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

\[
\begin{align*}
\{ n \geq 0 \} \\
p &:= 1 ; \\
i &:= 0 ; \\
\{ p = m^i \} &\textbf{while } i < n \textbf{ do } p := p \times m ; i := i + 1 \\
\{ p = m^n \}
\end{align*}
\]
VC derivation via control flow graph 2

Split graph at loop invariant:

\[ n \geq 0 \quad p := 1 \quad i := 0 \quad p = m^i \]

\[ p = m^i \quad \text{assume } i < n \quad p := p \times m \quad i := i + 1 \quad p = m^i \]

\[ p = m^i \quad \text{assume } \neg(i < n) \quad p = m^n \]
VC derivation via control flow graph 3

For each acyclic path \( c \) the VC is

\[
\{ P \} c \{ Q \} = \forall \bar{x}. P \Rightarrow \text{Pre}(c, Q),
\]

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

\[
\begin{align*}
VC_1 &= \forall n. n \geq 0 \Rightarrow 1 = m^0 \\
VC_2 &= \forall i, n, m, p. p = m^i \Rightarrow \text{Pre}(\text{assume } i < n; p := p \times m; i := i + 1, p = m^i) \\
&= \forall i, n, m, p. p = m^i \Rightarrow \text{Pre}(\text{assume } i < n; p := p \times m, \text{Pre}(i := i + 1, p = m^i)) \\
&= \forall i, n, m, p. p = m^i \Rightarrow \text{Pre}(\text{assume } i < n; p := p \times m, p = m^{i+1}) \\
&= \forall i, n, m, p. p = m^i \Rightarrow (i < n \Rightarrow p \times m = m^{i+1}) \\
VC_3 &= \forall i, m, p. p = m^i \Rightarrow (\neg(i < n) \Rightarrow p = m^n)
\end{align*}
\]

\( VC_3 \) 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

<table>
<thead>
<tr>
<th>Assignment</th>
<th>(M)</th>
<th>(C_1)</th>
<th>(C_2)</th>
<th>(C_3)</th>
<th>(C_4)</th>
<th>(C_5)</th>
<th>(C_6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>(\emptyset)</td>
<td>(u)</td>
<td>(u)</td>
<td>(u)</td>
<td>(u)</td>
<td>(u)</td>
<td>(u)</td>
</tr>
<tr>
<td>(\bullet a)</td>
<td>(\emptyset)</td>
<td>(u)</td>
<td>(u)</td>
<td>(0)</td>
<td>(u)</td>
<td>(u)</td>
<td>(0)</td>
</tr>
<tr>
<td>(\bullet a \bullet b)</td>
<td>(u)</td>
<td>(0)</td>
<td>(u)</td>
<td>(u)</td>
<td>(u)</td>
<td>(u)</td>
<td>(0)</td>
</tr>
<tr>
<td>(\bullet a \bullet b)</td>
<td>(0)</td>
<td>(1)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>(\bullet a \bar b)</td>
<td>(1)</td>
<td>(u)</td>
<td>(0)</td>
<td>(1)</td>
<td>(u)</td>
<td>(0)</td>
<td>(u)</td>
</tr>
<tr>
<td>(\bullet a \bar b)</td>
<td>(1)</td>
<td>(u)</td>
<td>(0)</td>
<td>(1)</td>
<td>(u)</td>
<td>(0)</td>
<td>(1)</td>
</tr>
<tr>
<td>(\bar a)</td>
<td>(u)</td>
<td>(u)</td>
<td>(1)</td>
<td>(u)</td>
<td>(u)</td>
<td>(1)</td>
<td>(u)</td>
</tr>
<tr>
<td>(\bar a)</td>
<td>(u)</td>
<td>(u)</td>
<td>(1)</td>
<td>(u)</td>
<td>(u)</td>
<td>(1)</td>
<td>(u)</td>
</tr>
</tbody>
</table>

Derivation shows that \(C_1 \land C_2 \land C_3 \land C_4 \land C_5 \land C_6\) 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
Q. Given program

```c
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

```c
p = 1;
i = 0;
while (i <= n) {
    p = p * m;
    i = i + 1;
}
assert(p >= 1);
```
Unroll loop 2 times and add assume statement for loop exiting at that point

```c
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);
```
Assign all variables exactly once. Compute guards for conditional statements. Add conditional expressions for merging values.

\[
p_1 = 1; \\
i_1 = 0; \\
g_1 = i_1 \leq n_1; \\
p_2 = p_1 \times m_1; \quad // \quad g_1 \\
i_2 = i_1 + 1; \quad // \quad g_1 \\
g_2 = (i_2 \leq n_1); \\
p_3 = p_2 \times m_1; \quad // \quad g_1 \quad \& \quad g_2 \\
i_3 = i_2 + 1; \quad // \quad g_1 \quad \& \quad g_2 \\
\text{assume( !}(i_3 \leq n_1) \text{));} \\
p_4 = g_1 \quad ? \quad (g_2 \quad ? \quad p_3 \quad : \quad p_2) \quad : \quad p_1; \\
i_4 = g_1 \quad ? \quad (g_2 \quad ? \quad i_3 \quad : \quad i_2) \quad : \quad i_1; \quad // \quad \text{Optional, since i4 unused} \\
\text{assert(p4} \geq 1); \\
\]

Comments track conditions under which assignments hold and help with computing value merge expressions.
CBMC VC derivation 4

Convert to logical expression.

\[ p_1 = 1 \]
\[ \land i_1 = 0 \]
\[ \land g_1 = (i_1 \leq n_1) \]
\[ \land p_2 = p_1 \times m_1 \]
\[ \land i_2 = i_1 + 1 \]
\[ \land g_2 = (i_2 \leq n_1) \]
\[ \land p_3 = p_2 \times m_1 \]
\[ \land i_3 = i_2 + 1 \]
\[ \land \neg (i_3 \leq n_1) \quad (\text{translation of assume statement}) \]
\[ \land p_4 = g_1 \ ? (g_2 \ ? p_3 : p_2) : p_1 \]
\[ \land i_4 = g_1 \ ? (g_2 \ ? i_3 : i_2) : i_1 \]
\[ \land \neg (p_4 \geq 1) \quad (\text{translation of assert statement}) \]

If this is found unsatisfiable, then assertion holds.