Disjoint Conditions

CS 536: Science of Programming, Spring 2022

A. Why?

- Combining arbitrary threads are in parallel can yield programs that have surprisingly different results from how each thread works when run sequentially.
- When threads don't interfere with each other's work, we can combine them into a parallel program without worrying about these strange and unexpected kinds of behavior.
- Simply having disjoint parallel programs for threads isn't sufficient to avoid interference problems.
- What's needed are disjoint conditions, which ensure that no thread can interfere with the conditions of another thread.

B. Objectives

After this class, you should know

- What disjoint conditions are, why we need them, and how to recognize them.
- What the disjoint parallelism rule for disjoint parallel programs with disjoint conditions allows.
- That maybe not all apparent interference actually causes problems.

C. Parallelism Rule for Parallel Programs?

- The *sequentialization proof rule* for DPPs lets us reason about DPPs, which is nice, but it has two major flaws.
- First, It requires disjoint parallel programs, and we know disjoint parallelism is a strong constraint on what programs we can write.
- Second, to use it sequentialization, we need many intermediate conditions. To prove $\{p\}\ [s_1 \| \ldots \| s_n]\ \{q\}$, we need to prove $\{p\}\ s_1; \ldots ; s_n\ \{q\}$, which (if we use *wp*) means finding a sequence of preconditions q_1, ..., q_n and proving $\{p\}\ \{q_n\}\ s_1; \{q_{n-1}\}\ s_2\ ; \{q_{n-2}\} \ldots \{q_1\}\ S_n\ \{q\}$.

  The proofs of q_1, q_2, ... q_n can get increasingly complicated because each q_i can depend on all the threads and conditions to its right.
- *Parallelism Rule*: Ideally, we'd like to take advantage not only of parallel execution of programs but also parallel writing of programs. Can we have *n* programmers write *n* sequential threads and then combine them into a parallel program? This behavior describes a *parallelism rule*. For two threads, the form would be as follows (it generalizes to *n* threads).

$$\frac{\{p_1\}\, s_1\, \{q_1\} \qquad \{p_2\}\, s_2\, \{q_2\}}{\{p_1 \wedge p_2\}\, [s_1 \parallel s_2]\, \{q_1 \wedge q_2\}}\,(Par\ ?)$$

In proof outline format, we write this as

```
                                {p₁ ∧ p₂}
[{p₁} s₁ {q₁} ‖ {p₂} s₂ {q₂}]   {q₁ ∧ q₂}.
```

- *Example 1*: In Hilbert style, we write
  1. $\{x \geq 0\}\ z := x\ \{z \geq 0\}$
  2. $\{y \leq 0\}\ w := -y\ \{w \geq 0\}$
  3. $\{x \geq 0 \wedge y \leq 0\}\ [z := x \parallel w := -y]\ \{z \geq 0 \wedge w \geq 0\}$??? parallelism 1, 2

  The proof outline form is

```
                        {x ≥ 0 ∧ y ≤ 0}
[                       {x ≥ 0}
  z := x                {z ≥ 0}
‖                       {y ≤ 0}
  w := –y               {w ≥ 0}
]                       {z ≥ 0 ∧ w ≥ 0}
```

- It's nice when this pattern works, but we'll see examples where the pattern produces incorrect programs. To get correctness, we have to impose side conditions that limit how we can combine threads. Altogether, we'll look at a couple of different kinds of side conditions, which work in different situations.

- *Example 2*: Here are some outlines where using a parallelism rule works.

  a) $\{z=0\}\ [\{z=0\}\ x := z+1\ \{x \leq z=0\} \parallel \{z=0\}\ y := z\ \{y = z=0\}\ ]$
  $\{x \geq z=0 \wedge y = z=0\}$

  b) $\{T\}\ [\{T\}\ a:=x+1\ \{a=x+1\} \parallel \{T\}\ b:=x+2\ \{b=x+2\}]$
  $\{a=x+1 \wedge b=x+2\} => \{a+1=b\}$

  c)
```
{x=y=z=c}[{x=c} x := x²        {x=c²}
‖ {y=c} y := y²                {y=c²}
‖ {z=c} z := (z–d)*(z+d)       {z=c²–d²}
] {x=c² ∧ y=c² ∧ z=c²–d²}              {x = y = z+d²}
```

- *Example 3:* Here are some outlines where using a parallelism rule fails. Outline *(a)* is incorrect because it leaves $x \neq y$. Outline *(b)* doesn't always leave $x > y$ (e.g., if we start with $x = y = 2$). Outline *(c)* never works.
  a) $\{x = 0\}\ [\{x = 0\}\ x := 1\ \{x = 1\} \parallel \{x = 0\}\ y := 0\ \{x = y\}]\ \{x = 1 \wedge x = y\}$
  b) $\{x \geq y \wedge y = z\}\ [\{x \geq y\}\ x := x+1\ \{x > y\} \parallel \{y = z\}\ y := y*2; z := z*2\ \{y = z\}]\ \{x > y \wedge y = z\}$
  c) $\{T\}\ [\{T\}\ [x := 2+2\ \{x = 2+2\} \parallel \{T\}\ x := 5\ \{x = 5\}]\ \{x = 2+2 \wedge x = 5\}\ \{2+2 = 5\}$

Disjoint Conditions

- Why do the proof outlines in *Example 3* fail? Outline *(c) should* fail because the semantics don't support it. But outlines *(a)* and *(b)* fail even though they have disjoint parallel programs.
- Looking more closely, we see outline *(a)* fails because thread 1 sets x to *1*, which invalidates the precondition $x = 0$ needed by thread 2. Similarly, outline *(b)* fails because thread 2 modifies y, which might invalidate the $x \geq y$ and $x > y$ pre- and postconditions in thread 1.
- It's not enough for outlines *(a)* and *(b)* to have disjoint parallel programs, i.e., it's not enough to guarantee that threads don't change each others' states. For correctness, we need each thread to not modify the variables that appear in the *conditions* of other threads.
- *Notation*: FV*(p, ..., q) ("free variables")* is the set of variables that appear free in any of the predicates p, ..., q. (Recall that a variable can have both free and bound occurrences, "bound" meaning "in the scope of a quantifier" for it. If a variable has at least one free occurrence in a predicate, then it is free in that predicate, regardless of whether or not there are bound occurrences.)
- *Definition*: With $\{p_1\}\ s_1\ \{q_1\}$ and $\{p_2\}\ s_2\ \{q_2\}$, the first triple *interferes with the conditions* of the second triple if $Change(s_1) \cap FV(p_2, q_2) \neq \emptyset$. We may use *disjoint from the conditions* as a synonym for "not interfering with." The two triples have *disjoint conditions* if neither interferes with the conditions of the other. (I.e., $Change(s_1) \cap FV(p_2, q_2) = \emptyset$ and $Change(s_2) \cap FV(p_1, q_1) = \emptyset$.)
- *Definition*: A parallel program outline $\{p\}\ [\{p_1\}\ S_1\ \{q_1\} \parallel ... \parallel \{p_n\} S_n \{q_n\}]\ \{q\}$ has *disjoint conditions* if its threads have pairwise disjoint conditions. i.e., threads i and j have disjoint conditions, for all indexes i and j (with $i \neq j$).

- *Example 4:* All of the outlines in *Example 2* have pairwise disjoint programs and conditions. To show this, we can add a Disjoint Conditions column to the table we used for checking for disjoint programs. (Note that not all the sequential thread outlines are full outlines.)

a. $\{z=0\}$ $[\{z=0\}\ x := z+1\ \{x \geq z=0\} \parallel \{z=0\}\ y := z\ \{y = z=0\}\ ]$
$\{x \geq z=0 \wedge y = z=0\}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	*x*	*y z*	*y z*	*Y*	*Y*
2	1	*y*	*x z*	*x z*	*Y*	*Y*

b. $\{T\}$ $[\{T\}\ a:=x+1\{a=x+1\} \parallel \{T\}\ b:=x+2\ \{b=x+2\}]$
$\{a=x+1 \wedge b=x+2\} \Rightarrow \{a+1=b\}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	*a*	*b x*	*b x*	*Y*	*Y*
2	1	*b*	*a x*	*a x*	*Y*	*Y*

c. $\{x=y=z=c\}$
$[\{x=c\}\ x := x^2\ \{x=c^2\}$
$\parallel \{y=c\}\ y := y^2\ \{y=c^2\}$
$\parallel \{z=c\}\ z := (z-d)*(z+d)\ \{z=c^2-d^2\}$
$]\ \{x=c^2 \wedge y=c^2 \wedge z=c^2-d^2\}$
$\{x = y = z+d^2\}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	*x*	*y*	*c y*	*Y*	*Y*
1	3	*x*	*d z*	*c d z*	*Y*	*Y*
2	1	*y*	*x*	*c x*	*Y*	*Y*
2	3	*y*	*d z*	*c d z*	*Y*	*Y*
3	1	*z*	*x*	*c x*	*Y*	*Y*
3	2	*z*	*y*	*c y*	*Y*	*Y*

- *Example 5:* All of the outlines in *Example 4* lack pairwise disjoint conditions. Outlines *(a)* and *(b)* have disjoint programs but outline *(c)* does not.

a. $\{x = 0\}\ [\{x = 0\}\ x := 1\ \{x = 1\} \parallel \{x = 0\}\ y := 0\ \{x = y\}]\ \{x = 1 \wedge x = y\}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	*x*	*y*	*x y*	*Y*	*N*
2	1	*y*	*x*	*x*	*Y*	*Y*

b. $\{x \le y \wedge y = z\}\ [\{x \le y\}\ x := x–1\ \{x < y\} \parallel \{y = z\}\ y := y*2; z := z*2\ \{y = z\}]$ $\{x < y \wedge y = z\}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	*x*	*y z*	y z	*Y*	*Y*
2	1	*y z*	*x*	*x y*	*Y*	*N*

c. $\{T\}\ [\{T\}\ x := 2+2\ \{x = 2+2\} \parallel \{T\}\ x := 5\ \{x = 5\}]\{x = 2+2 \wedge x = 5\}\{2+2 = 5\}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	*x*	*x*	*x*	*N*	*N*
2	1	*x*	*x*	*x*	*N*	*N*

Disjoint Conditions; Disjoint Parallelism Rule

- If two outlines have disjoint programs and conditions, then their evaluations can be arbitrarily interleaved without fear. Having disjoint programs guarantees that no thread modifies the parts of the state used by the calculations of other threads. Having disjoint conditions guarantees that no thread modifies the parts of the state used to determine satisfaction of other threads' conditions.
- Because of this lack of runtime interference, we can use a a parallelism rule for disjoint parallel programs with disjoint conditions. The rule has the form of a parallelism rule (the parallel program uses the conjunctions of its threads' preconditions and postconditions as its precondition and postcondition). Having disjoint programs and conditions is the side condition that guarantees correctness of the rule.

Disjoint Parallelism Rule

$$\frac{\{p_1\}s_1\{q_1\}\{p_2\}s_2\{q_2\}...\{p_n\}s_n\{q_n\}\ 1,2,...,n \text{ pairwise disjoint w/ disjoint conditions}}{\{p_1 \wedge p_2 \wedge ... \wedge p_n\}[s_1 \parallel s_2 \parallel ... \parallel s_n]\{q_1 \wedge q_2 \wedge ... \wedge q_n\}}$$

Example 6: The outlines from Example 2 all have disjoint programs and conditions, so all of them can be proved using the disjoint parallelism rule. The presentations in Example 2 illustrate the uses of the rule.

Removing Interference of Conditions

- Sometimes, interference with conditions can be removed by adding logical (ghost) variables.
- *Example 7*: All of the outlines from Example 3 have interfering conditions, but we can remove the interference for outlines *(a)* and *(b)*. Since *(c)* doesn't have disjoint programs, modifying its conditions won't help.

a. Original: *{x = 0} [{x = 0} x := 1 {x = 1} ‖ {x = 0} y := 0 {x = y}] {x = 1 ∧ x = y}.*
The interference is that thread 1 setting *x* to *1* interferes with the *x* in *x = y* in thread 2. Said the other way, the *x* in *x = y* refers to the value of x before *x := 1*. Introducing a ghost variable for the initial value of x takes care of that. We also have to change the final postcondition to get one that's true.

Without interference:

{x = x_0 ∧ x = 0}
[{x = x_0 ∧ x = 0} x := 1 {x = 1 ∧ x_0 = 0}
‖ {x = x_0 ∧ x = 0} y := 0 {x = x_0 ∧ x = 0 ∧ y = 0} => {x_0 = y}]
{x = 1 ∧ x_0 = 0 ∧ x_0 = y} => {x = 1 ∧ x–1 = y}

b. Original: *{x ≤ y ∧ y = z} [{x ≤ y} x := x–1 {x < y}*
*‖ {y = z} y := y*2; z := z*2 {y = z}] {x < y ∧ y = z}.*
The interference is *y := y*2* in thread 2 versus *x ≤ y* and *x < y* in thread 1. Again, introducing ghost variables helps.

Without interference:

$\{x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge x \le y \wedge y = z\}$
$[\ \{x = x_0 \wedge x \le y_0\}$
$\quad x := x-1$
$\quad \{x_0 \le y_0 \wedge x = x_0-1\}$
$\|\ \{y = y_0 \wedge z = z_0 \wedge y = z\}$
$\quad y := y*2; z := z*2$
$\quad \{y_0 = z_0 \wedge y = y_0*2 \wedge z = z_0*2\}$
$]$
$\{(x_0 \le y_0 \wedge x = x_0-1) \wedge (y_0 = z_0 \wedge y = y_0*2 \wedge z = z_0*2)\}$
$\Rightarrow \{x < y \div 2 \wedge y = z\}$

- *Note:* This is another case, like with loop invariants, where you have to include intermediate conditions (i.e., not preconditions or postconditions) in a minimal proof outline: we can't infer the stronger preconditions of the threads by just using wp/sp.

D. *Not All Changes are Interference*

- We'll look into this in more detail next class, but can we loosen the requirement of disjoint conditions? (After all, requiring disjoint threads already disallows too many programs we want to write.)
- The basic observation is that often we can change the values of variables without changing whether a condition is satisfied or not. e.g., the condition $x > 0$ isn't invalidated by setting $x := 1$ or $x := x+1$ (or $x := x$, for that matter).
- Say a thread includes a condition p (either as a precondition or postcondition; it doesn't matter), and a different thread is about to execute $\{q_1\}\ x := e\ \{q_2\}$. Surprisingly, q_2 isn't important; what is important is that executing $x := e$ only causes harm if it makes p false. More precisely, if p and q_1 hold and we execute $x := e$, we don't have interference if p is still satisfied after the assignment.
- As long as $\{p \wedge q_1\}\ x := e\ \{p\}$ is valid, we know that whatever change $x := e$ causes, it won't disturb execution of the other thread, at least as far as p holding.
- More generally, with p and $\{q_1\}\ s_1\ \{...\}$ (we don't care about the postcondition of s_1), we're free of interference if $\{p \wedge q_1\}\ s_1\ \{p\}$ is valid.
- We can build on this observation to get a notion of interference-freedom that works as a side condition for a nicer parallelism rule. It's not trivial because a thread has multiple statements that we have to check against multiple conditions in the other

thread. But we can reduce the burden by looking more carefully at exactly what statements we need to check for interference.

- E.g., $x_1 := e_1; x_2 := e_2; x_3 := e_3$ has 5 statements (three assignments and two sequences). We'll have the check the assignments, of course, but do we need to check the two sequences? As it turns out, no, we don't.