Formal Methods Module III: Verification of parallel programs

Non-deterministic programs

General notes about parallelism

- Parallel programs are compositions of sequential processes (threads).
- Processes are implemented as (possibly non-deterministic) sequential programs.
- Two basic inter-process communication mechanisms:
  - shared variables;
  - message passing.

Principles of verifying parallel programs

- Observation:
  - The behaviour of whole system does not depend only on the interacting processes alone
  - but also on the communication mechanism between the processes
  - and the order (timing) of communication actions.
- Thus, the communication must be made <u>explicit</u> to verify the program in whole!

Example of necessity to make the interleavings of processes explicit

- What is the result of executing a simple parallel program?
  - Process 1:: `X := 0; Y := X + 1;`
  - Process 2:: `X := 1; Y := X + 2;`

- Possible interleaving of executions:
  - <P1.1, P1.2, P2.1, P2.2> → {X=1, Y=3}
  - <P2.1, P2.2, P1.1, P1.2> → {X=0, Y=1}
  - <P1.1, P2.1, P2.2, P2.1> → {X=1, Y=2}
  - ...
- Due to the interleavings the number of possible final results explodes

General verification strategy

- We prefer to reuse the Hoare logic for while-programs, i.e. to prove processes at first *locally and thereafter whole system.*
- To verify local correctness we need assertions (contracts) about the local effect of communication (i.e. extra lemmas about it).
- The communication assertions need to be generated and verified:
  - the *interference test* (IFT) if communication via shared variables ;
  - the *co-operation test* (COOP) if communication via message passing.
- Finally, *whole* system correctness is verified by using local proofs, communication assertions and parallel composition rule.

Non-deterministic sequential programs

- Languages GCL and GCL+ are
  - *guarded command languages* designed by E. Dijkstra
  - they include non-deterministic counterparts of
    - `if` - command and
    - `while` – command
  - they differ slightly by their syntactic structure
  - GCL is more compact than GCL+.

Syntax of GCL and GCL+

- $Pvar$ – set of program variables:
  - $x \in Pvar$
- VAL- set of possible values including natural numbers:
  - $a \in VAL$
- *Arithmetic expressions*:
  - $e ::= a \mid x \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 \cdot e_2)$
- *Boolean expressions*:
  - $b ::= e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$

GCL / GCL+

- *Commands*:

$C ::=$

$\overline{x} := \overline{e}$

$\mid C_1 \; ; \; C_2$

$\mid \texttt{if} \; []^{n}_{i=1} \; b_i \rightarrow C_i \; \texttt{fi}$

$\mid \texttt{do} \; []^{n}_{i=1} \; b_i \rightarrow C_i \; \texttt{od}$ (different in GCL+)

GCL / GCL+ (continued)

- *Assignment*:
  - $\overline{x} := \overline{e}$
  - assigns value of vector $\overline{e}$ to the variable vector $\overline{x}$
- *Sequential composition*:
  - $C_1 ; C_2$
  - *first execute* C_1 *and continue with the execution of* C_2 *if and when* C_1 *terminates.*

GCL / GCL+ (continued)

- Guarded command:

$$\texttt{if } []^{n}_{i=1}\, b_i \rightarrow C_i \text{ fi}$$

also written as

$$\texttt{if } b_1 \rightarrow C_1 \texttt{ [] } \dots \texttt{ [] } b_n \rightarrow C_n \texttt{ fi}$$

  - *abort* if none of the guards b_i evaluates to `true`;
  - otherwise, nondeterministically select one of the b_i that evaluates to `true` and execute the corresponding C_i.

GCL (continued)

- Iteration:

  $\texttt{do } []^{n}_{i=1}\ b_i \rightarrow C_i \texttt{ od}$ % in GCL only

  - repeats execution of guarded command C_i as long as at least one of the guards b_i evaluates to `true`;
  - when none of the guards evaluates to `true`, the iteration terminates (acts like *skip*).

GCL+

Commands:

C ::=

$\langle b \rightarrow \bar{x} := \bar{e} \rangle$

| C_1 ; C_2

| `if` $[]^{n}_{i=1}\, b_i \rightarrow C_i$ `fi`

| `do` C_B [] (C_E ; `exit`) `od`

Same as in GCL (first three alternatives)

- where C_i, C_B, C_E are guarded commands (nesting),
- (C_E ; `exit`) is terminating branch of the loop.

GCL+ (continued)

- Iteration:

  `do` C_B `[] (`C_E `; exit) od`

  - is the repeated execution of guarded command C_B as long as at least one of the guards in C_B evaluates to *true*
  - or the guard of the finishing command C_E evaluates to *true*.

- Command C is guarded command, if C has a form:
  - $\langle b \rightarrow \overline{v} := \overline{e} \rangle$ (atomic) guarded assignment;
  - C_1 ; C_2 where C_1 is a guarded command;
  - `if` $[]^{n}_{i=1}\ b_i \rightarrow C_i$ `fi` where every C_i is a guarded command

Proof system for GCL+ programs

- The "assignment" and "skip" axioms of deterministic sequential programs are same for GCL+.

Axiom 3 (*guard*):

$\{b \Rightarrow Q\}$ `b` $\{Q\}$

- Note: guard evaluation is an atomic operation.

Axiom 4 (*guarded assignment*):

- $\{b \Rightarrow Q[e/x]\}\ \langle$ `b` $\rightarrow$ `x:=e` $\rangle\ \{Q\}$

- Note:
  - Given axiomatic system is not minimal,
  - axioms 1-3 can be deduced from axiom 4.

GCL+ inference rules (continuation)

- Weakening, strengthening and sequential composition rules apply in GCL+.

Rule 3 (choice):

$$\frac{\forall i \in \{1, \ldots, n\}: \{P\}\ C_i\ \{Q\}}{\{P\}\ \texttt{if}\ \square^{n}_{i=1}\ C_i\ \ \texttt{fi}\ \{Q\}}$$

Rule 4 (guarded command):

$$\frac{\vdash \forall i \in \{1, \ldots, n\}: \{P \wedge b_i\}\ C_i\ \{Q\}}{\vdash \{P\}\ \texttt{if}\ \square^{n}_{i=1}\ b_i \rightarrow C_i\ \texttt{fi}\ \{Q\}}$$

GCL+ inference rules (continuation)

- Rule 5 (`exit`-loop):

$$\frac{\vdash \{P\}\ C_B\ \{P\}, \qquad \vdash \{P\}\ C_E\ \{Q\}}{\vdash \{P\}\ \texttt{do}\ C_B\ \square\ (C_E;\ \texttt{exit})\ \texttt{od}\ \{Q\}} \qquad P\text{- invariant}$$

- Rule 6 (`do`-loop):

$$\frac{\vdash \quad \forall i \in \{1,\ldots,n\}\colon \{P \wedge b_i\}\ C_i\ \{P\}}{\vdash \{P\}\ \texttt{do}\ \square^{n}_{i=1}\ b_i \rightarrow C_i\ \texttt{od}\ \{P \wedge \neg b_G\}}$$

where $b_G \cong \bigvee^{n}_{i=1} b_i$

GSL+ verification example

<u>Integer division:</u>

- `x` – dividend (non-negative integer)
- `y` – divisor (positive integer)
- `q` – quotient
- `r` – reminder

We are looking for a GSL+ program *Div*, for the specification

$$\{x \geq 0 \wedge y > 0\}\ Div\ \{post\_div\},$$

where

$post\_div \equiv x = q \cdot y + r \ \wedge 0 \leq r < y,$

Div does not change `x` and `y`

GSL+ verification example (continuation)

Solution 1:

$Div1 \equiv$

```
q, r := 0, x;                // atomic assignment
do
      y ≤ r  →   q,r := q+1, r-y
od
```

construct an invariant I by strengthening the post-condition of the loop

- Example:
  - from $(I \wedge \neg (y \leq r)) \Rightarrow post\_div$,
  - we get $I \equiv x = q \cdot y + r \wedge 0 \leq r$

GSL+ verification example (continuation)

Annotate the program, using the invariant $I \equiv x = q \cdot y + r \wedge 0 \leq r$

$\{x \geq 0 \wedge y > 0\}$

```
    q,r  := 0,x;
    do   {I}
         y ≤ r  →    q,r := q+1, r−y
    od   {I ∧ ¬ (y ≤ r)}
```

$\{x = q \cdot y + r \ \wedge 0 \leq r < y\}$

Check the partial correctness of given annotations:

1. $$\frac{(x \geq 0 \wedge y > 0) \Rightarrow (x = 0 \cdot y + x \wedge 0 \leq x)}{\{x \geq 0 \wedge y > 0\}\ q, r\ := 0, x\ \{I\}}$$

2. $$\frac{(x = q \cdot y + r\ \wedge 0 \leq r\ \wedge y \leq r) \Rightarrow (x = (q+1) \cdot y + (r\text{-}y) \wedge 0 \leq (r\ \text{-}y))}{\{I \wedge (y \leq r)\}\ q, r := q+1, r - y\ \{I\}}$$

3. $(I \wedge \neg(y \leq r)) \Rightarrow x = q \cdot y + r\ \wedge 0 \leq r < y$

Exercise: GCD

Show that the following program finds the $gcd(x, y)$ and returns the result in X.

```
X,Y := x,y
do
   X>Y → X:=X-Y
[]
   Y>X → Y:=Y-X
od
```

Use axioms of gcd:

- $\gcd(a,0) = a$
- $\gcd(a, a) = a$
- $a>b \Rightarrow \gcd(a, b)= \gcd(a-b, b)$
- $a<b \Rightarrow \gcd(a, b)= \gcd(a, b-a)$

Exercise 2

Annotate and verify the program that computes max of `x` and `y`

```
[
x≥y → m:=x
[]
y≥x → m:=y
]
```