Software Engineering using Formal Methods

Reasoning about Programs with Loops and Method Calls

Wolfgang Ahrendt

17 October 2013

Program Logic Calculus – Repetition

Calculus realises symbolic interpreter:

- works on first active statement
- decomposition of complex statements into simpler ones
- simple assignments to updates
- accumulated update captures changed program state (abbr. w. $\mathcal{U}$)
- control flow branching induces proof splitting
- application of update computes weakest precondition of $\mathcal{U}'$ wrt. ϕ

$$\Gamma' \Longrightarrow \{\mathcal{U}'\}\phi \qquad \ldots$$

$$\ldots \qquad \ldots$$

$$\text{'branch1'} \quad \Gamma, \{\mathcal{U}\}(\mathtt{isValid} \doteq \mathtt{TRUE}) \Longrightarrow \{\mathcal{U}\}\langle\{\mathtt{ok{=}\mathbf{true};}\}\ldots\rangle\phi$$

$$\text{'branch2'} \quad \Gamma, \{\mathcal{U}\}(\mathtt{isValid} \doteq \mathtt{FALSE}) \Longrightarrow \{\mathcal{U}\}\langle\ldots\rangle\phi$$

$$\Gamma \Longrightarrow \{\mathtt{t} := \mathtt{j} \| \mathtt{j} := \mathtt{j} + 1 \| \mathtt{i} := \mathtt{j}\}\{\mathcal{U}\}\{\mathcal{U}\}\langle\mathbf{if}\,\mathtt{(isValid)\{ok{=}\mathbf{true};\}}\ldots\rangle\phi$$

$$\ldots$$

$$\Gamma \Longrightarrow \{\mathtt{t} := \mathtt{j}\}\langle\mathtt{j{=}j{+}1;i{=}t;}\mathbf{if}\mathtt{(isValid)\{ok{=}\mathbf{true};\}}\ldots\rangle\phi$$

$$\Gamma \Longrightarrow \langle\mathtt{t{=}j;j{=}j{+}1;i{=}t;}\mathbf{if}\mathtt{(isValid)\{ok{=}\mathbf{true};\}}\ldots\rangle\phi$$

$$\Gamma \Longrightarrow \langle\mathtt{i{=}j{+}{+};}\mathbf{if}\mathtt{(isValid)\{ok{=}\mathbf{true};\}}\ldots\rangle\phi$$

Are parallel updates sufficient?

How to express using updates that a formula ϕ is evaluated in a state where

- program variable `i` has been set to 5? $\{\mathtt{i} := 5\}\phi$
- program variable `i` has been increased by 1? $\{\mathtt{i} := \mathtt{i+1}\}\phi$
- program variables `i` and `j` swapped values? $\{\mathtt{i} := \mathtt{j} \parallel \mathtt{j} := \mathtt{i}\}\phi$
- all components of an array `arr` of length 2 have value 0?

  $$\{\mathtt{arr[0]} := 0 \parallel \mathtt{arr[1]} := 0\}\phi$$
- all components of an array `arr` of length `n` have value 0?

For example to deal with things like

$$\langle \mathtt{\textbf{int}[]\ a = \textbf{new}\ \textbf{int}[n];} \rangle$$
$$\forall\, \mathtt{\textbf{int}}\ x; (0 \leq x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 0)$$

Quantified Updates

Definition (Quantified Update)

For T well-ordered type (no ∞ descending chains): quantified update:

$$\{\texttt{\textbackslash for}\ T\ x; \texttt{\textbackslash if}\ \phi(x); l(x) := r(x)\}$$

- For all objects d in T such that $\phi(d)$
  perform the updates $\{l(d) := r(d)\}$ in parallel
- If there are several l with conflicting d then choose T-minimal one

- The conditional expression is optional
- Typically, x occurs in ϕ, l, and r (but doesn't need to)
- There is a normal form for updates computed efficiently by KeY

Quantified Updates Cont'd

Example (Initialization of field `a` for all objects in class `C`)

$$\{\backslash\texttt{\textbf{for}}\ \texttt{C}\ o; o.\mathrm{a} := 0\}$$

Example (Initialization of components of array `a`)

$$\{\backslash\texttt{\textbf{for}}\ \texttt{\textbf{int}}\ i; \mathrm{a}[i] := 0\}$$

Example (Integer types are well-ordered in KeY)

$$\{\backslash\texttt{\textbf{for}}\ \texttt{\textbf{int}}\ i; \mathrm{a}[0] := i\}(\mathrm{a}[0] \doteq 0)$$

- Non-standard order for $\mathbb{Z}$ (with 0 smallest and preserving $<$ for arguments of same sign)
- Proven automatically by update simplifier

Loop Invariants

Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\,\mathtt{if}\,(\mathtt{b})\,\{\mathtt{p};\ \mathtt{while}\,(\mathtt{b})\ \mathtt{p}\}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\,\mathtt{while}\,(\mathtt{b})\ \mathtt{p}\ \omega]\phi, \Delta}$$

(We omitted $\mathcal{U}$ previous lectures, for simplicity.)

How to handle a loop with...

- 0 iterations? Unwind $1\times$
- 10 iterations? Unwind $11\times$
- 10000 iterations? Unwind $10001\times$
- an unknown number of iterations?

We need an invariant rule (or some form of induction)

Loop Invariants Cont'd

Idea behind loop invariants

- A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates, then *Inv* holds afterwards
- Construct *Inv* such that, together with loop exit condition, it implies postcondition of loop

Basic Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}\, \mathit{Inv}, \Delta & \text{(valid when entering loop)} \\ \mathit{Inv},\ b \doteq \mathtt{TRUE} \Longrightarrow [\mathrm{p}]\mathit{Inv} & \text{(preserved by p)} \\ \mathit{Inv},\ b \doteq \mathtt{FALSE} \Longrightarrow [\pi\ \omega]\phi & \text{(assumed after exit)} \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \mathtt{while}(\mathtt{b})\ \mathtt{p}\ \omega]\phi, \Delta}$$

Loop Invariants Cont'd

Basic Invariant Rule: Problem

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta & \text{(valid when entering loop)}\\ \mathit{Inv},\ b \doteq \mathtt{TRUE} \Longrightarrow [\mathtt{p}]\mathit{Inv} & \text{(preserved by p)}\\ \mathit{Inv},\ b \doteq \mathtt{FALSE} \Longrightarrow [\pi\ \omega]\phi & \text{(assumed after exit)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathtt{while(b)\ p}\ \omega]\phi, \Delta}$$

- Context Γ, Δ, $\mathcal{U}$ must be omitted in 2nd and 3rd premise:
  - $\mathcal{U}$ represents state when entering loop, not after some loop iterations
  - keeping Γ, Δ without $\mathcal{U}$ meant executing `p` in prestate of program
- But: context contains important preconditions and class invariants
- Relevant context information must be added to *Inv* ☹

Example

(Implicit) Class Invariant: $\mathtt{a} \not\doteq \mathbf{null}$

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \mathtt{i}\ \ \&\ \ \mathtt{i} \leq \mathtt{a.length}$
$\&\ \ \forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{i} \rightarrow \mathtt{a}[x] \doteq 1)$
$\&\ \ \mathtt{a} \not\doteq \mathbf{null}$

Keeping the Context

- Want to keep part of the context that is unmodified by loop
- **assignable** clauses for loops can tell what might be modified

```
@ assignable i, a[*];
```

- How to erase all values of **assignable** locations in formula Γ ?
- Anonymising update $\mathcal{V}$ erases information about assignable locations

$\{\mathtt{i} := c\}$ (c fresh constant symbol)
$\{\textbf{\textbackslash for}\ x;\ \mathtt{a}[x] := f(x)\}$ (f fresh function symbol)

$\mathcal{V} = \{\mathtt{i} := c \,||\, \textbf{\textbackslash for}\ x;\ \mathtt{a}[x] := f(x)\}$
(c, f fresh constant resp. function symbol)

Loop Invariants Cont'd

Improved Invariant Rule

$$\frac{\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}\,\mathit{Inv}, \Delta & \text{(valid when entering loop)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(\mathit{Inv}\ \&\ b \doteq \mathtt{TRUE} \rightarrow [\mathrm{p}]\mathit{Inv}), \Delta & \text{(preserved by p)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(\mathit{Inv}\ \&\ b \doteq \mathtt{FALSE} \rightarrow [\pi\ \omega]\phi), \Delta & \text{(assumed after exit)} \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \mathbf{while}(\mathrm{b})\ \mathrm{p}\ \omega]\phi, \Delta}$$

- Context is kept as far as possible
- Invariant not 'responsible' for un-assignable locations
- Missing assignable clause (equiv. to **`assignable \everything`**):
  - $\mathcal{V} = \{* := *\}$ wipes out **all** information
  - Equivalent to basic invariant rule
  - Avoid this! Always give a specific **`assignable`** clause

Example with Improved Invariant Rule

(Implicit) Class Invariant: $\mathtt{a} \not\doteq \mathbf{null}$ not needed for loop invariant

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \mathtt{i}\ \ \&\ \ \mathtt{i} \leq \mathtt{a.length}$
$\&\ \ \forall\, \mathbf{int}\ x;\ (0 \leq x < \mathtt{i} \rightarrow \mathtt{a}[x] \doteq 1)$

Example in JML/JAVA – Loop.java

```
public int[] a;
/*@ public normal_behavior
  @  ensures (\forall int x; 0<=x && x<a.length; a[x]==1);
  @  diverges true;
  @*/
public void m() {
  int i = 0;
  /*@ loop_invariant
    @  (0 <= i && i <= a.length &&
    @   (\forall int x; 0<=x && x<i; a[x]==1));
    @ assignable i, a[*];
    @*/
  while(i < a.length) {
    a[i] = 1;
    i++;
  }
}
```

Example from last week

$\forall$ `int` x;
  $(x \doteq \texttt{n} \wedge x >= 0 \rightarrow$
    $[$
```
i = 0; r = 0;
while (i<n) { i = i + 1; r = r + i;}
r=r+r-n;
```
    $]\texttt{r} \doteq ?x * x)$

How can we prove that the above formula is valid
(i.e. satisfied in all states)?

Solution:

```
@ loop_invariant
@     i>=0 && 2*r == i*(i + 1) && i <= n;
@ assignable i, r;
```

File: Loop2.java

Hints

Proving `assignable`

- The invariant rule assumes that **`assignable`** is correct
  E.g., with **`assignable \nothing;`** one can prove nonsense
- Invariant rule of KeY generates proof obligation that ensures correctness of **`assignable`**

Setting in the KeY Prover when proving loops

- Loop treatment: Invariant
- Quantifier treatment: No Splits with Progs
- If program contains `*`, `/`:
  Arithmetic treatment: DefOps
- Is search limit high enough (time out, rule apps.)?
- When proving partial correctness, add **`diverges true;`**

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- v is strictly decreased by the loop body

Proving termination in JML/Java

- Remove directive `diverges true;`
- Add directive `decreasing v;` to loop invariant
- KeY uses suitable invariant rule and PO (with $\langle \ldots \rangle \phi$)

Example (The `array` loop)

```
@ decreasing  a.length - i;
```

Files:

- `LoopT.java`
- `Loop2T.java`

Method Calls – Repetition

Method Call with actual parameters $arg_0, \ldots, arg_n$

$$\{arg_0 := t_0 \,||\, \ldots \,||\, arg_n := t_n \,||\, c := t_c\}\langle c.\mathtt{m}(arg_0, \ldots, arg_n);\rangle\phi$$

where $\mathtt{m}$ declared as $\mathbf{void}\ \mathtt{m}(\mathtt{T_0}\ \mathtt{p_0}, \ldots, \mathtt{T_n}\ \mathtt{p_n})$

Actions of rule methodCall

- for each formal parameter $\mathtt{p_i}$ of $\mathtt{m}$:
  declare and initialize new local variable $\mathtt{T_i}\ \mathtt{p\#i} = arg_i;$
- look up implementation class C of $\mathtt{m}$, or split proof
  if implementation cannot be uniquely determined
- create method invocation $c.\mathtt{m}(\mathtt{p\#0}, \ldots, \mathtt{p\#n})\mathtt{@}C$

Method Calls Cont'd

Method Body Expand

1. Execute code that binds actual to formal parameters $\mathtt{T_i}$ $\mathtt{p\#i} = arg_i$;
2. Call rule methodBodyExpand

$$\frac{\Gamma \Longrightarrow \langle \pi\ \mathtt{method\text{-}frame(source=C,\ this=c)\{\ body\ \}}\ \omega \rangle \phi, \Delta}{\Gamma \Longrightarrow \langle \pi\ \mathtt{c.m(p\#0,\ldots,p\#n)@C;}\ \omega \rangle \phi, \Delta}$$

Symbolic Execution

Only static information available, proof splitting;
Runtime infrastructure required in calculus

File: `inlineDynamicDispatch.key`

Problem

Formal specification of JAVA API and other called methods

How to perform symbolic execution when JAVA API method is called?

1. Method has reference implementation in JAVA
   Inline method body and execute symbolically

   Problems Reference implementation not always available
   Too expensive
   Impossible to deal with recursion

2. Use method contract instead of method implementation

Method Contract Rule – Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
  @ requires normalPre;
  @ ensures normalPost;
  @ assignable mod;
  @*/
```

$$\frac{\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\mathtt{normalPre}), \Delta \quad \text{(precondition)} \qquad \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod}(\mathcal{F}(\mathtt{normalPost}) \rightarrow \langle \pi\ \omega \rangle \phi), \Delta \quad \text{(normal)}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \mathtt{result} = \mathtt{m}(\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n})\ \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL (see last lecture)
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Method Contract Rule – Exceptional Behavior Case

Warning: Simplified version

```
/*@ public exceptional_behavior
  @ requires excPre;
  @ signals (Exception exc) excPost;
  @ assignable mod;
  @*/
```

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\mathcal{F}(\mathtt{excPre}), \Delta \quad \text{(precondition)} \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod}((\mathtt{exc} \doteq\not\ \mathbf{null} \wedge \mathcal{F}(\mathtt{excPost})) \\ \qquad\qquad\qquad \rightarrow \langle \pi\, \mathbf{throw}\ \mathtt{exc};\ \omega\rangle\phi), \Delta \quad \text{(exceptional)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \mathtt{result} = \mathtt{m}(\mathtt{a}_1, \ldots, \mathtt{a}_n)\ \omega\rangle\phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Method Contract Rule – Combined

Warning: Simplified version

KeY uses actually only one rule for both kinds of cases.

Therefore translation of postcondition ϕ_{post} as follows (simplified):

$$\phi_{post\_n} \equiv \mathcal{F}(\texttt{\textbackslash old}(\texttt{normalPre})) \land \mathcal{F}(\texttt{normalPost})$$
$$\phi_{post\_e} \equiv \mathcal{F}(\texttt{\textbackslash old}(\texttt{excPre})) \land \mathcal{F}(\texttt{excPost})$$

$$\frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}(\mathcal{F}(\texttt{normalPre}) \lor \mathcal{F}(\texttt{excPre})), \Delta & \text{(precondition)}\\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod_{normal}}(\phi_{post\_n} \to \langle \pi\, \omega \rangle \phi), \Delta & \text{(normal)}\\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}_{mod_{exc}}((\texttt{exc} \doteq\!\!\!\!/\; \textbf{null} \land \phi_{post\_e}) & \\ \qquad\qquad \to \langle \pi\, \textbf{throw}\ \texttt{exc};\ \omega \rangle \phi), \Delta & \text{(exceptional)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\, \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{n})\ \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- $\mathcal{V}_{mod}$: anonymising update (similar to loops)

Understanding Proof Situations

Reasons why a proof may not close

- bug or incomplete specification
- bug in program
- maximal number of steps reached: restart or increase # of steps
- automatic proof search fails and manual rule applications necessary

Understanding open proof goals

- follow the taken control-flow from the root to the open goal
- branch labels may give useful hints
- identify (part of) the post-condition or invariant that cannot be proven
- sequent remains always in "pre-state". I.e., constraints like $\mathtt{i} \geq 0$ refer to the value of `i` before executing the program (exception: sub-formulae prefixed by update or modality)
- remember: $\Gamma \Longrightarrow \mathrm{o} \doteq \mathbf{\mathtt{null}}, \Delta$ is equivalent to $\Gamma, \mathrm{o} \not\doteq \mathbf{\mathtt{null}} \Longrightarrow \Delta$

Summary

- Most JAVA features covered in KeY
- Several of remaining features available in experimental version
  - Simplified multi-threaded JMM
  - Floats
- Degree of automation for loop-free programs is high
- Proving loops requires user to provide invariant
  - Automatic invariant generation sometimes possible
- Symbolic execution paradigm lets you use KeY w/o understanding details of logic

Literature for this Lecture

Essential

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 10: Using KeY

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 3: Dynamic Logic, Sections 3.1, 3.2, 3.4, 3.5, 3.6.1, 3.6.2, 3.6.3, 3.6.4, 3.6.5, 3.6.7, 3.7