp(p_i, t)$ or $\psi \Rightarrow wp(\neg p_i, t)$

Variant 2: BDD-based full decomposition [Das-Dill-Park]

– Approximating with monomials is highly restrictive
$\Rightarrow$ can lead to very coarse approximations
– However, more precise computations can lead to exponential number of calls to decision procedures $\Rightarrow$ infeasible
– split region ϕ recursively in fragments that can lead to states that satisfy p_i, or $\neg p_i$, respectively

$post^a(\phi, t) = post_1(\phi, t)$, where
$post_k(\phi, t) = p_k \wedge post_{k+1}(\phi \wedge pre^c(\gamma(p_k), t), t) \vee \neg p_k \wedge post_{k+1}(\phi \wedge \neg pre^c(\gamma(p_k), t),$
for $1 \leq k \leq n$, where $pre^c(r, t) = \{s \mid \exists s' \in r \,.\, s \xrightarrow{t} s'\}$
şi $post_{n+1}(\phi, t)$ is true if ϕ is satisfiable, and false otherwise.

Variant 3: constructing an abstract program [Ball-Rajamani]

– Previous variants require computing $post^a$ *dynamically* for *any* combination of predicates that appear in exploration
– this number is worst-case *exponential*
– Solutione: separate effects of program on each predicate
$\Rightarrow$ compute *once* for each statement its effect on predicate p_i
$\Rightarrow$ produce an *abstract boolean* program in which every statement has as effect the assignment of every predicate with a new value:
– true, for predicate combinations that imply $wp(\gamma(p_i), t)$
– false, for predicate combinations that imply $wp(\gamma(\neg p_i), t)$
– unknown, otherwise
– also called *cartesian* abstraction (independent for each predicate)

Predicate abstraction in software verification practice

Example: SLAM project [Microsoft Research]
(Software (Specifications), Languages, Analysis and Model checking)

Goal: verification of safety properties (invariants)
example: a program observes API usage rules
(such as: calls to `lock()` and `unlock()` alternate

– focused mainly on detecting *interface* errors
– applied to device drivers for Windows NT/XP

Characteristics:
– needs no user annotation of program
(only specifying rules as automata monitoring correct behavior)
– automated counterexample-guided abstraction refinement

Counterexample-guided abstraction refinement

– abstract model is program control flow graph augmented with chosen set of boolean predicates over program variables
(initial set of predicates may be empty)
– this finite representation is model checked to find violation of specification
– if the model is correct, the program is correct (conservative abstraction)
– if a counterexample is found, it is explored symbolically in the concrete program, retaining (cor)relations among variables
– if the counterexample is feasible, an error has been found
– counterexample may be infeasible, if the conjunction of the conditions needed to traverse the required branches is unsatisfiable (false)
$\Rightarrow$ counterexample due to coarse abstraction
– unsatisfiable core of formula suggests predicates to refine abstraction
– procedure is repeated with new (augmented) set of predicates

This is a *semialgorithm*; termination is not guaranteed.

Sample program

```
do {      /* fragment of device driver, [Ball & Rajamani '01] */
  KeAcquireSpinLock(&devExt->writeListLock);
  nPacketsOld = nPackets;
  request = devExt->WriteListHeadVa;
  if(request && request->status) {
    devExt->WriteListHeadVa = request->Next;
    KeReleaseSpinLock(&devExt->writeListLock);
    irp = request->irp;
    if (request->status > 0) {
      irp->IoStatus.Status = STATUS_SUCCESS;
      irp->IoStatus.Information = request->Status;
    } else {
      irp->IoStatus.Status = STATUS_UNSUCCESSFUL;
      irp->IoStatus.Information = request->Status;
    }
    SmartDevFreeBlock(request);
    IoCompleteRequest(irp, IO_NO_INCREMENT);
    nPackets++;
  }
} while (nPackets != nPacketsOld);
KeReleaseSpinLock(&devExt->writeListLock);
```

Specifying properties

```
state {
  enum { Unlocked=0, Locked=1 }
    state = Unlocked;
}

KeAcquireSpinLock.return {          KeReleaseSpinLock.return {
  if (state == Locked) abort;         if (state == Unlocked) abort;
  else state = Locked;                else state = Unlocked;
}                                   }
```

Specification translated into C; original program is instrumented
(original program correct ⇔ instrumented program cannot reach error)

Generating the boolean program

– start from the predicates in the specification
– use nondeterministic `if` where truth value unknown
– remove irrelevant instructions (`skip`)

```
do {
A: KeAcquireSpinLock_return();
  skip;
  if(*) {
B:   KeReleaseSpinLock_return();
    if (*) {
      skip;
    } else {
      skip;
    }
  }
} while (*);
C: KeReleaseSpinLock_return();
```


Model checking the boolean program

Bebop: calculates reached states for every statement of boolean program, using an interprocedural dataflow analysis algorithm

state = assignment to variables in scope
set of states = boolean function, represented as BDD
computation with sets of states: captures correlation between variables

– does not expand procedures, exploits locality of variables
– uses an explicit control flow graph
– complexity: linear in size of CFG; exponential in number of vars in scope

For the given example: model checker finds that `A: KeAcquireSpinLock()` could be called twice successively (an error)

Contraexample and generation of new predicates

A theorem prover is used to check if the counterexample in the abstract program is really a counterexample in concrete program

Evaluates program statements using symbolic constants until it finds that the assignment at the end of the path is feasible, or finds an inconsistency along the way.
For an inconsistency, a minimal unsatisfiable formula is found and the corresponding predicates are generated.
In the example `nPacketsOld = nPackets` and `nPacketsOld != nPackets`
decision procedures are incomplete ⇒ might return "don't know"
– the boolean program is then regenerated

The second boolean program

```
do {
A: KeAcquireSpinLock_return();
  b = T;     /* b == (nPackets == nPacketsOld) */
  if(*) {
B:   KeReleaseSpinLock_return();
    if (*) {
      skip;
    } else {
      skip;
    }
    b := choose(F, b);     /* choose(p1, p2) == p1 ?  T : p2 ?  F : nondet */
  }
} while (!b);
C: KeReleaseSpinLock_return();
```

The second, refined abstraction is sufficient to prove correctness.

Predicate abstraction in practice

– at present: programs of about 10kloc and tens/hundreds boolean variables can be analyzed in (tens of) minutes
– with optimisations, 100kloc may be reached

Available verifiers for C: BLAST (UC Berkeley), MAGIC (CMU)

Optimisation: *lazy abstraction* [Henzinger, Jhala, Majumdar, Sutre '02]
– does not refine abstraction at each iteration
– current abstraction is refined with new predicates only in code fragments where this is necessary (on-the-fly)
⇒ preserves locality (e.g., different abstractions for then/else branches)