Refinement-based Verification of FreeRTOS in VCC [1]

Sumesh Divakaran

Department of Computer Science
Government Engineering College Idukki

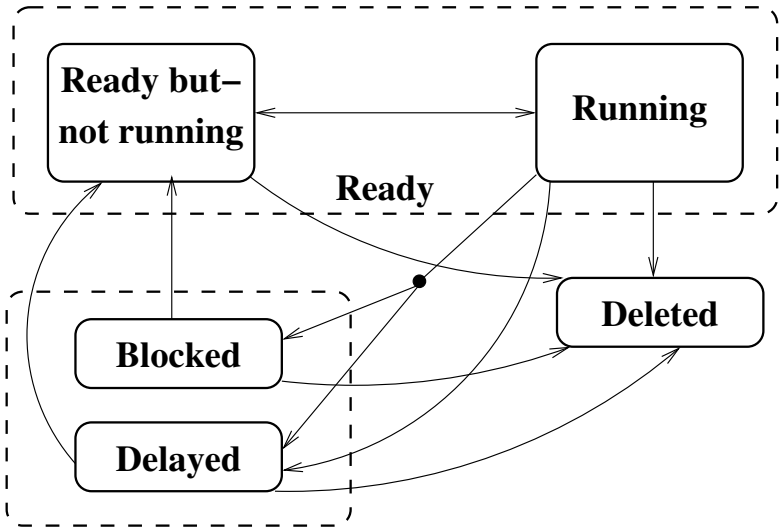
11 December 2017, ACM Winter School in SE @ TCS, Pune

[1] PhD work @ IISc under the supervision of Prof. Deepak D'Souza

Outline

Motivating example for Functional Correctness

FreeRTOS: A popular open-source real time operating system for embedded software applications. It provides API to create and manage multiple tasks.



Task states

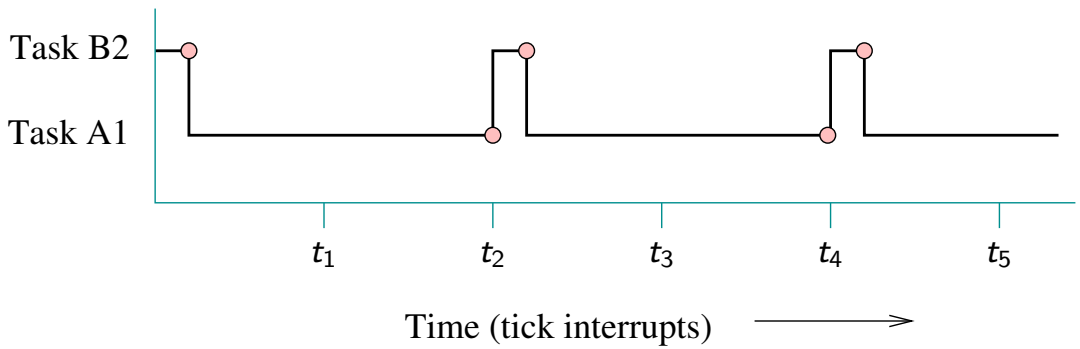
Functional correctness: Need to prove that each behaviour exhibited by the implementation is an expected behaviour of the system

Example FreeRTOS application and its behaviour

```
int main(void){
    xTaskCreate(foo, "Task A1", 1, ...);
    xTaskCreate(bar, "Task B2", 2, ...);
    vTaskStartScheduler();
}

void foo(void* params){
    for(;;);
}

void bar(void* params){
    for(;;){
        vTaskDelay(2);
    }
}
```



A mechanism for proving functional correctness

1. Develop an abstract mathematical model of the system, precisely representing the required behaviours

   One can use an *Abstract Data Type (ADT)* to model the requirements of a system

2. Prove that the given concrete implementation *conforms* to the mathematical model

   One can use *refinement* to establish that a concrete implementation conforms to a mathematical model (ADT)

ADT type

An ADT type is a finite set N of operations. For example, *FreeRTOS* could be an ADT type with operations: `xTaskCreate`, `vTaskStartScheduler` and `vTaskDelay`

- **Each operation n in N has an associated input type I_n and an output type O_n, each of which is simply a set of values.** For example, consider the operation `vTaskDelay` in the type *FreeRTOS*

$$I_{\texttt{vTaskDelay}} = \mathbb{N} \text{ and } O_{\texttt{vTaskDelay}} = \{\}$$

- We require that the set of operations N includes a designated *initialization operation* called *init*.

A simple example for ADT type:

DoubleUIntType

ADT type $DoubleUIntType = \{init, increment, decrement\}$ with

$$
\begin{aligned}
I_{init} &= \{nil\},\\
O_{init} &= \{ok\},\\
I_{increment} &= \{nil\},\\
O_{increment} &= \{ok, fail\},\\
I_{decrement} &= \{nil\},\\
O_{decrement} &= \{ok, fail\}.
\end{aligned}
$$

Here *nil* is a "dummy" argument for the operations. The operations are assumed to return the dummy value *ok* on successful completion.

ADT definition

An *ADT* of type N is a structure of the form

$$\mathcal{A} = (Q, U, \{op_n\}_{n \in N})$$

where

- Q is the set of states of the ADT,
- $U \in Q$ is an arbitrary state in Q used as an *uninitialized* state,
- Each op_n is a (possibly non-deterministic) *realisation* of the operation n given by $op_n \subseteq (Q \times I_n) \times (Q \times O_n)$ (a relation which relates a (state, input) pair to a set of (state, output) pairs)
- Further, we require that the *init* operation depends only on its argument and not on the originating state: thus $init(p, a) = init(q, a)$ for each $p, q \in Q$ and $a \in I_{init}$.

ADT example: *DoubleUInt* of type *DoubleUIntType*

DoubleUInt

$DoubleUInt = (Q, U, \{op_n\}_{n \in DoubleUIntType})$ where

$$Q = \{n \in \mathbb{N} \mid n < (UMAX + 1) * (UMAX + 1)\}$$

where the operations:

op_{init}, $op_{increment}$ and $op_{decrement}$ are given by:

$$op_{init}(n, nil) = \{(0, ok) \mid \forall\, n \in \mathbb{N}, n < (UMAX + 1) * (UMAX + 1)\}$$

$$op_{increment}(n, nil) = \{(n + 1, ok) \mid \forall\, n \in \mathbb{N}, n + 1 < (UMAX + 1) * (UMAX + 1)\}$$

$$op_{decrement}(n, nil) = \{(n - 1, ok) \mid \forall\, n \in \mathbb{N}, 0 < n < (UMAX + 1) * (UMAX + 1)\}$$

Language of sequences of operation calls of an ADT

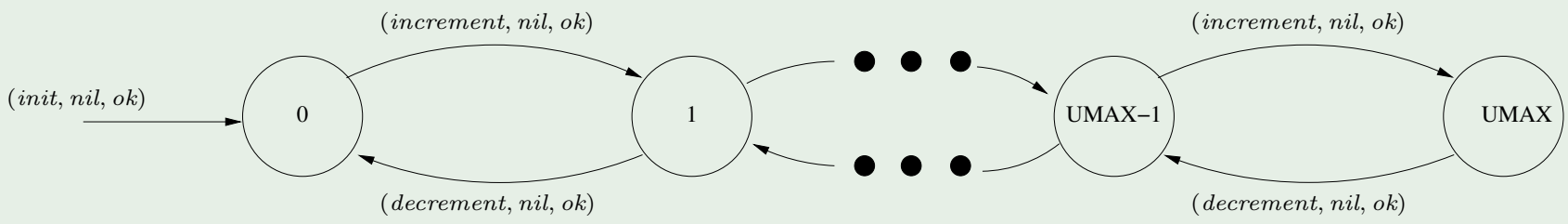
- An ADT $\mathcal{A} = (Q, U, \{op_n\}_{n \in N})$ of type N induces a transition system $\mathcal{S}_\mathcal{A} = (Q, \Sigma_N, U, \Delta)$ where
  - $\Sigma_N = \{(n, a, b) \mid n \in N, a \in I_n, b \in O_n\}$ is the set of *operation call* labels corresponding to the ADT type N. The action label (n, a, b) represents a call to operation n with input a that returns the value b.
  - Δ is given by

$$(p, (n, a, b), q) \in \Delta \text{ iff } op_n(p, a, q, b).$$

- We define the language of *initialised sequences of operation calls* of $\mathcal{A}$, denoted $L_{init}(\mathcal{A})$, to be $L(\mathcal{S}_\mathcal{A}) \cap \{(init, a, b) \cdot \Sigma_N^* \mid a \in I_{init} \text{ and } b \in O_{init}\}$.

Example: Transition system induced by *DoubleUInt*

TS induced by *DoubleUInt*



Totalized version of an ADT $\mathcal{A}$

Given an ADT $\mathcal{A} = (Q, U, \{op_n\}_{n \in N})$ over a data type N, define the totalized version of $\mathcal{A}$, to be an ADT $\mathcal{A}^+$ of type N^+:

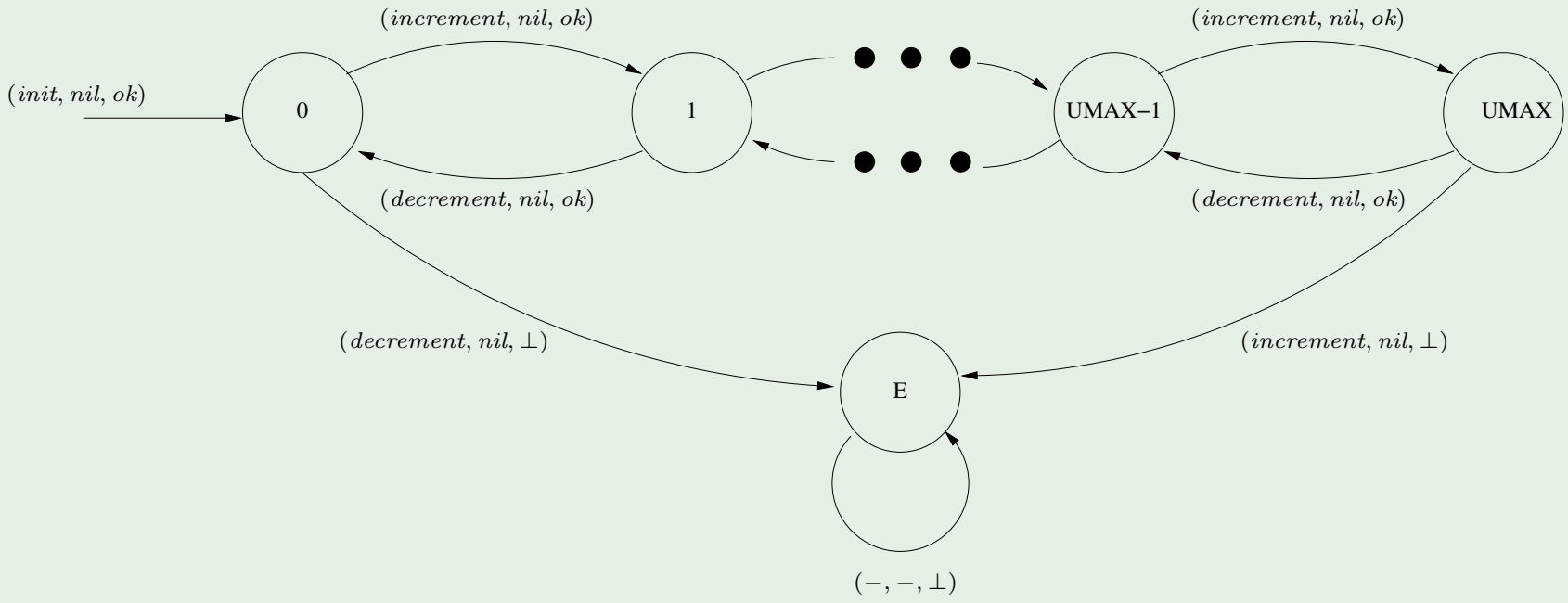
$$\mathcal{A}^+ = (Q \cup \{E\}, U, \{op_n^+\}_{n \in N}), \text{ where}$$

- N^+ has input type I_n and output type $O_n^+ = O_n \cup \{\perp\}$, where $\perp$ is a new output value.
- E is a new "error" state
- op_n^+ is the completed version of operation op_n, obtained as follows:
  - If $(q, a) \notin pre(op_n)$, then add (q, a, E, b') to op_n^+ for each $b' \in O_n^+$.
  - Add $(E, a, E, b') \in op_n^+$ for each $a \in I_n$ and $b' \in O_n^+$.

Here $pre(op_n)$ is the set of state-input pairs on which op_n is defined. Thus $(p, a) \in pre(op_n)$ iff $\exists\, q, b$ such that $op_n(p, a, q, b)$.
If op_n is invoked outside this precondition, the data-structure is assumed to "break" and allow any possible interaction sequence after that.
$\mathcal{A}^+$ represents the interaction sequences that a client of $\mathcal{A}$ may encounter while using $\mathcal{A}$ as a data-structure.

Example: Transition system induced by *DoubleUInt*$^+$

TS induced by *DoubleUInt*$^+$



Refinement between ADTs

Let $\mathcal{A}$ and $\mathcal{B}$ be ADTs of type N. We say $\mathcal{B}$ refines $\mathcal{A}$, written

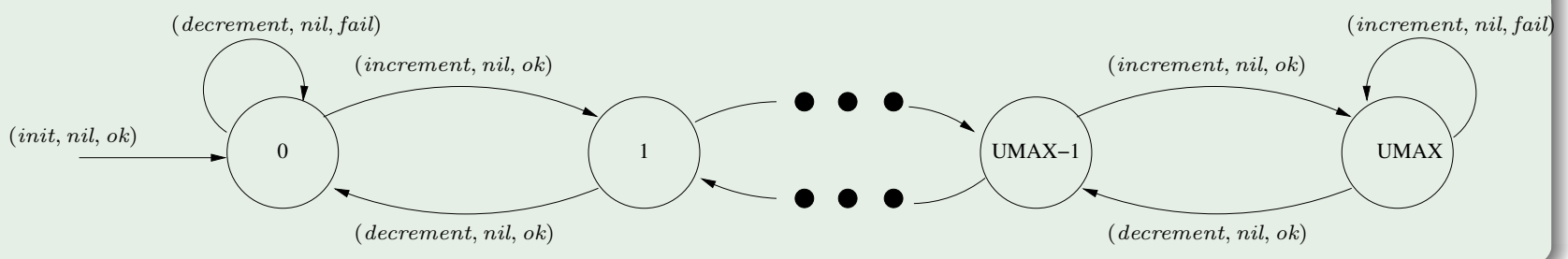
$$\mathcal{B} \preceq \mathcal{A}, \text{ iff } L_{init}(\mathcal{B}^+) \subseteq L_{init}(\mathcal{A}^+).$$

Thus every interaction sequence that a client may see with $\mathcal{B}$ is also an interaction sequence it could have seen with $\mathcal{A}$.

This notion of refinement is from Hoare, He, Sanders et al, *Data Refinement Refined*, Oxford Univ Report, 1985.

Example of refinement:

TS induced by *DoubleUInt'*



Refinement Condition (RC)

(init)

$a \in pre(init_{\mathcal{A}})$ and $(init, a, b)$ q'_a

$\Longrightarrow$

$(init, a, b)$ q_a ρ $(init, a, b)$ q'_a

(g–weak)

(n, a, b) p ρ p'

$\Longrightarrow$

(n, a, b) p ρ (n, a, b') p'

(sim)

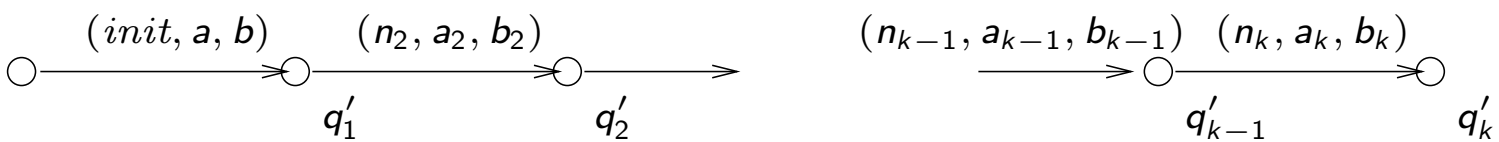
(n, a, b') p ρ (n, a, b) p' q'

$\Longrightarrow$

(n, a, b) p q ρ ρ (n, a, b) p' q'

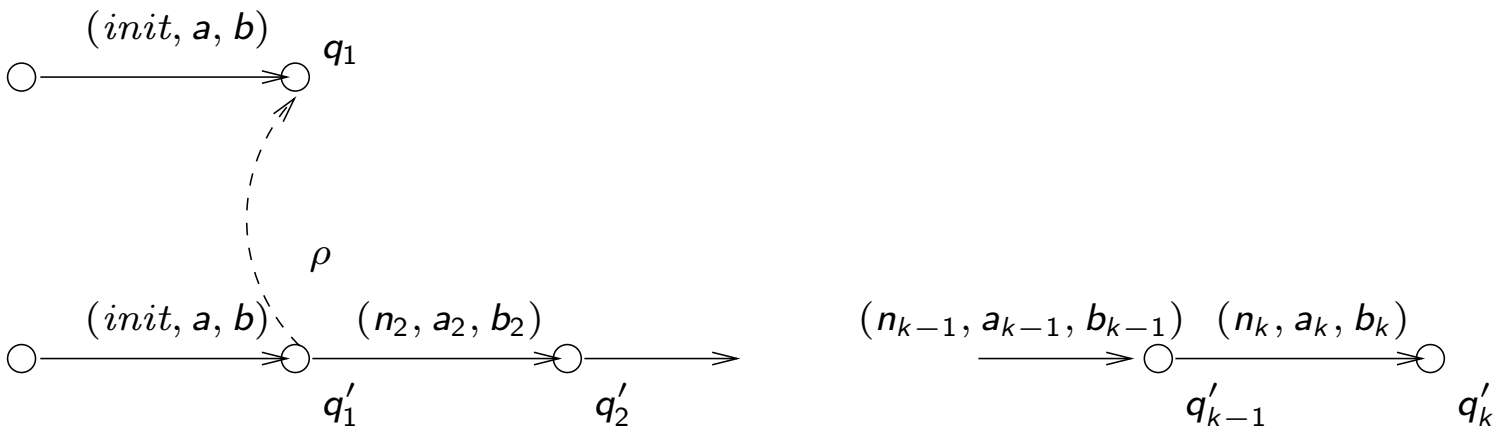
Condition (RC) is sufficient for refinement

If $\mathcal{A}$ and $\mathcal{C}$ are ADTs of the same type, and ρ is an abstraction relation from $\mathcal{C}$ to $\mathcal{A}$ satisfying condition (RC), then $\mathcal{C}$ refines $\mathcal{A}$.



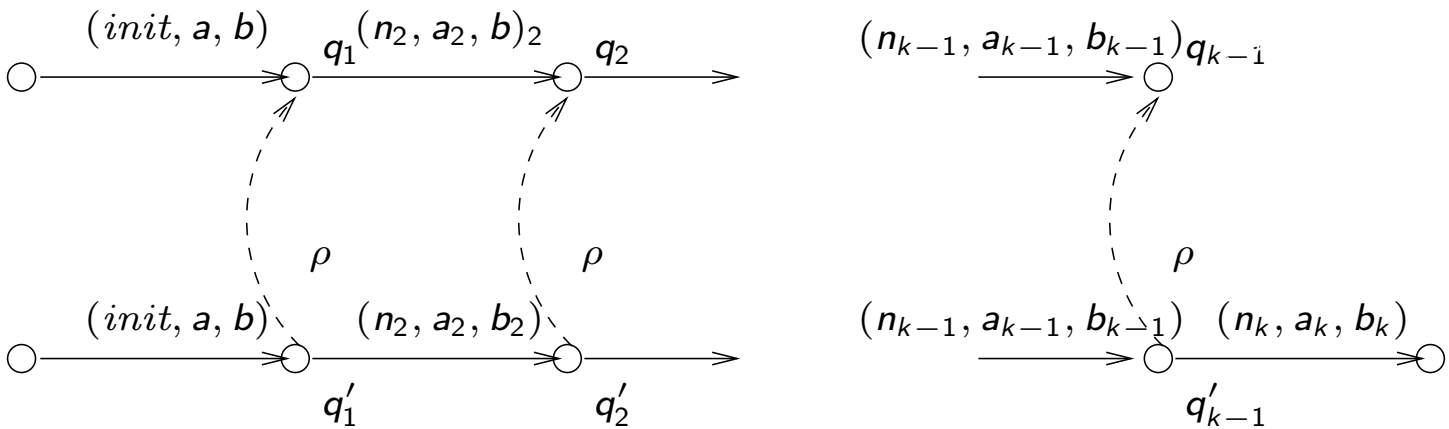
Condition (RC) is sufficient for refinement

If $\mathcal{A}$ and $\mathcal{C}$ are ADTs of the same type, and ρ is an abstraction relation from $\mathcal{C}$ to $\mathcal{A}$ satisfying condition (RC), then $\mathcal{C}$ refines $\mathcal{A}$.



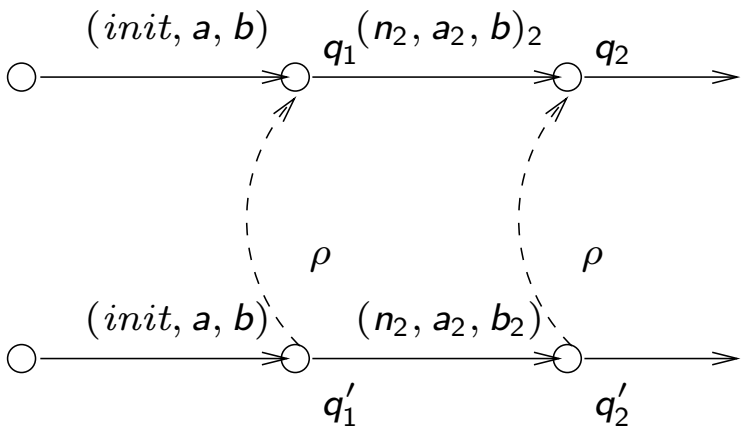
Condition (RC) is sufficient for refinement

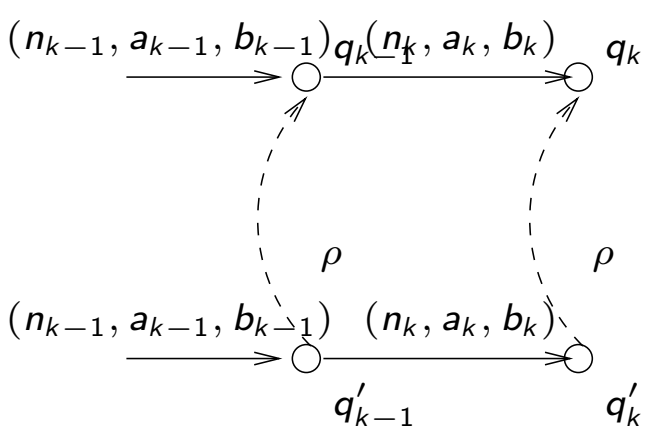
If $\mathcal{A}$ and $\mathcal{C}$ are ADTs of the same type, and ρ is an abstraction relation from $\mathcal{C}$ to $\mathcal{A}$ satisfying condition (RC), then $\mathcal{C}$ refines $\mathcal{A}$.



Condition (RC) is sufficient for refinement

If $\mathcal{A}$ and $\mathcal{C}$ are ADTs of the same type, and ρ is an abstraction relation from $\mathcal{C}$ to $\mathcal{A}$ satisfying condition (RC), then $\mathcal{C}$ refines $\mathcal{A}$.





Refinement for proving functional correctness

Refinement can be used to prove the functional correctness of an ADT implementation

- Let $\mathcal{C}$ be a given implementation of an ADT
- Develop an abstract mathematical model $\mathcal{A}$ *precisely* modeling the behaviours of the ADT
- Check whether $\mathcal{C}$ refines $\mathcal{A}$
  - If yes, then any behaviour exhibited by $\mathcal{C}$ is also allowed by $\mathcal{A}$ and hence, the given implementation is *functionally correct*.
  - If no, then there exists a behaviour exhibited by $\mathcal{C}$, which is not allowed by $\mathcal{A}$ and hence, the given implementation is *not correct*.

Concrete implementation of *DoubleUInt*

```
struct DIC {
   unsigned low,high;
   _(invariant low <= UMAX)
   _(invariant high <= UMAX)
}doubleIntCon;
struct DIC *ptrDIntCon = &doubleIntCon;

void initCon()
  _(requires \thread_local(&ptrDIntCon))
  _(writes \span(ptrDIntCon))
  _(ensures \wrapped(ptrDIntCon))
{
   ptrDIntCon->high = ptrDIntCon->low = 0;
   _(wrap ptrDIntCon)
}
```

Concrete implementation of *DoubleUInt* Contnd.

```
void incrementCon()
  _(requires \wrapped(ptrDIntCon))
  _(requires (ptrDIntCon->high < UMAX
              || ptrDIntCon->low < UMAX) )
  . . .
{
   _(unwrap ptrDIntCon)
   if(ptrDIntCon->low < UMAX)
      ptrDIntCon->low++;
   else
   {
      ptrDIntCon->high++;
      ptrDIntCon->low = 0;
   }
   _(wrap ptrDIntCon)
}
```

Abstract model of *DoubleUInt*

```
struct DIA {
    unsigned dummy;
    _(ghost \natural number)
    _(invariant number < ((UMAX+1)*(UMAX+1)) )
}doubleIntAbs;
struct DIA *ptrDIntAbs = &doubleIntAbs;

void initAbs()
  _(writes \span(ptrDIntAbs))
  _(ensures \wrapped(ptrDIntAbs))
{
    _(ghost ptrDIntAbs->number = 0)
    _(wrap ptrDIntAbs)
}
```

Abstract model of *DoubleUInt* Contnd.

```
void incrementAbs()
  _(requires \wrapped(ptrDIntAbs))
  _(requires (ptrDIntAbs->number+1) < ((UMAX+1)*(UMAX+1)))
  _(writes ptrDIntAbs)
  _(ensures \wrapped(ptrDIntAbs))
{
   _(unwrap ptrDIntAbs)
   _(ghost ptrDIntAbs->number = ptrDIntAbs->number + 1)
   _(wrap ptrDIntAbs)
}
```

Phrasing refinement conditions in VCC

```
typedef struct AC {
  abstract state
  invariants on abs state
  concrete state
  invariants on conc state
  gluing invariant on joint abs-conc state
} AC;

operation n(AC *p, arg a)
_(requires \wrapped(p)) // glued joint state
_(requires G)  // precondition G of abs op
_(ensures \wrapped(p)) // restores glued state
_(decreases 0) // conc op terminates whenever G is true
{
  _(unwrap p)
  // abs op body
  // conc op body
  _(wrap p)
}

init(*p)
_(ensures \wrapped(p)) {...}
```

Refinement checking for *DoubleUInt*

```
struct DIJ {
  unsigned dummy;
  _(ghost \natural number)
  _(invariant number < ((UMAX+1)*(UMAX+1)) )
  unsigned low,high;
  _(invariant low <= UMAX)
  _(invariant high <= UMAX)
  _(invariant number == high * (UMAX+1) + low)
}doubleIntJoint;

struct DIJ *ptrDIntJoint = &doubleIntJoint;
```

Refinement checking for *DoubleUInt* Contnd.

```
void initDIntJoint()
  _(ensures \wrapped(ptrDIntJoint))
{
   _(ghost ptrDIntJoint->number = 0)
   ptrDIntJoint->high = 0;
   ptrDIntJoint->low = 0;
   _(wrap ptrDIntJoint)
}
```

Refinement checking for *DoubleUInt* Contnd.

```
void IncrementJoint()
  _(requires \wrapped(ptrDIntJoint))
  _(requires ptrDIntJoint->number+1 < (UMAX+1) * (UMAX+1))
  _(ensures \wrapped(ptrDIntJoint))
{
   _(unwrap ptrDIntJoint)
   _(ghost ptrDIntJoint->number = ptrDIntJoint->number+ 1)
   if(ptrDIntJoint->low < UMAX)
      ptrDIntJoint->low = ptrDIntJoint->low + 1;
   else
   {
      ptrDIntJoint->high = ptrDIntJoint->high + 1;
      ptrDIntJoint->low = 0;
   }
   _(wrap ptrDIntJoint)
}
```

Checking Refinement in VCC

Demo.

About FreeRTOS

A popular open-source operating system for embedded software applications

It provides an application programmer ways to:

- Create and manage multiple tasks.
- Schedule tasks based on priority-based pre-emption.
- Let tasks communicate (via message queues, semaphores, mutexes).
- Let tasks delay and timeout on blocking operations.

About FreeRTOS implementation

- Written mostly in C.
- Assembly language for processor-specific code.
- Portable:
  - Processor independent code is in 3 C files.
  - Processor dependent code (called a "port" in FreeRTOS) is organised by Compiler-Processor pairs.
  - (19 compilers, 27 processors supported).
- Small footprint (≈3,000 lines), engineered for efficiency.
- Well-written, and well-documented through comments.

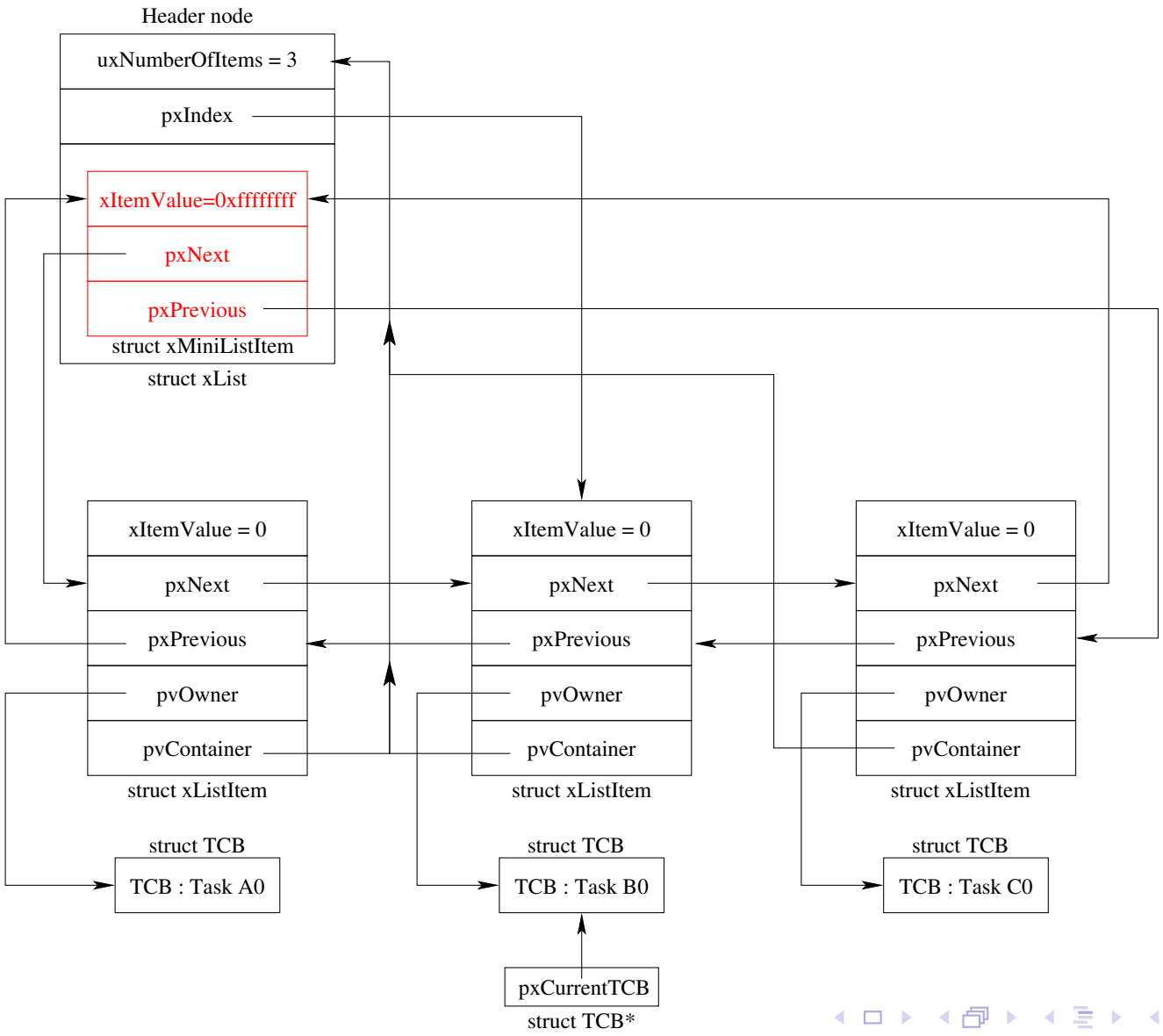
Scheduler APIs considered in the verification

FreeRTOS API	**Basic Functionality**
`vTaskStartScheduler`	Schedule the longest waiting highest priority task.
`xTaskCreate`	Create the given task and add it to the ready queue.
`vTaskDelete`	Delete the given task (move it to the deleted queue).
`vTaskDelay`	Delay the current task for a given period of time.
`vTaskDelayUntil`	Delay the current task relative to its previous wake-time.
`vTaskIncrementTick`	Increment the tick-count and awaken delayed tasks.
`vTaskPrioritySet`	Change the priority of the given task.
`vTaskSuspend`	Suspend the given task.
`vTaskResume`	Resume the given task which is in the *suspended* state.
`xTaskGetTickCount`	Get the value of the current tick-count.
`vTaskPriorityInherit`	Implements the priority inheritance scheme in FreeRTOS.

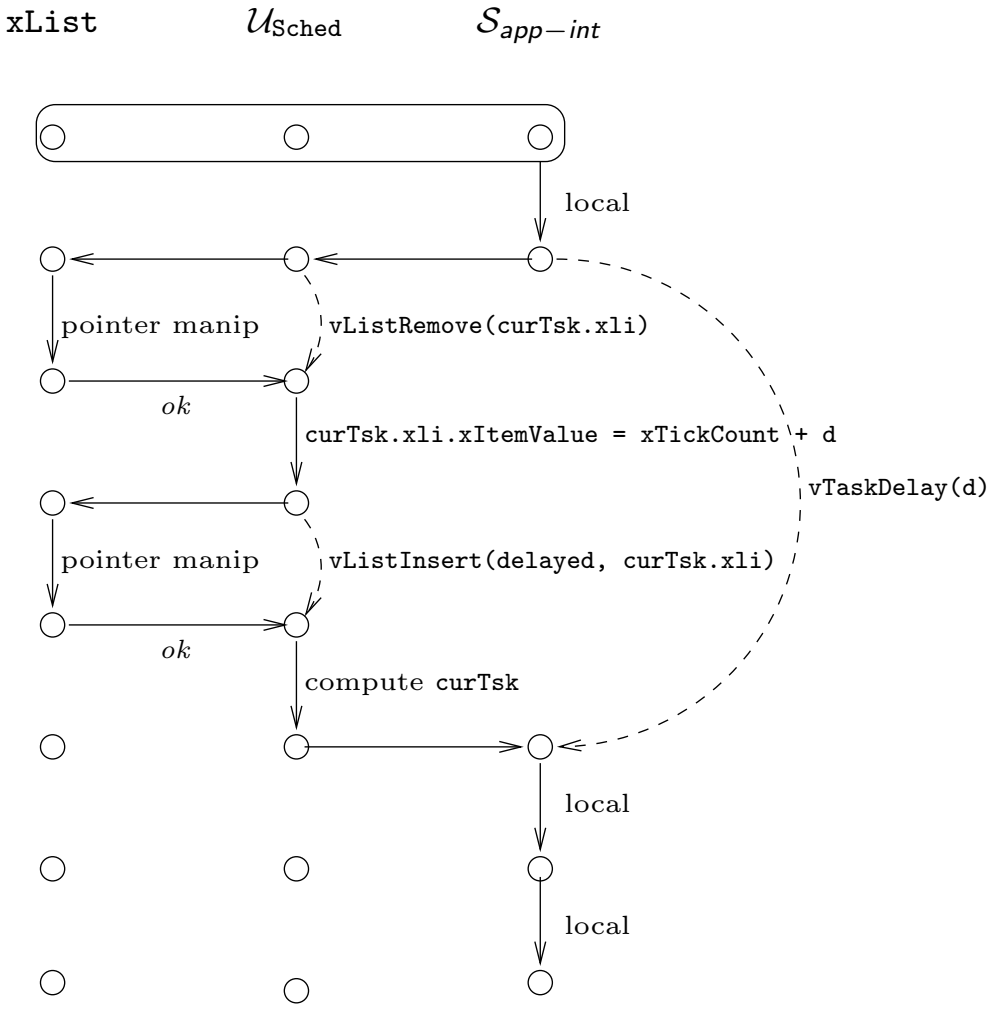
Extracts from code: vTaskDelay()

```
void vTaskDelay(portTickType xTicksToDelay)
{
  ...
  if(xTicksToDelay > (portTickType) 0)
  {
    vListRemove(&(pxCurrentTCB->xGenListItem));
    xTimeToWake = xTickCount + xTicksToDelay;
    vListInsert(pxDelayedTaskList,&(pxCurrentTCB->xGenListItem))
    ...
  }
}
```

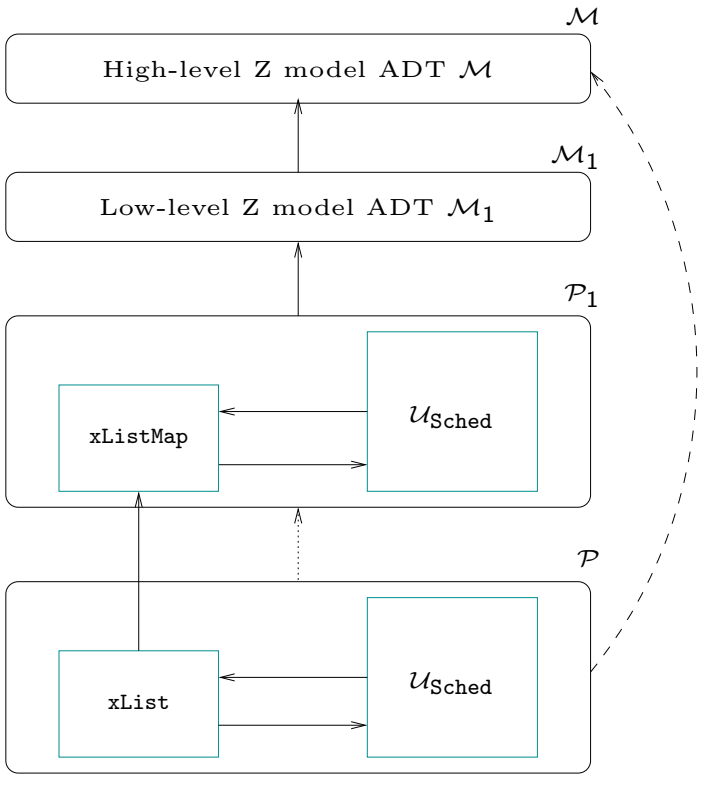
xList - the core data structure in FreeRTOS



Application running with FreeRTOS scheduler



FreeRTOS - verification strategy



Steps in verification

- Developing abstract model ($\mathcal{M}$) of the ADT whose type is defined by the prototypes of APIs in tasks.c.
- Refining $\mathcal{M}$ to $\mathcal{M}_1$ to capture implementation details.
- Abstracting concrete implementation $\mathcal{P}$ to $\mathcal{P}_1$ by replacing `xList` library with an abstract `xListMap` library.
- Proving refinements between successive levels.

Abstract model of the scheduler in Z

Scheduler

$$
\begin{array}{l}
maxPrio, maxNumVal, tickCount, topReadyPriority : \mathbb{N} \\
tasks : \mathbb{P}\ TASK \\
priority : TASK \pfun \mathbb{N} \\
running\_task, idle : TASK \\
ready : \mathrm{seq}\ (\mathrm{iseq}\ TASK) \\
delayed : \mathrm{seq}\ TASK \times \mathbb{N} \\
blocked : \mathrm{seq}\ TASK \\
\ldots \\
\hline
idle \in tasks \land idle \in \mathrm{ran} \frown / (\mathrm{ran}\ ready) \\
running\_task \in tasks \land topReadyPriority \in \mathrm{dom}\ ready \\
\forall i, j : \mathrm{dom}\ delayed \mid (i < j) \bullet delayed(i).2 \leq delayed(j).2 \\
\forall tcn : \mathrm{ran}\ delayed \mid tcn.2 > tickCount \\
running\_task = head\ ready(topReadyPriority) \\
\mathrm{dom}\ priority = tasks \land tickCount \leq maxNumVal \\
\forall i, j : \mathrm{dom}\ blocked \mid (i < j) \implies priority(blocked(i)) \geq priority(blocked(j)) \\
\ldots
\end{array}
$$

Z model of `TaskDelay` operation

TaskDelay

$$
\begin{array}{l}
\Delta Scheduler \\
delay? : \mathbb{N} \\
delayedPrefix, delayedSuffix : \operatorname{seq} TASK \times \mathbb{N} \\
running! : TASK \\
\hline
delay > 0 \land delay \leq maxNumVal \\
tickCount + delay? \leq maxNumVal \\
delayed = delayedPrefix \frown delayedSuffix \\
\forall\, tcn : \operatorname{ran} delayedPrefix \mid tcn.2 \leq (tickCount + delay?) \\
delayedSuffix \neq \langle\rangle \implies (head\ delayedSuffix).2 > (tickCount + delay?) \\
delayed' = delayedPrefix \frown \langle (running\_task, (tickCount + delay?)) \rangle \frown delayedSuffix \\
\ldots
\end{array}
$$

Extracts from verification of vTaskDelay()

```
void vTaskDelay(unsigned delay _(out unsigned dIndex))
  _(requires (delay>0))
  ...
  _(ensures \old(pxCurrentTCB)->xGLI.xItemValue ==
            SchedP1.tickCount+delay))
  _(ensures ((dIndex <= \old(SchedP1.dLength)) &&
   (\forall unsigned i;
   (i < dIndex) ==>(\old(SchedP1.D[i].xli.xItemValue) <=
                    \old(pxCurrentTCB)->xGLI.xItemValue)) &&
     (\forall unsigned i;
       ((i >= dIndex) && (i < \old(SchedP1.dLength))) ==>
        (\old(SchedP1.D[i].xli.xItemValue) >
          \old(pxCurrentTCB)->xGLI.xItemValue))))
```

Extracts from verification of vTaskDelay() Contd.

```
void vTaskDelay(unsigned delay _(out unsigned dIndex))
  _(requires (delay>0))
  ...
   _(ensures \forall unsigned i; (i < dIndex) ==>
     (SchedP1.D[i] == \old(SchedP1.D[i]))
  _(ensures SchedP1.D[dIndex] == \old(pxCurrentTCB)->xGLI)
  _(ensures \forall unsigned i;
     ((i > dIndex) && (i <= \old(SchedP1.dLength))) ==>
      (SchedP1.D[i] == \old(SchedP1.D[i - 1]))
  _(ensures SchedP1.dLength == (\old(SchedP1.dLength) + 1))
  ...
{
     //body of the method
}
```

FreeRTOS - verification effort

Z Model $\mathcal{M}_1$		Z Model $\mathcal{M}_2$		API funcs in $\mathcal{P}$		
Schemas	LOC	Schemas	LOC	Funcs	LOC	LOA
50	766	60	1239	17	361	2347

`xListMap`			`xList`		
Funcs	LOC	LOA	Funcs	LOC	LOA (`xListJoint`)
15	306	1033	15	121	1450

Bugs found in FreeRTOS verification

- `vTaskCreate`
  1. If the running task say $A1$ creates a new task of same priority say $B1$, then FreeRTOS schedules $A1$ again before scheduling $B1$.
  2. If user creates tasks $A1, B1$ and $C1$ before starting the scheduler, then FreeRTOS schedules these tasks in the order $C1, A1, B1$.
- `vTaskPrioritySet`

  Violates the invariant on a priority queue when the task is waiting and hence no effect until it becomes ready.

Bugs found in FreeRTOS verification Contd.

- `vTaskPriorityInherit`

  Fails to meet the goal when the task is blocked
- `vTaskSuspend,vTaskResume`

  The suspended task loses its context in delayed/blocked lists
- `vTaskDelay`

  Fails to restore a property on the kernel variable uxTopReadyPriority. The same problem exists in some other APIs. Even though this is not a *bug*, the situation can be handled more efficiently.

???

???

Thank You!