Lecture #9
Verification of parallel programs

ITI8531: Software synthesis and verification

Spring 2017

J.Vain

What is a prallel program?

- Parallel programs are compositions of sequential processes (threads).
- Processes are implemented as sequential programs (possibly non-deterministic).
- Processes communicate using 2 mechanisms:
  - shared variables;
  - message passing.

What makes verifying parallel programs so special?

- Observation:
  - The behaviour of whole system does not depend only on the processes alone
  - but interaction matters, i.e.
    - the communication mechanism between processes
    - and the order (timing) of how the processes interact
- Thus, to verify a parallel program also *communication* must be addressed explicitly by proof rules

Why interleaving of processes matters? An example

- 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 execution steps:
  - <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 interleaving the number of possible final results grows exponentially in the length of processes.

General verification strategy

- We prefer compositional HL also when verifying parallel programs.
- Processes are proven *locally* at first and *whole system* thereafter.
- To verify local correctness we need assertions about how communication affects processes locally (*extra axioms* about it).
- The communication assertions need to be generated and verified:
  - the *interference test* (IFT) if communication via <u>shared variables</u>;
  - the *co-operation test* (COOP) if communication via <u>message passing</u>.
- After local proofs are done using HL and communication axioms *whole* system is verified using parallel composition rule.

Parallel processes are generally non-deterministic sequential programs

- We use E. Dijkstra’s Guarded Command Language (GCL) for programming non-deterministic sequential processes

- GCL includes non-deterministic counterparts of
  - `if` - command and
  - `while` – command

Syntax of 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

- *Commands*:

  C ::=

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

  | C_1 `;` C_2

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

  | `do` $[]^{n}_{i=1}$ $b_i \rightarrow C_i$ `od`

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 (continued)

- Guarded command (symbolically):

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

- Each alternative is written (explicitly) in the program

```
if    $b_1 \rightarrow C_1$
   [] $b_2 \rightarrow C_2$
    …
   [] $b_n \rightarrow C_n$
fi
```

- Meaning:
  - *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:

  `do` $[]^{n}_{i=1}\ b_i \rightarrow C_i$ `od`

  - 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 inference rules

Guarded assignment rule:

$$\frac{\vdash P \wedge b \Rightarrow Q[e/x]}{\vdash \{P\}\ \langle \mathtt{b} \ \rightarrow\ \mathtt{x:=e} \rangle\ \{Q\}}$$

General *guarded command* rule:

$$\frac{\vdash \forall i \in \{1, \ldots, n\}\colon \{P \wedge b_i\}\ C_i\ \{Q\}}{\vdash \{P\}\ \mathtt{if}\ []^{n}_{i=1}\ b_i\ \rightarrow\ C_i\ \mathtt{fi}\ \{Q\}}$$

GCL inference rules (continuation)

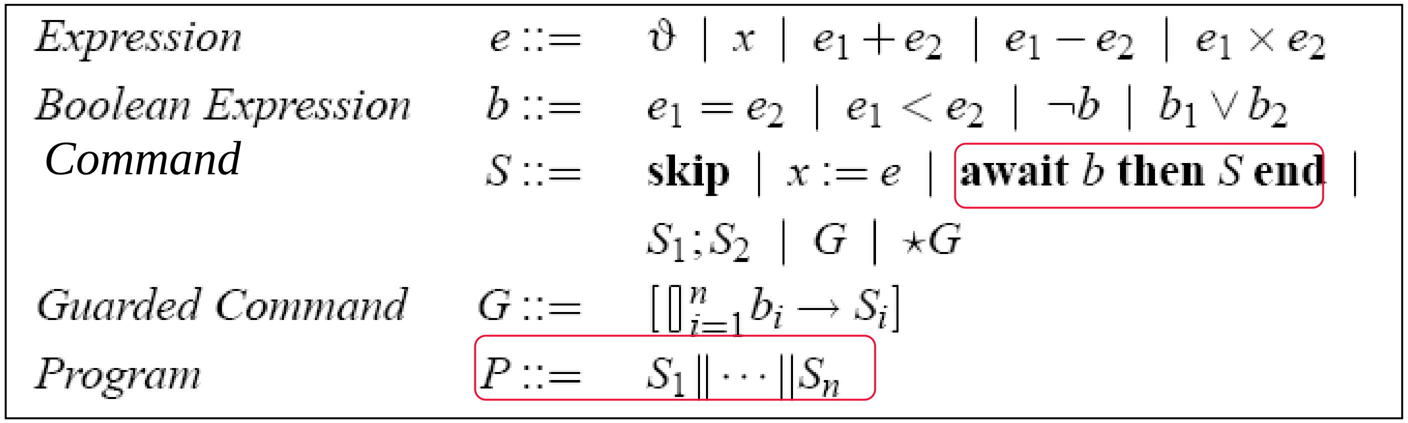
- Non-deterministic loop

$$\frac{\vdash P \Rightarrow I \qquad \vdash \forall i=1,n : \{I \wedge b_i\}\ S_i\ \{I\} \qquad \vdash (I \wedge \neg\, b_G) \Rightarrow Q}{\vdash \{P\}\ \text{do}\ \ \{I\}\ [\Box^n{}_{i=1}\, b_i \rightarrow S_i\,]\ \text{od}\ \ \{Q\}}$$

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

$\qquad I$ - *invariant*

Parallel programming language with shared variables



Expression	$e ::=$	$\vartheta \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Boolean Expression	$b ::=$	$e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
Command	$S ::=$	$\mathbf{skip} \mid x := e \mid \boxed{\mathbf{await}\ b\ \mathbf{then}\ S\ \mathbf{end}} \mid$
		$S_1 ; S_2 \mid G \mid \star G$
Guarded Command	$G ::=$	$[[]_{i=1}^{n} b_i \rightarrow S_i]$
Program	$\boxed{P ::=}$	$\boxed{S_1 \| \cdots \| S_n}$

Parallel programming language with shared variables

- **await** b **then** S **end** is blocked in any state in which b evaluates to false. If b evaluates to true then S can be executed atomically, i.e., without being interrupted by other components.

- $S_1 \| \cdots \| S_n$ indicates parallel execution of the *commands* $S_1, \ldots, S_n$. The components $S_1, \ldots, S_n$ of a parallel composition are often called *processes*.

- **await** $b \equiv$ **await** b **then skip end**
- $< S > \equiv$ **await** $true$ **then** S **end**

The brackets $< \cdots >$ are sometimes called "Lamport brackets" and $< S >$ is also called a "bracketed section" or "atomic region".

Execution model: atomicity and interleaving

- An assignment $x := e$ is executed *atomically*, that is, during its execution other parallel processes may not change x or the variables occurring in e.
- For an await statement **await** b **then** S **end** we assume that S is executed atomically in a state where b holds.
- Concurrent processes proceed asynchronously. No assumptions are made about the relative speed at which processes execute their actions.

Interleaving semantics*: only one atomic action of one of the processes that is not in the waiting state is executed at a time. It is called interleaving of atomic actions.*

What is the value of x after the execution of the following program?

(x := 0; x := x +2) || (x := 1; x := x +3)

It can be either 2, 4, 5 or 6.

Interference of processes

- Annotation specifies the constraint what program variables have to satisfy when the execution has reached the place/state where the annotation is written.
- It is difficult to locate the place for annotations in parallel programs because the global annotations should take into account all possible interleavings.
- It is not enough to prove the correcness of processes locally.
- Local annotations suffice only if we can prove that other processes do not violate the validity of assertions in the process.

Interference freedom

<u>Definition:</u>

The annotated triples $\{p_i\}\ AS_i\ \{q_i\}$ $i = 1, \ldots, n$ are *interference free* iff for all $i, j \in \{1, \ldots, n\}$, $i \neq j$, and for every assertion r in any $\{p_j\}\ AS_j\ \{q_j\}$ we have that if S_i is either a command

```
x := e
```

or

```
await b then S0 end
```

with precondition r_i in parallel process $\{p_i\}\ AS_i\ \{q_i\}$ then

$\{r_i \wedge r\}\ S_i\ \{r\}$.

SVL parallel composition rule

$$\frac{A_1 \vdash \{P_1\}\ S_1\ \{Q_1\} \quad A_2 \vdash \{P_2\}\ S_2\ \{Q_2\} \quad \vdash P \Rightarrow P_1 \wedge P_2 \quad \vdash Q_1 \wedge Q_2 \Rightarrow Q \quad IFT(S_1 \| S_2)}{\vdash \{P\}[\{P_1\}\ S_1\ \{Q_1\} \parallel \{P_2\}\ S_2\ \{Q_2\}]\ \{Q\}}$$

$IFT(S_1 \| S_2)$ – processes S_1 and S_2 are Interference Free

IFT - Interference Freedom Test

Interference freedom test (*IFT*):

- Let $S_1 \parallel S_2$.
- For each pair of annotated assignments $\{P_1\}\ V_1{:=}E_1\ \{Q_1\}$ and $\{P_2\}\ V_2{:=}E_2\ \{Q_2\}$ where $\{P_1\}\ V_1{:=}E_1\ \{Q_1\} \in \mathrm{A}(S_1)$ and $\{P_2\}\ V_2{:=}E_2\ \{Q_2\} \in \mathrm{A}(S_2)$, interference test consists of 4 proof obligations (where $\mathrm{A}(S)$ denotes annotated program S):
- S_1 does not violate the local precondition P_2 of S_2:

$$\{P_1 \wedge P_2\}\ V_1 := E_1\ \{P_2\}$$

- S_1 does not violate the local postcondition Q_2 of S_2:

$$\{P_1 \wedge Q_2\}\ V_1 := E_1\ \{Q_2\}$$

- S_2 does not violate the local precondition P_1 of S_1:

$$\{P_2 \wedge P_1\}\ V_2 := E_2\ \{P_1\}$$

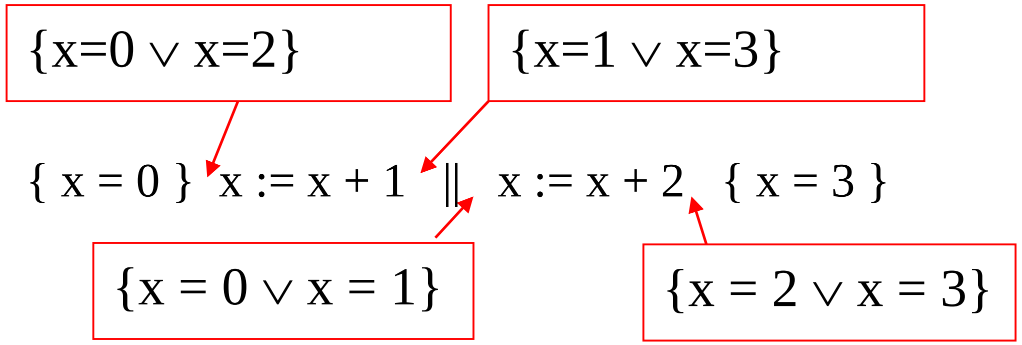
- S_2 does not violate the local postcondition Q_1 of S_1:

$$\{P_2 \wedge Q_1\}\ V_2 := E_2\ \{Q_1\}$$

Example

Prove that $\{ x = 0 \}\ x := x + 1 \ \|\ x := x + 2\ \{ x = 3 \}$

Add the annotations:



The global precondition implies the local preconditions of the processes and the local postconditions imply the global postcondition:

$\vdash\ (x = 0) \Rightarrow (x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)$

$\vdash\ (x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3) \Rightarrow (x = 3)$

Each process has local specification:

$\vdash \{x = 0 \vee x = 2\}\ x := x + 1\ \{x = 1 \vee x = 3\}$

$\vdash \{x = 0 \vee x = 1\}\ x := x + 2\ \{x = 2 \vee x = 3\}$

Example: interference test

$\boxed{\{x = 0 \lor x = 2\}}$ $\boxed{\{x = 1 \lor x = 3\}}$

$\{ x = 0 \}$ `x := x + 1   ||   x := x + 2` $\{ x = 3 \}$

$\boxed{\{x = 0 \lor x = 1\}}$ $\boxed{\{x = 2 \lor x = 3\}}$

$P1$ does not interfere to $P2$ local precondition

$$\vdash \{(x = 0 \lor x = 2) \land (x = 0 \lor x = 1)\}\ x := x + 1 \{ x = 0 \lor x = 1\}$$

$P1$ does not interfere to $P2$ local postcondition

$$\vdash \{(x = 0 \lor x = 2) \land (x = 2 \lor x = 3)\}\ x := x + 1 \{ x = 2 \lor x = 3\}$$

$P2$ does not interfere to $P1$ local precondition

$$\vdash \{(x = 0 \lor x = 1) \land (x = 0 \lor x = 2)\}\ x := x + 2 \{ x = 0 \lor x = 2\}$$

$P2$ does not interfere to $P1$ local postcondition

$$\vdash \{(x = 0 \lor x = 1) \land (x = 1 \lor x = 3)\}\ x := x + 2 \{ x = 1 \lor x = 3\}$$

A problem

We cannot prove

$$\vdash \{ x = 0 \}\ x := x + 1 \ \|\ x := x + 1 \ \{ x = 2 \}$$

because VCs

$$\vdash \{(x = 0 \vee x = 1)\}\ x := x + 1 \{ x = 1 \vee x = 2\}$$
$$\vdash \{(x = 0 \vee x = 1)\}\ x := x + 1 \{ x = 1 \vee x = 2\}$$

are not interference free, i.e:

$$\nvdash \{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 1)\}\ x := x + 1 \{ x = 0 \vee x = 1\}$$

and the conjunction of local postconditions does no imply postcondition

$$\nvdash \{ x = 1 \vee x = 2\} \wedge \{ x = 1 \vee x = 2\} \Rightarrow \{ x = 2 \}$$

Intermediate remarks

- Proving the properties of parallel programs is computationally hard
  - There is an exponential number of verification conditions (VC) one for every command in all processes for every assignment
  - Given proof method is not compositional for parallel composition
  - Not possible to verify ||-composition of processes knowing only the pre- and postconditions of the local processes.
- If the specification is proved for the whole parallel program then it is possible to compose it sequentially to other programs looking only at pre- and postcondition.

Message passing parallel programs

We have studied the formal (syntactic and) verification of

- non-deterministic programs – generalization of deterministic sequential programs;
- parallel programs with shared variables - an abstraction of multi-threaded programs (in multiprocessor computer).

We will look next parallel programs with message passing - an abstraction of distributed (networked) programs.

Communication primitives of parallel programs

- We have communication primitives $C!e$ and $C?x$ sending a value to channel C and reading the value from C.

Notations

$C \in CHANNEL$;

e – arithmetic expression on the local variables of the process;

x – local variable;

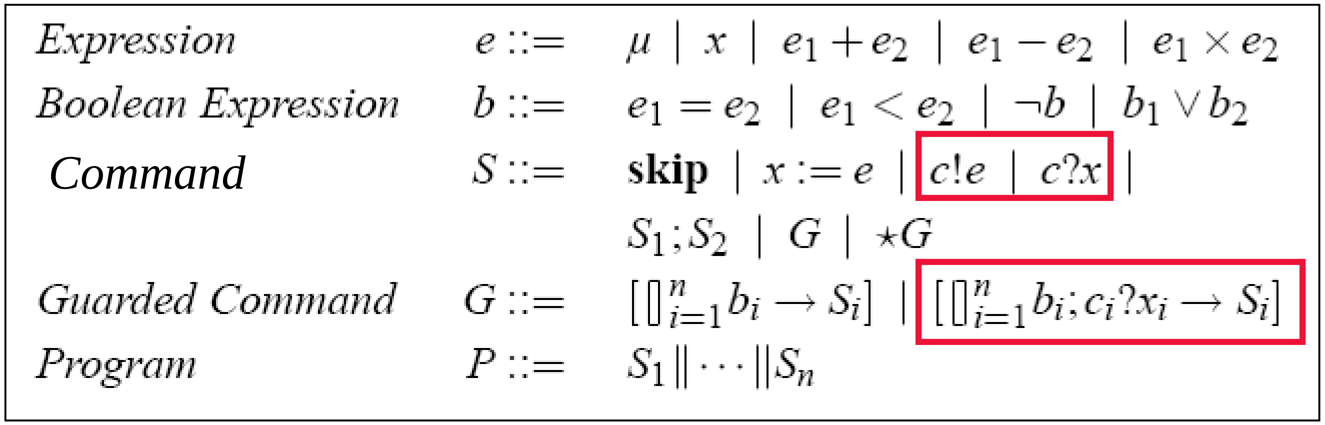
$C!e$ – the value of an expression e is sent to channel C;

$C?x$ – a value is read from channel C and assigned to variable x.

Synchrony!

Commands $C!e$ and $C?x$ are executed synchronously.

Parallel programs with message passing



Expression	$e ::=$	$\mu \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2$
Boolean Expression	$b ::=$	$e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
Command	$S ::=$	$\mathbf{skip} \mid x := e \mid \boxed{c!e \mid c?x} \mid$
		$S_1 ; S_2 \mid G \mid \star G$
Guarded Command	$G ::=$	$[\Box_{i=1}^{n} b_i \rightarrow S_i] \mid \boxed{[\Box_{i=1}^{n} b_i ; c_i?x_i \rightarrow S_i]}$
Program	$P ::=$	$S_1 \parallel \cdots \parallel S_n$

We use here sligthly different notation

$\texttt{if } []^{n}_{i=1}\, b_i \rightarrow S_i \texttt{ fi} \quad ⇨ \quad []^{n}_{i=1}\, b_i \rightarrow S_i$

$\texttt{do } []^{n}_{i=1}\, b_i \rightarrow S_i \texttt{ od} \quad ⇨ \quad \star([]^{n}_{i=1}\, b_i \rightarrow S_i)$

Communication commands

- Output command $c!e$ is used to send the value of expression e on channel c as soon as a corresponding input command is available. Since we assume synchronous communication, such an output command is suspended until a parallel process executes an input command $c?x$.
- Input command $c?x$ is used to receive a value via channel c and assign this value to the variable x. As for the output command , such an input statement has to wait for a corresponding partner before a (synchronous) communication can take place.

Parallel programs with message passing

- Guarded command $[\|_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i]$. A guard (the part before the arrow) is *open* if its boolean part evaluates to true. If none of the guards is open, the guarded command terminates after evaluation of the booleans. Otherwise, wait until the communication of one of the open guards can be performed and continue with the corresponding S_i.
- $S_1\|\cdots\|S_n$ indicates parallel execution of the commands $S_1,\ldots,S_n$. The components $S_1,\ldots,S_n$ of a parallel composition are often called *processes*.

For a guarded command $G \equiv [\|_{i=1}^{n} b_i \rightarrow S_i]$ or $G \equiv [\|_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i]$, we define $b_G \equiv b_1 \vee \ldots \vee b_n$. An io-guard $b_i; c_i?x_i$ is often shortened to $c_i?x_i$ if $b_i \equiv \mathit{true}$.

Syntactic restrictions

- For $S_1; S_2$ we require that if S_1 contains $c!e$ then S_2 does not contain $c?x$, and if S_1 contains $c?x$ then S_2 does not contain $c!e$.
- For $[\Box_{i=1}^{n}\, b_i \to S_i]$ we require that, for all $i, j \in \{1, \ldots, n\}$, $i \neq j$, if S_i contains $c!e$ then S_j does not contain $c?x$.
- For $[\Box_{i=1}^{n}\, b_i; c_i?x_i \to S_i]$ we require that, for all $i, j \in \{1, \ldots, n\}$, S_j does not contain $c_i!e$, and if S_i contains $c!e$ then, for $j \neq i$, S_j does not contain $c?x$.
- For $S_1 \| \cdots \| S_n$ we require that, for all $i, j \in \{1, \ldots, n\}$, $i \neq j$, if S_i contains $c!e_1$ then S_j does not contain $c!e_2$, and if S_i contains $c?x_1$ then S_j does not contain $c?x_2$.

Parallel processes do not share program variables!

- For $S_1 \| \cdots \| S_n$, we require, for all $i, j \in \{1, \ldots, n\}$, $i \neq j$, that $var(S_i) \cap var(S_j) = \emptyset$.

Recall: proof method for parallel programs with shared variables

- The method of Owicki and Gries
  - First, a *local* correcness proof is given for each process
  - a consitency check - *interference* test is applied to the local proofs.
- Similar two-stage method is applicable to parallel processes with message passing, but the *cooperation* tests are verified instead of *interference* tests

DML parallel composition

$$\frac{A_1 \vdash \{P_1\} S_1 \{Q_1\} \quad A_2 \vdash \{P_2\}\ S_2\ \{Q_2\} \quad \vdash P \Rightarrow P_1 \wedge P_2 \quad \vdash Q_1 \wedge Q_2 \Rightarrow Q \quad Coop(A_1\ A_2)}{\vdash \{P\}[\{P_1\}\ S_1\ \{Q_1\} \parallel \{P_2\} S_2\ \{Q_2\}]\{Q\}}$$

non-deterministic choice

$$\frac{\forall i = 1, l{:}\ \ A_i \vdash \{P\}\ S_i\ \{Q\},}{A \vdash \{P\}\ \ [\Box^{l}{}_{i=1}\ S_i]\ \ \{Q\} \quad A =_{def} \cup^{l}{}_{i=1} A_i}$$

Cooperation test $Coop(A_1\ A_2)$

$Coop(A_1\ A_2)$ establishes the validity of sets of axioms A_1 and A_2 about the communication correctness:

Assuming:

- there is a matching pair of communication operations over channel C, i.e.

$C!\ E$ and $C?v$ where E is an expression and v is a variable, the matching pair has local pre- and post-conditions:

- S_i: ... $\{P_i'\}\ C!E\ \{Q_i'\}$.... and
- S_j: ... $\{P_j''\}\ C?v\ \{Q_j''\}$... respectively,

then the test $Coop()$ for this pair means proving the validity of tripple

- $\vdash \{P_i' \wedge P_j''\}\ v{:=}\ E\ \{Q_i' \wedge Q_j''\}$. (*)

When the tripple (*) is proved correct then $\{P_i'\}\ C!E\ \{Q_i'\}$ and $\{P_j''\}\ C?v\ \{Q_j''\}$ are treated respectively as axioms $a^i{}_k \in A_i$ and $a^j{}_k \in A_j$ in the local proofs of processes S_i and S_j where these tripples occur.

Example

$$\{y=3\}\ (c?x\,;\,x:=x+1\,;\,d!(x+2))\ \|\ (c!y\,;\,d?y\,;\,y:=y+2)\ \{x=4\wedge y=8\}.$$

$\{true\}\ c?x;\ \{x=3\}\ x:=x+1;\ \{x=4\}\ d!(x+2)\ \{x=4\}$, *and*
$\{y=3\}\ c!y;\ \{y=3\}\ d?y;\ \{y=6\}\ y:=y+2\ \{y=8\}$.

local annotations

cooperation tests

- *for the pair* $\{true\}\ c?x;\ \{x=3\}$ *and* $\{y=3\}\ c!y;\ \{y=3\}$ *we have to prove* $\{true\wedge y=3\}\ \boxed{x:=y}\ \{x=3\wedge y=3\}$, *and*
- *for the pair* $\{x=4\}\ d!(x+2)\ \{x=4\}$ *and* $\{y=3\}\ d?y;\ \{y=6\}$ *we have to show* $\{x=4\wedge y=3\}\ y:=x+2\ \{x=4\wedge y=6\}$.

parallel composition

$$\{y=3\}\ (c?x\,;\,x:=x+1\,;\,d!(x+2))\ \|\ (c!y\,;\,d?y\,;\,y:=y+2)\ \{x=4\wedge y=8\}.$$

Example of using auxiliary variables (for identifying matching pairs)

$\{true\}\ (c!1;c!2)\|(c?x;c?x)\ \{x=2\}$

$\{k=0\}\ <c!1;k:=k+1>;\ \{k=1\}\ <c!2;k:=k+1>\ \{k=2\}$

$\{k=0\}\ c?x;\ \{k=1\}\ c?x\ \{k=2\wedge x=2\}.$

- *for* $\underbrace{\{k=0\}\ <c!1;k:=k+1>\ \{k=1\}}$ *and* $\underbrace{\{k=0\}\ c?x\ \{k=1\}}$ *we can prove* $\{k=0\}\ x:=1;k:=k+1\ \{k=1\}$, *and*
- *for* $\underbrace{\{k=1\}\ <c!2;k:=k+1>\ \{k=2\}}$ *and* $\underbrace{\{k=1\}\ c?x\ \{k=2\wedge x=2\}}$ *we have* $\{k=1\}\ x:=2;k:=k+1\ \{k=2\wedge x=2\}$.

- *for* $\underbrace{\{k=0\}\ <c!1;k:=k+1>\ \{k=1\}}$ *and* $\underbrace{\{k=1\}\ c?x\ \{k=2\wedge x=2\}}$, *we have* $\{k=0\wedge k=1\}\ x:=1;k:=k+1\ \{k=1\wedge k=2\wedge x=2\}$, *and*
- *for* $\{k=1\}\ <c!2;k:=k+1>\ \{k=2\}$ *and* $\{k=0\}\ c?x\ \{k=1\}$ *we can prove* $\{k=1\wedge k=0\}\ x:=2;k:=k+1\ \{k=2\wedge k=1\}$.

Assignment

Show that

$$\{true\}\ S_1 \,||\, S_2 \,||\, S_3\ \{x = u\},$$

where

$S_1 \equiv C!x,$

$S_2 \equiv C?y;\ D!y$

$S_3 \equiv D?u$

Assignment

- Let R and T be nonempty sets of natural numbers. Consider the following partitioning algorithm $S_1 \| S_2$, where

$$
\begin{aligned}
S_1 \equiv\ & max := max(R); c?mn; d!max;\\
& \star[max > mn \rightarrow R := (R \setminus \{max\}) \cup \{mn\}; max := max(R);\\
& \qquad\qquad c?mn; d!max]
\end{aligned}
$$

$$
\begin{aligned}
S_2 \equiv\ & min := min(T);\ c!min; d?mx;\\
& \star[mx > min \rightarrow T := (T \setminus \{min\}) \cup \{mx\}; min := min(T);\\
& \qquad\qquad c!min; d?mx]
\end{aligned}
$$

Prove, by means of the method of Levin & Gries,

$$\{R = R_0 \neq \emptyset \wedge T = T_0 \neq \emptyset \wedge R \cap T = \emptyset\}\ \ S_1 \| S_2$$
$$\{|R| = |R_0| \wedge |T| = |T_0| \wedge R \cup T = R_0 \cup T_0 \wedge max(R) < min(T)\}$$

where, for a set A, $|A|$ denotes the number of elements of A, and R_0 and T_0 are logical variables denoting a finite set of natural numbers.