

Sheep in wolf's clothing: Implementation models for dataflow multi-threaded software

Keryan Didier¹ Albert Cohen² Adrien Gauffriau³
Dumitru Potop-Butucaru¹

¹INRIA ²ENS & Google ³Airbus

ACSD'19 27 June 2019

Critical systems context

Threaded implementation

Posix threads

Non-determinism	-
Data races possible	-
Deadlocks possible	-
Asynchronous small-step semantics	-
Portability	
Flexibility	+++
General-purpose	



Critical systems context

Threaded implementation and dataflow synchronous specification

Posix threads		Dataflow synchronous formalisms
Non-determinism	-	+ Determinism
Data races possible	-	+ No data races
Deadlocks possible	-	+ No deadlocks
Asynchronous small-step semantics	-	+ Big-step semantics
Portability		
Flexibility	+++	
General-purpose		



Critical systems context

Threaded implementation and dataflow synchronous specification

Posix threads		Dataflow synchronous formalisms
Non-determinism	-	+ Determinism
Data races possible	-	+ No data races
Deadlocks possible	-	+ No deadlocks
Asynchronous small-step semantics	-	+ Big-step semantics
Portability		
Flexibility	+++	
General-purpose		



Our thesis : in practice, threaded implementations of dataflow specification preserve a fundamentally dataflow structure

Application

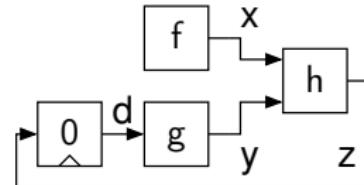
Dataflow synchronous specification in Lustre

```
fun f:()->(float)
fun g:(int)->(int)
fun h:(float,int)->(int)
var
    x : float; y : int; z : int; d : int;
let
    x = f();
    y = g(d);
    z = h(x,y);
    d = 0 fby z;
tel
```

Application

Dataflow synchronous specification in Lustre

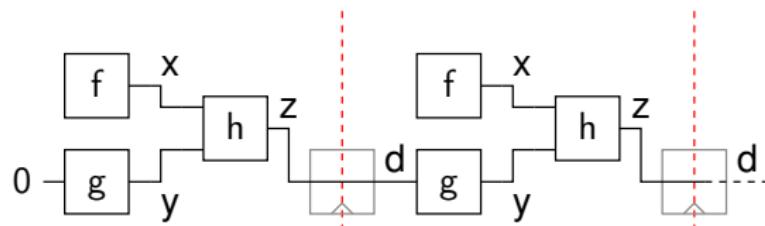
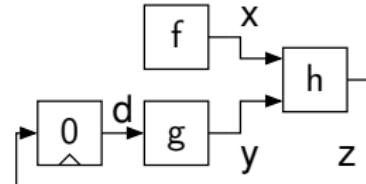
```
fun f:()->(float)
fun g:(int)->(int)
fun h:(float,int)->(int)
var
    x : float; y : int; z : int; d : int;
let
    x = f();
    y = g(d);
    z = h(x,y);
    d = 0 fby z;
tel
```



Application

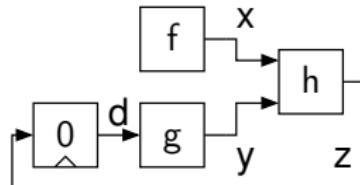
Dataflow synchronous specification in Lustre

```
fun f:()-(float)
fun g:(int)->(int)
fun h:(float,int)->(int)
var
    x : float; y : int; z : int; d : int;
let
    x = f();
    y = g(d);
    z = h(x,y);
    d = 0 fby z;
tel
```



Application

Two cores implementation (static allocation and scheduling)



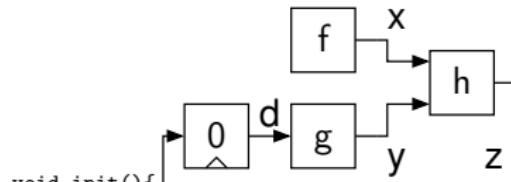
```
void thread_cpu0(){
    for(;;){
        f(&x);
        dcache_flush();

        mutex_lock(&m0);
        dcache_inval();
        h(x,y,&z);
        dcache_flush();
        mutex_unlock(&m1);
    }
}
```

```
void thread_cpu1(){
    for(;;){
        mutex_lock(&m1);
        dcache_inval();
        g(z,&y);
        dcache_flush();
        mutex_unlock(&m0);
    }
}
```

Application

Two cores implementation (static allocation and scheduling)

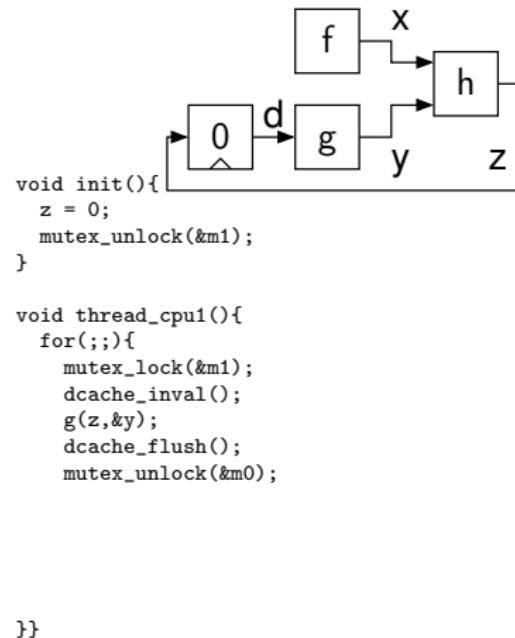


```
void init(){  
    z = 0;  
    mutex_unlock(&m1);  
}  
  
void thread_cpu0(){  
    for(;;){  
        f(&x);  
        dcache_flush();  
  
        mutex_lock(&m0);  
        dcache_inval();  
        h(x,y,&z);  
        dcache_flush();  
        mutex_unlock(&m1);  
    }  
}  
  
void thread_cpu1(){  
    for(;;){  
        mutex_lock(&m1);  
        dcache_inval();  
        g(z,&y);  
        dcache_flush();  
        mutex_unlock(&m0);  
    }  
}
```

Application

Two cores implementation (static allocation and scheduling)

```
ldscript fragment:  
x=0x22000; y=0x32000; z=0x22004; i=0x220064;  
stack0=0x30000; stack1=0x40000;  
.=0x20000; .bank2:{*(.text.cpu0);  
    .=0x100 ; *(.text.f) ;  
    .=0x500 ; *(.text.h) ;  
}  
.=0x30000; .bank3:{*(.text.cpu1);  
    .=0x200 ; *(.text.g) ;  
}  
  
void thread_cpu0(){  
    for(;;){  
        f(&x);  
        dcache_flush();  
  
        mutex_lock(&m0);  
        dcache_inval();  
        h(x,y,&z);  
        dcache_flush();  
        mutex_unlock(&m1);  
    }  
}
```



Application

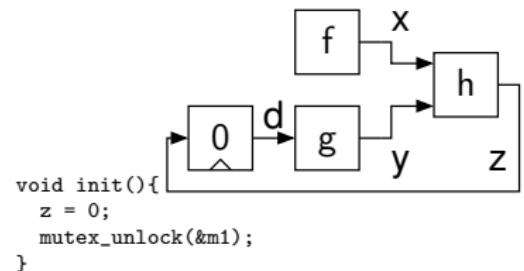
Two cores implementation (static allocation and scheduling)

```
ldscript fragment:  
x=0x22000; y=0x32000; z=0x22004; i=0x220064;  
stack0=0x30000; stack1=0x40000;  
.=0x20000; .bank2:{*(.text.cpu0);  
    .=0x100 ; *(.text.f) ;  
    .=0x500 ; *(.text.h) ;  
}  
.=0x30000; .bank3:{*(.text.cpu1);  
    .=0x200 ; *(.text.g) ;  
}
```

```
void thread_cpu0(){  
    for(;;){  
        f(&x);  
        dcache_flush();
```

```
        mutex_lock(&m0);  
        dcache_inval();  
        h(x,y,&z);  
        dcache_flush();  
        mutex_unlock(&m1);
```

```
    }}
```



```
void thread_cpu1(){  
    for(;;){  
        mutex_lock(&m1);  
        dcache_inval();  
        g(z,&y);  
        dcache_flush();  
        mutex_unlock(&m0);
```

```
    }}
```

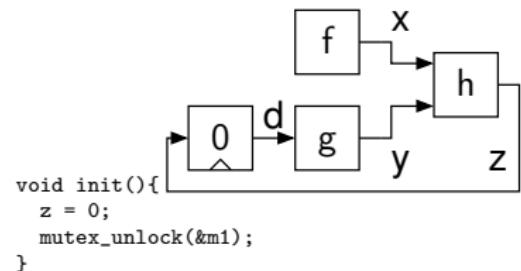
Application

Two cores implementation (static allocation and scheduling)

```
ldscript fragment:  
x=0x22000; y=0x32000; z=0x22004; i=0x220064;  
stack0=0x30000; stack1=0x40000;  
.=0x20000; .bank2:{*(.text.cpu0);  
    .=0x100 ; *(.text.f) ;  
    .=0x500 ; *(.text.h) ;  
}  
.=0x30000; .bank3:{*(.text.cpu1);  
    .=0x200 ; *(.text.g) ;  
}
```

```
void thread_cpu0(){  
    for(;;){  
        f(&x);  
        dcache_flush();
```

```
        mutex_lock(&m0);  
        dcache_inval();  
        h(x,y,&z);  
        dcache_flush();  
        mutex_unlock(&m1);  
    }}
```



```
void thread_cpu1(){  
    for(;;){  
        mutex_lock(&m1);  
        dcache_inval();  
        g(z,&y);  
        dcache_flush();  
        mutex_unlock(&m0);  
    }}
```

Dataflow representation of the implementation

Functional aspects

```
void init(){
    z = 0;
    mutex_unlock(&m1);
}

void thread_cpu0(){
    for(;;){
        f(&x);                                x = f()
        dcache_flush();
    }
}

void thread_cpu1(){
    for(;;){
        mutex_lock(&m1);
        dcache_inval();
        g(z,&y);                            y = g(d)
        dcache_flush();
        mutex_unlock(&m0);
    }
}

mutex_lock(&m0);
dcache_inval();
h(x,y,&z);                          z = h(x ,y )
dcache_flush();
mutex_unlock(&m1);
}}
```

The diagram illustrates the functional aspects of the implementation. It shows two threads, `thread_cpu0` and `thread_cpu1`, running in parallel. The `thread_cpu0` thread performs a computation $x = f()$ and then flushes the data cache. The `thread_cpu1` thread performs a computation $y = g(d)$, which depends on the value of d (which is initialized to 0). Both threads use mutex locks (`mutex_lock` and `mutex_unlock`) and invalidation functions (`dcache_inval` and `dcache_flush`) to ensure consistency. Red arrows indicate dependencies: one from the `z` assignment in `thread_cpu0` to the `y` assignment in `thread_cpu1`, and another from the `y` assignment in `thread_cpu1` back to the `z` assignment in `thread_cpu0`.

- ▶ Specification dataflow

Application

Two cores implementation (static allocation and scheduling)

```
void init(){
    z = 0;
    mutex_unlock(&m1);
}

void thread_cpu0(){
    for(;;){
        f(&x);
        dcache_flush();
        x0 = f()
        x = x0
    }
}

void thread_cpu1(){
    for(;;){
        mutex_lock(&m1);
        dcache_inval();
        g(z,&y);
        dcache_flush();
        mutex_unlock(&m0);
        d1 = d
        y1 = g(d1)
        y = y1
    }
}

mutex_lock(&m0);
dcache_inval();
h(x,y,&z);
dcache_flush();
mutex_unlock(&m1);
}}
```

The diagram illustrates the memory hierarchy representation for variable duplications. It shows two cores, CPU0 and CPU1, with their respective code snippets. The variable 'y' is declared in the CPU0 loop and is copied to 'y1' in the CPU1 loop via a red arrow, indicating a duplication of memory access.

- ▶ Memory hierarchy representation (variable duplications)

Application

Two cores implementation (static allocation and scheduling)

```
void init(){
    z = 0;
    mutex_unlock(&m1);
}

void thread_cpu0(){
    for(;;){
        f(&x);
        dcache_flush();
        x0 = f();
        x = x0
    }
}

void thread_cpu1(){
    for(;;){
        mutex_lock(&m1);
        dcache_inval();
        g(z,&y);
        dcache_flush();
        mutex_unlock(&m0);
        _ = u1
        d1 = d
        y1 = g(d1)
        y = y1
        v = top
    }
}
```

The diagram illustrates the execution flow between two threads, CPU0 and CPU1, over time steps. The code is divided into three main sections: the initial setup in void init(), the execution of thread_cpu0() by CPU0, and the execution of thread_cpu1() by CPU1.

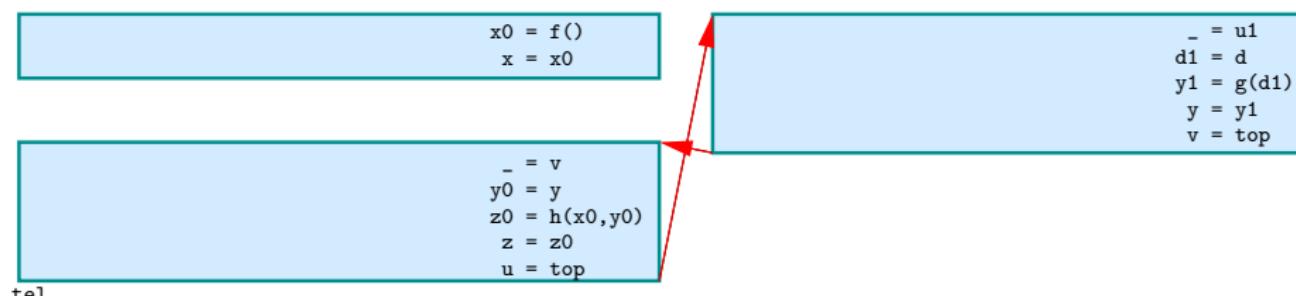
- Initial Setup:** The `init()` function initializes variable `z` to 0 and releases the mutex `m1`.
- Thread CPU0 Execution:** The `thread_cpu0()` function runs in a loop. It calls `f(&x)`, flushes the data cache, reads the result `x0` into `x`, and then updates `x0` to the result of `f()`.
- Thread CPU1 Execution:** The `thread_cpu1()` function runs in a loop. It locks the mutex `m1`, invalidates the data cache, calls `g(z, &y)`, flushes the data cache, releases the mutex `m0`, and then updates variables `_`, `d1`, `y1`, `y`, and `v` based on the results of `g()`.
- Scheduling:** A red arrow points from the end of the `for(;;)` loop of `thread_cpu0()` back to the start of the `for(;;)` loop of `thread_cpu1()`, indicating a scheduled switch between the two threads.

- ▶ Mutex operations

Dataflow representation of the implementation

Dataflow synchronous implementation model

```
fun f:()->(float)
fun g:(int)->(int)
fun h:(float,int)->(int)
var
  x:int y:float z:int d:int
  x0:int y0:float y1:float
  z0:int d1:int
  u:event u1:event v:event
let
  d = 0 fby z
  u1 = top fby u
```



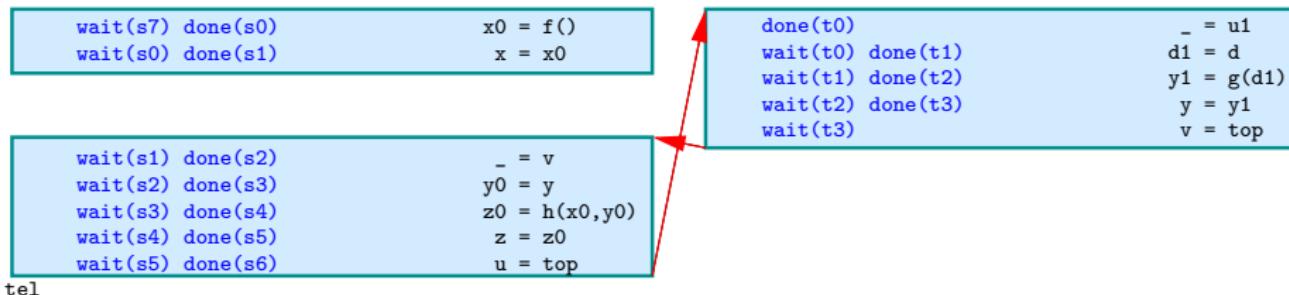
tel

- ▶ Statements all interpreted, but no structure nor sequencing

Dataflow representation of the implementation

Extended dataflow synchronous implementation model

```
fun f:()->(float)
fun g:(int)->(int)
fun h:(float,int)->(int)
var
  x:int y:float z:int d:int
  x0:int y0:float y1:float
  z0:int d1:int
  u:event u1:event v:event s0,s1,s2,s3,s4,s5,s6,s7,t0,t1,t2,t3:event
let
  d = 0 fby z
  u1 = top fby u
  s7 = top fby s6
```



tel

- ▶ Explicit sequencing
- ▶ Functionally complete : same traces as C code under async. semantics

Dataflow representation of the implementation

Non-functional annotations

```
fun f:()->(float)
fun g:(int)->(int)
fun h:(float,int)->(int)
var
  x:int y:float z:int d:int
  x0:int y0:float y1:float
  z0:int d1:int
  u:event u1:event v:event s0,s1,s2,s3,s4,s5,s6,s7,t0,t1,t2,t3:event
let
  d = 0 fby z
  u1 = top fby u
  s7 = top fby s6
```

thread

thread

wait(s7) done(s0)	x0 = f()
wait(s0) done(s1)	x = x0

done(t0)	_ = u1
wait(t0) done(t1)	d1 = d
wait(t1) done(t2)	y1 = g(d1)
wait(t2) done(t3)	y = y1
wait(t3)	v = top

wait(s1) done(s2)	_ = v
wait(s2) done(s3)	y0 = y
wait(s3) done(s4)	z0 = h(x0,y0)
wait(s4) done(s5)	z = z0
wait(s5) done(s6)	u = top

tel

- ▶ Thread structure

Dataflow representation of the implementation

Non-functional annotations

```
fun f:()->(float) at 0x20100
fun g:(int)->(int) at 0x30300
fun h:(float,int)->(int) at 0x20500
var
  x:int at 0x22000 y:float at 0x32000 z:int at 0x22004 d:int at z
  x0:int at x on cpu0 y0:float at y on cpu0 y1:float at y on cpu1
  z0:int at z on cpu0 d1:int at z on cpu1
  u:event at m1 u1:event at m1 v:event at m0 s0,s1,s2,s3,s4,s5,s6,s7,t0,t1,t2,t3:event
let
  d = 0 fby z
  u1 = top fby u
  s7 = top fby s6
thread on cpu0 at 0x20000 stack 0x30000
```

thread on cpu1 at 0x30000 stack 0x40000

wait(s7) done(s0)	x0 = f()
wait(s0) done(s1)	x = x0

done(t0)	_ = u1
wait(t0) done(t1)	d1 = d
wait(t1) done(t2)	y1 = g(d1)
wait(t2) done(t3)	y = y1
wait(t3)	v = top

wait(s1) done(s2)	_ = v
wait(s2) done(s3)	y0 = y
wait(s3) done(s4)	z0 = h(x0,y0)
wait(s4) done(s5)	z = z0
wait(s5) done(s6)	u = top

tel

- ▶ Allocation

Dataflow representation of the implementation

Non-functional annotations

```
fun f:()->(float) at 0x20100
fun g:(int)->(int) at 0x30300
fun h:(float,int)->(int) at 0x20500
var
    x:int at 0x22000 y:float at 0x32000 z:int at 0x22004 d:int at z
    x0:int at x on cpu0 y0:float at y on cpu0 y1:float at y on cpu1
    z0:int at z on cpu0 d1:int at z on cpu1
    u:event at m1 u1:event at m1 v:event at m0 s0,s1,s2,s3,s4,s5,s6,s7,t0,t1,t2,t3:event
let
    d = 0 fby z
    u1 = top fby u
    s7 = top fby s6
thread on cpu0 at 0x20000 stack 0x30000
```

thread on cpu1 at 0x30000 stack 0x40000

```
wait(s7) done(s0)          x0 = f()
wait(s0) done(s1) [flush:0x22000] x = x0
```

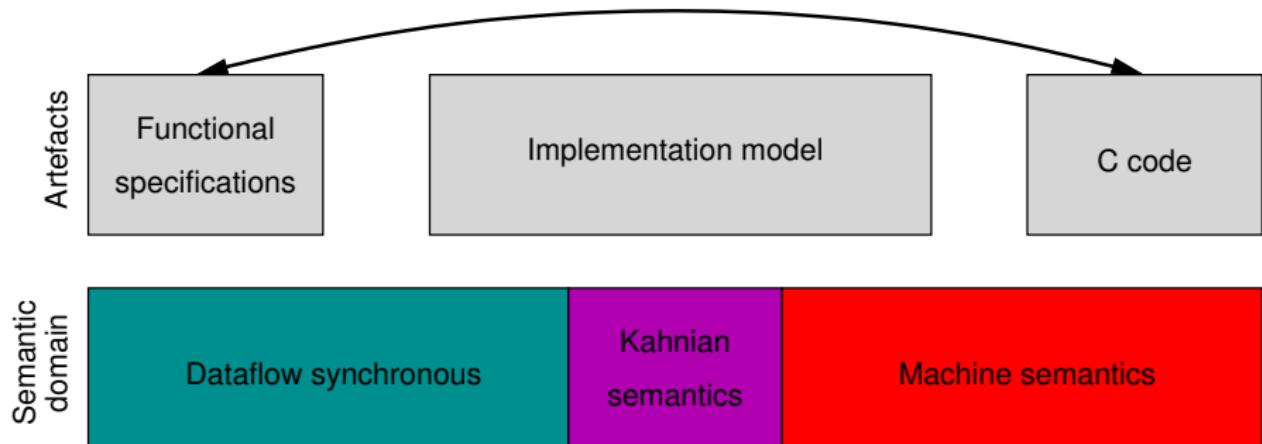
```
done(t0)                  [lock:m1]      _ = u1
wait(t0) done(t1) [inval:0x22004] d1 = d
wait(t1) done(t2)          y1 = g(d1)
wait(t2) done(t3) [flush:0x32000] y = y1
wait(t3) done(t4) [unlock:m0]   v = top
```

```
wait(s1) done(s2) [lock:m0]      _ = v
wait(s2) done(s3) [inval:0x32000] y0 = y
wait(s3) done(s4)          z0 = h(x0,y0)
wait(s4) done(s5) [flush:0x22004] z = z0
wait(s5) done(s6) [unlock:m1]   u = top
```

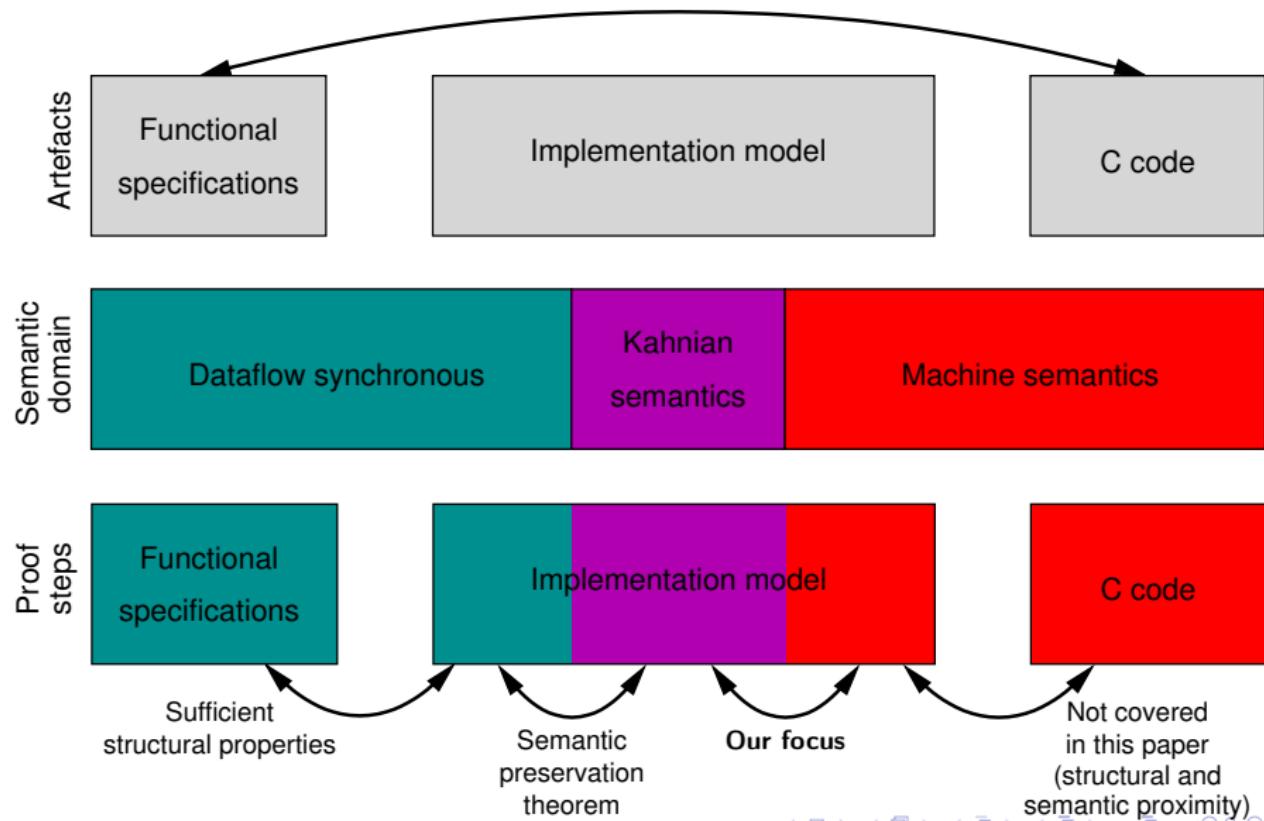
tel

- ▶ Machine semantics of memory coherency and synchronization operations

Proving the correctness of an implementation



Proving the correctness of an implementation



Correction formalization

From asynchronous to machine semantics

The proof is not complete, we merely provide a paper formalization of the proof objectives.

- ▶ **Implementability of the Kahnian interpretation**

Boundedness Every FIFO of the Kahn network must be statically bounded for implementation in memory

Explicit synchronizations Synchronization no longer on the data but exclusively on pure synchronization event variables

- ▶ **Mapping correctness**

Execution without errors Execution under machine semantics do not lead to error state (static check of synchronization behavior)

Semantic preservation Same sequence of inputs of function in Kahnian and machine semantics

Conclusion

Main claim : in practice, threaded implementations of dataflow specification preserve a fundamentally dataflow structure

- ▶ True for implementations we synthesize
- ▶ Future work : determine if it is true for other implementation methods

Impact on implementation correctness proof?

- ▶ Proposal for new proof structuring (on paper)
- ▶ It is still difficult (future work)

No real-time yet

- ▶ Future work