Software Formal Verification Revision

Paul Jackson

School of Informatics
University of Edinburgh

Formal Verification
Spring 2017

What you need to know for exam

In general exam covers

- All material from lecture slides
- All material from labs

Further specific remarks on topics in Software Verification half follow:

Spark verification features

- You are expected to be able to read and understand Spark programs at level presented in lecture and labs.
- Do need to be able to write Spark assertions (e.g. loop invariants, pre-conditions, post-conditions).
    - Definitely review Spark labs

SPARK tool-set

- You need to have high-level appreciation of architecture of tool-set.
- Exam does not require specific knowledge of WhyML language.
- Re SMT solvers and SMT-LIB
    - You are expected to be able to understand SMT-LIB examples at level of lecture presentation
    - You should be familiar with the common theories SMT solvers support (e.g. linear vs. non-linear arithmetic, integer and real arithmetic, bitvectors, arrays, uninterpreted functions)
    - Do walk through the Z3 tutorial linked-to from the course home page.

WP-based methodology and tools

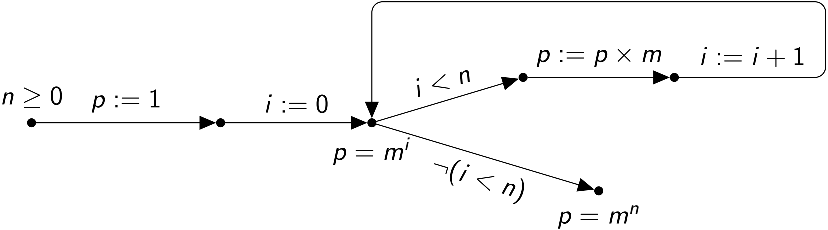
- Appreciation of methodology points is important
- No need to memorise names and capabilities of various tools

Programming language semantics

- Important to know the main definitions (big-step semantics, Hoare triples, (weakest) precondition computation, VC computation.)
  - VC computation best understood intuitively - components of VC from decomposition of control flow-graph into acyclic segments and paths.

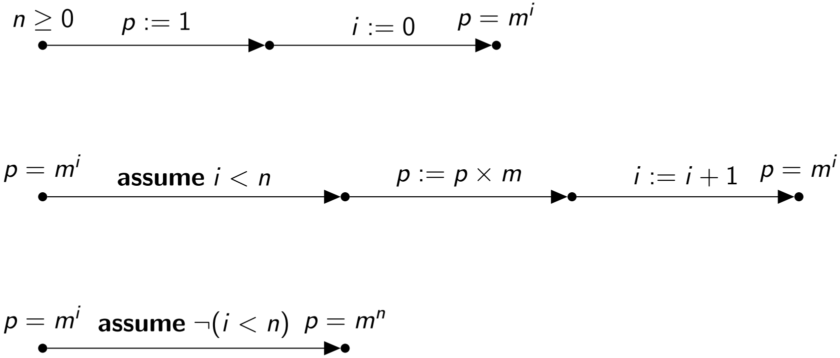
VC derivation via control flow graph 1

$\{n \geq 0\}$
$p := 1\ ;$
$i := 0\ ;$
$\{p = m^i\}$ **while** $i < n$ **do** $p := p \times m\ ;\ i := i + 1$
$\{p = m^n\}$



VC derivation via control flow graph 2

Split graph at loop invariant:



$n \geq 0$ $p := 1$ $i := 0$ $p = m^i$

$p = m^i$ **assume** $i < n$ $p := p \times m$ $i := i + 1$ $p = m^i$

$p = m^i$ **assume** $\neg(i < n)$ $p = m^n$

VC derivation via control flow graph 3

For each acyclic path c the VC is

$$\{P\}\, c\, \{Q\} \;=\; \forall \bar{x}.\; P \Rightarrow \mathrm{Pre}(c\;,\; Q) \quad ,$$

so the full VC is $VC_1 \wedge VC_2 \wedge VC_3$, where

$$\begin{aligned}
VC_1 &= \forall n.\; n \geq 0 \Rightarrow 1 = m^0 \\[1em]
VC_2 &= \forall i, n, m, p.\; p = m^i \Rightarrow \\
&\quad\; \mathrm{Pre}(\textbf{assume}\ i < n\;;\; p := p \times m\;;\; i := i + 1\;,\; p = m^i) \\
&= \forall i, n, m, p.\; p = m^i \Rightarrow \\
&\quad\; \mathrm{Pre}(\textbf{assume}\ i < n\;;\; p := p \times m\;,\; \mathrm{Pre}(i := i + 1\;,\; p = m^i)) \\
&= \forall i, n, m, p.\; p = m^i \Rightarrow \\
&\quad\; \mathrm{Pre}(\textbf{assume}\ i < n\;;\; p := p \times m\;,\; p = m^{i+1}) \\
&= \forall i, n, m, p.\; p = m^i \Rightarrow \mathrm{Pre}(\textbf{assume}\ i < n\;,\; p \times m = m^{i+1}) \\
&= \forall i, n, m, p.\; p = m^i \Rightarrow (i < n \Rightarrow p \times m = m^{i+1}) \\[1em]
VC_3 &= \forall i, m, p.\; p = m^i \Rightarrow (\neg(i < n) \Rightarrow p = m^n
\end{aligned}$$

VC_3 does not hold. What is missing from the loop invariant?

SAT and SMT algorithms

- You are expected to be able to run through calculations of
  - basic DPLL algorithm execution (backtracking, no backjumping)
  - formation of implication graphs and inference of learned clauses, including backjumping clauses, from these graphs
- You should have some intuition for all the rules covered

Basic SAT algorithm derivation

Assignment	Clauses														Rule
M	C_1		C_2			C_3		C_4			C_5		C_6		
	$\bar{b} \vee$	c	$\bar{a} \vee$	$\bar{b} \vee$	$\bar{c}$	$b \vee$	d	$\bar{a} \vee$	$b \vee$	$\bar{d}$	$a \vee$	e	$a \vee$	$\bar{e}$	
$()$	u	u	u	u	u	u	u	u	u	u	u	u	u	u	
$\bullet a$	u	u	0	u	u	u	u	0	u	u	1	u	1	u	Decide a
$\bullet a \bullet b$	$\underline{0}$	$\underline{u}$	0	0	u	1	u	0	1	u	1	u	1	u	Decide b
$\bullet a \bullet b\ c$	0	1	$\underline{0}$	$\underline{0}$	$\underline{0}$	1	u	0	1	u	1	u	1	u	UnitProp
$\bullet a\ \bar{b}$	1	u	0	1	u	$\underline{0}$	$\underline{u}$	0	0	u	1	u	1	u	Backtrack
$\bullet a\ \bar{b}\ d$	1	u	0	1	u	0	1	$\underline{0}$	$\underline{0}$	$\underline{0}$	1	u	1	u	UnitProp
$\bar{a}$	u	u	1	u	u	u	u	1	u	u	$\underline{0}$	$\underline{u}$	0	u	Backtrack
$\bar{a}\ e$	u	u	1	u	u	u	u	1	u	u	0	1	$\underline{0}$	$\underline{0}$	UnitProp
fail															Fail

Derivation shows that $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$ is unsatisfiable.

Notes on SAT algorithm derivation

- Rule priority: Fail, Backtrack, UnitPropagate, Decide (high to low)
    - See SAT-SMT slides for motivation
- Each rule might be applicable in more than one way
- Here:
    - Decide chooses earliest unassigned literal in alphabet and makes it un-negated
    - UnitPropagate chooses clause C_i with lowest index i
- Underlining indicate clauses that rules operate on.
- In practice, heuristics used to optimise performance. See SAT-SMT slides.

CBMC

- Make sure you appreciate the similarities and differences between the CBMC approach and the SPARK toolset approach.
- Given a simple C program decorated with one or more assertions, you should be able to derive SMT-level VCs that CBMC might check.
    - Loop unrolling
    - Static single assignment transformation
    - Use of conditional expressions at merge points in control flow

CBMC VC derivation 1

Q. Given program

```
int i;
int p;
p = 1;
for (i = 0; i <= n; i++) {
  p = p * m;
}
assert p >= 1;
```

What VC might CBMC generate, if loop is unrolled two times and we assume loop will not execute a third time?

A. Transform first to while loop, since easier to unroll

```
p = 1;
i = 0;
while (i <= n) {
  p = p * m;
  i = i + 1;
}
assert(p >= 1);
```

CBMC VC derivation 2

Unroll loop 2 times and add assume statement for loop exiting at that point

```
p = 1;
i = 0;
if (i <= n) {
  p = p * m;
  i = i + 1;
  if (i <= n) {
    p = p * m;
    i = i + 1;
    assume( !(i <= n) );
  }
}
assert(p >= 1);
```

CBMC VC derivation 3

Assign all variables exactly once. Compute guards for conditional statements. Add conditional expressions for merging values.

```
p1 = 1;
i1 = 0;
g1 = i1 <= n1;
  p2 = p1 * m1; // g1
  i2 = i1 + 1;  // g1
  g2 = (i2 <= n1);
    p3 = p2 * m1;  // g1 & g2
    i3 = i2 + 1;   // g1 & g2
    assume( !(i3 <= n1) );
p4 = g1 ? (g2 ? p3 : p2) : p1;
i4 = g1 ? (g2 ? i3 : i2) : i1; // Optional, since i4 unused
assert(p4 >= 1);
```

Comments track conditions under which assignments hold and help with computing value merge expressions.

CBMC VC derivation 4

Convert to logical expression.

$$
\begin{aligned}
& p_1 = 1 \\
& \wedge\ i_1 = 0 \\
& \wedge\ g_1 = (i_1 \leq n_1) \\
& \wedge\ p_2 = p_1 * m_1 \\
& \wedge\ i_2 = i_1 + 1 \\
& \wedge\ g_2 = (i_2 <= n_1) \\
& \wedge\ p_3 = p_2 * m_1 \\
& \wedge\ i_3 = i_2 + 1 \\
& \wedge\ \neg(i_3 \leq n_1) \quad \textit{(translation of } \texttt{assume} \textit{ statement)} \\
& \wedge\ p_4 = g_1 \mathbin{?} (g_2 \mathbin{?} p_3 : p_2) : p_1 \\
& \wedge\ i_4 = g_1 \mathbin{?} (g_2 \mathbin{?} i_3 : i_2) : i_1 \\
& \wedge\ \neg(p_4 \geq 1) \quad \textit{(translation of } \texttt{assert} \textit{ statement)}
\end{aligned}
$$

If this is found unsatisfiable, then assertion holds.