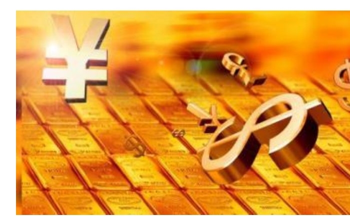
Mechanized Verification of Preemptive OS Kernels

Xinyu Feng

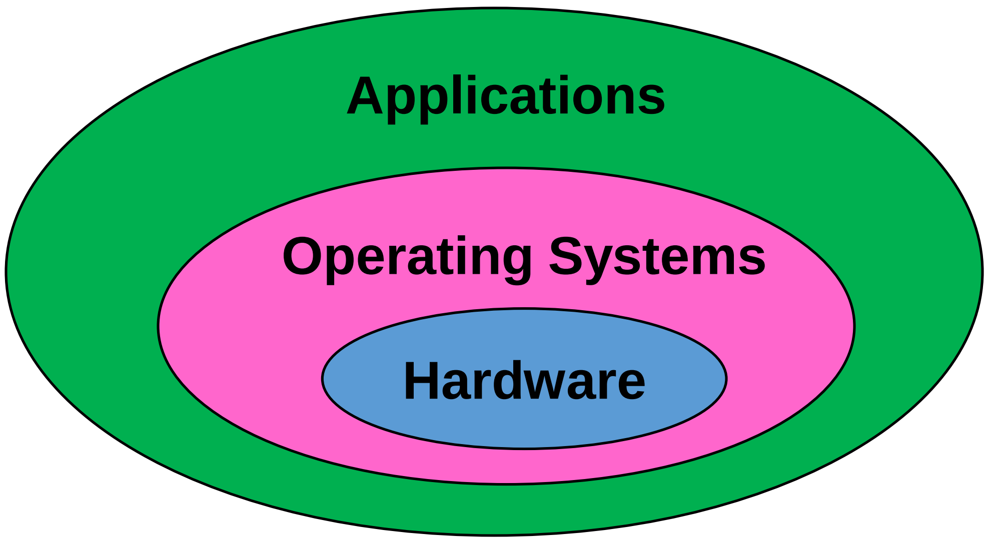
University of Science and Technology of China

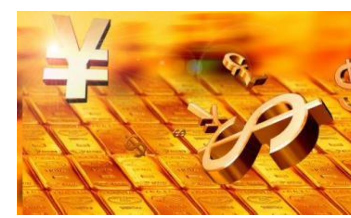
Why OS Kernel Verification?

Computer Systems



Why OS Kernel Verification?





Correctness of OS is crucial for safety and security of the whole system

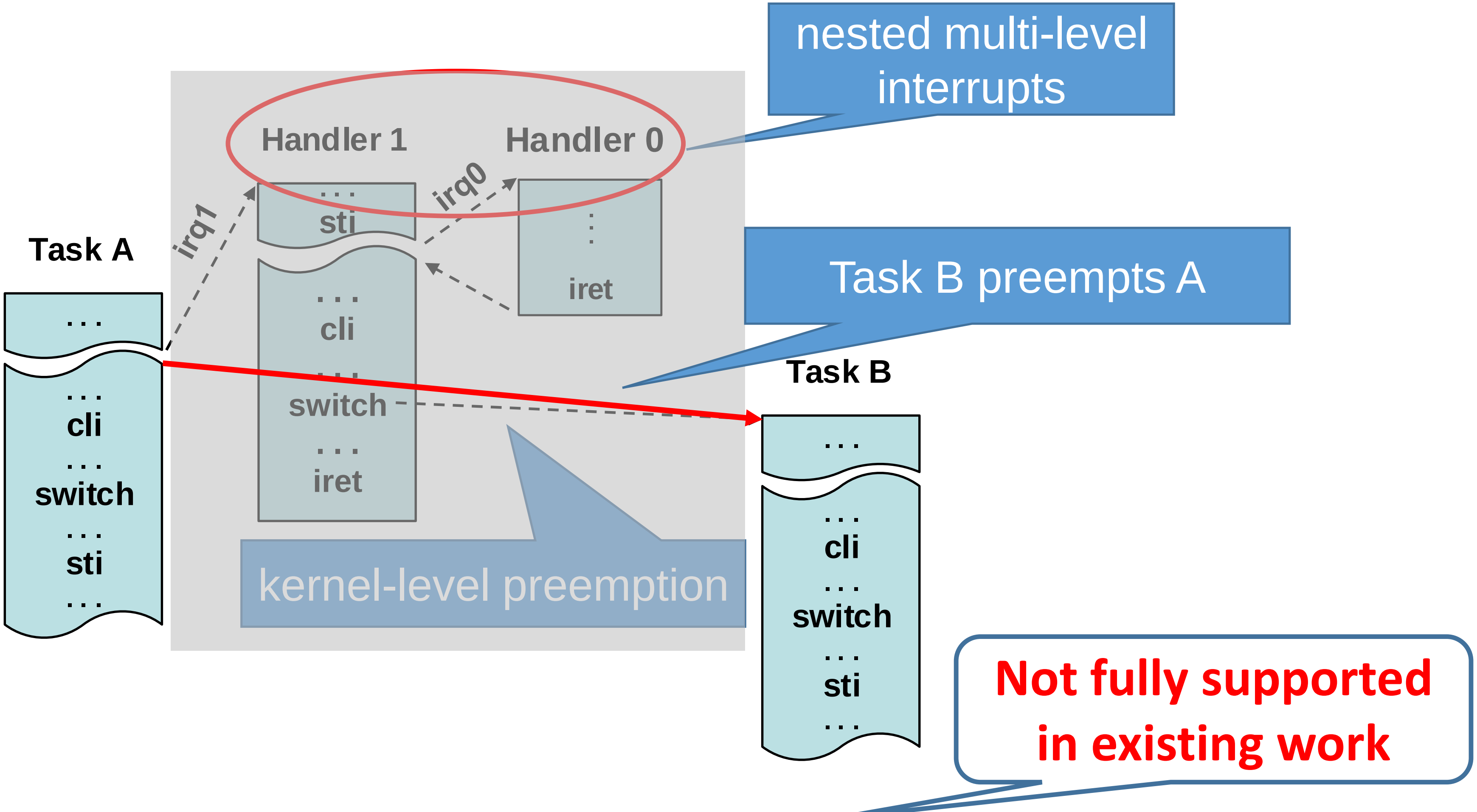
Why OS Kernel Verification?

- Fundamental, but also **simpler to verify**!
  (comparing to applications)
  - Less domain knowledge required
    – every programmer knows OS
  - Stable specifications
  - Slow evolution
  - Specs validated by application-level verification

OS Kernel Verification: Challenges

- Low-level programs
  - C + inline assembly, interrupts, task management, …
- Larger code base (than algorithm verification)
- Code at different abstraction layers
  - E.g., threads vs. schedulers
- Involves both libraries (sys. calls) and runtime (scheduler)
  - What is a proper specification?
- Rich concurrency
  - Multi-tasking, multi-core, interrupts

Preemption and nested interrupts



Preemptions and multi-level interrupts are crucial for real-time systems.

They also make system highly concurrent and complex

Concurrency & Preemption in Previous work

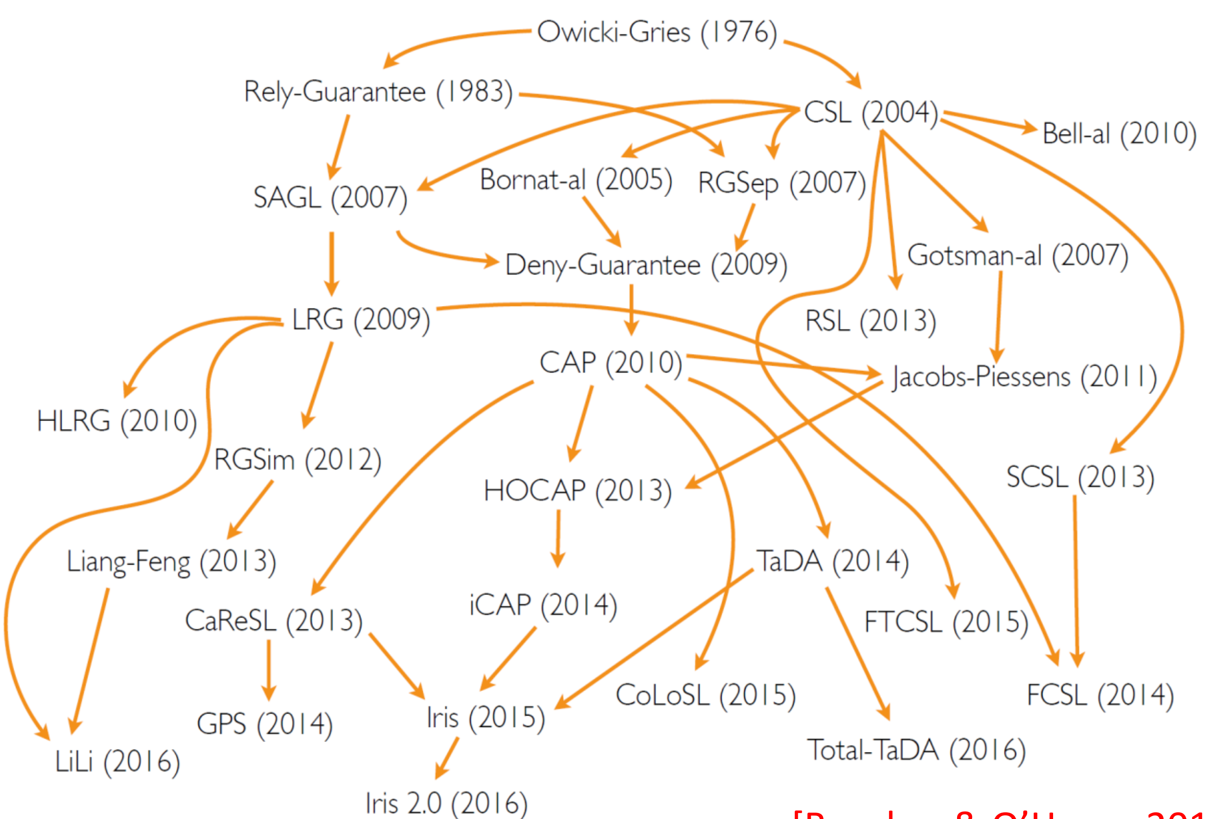
- seL4 [Klein et al. 2009 …]
  - Mostly sequential
  - Limited support of interrupts at fixed program points
- Verisoft [Rieden et al. 2007 …]
  - Kernel is sequential
- Verve [Yang & Hawblitzel. PLDI 2010]
  - Allows preemption, but no nested interrupts
  - Mostly about safety, limited functionality verification
- CertiKOS [Gu et al. 2015, Chen et al. 2016, Gu et al. 2016]
  - Evolving: sequential → limited interrupts → multicore
  - Still no preemption

Concurrency & Preemption in Previous work (2)

- eChronos OS [Andronick et al. 2015, 2016]
  - Supports preemption and nested interrupts
  - But verification at the model level only
  - Verifies scheduling invariants, no API correctness

Challenges for Verifying Preemptive OS Kernels

- Verifying concurrent programs is difficult
  - Non-deterministic interleaving



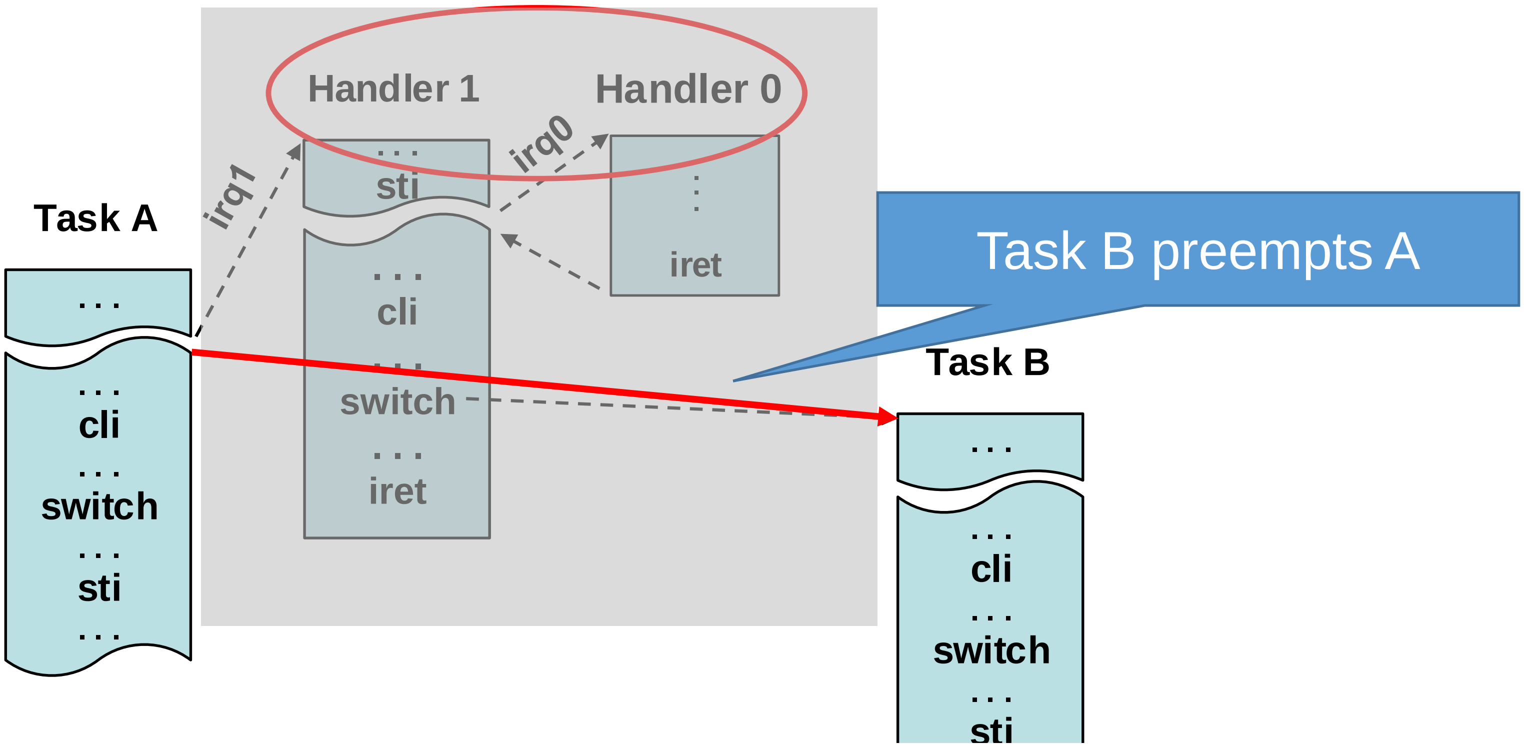
[Brookes & O'Hearn 2016], courtesy of Ilya Sergey

Challenges for Verifying Preemptive OS Kernels

- Verifying concurrent programs is difficult
- Verifying concurrent kernels is even more challenging
  - More difficult to establish **refinement** with concurrency
    - Theories not fully developed until recently
      [Turon et al. POPL'13, ICFP'13] [Liang et al. PLDI'13, CSL-LICS'14]
  - Kernel-level preemption can be more complex than multi-tasking/multi-processor concurrency

A natural correctness spec. for OS kernels

Kernel-level preemption can be more complex than multi-tasking/multi-processor concurrency



Interrupt management is now a verification target:

lower abstraction layer and non-uniform concurrency model

More low-level details:

e.g., can context switch only when there are no nested interrupts

This talk

- **Verification framework for preemptive OS kernels**
  - Refinement reasoning about concurrent kernels
  - Multi-Level nested interrupts and preemption

- **Verification of key modules of a commercial OS kernel µC/OS-II in Coq**

The first mechanized verification of a commercial preemptive OS kernel.

[Xu et al. CAV'16]

Outline

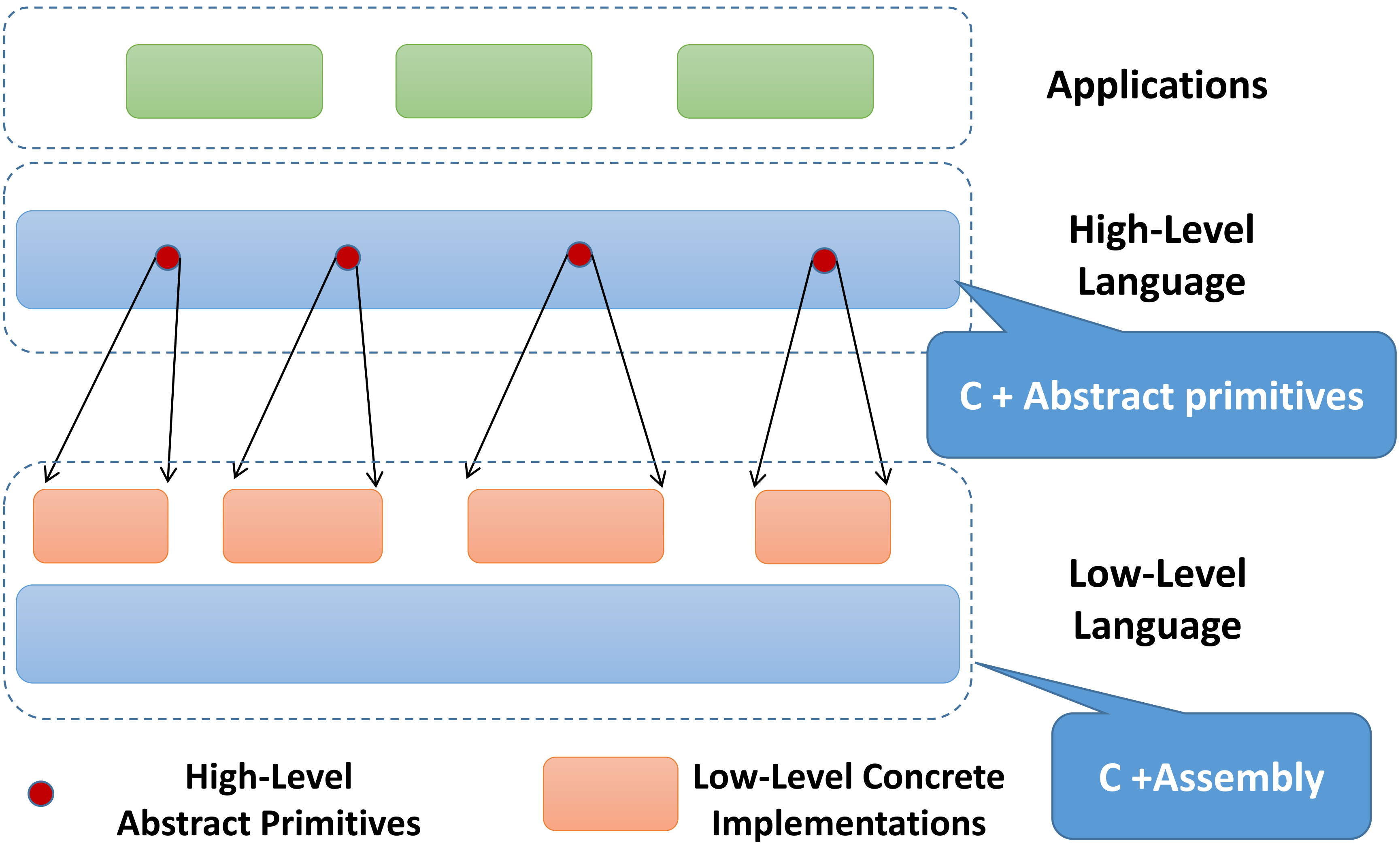
- OS Correctness Specification
- Verification Framework
  - System modeling
  - CSL-R: Program logic for refinement & multi-level interrupts
  - Coq tactics
- Verifying µC/OS-II

OS Correctness

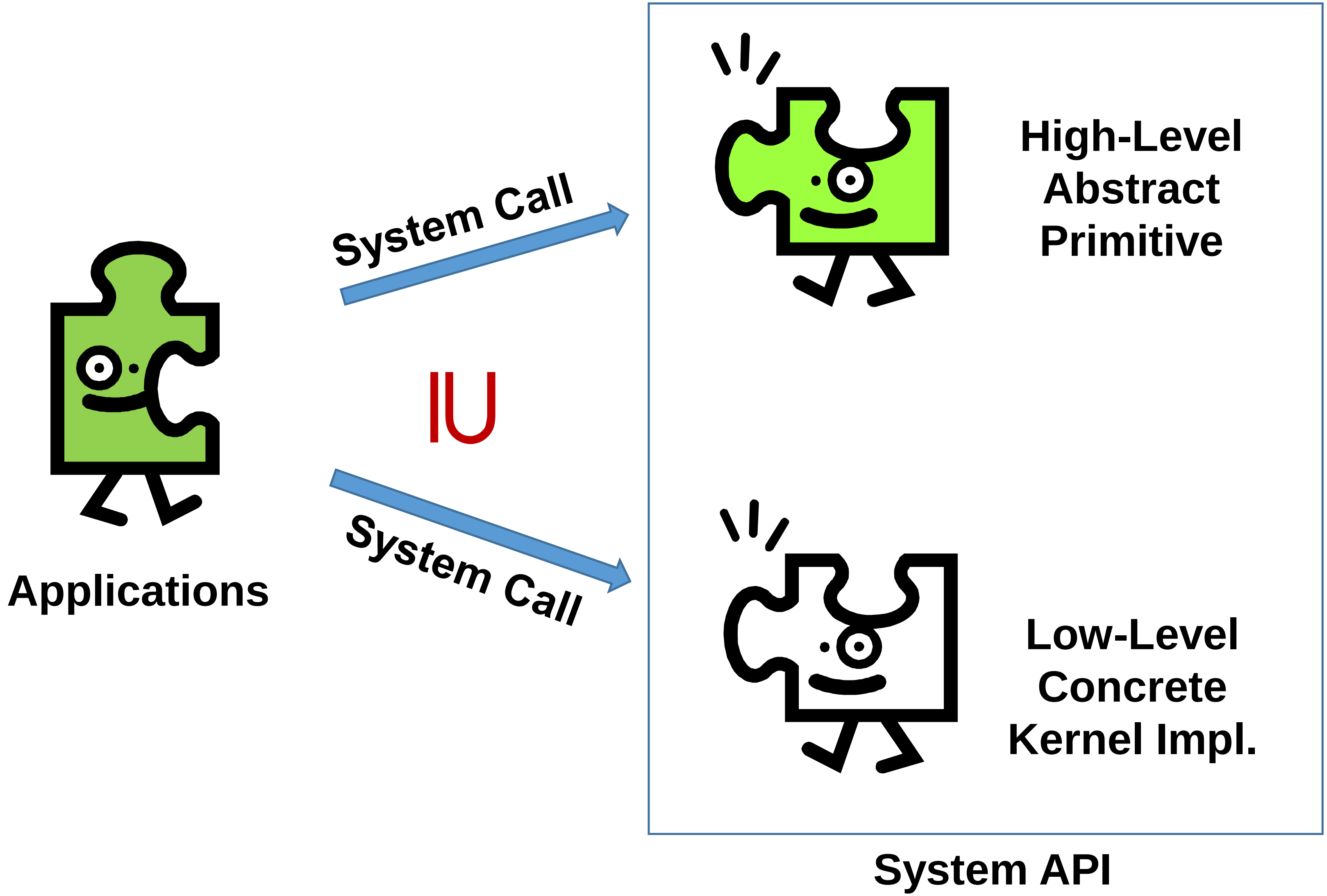
- **OS provides abstraction for programmers**
  - Hides details of the underlying hardware
  - Provides an abstract programming model

- **OS Correctness** : **refinement** between high-level abstraction and low-level concrete implementation

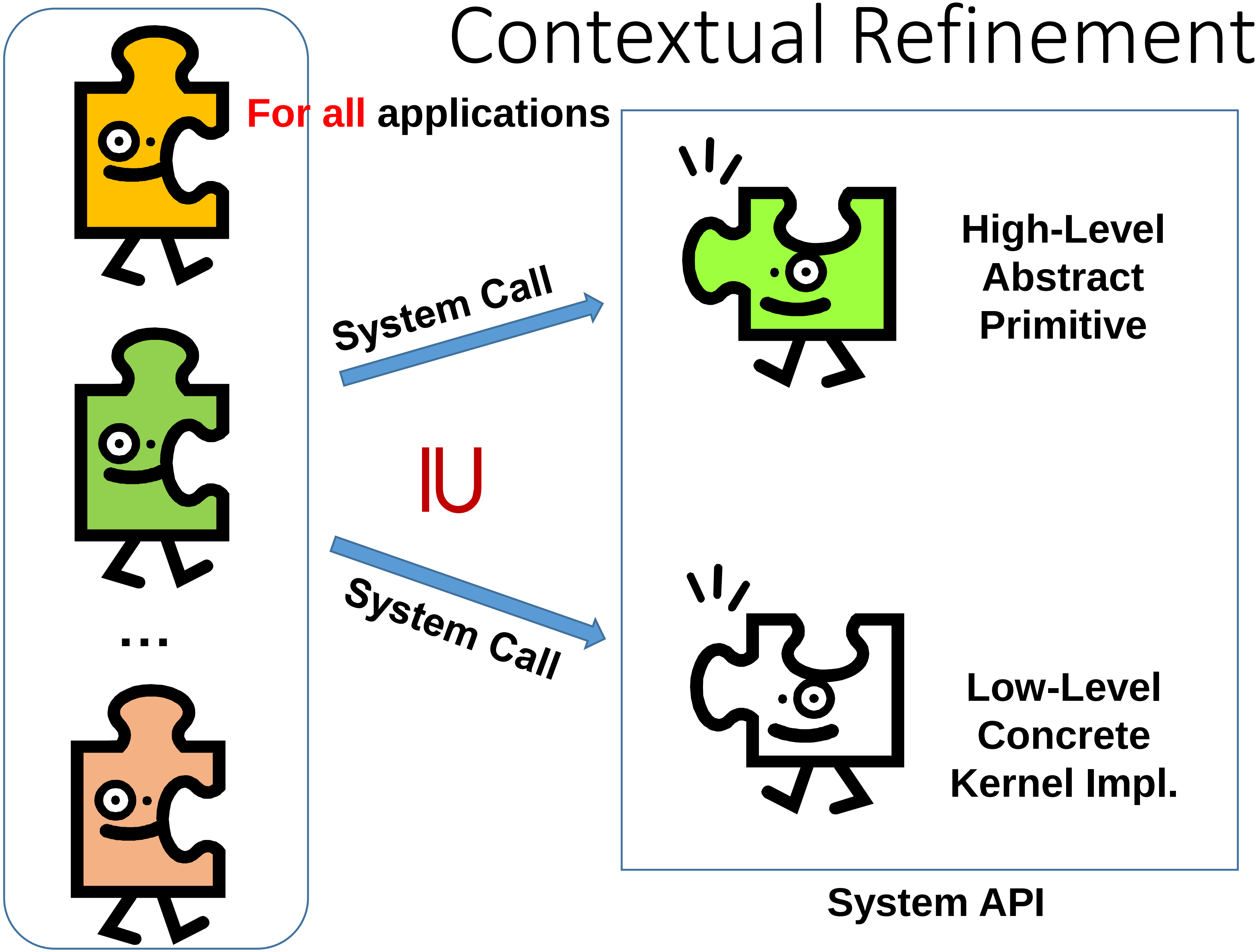
OS Correctness



Refinement


System Call
System Call
⊑
Applications
High-Level
Abstract
Primitive
Low-Level
Concrete
Kernel Impl.
System API

Contextual Refinement



Contextual Refinement as OS API Correctness

$O \sqsubseteq_{ctxt} S$ iff

$\forall A.\ \mathrm{ObsBeh}(A[O]) \subseteq \mathrm{ObsBeh}(A[S])$

With some assumptions about A

The set of observable behaviors

A: **Application** O: **Concrete Impl. of OS API**

S: **Abstract Prim.**

Contextual Refinement as OS API Correctness

But OS correctness is more than API correctness:

Correctness of runtime services, e.g., scheduler
(not exported as an API)

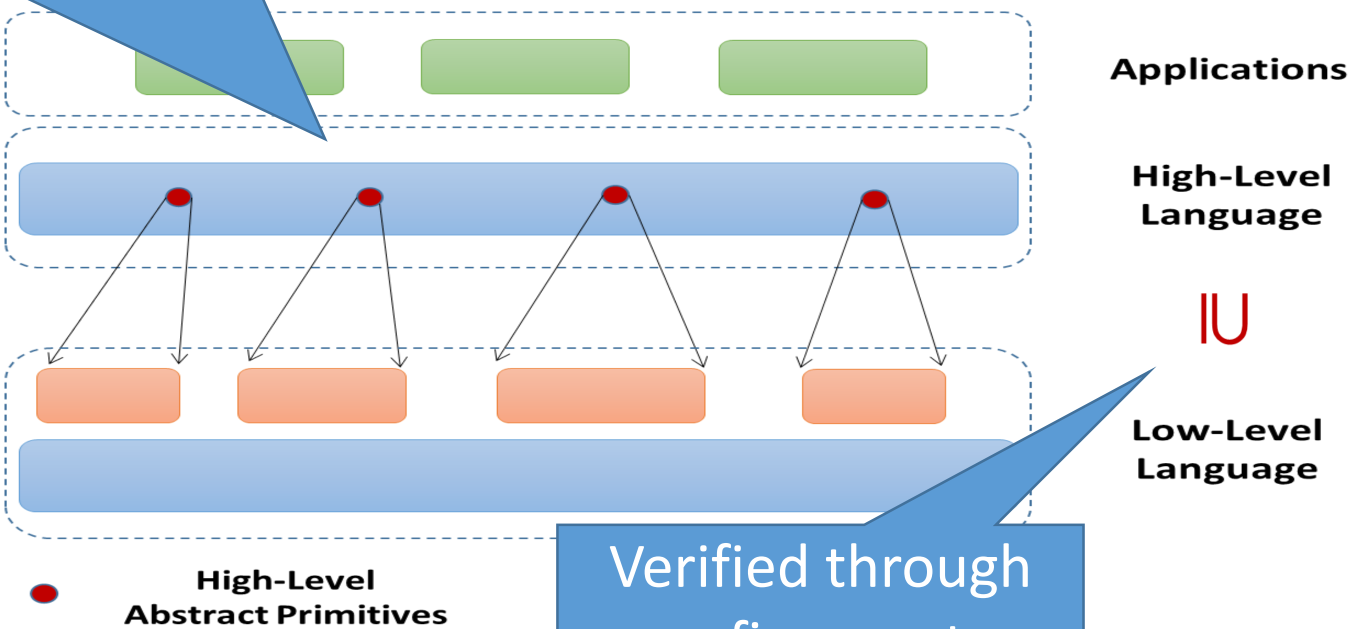
Whole system properties,
e.g., isolation and security, real-time properties, …

Cannot be specified as abstract API primitives!

How to specify their correctness?

Runtime services and Sys. Props

Runtime: specified as part of the high-level language semantics (e.g., scheduling)

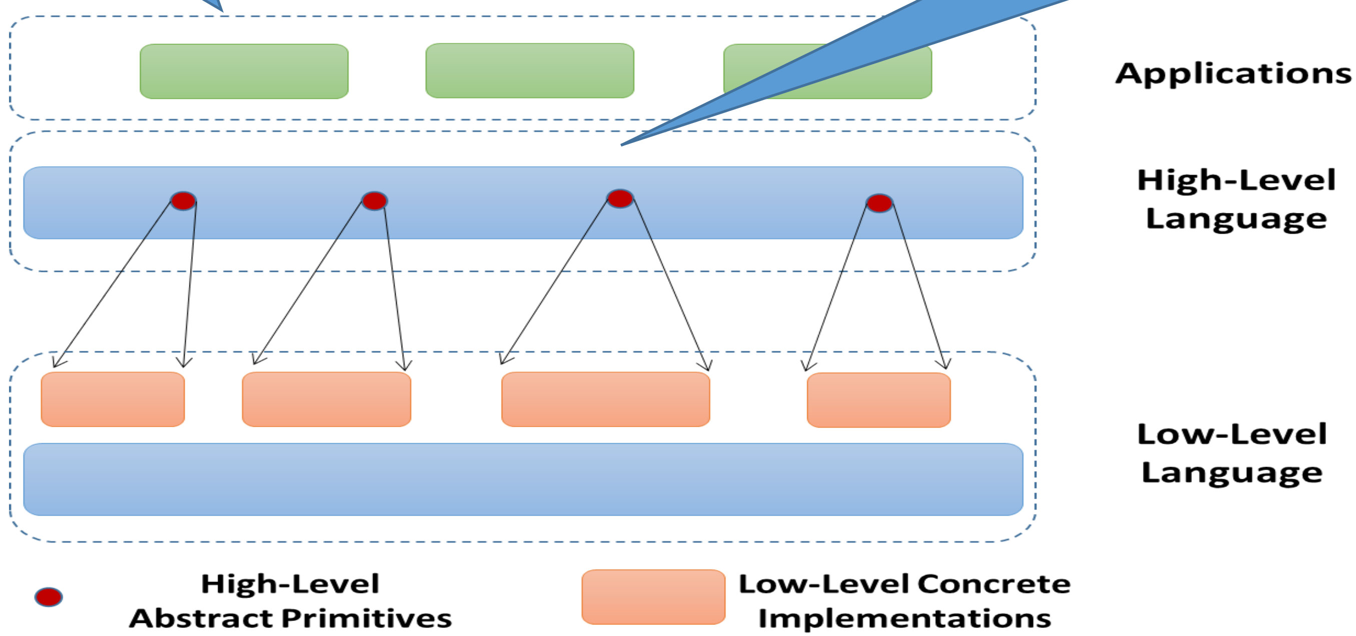


Verified through refinement

Runtime services and Sys. Props

Whole system properties: specified as **trace properties** of **all** apps (with high-level views)

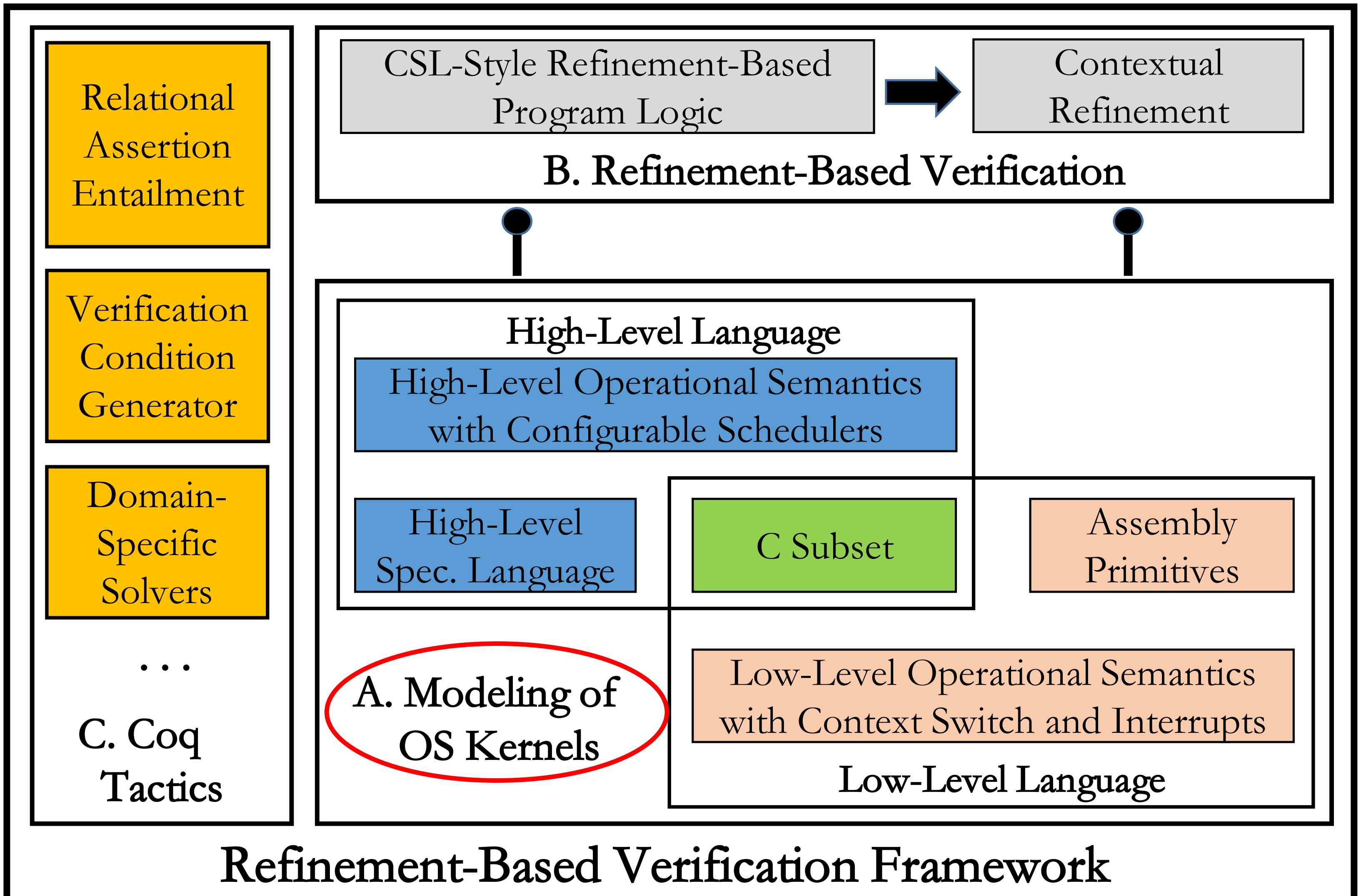
Proved at **high-level** only, propagated to low-level through **contextual refinement!**



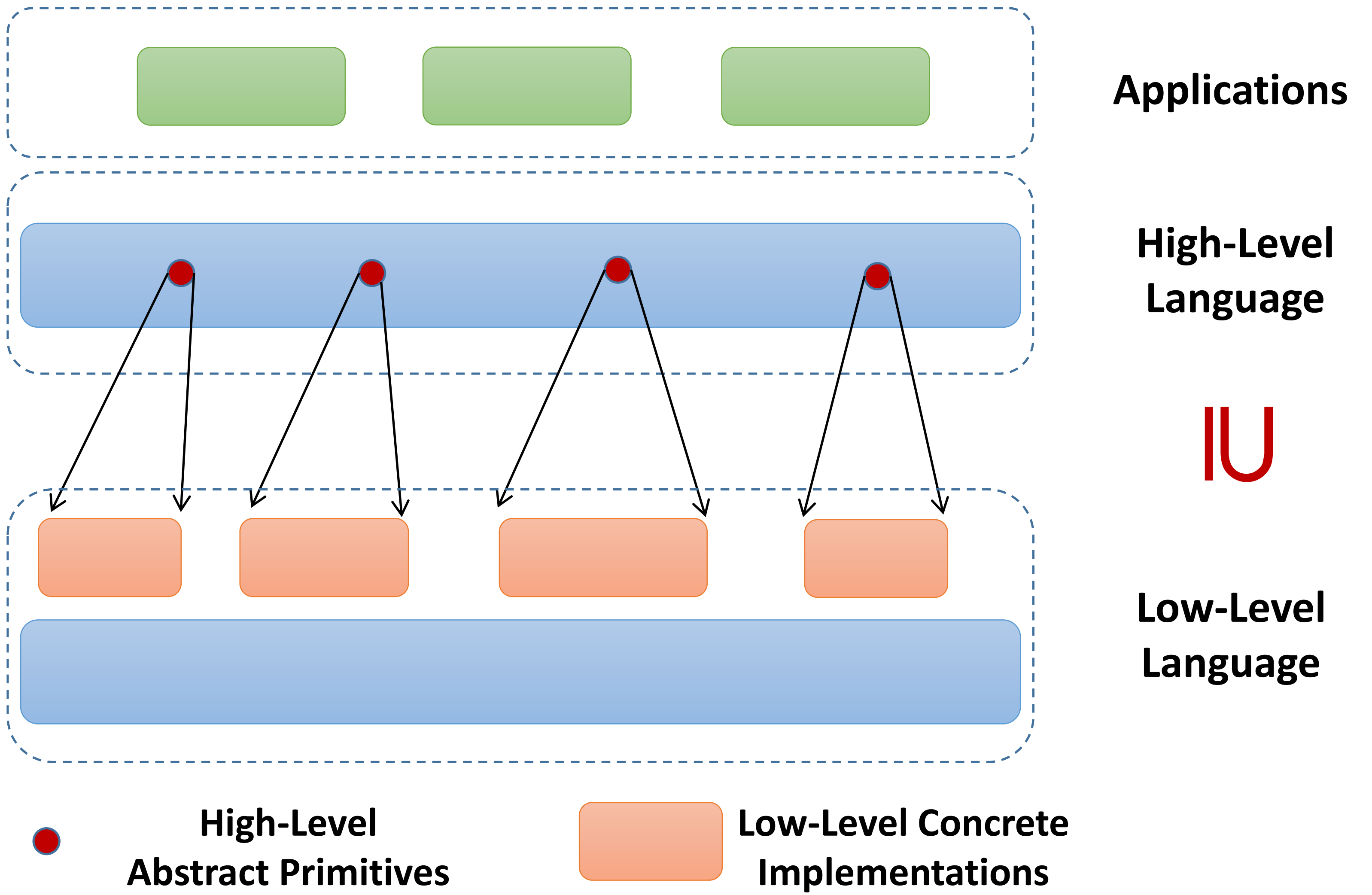
Outline

- OS Correctness Specification
- Verification Framework
  - System modeling
  - CSL-R: Program logic for refinement & multi-level interrupts
  - Coq tactics
- Verifying μC/OS-II

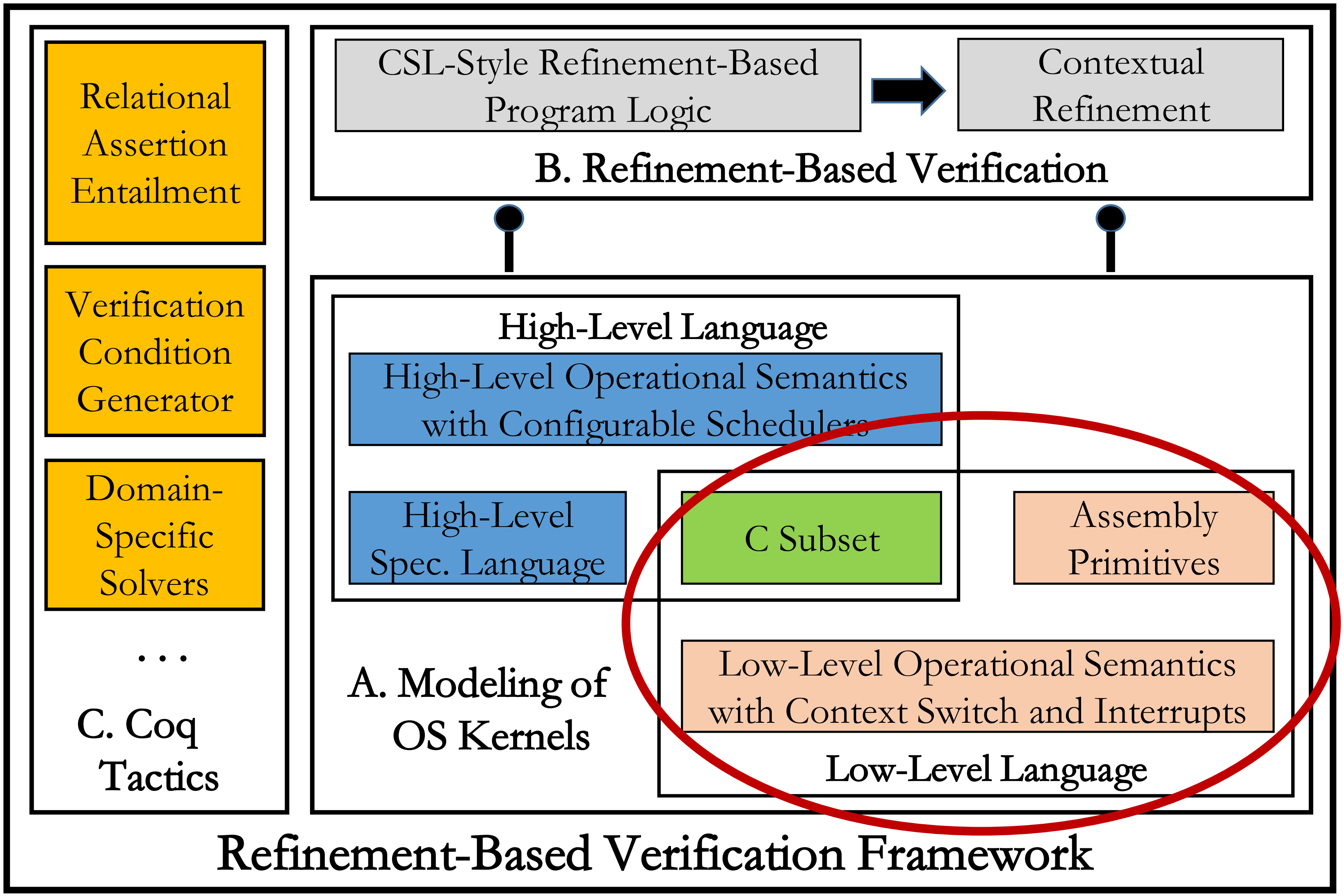
Our Verification Framework



OS Correctness



Our Verification Framework



The Low-Level Language

C Subset

L ::= C | Pr | L;L | · · ·

C ::= **while** e **{** C **}** | **if** e **{** C1 **}** **{** C2 **}** | f(e) | e=e | · · ·

e ::= &e | *e | e[e] | e.id | …

The Low-Level Language

```
#=====================================================================
OSCtxSw:                    # Task switching from task level
    pushfl
    pushal                  # Save current task's context
    mov OSTCBCur,%ebx
    mov %esp,(%ebx)         # OSTCBCur->OSTCBStkPtr = ESP
    call    OSTaskSwHook            # Call user defined task switch hook
    mov OSTCBHighRdy,%eax    # OSTCBCur <= OSTCBHighRdy
    mov %eax,OSTCBCur
    mov OSPrioHighRdy,%al    # OSPrioCur <= OSPrioHighRdy
    mov %al,OSPrioCur
    mov OSTCBHighRdy,%ebx    # ESP = OSTCBHighRdy->OSTCBStkPtr
    mov (%ebx),%esp
    popal
    popfl
    ret                 # Return to new task
#=====================================================================
```

Pr ::= encrt | excrt | switch | …

Explicit interrupts management and context switch

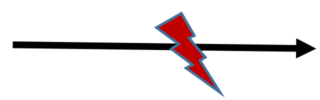
```
#define  OS_ENTER_CRITICAL()  __asm__("pushf \n\t cli") /* Disable interrupts*/
#define  OS_EXIT_CRITICAL()   __asm__("popf")      /* Enable  interrupts*/
```

Semantics

Small-step, even for expressions:
Try to be faithful to the granularity of machine-code

(x + y)/ 2 ⇒

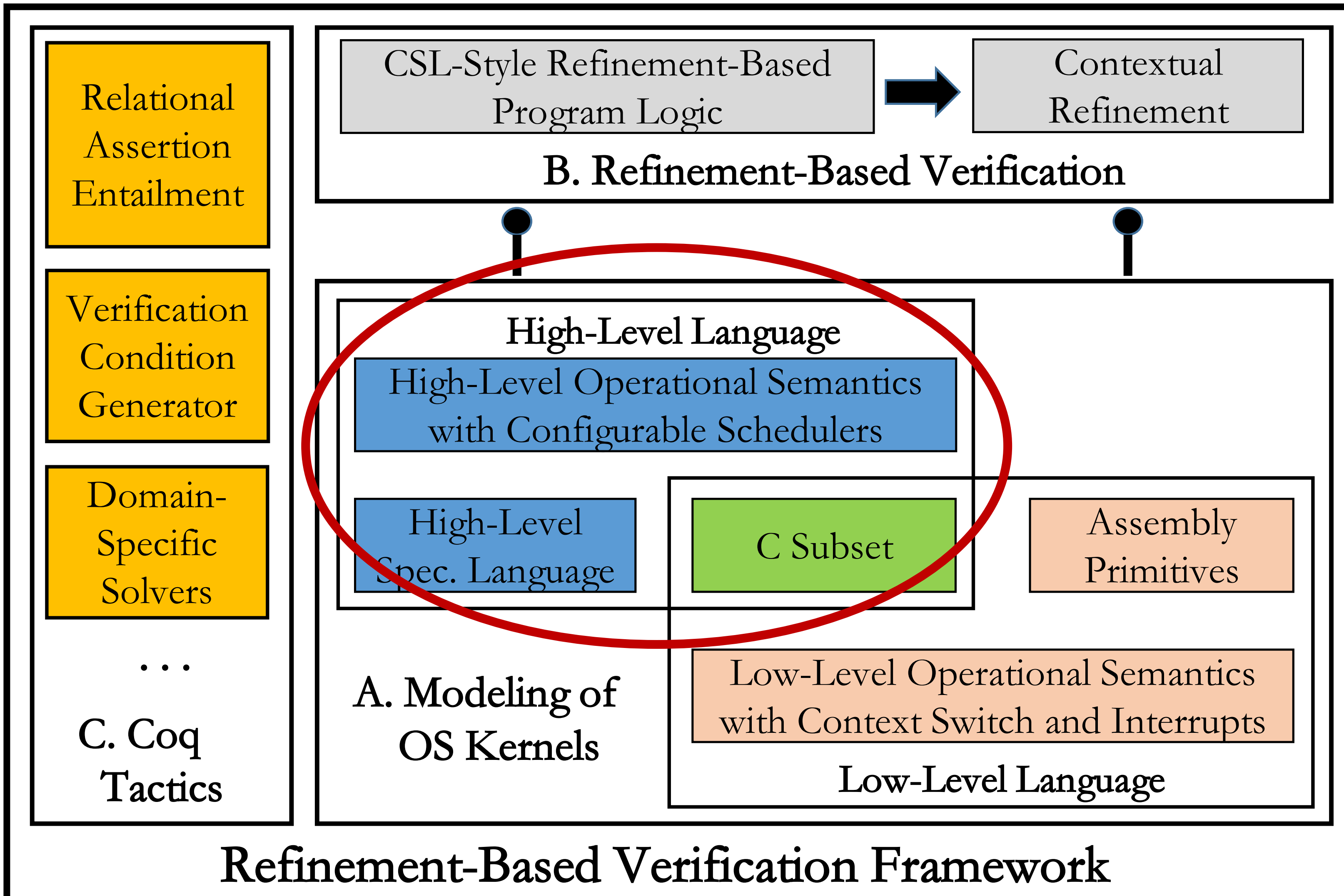
eval x
eval y
eval (x+y)
eval (x+y)/2



Interrupt handler:
y = y + 2

Semantics similar to CompCertTSO [Sevcik et al. 2011]
(but is interleaving semantics instead of TSO model)

Our Verification Framework



High-Level Language

C subset

High-level API specification language

H ::= C | S | H;H | ….

S ::= **sched** | **γ(v)** | S;S | S+S | ⋯

explicit scheduling points

High-Level Language

H ::= C | S | H;H | ….

S ::= **sched** | **γ(v)** | S;S | S+S | · · ·

abstract **atomic** transitions
(over the abstract kernel states)

Example

encrt

__asm__("pushf \n\t cli")
/*Disable interrupts*/

```
void  OSTimeDly (INT16U ticks)

> 0) {
    OS_ENTER_CRITICAL();
     ……

    OS_EXIT_CRITICAL();

S_Sched();

  return;
}
```

Suspend the current thread, and remove it from the READY thread queue

call scheduler

__asm__("popf")
/*Enable interrupts*/

excrt

Example

Low-level Code **VS.** **High-Level Spec**

```
void  OSTimeDly (INT16U ticks)
{
   if (ticks > 0) {
      OS_ENTER_CRITICAL();
       ……

      OS_EXIT_CRITICAL();

      OS_Sched();
   }
   return;
}
```

ticks <= 0

+

ticks>0;

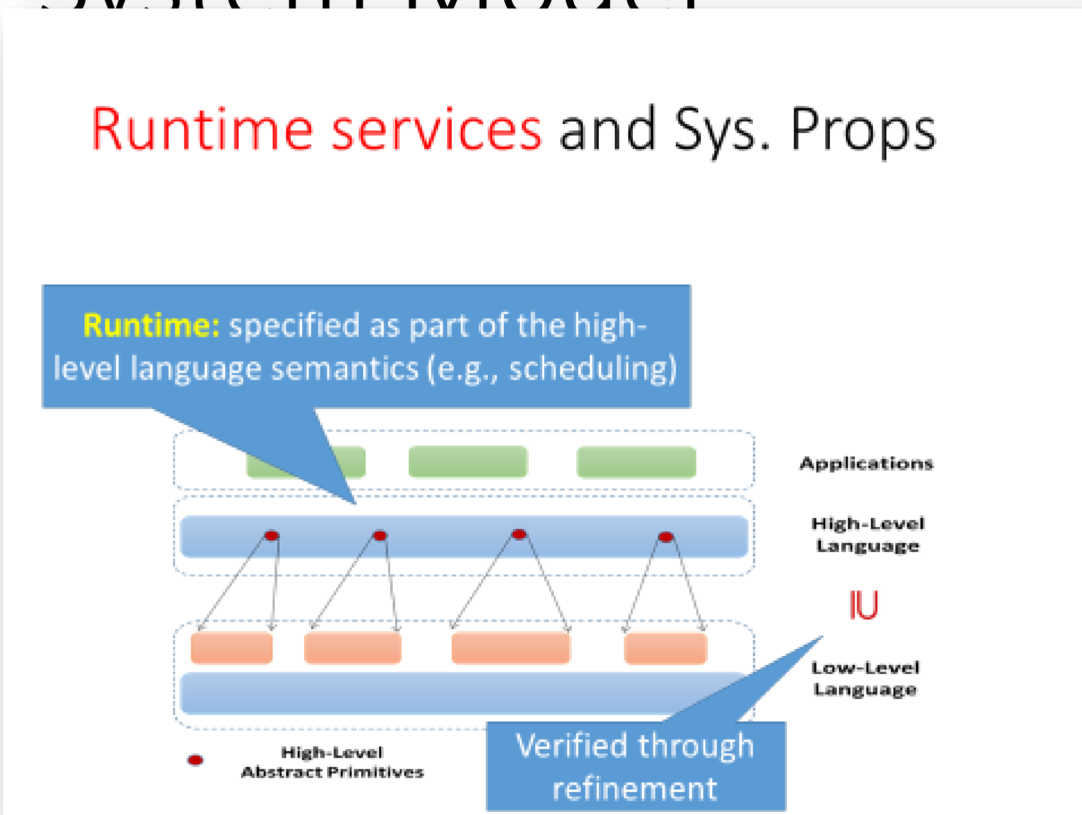
γ_{dly}(ticks);

sched

System Model

- Low-level impl. O: (η_a, θ, η_i)
  - η_a : API implementations
  - θ : Interrupt handlers
  - η_i : Internal functions
- High-level spec. S: $(\varphi, \varepsilon, \chi)$
  - φ : API specs. (high-level primitives for APIs)
  - ε : Abstract events (high-level primitives for int. handlers)
  - χ: Abstract scheduler
    - Scheduling policy can be customized by instantiating χ

System Model



for APIs)

itives for int. handlers)

- χ: Abstract scheduler
  - Scheduling policy can be customized by instantiating χ
  - Shows abstractions for runtime

System Model

- Low-level impl. O: (η_a, θ, η_i)
  - η_a : API implementations
  - θ : Interrupt handlers
  - η_i : Internal functions

Verification goal:

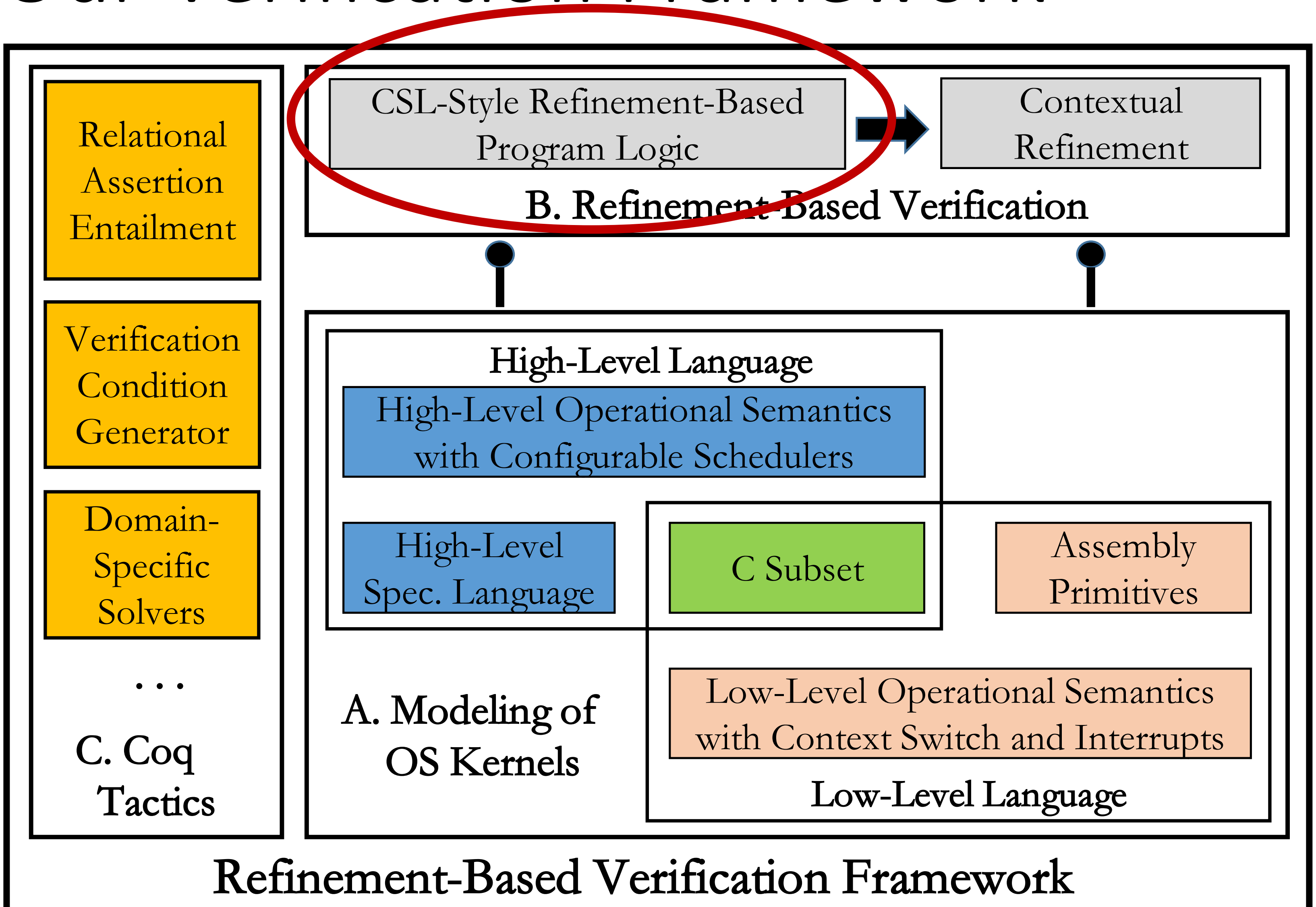
$$(\eta_a, \theta, \eta_i) \subseteq_{\mathbf{ctxt}} (\varphi, \varepsilon, \chi)$$

- High-level spec. S: $(\varphi, \varepsilon, \chi)$
  - φ : API specs. (high-level primitives for APIs)
  - ε : Abstract events (high-level primitives for int. handlers)
  - χ: Abstract scheduler
    - Scheduling policy can be customized by instantiating χ
    - Shows abstractions for runtime

Outline

- OS Correctness Specification
- Verification Framework
  - System modeling
  - CSL-R: Program logic for refinement & multi-level interrupts
  - Coq tactics
- Verifying μC/OS-II

Our Verification Framework



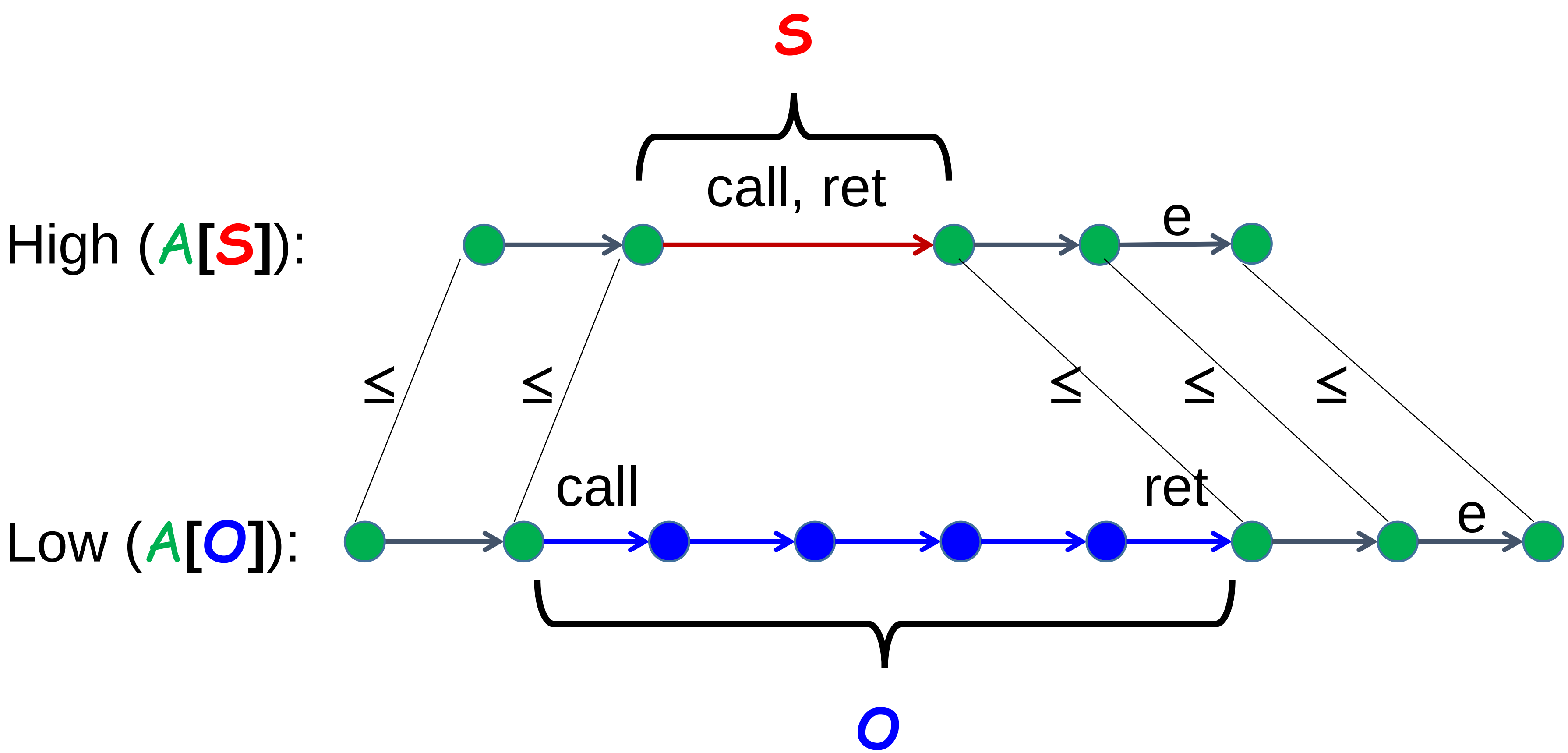
Program Logic for Refinement and Multi-Level Interrupts

- Relational program logic for simulation/refinements
  [Liang et al. PLDI'13, CSL-LICS'14]

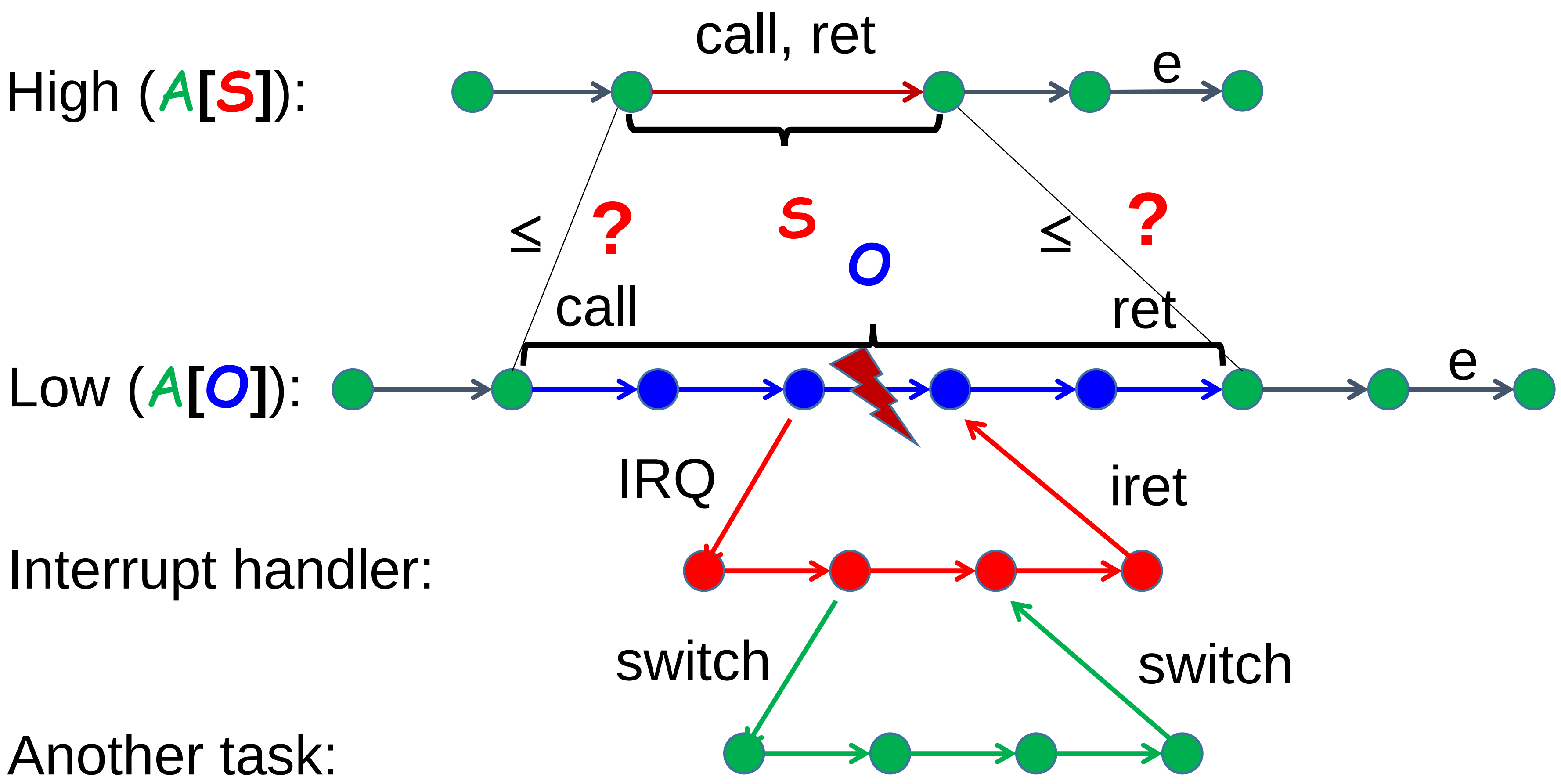
- Ownership-Transfer semantics for interrupts
  [Feng et al. PLDI'08]

- Combining the two: CSL-R for refinement reasoning with multi-level interrupts

Refinement Verification via Simulation

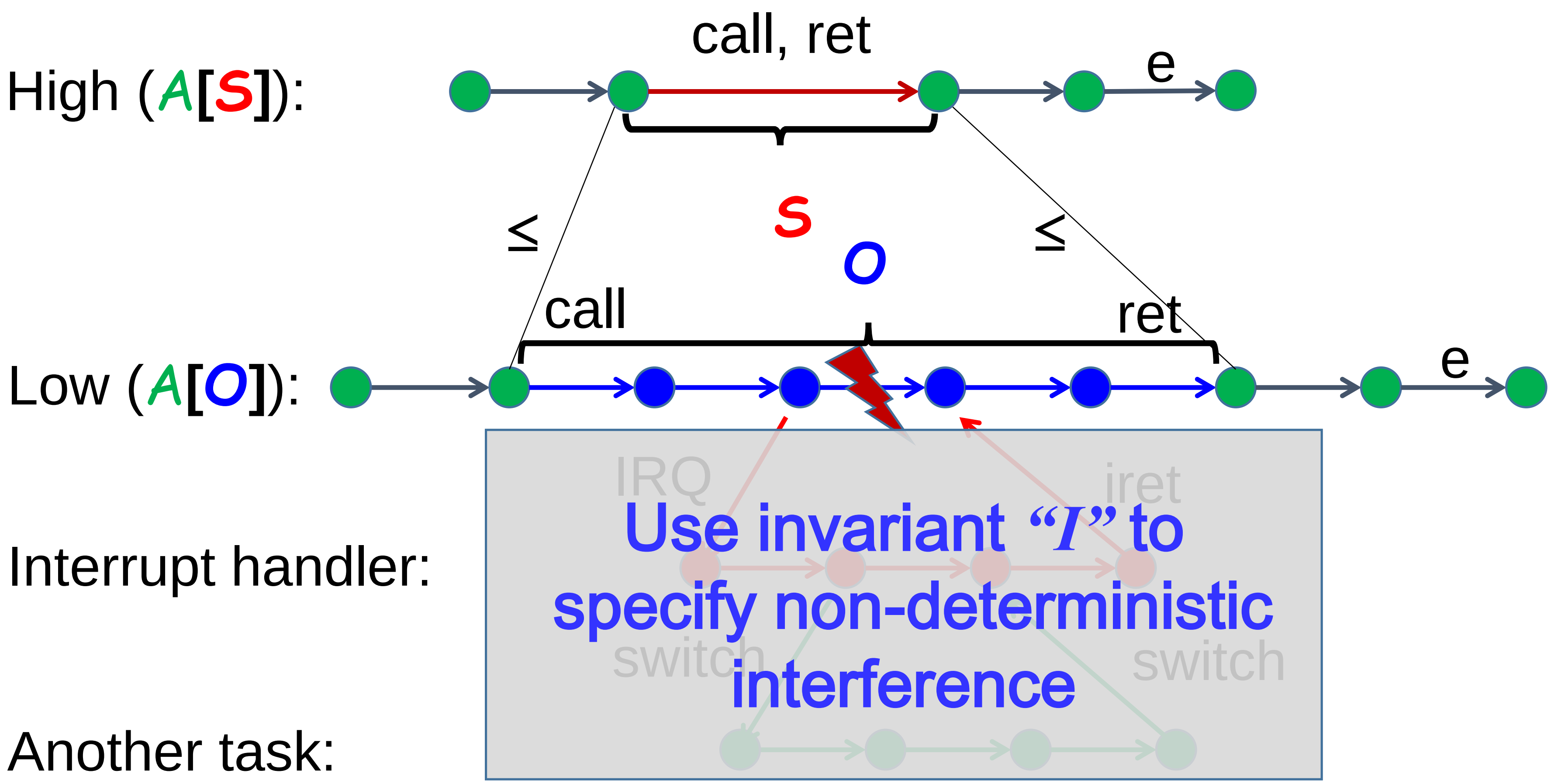


Simulation with Interrupts & Multitasking

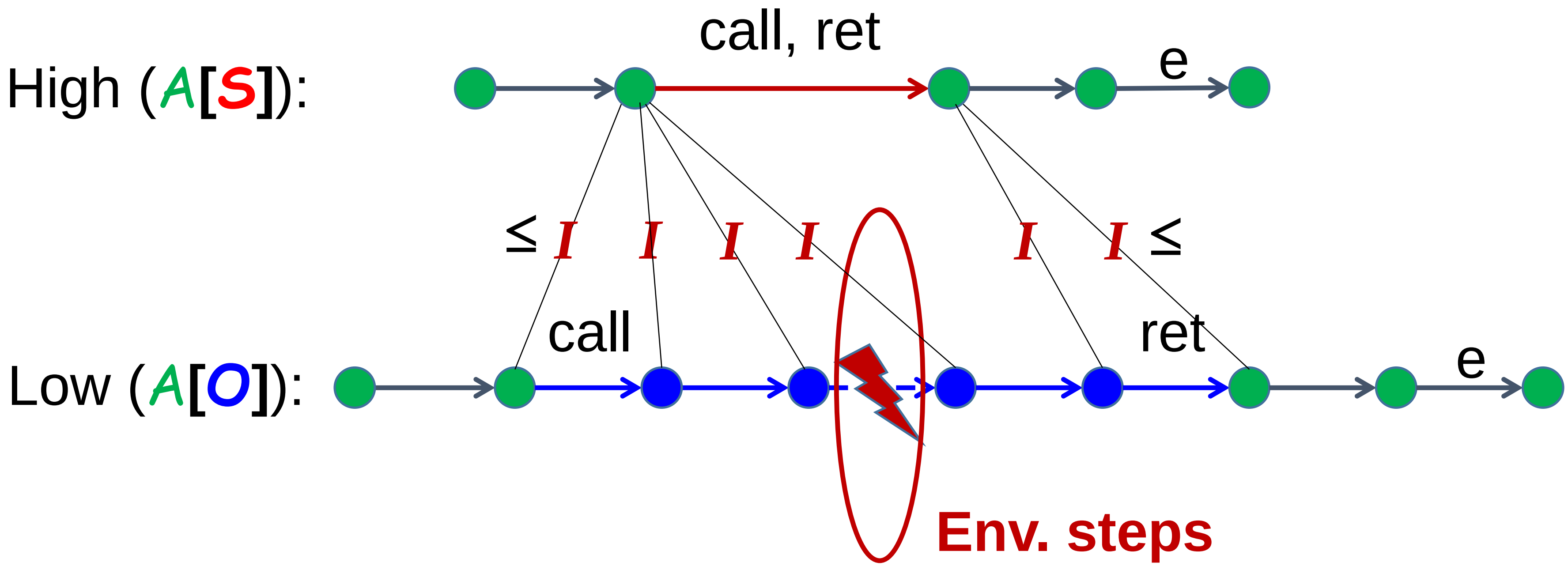


How to do compositional verification?

Simulation with Interrupts & Multitasking



Simulation with Interrupts & Multitasking



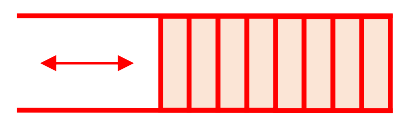
Adapted from RGSim [Liang et al. POPL'12] **and the relational program logic** [Liang et al. PLDI'13, CSL-LICS'14]

Program Logic for Simulation

High-Level
abstract primtive

S

High-Level
abstract states

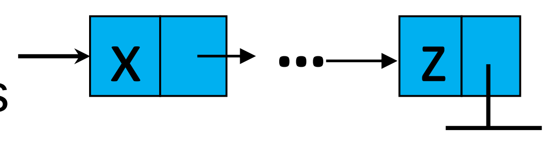


$\geq$

Low-Level
concrete code

C

Low-Level
concrete states

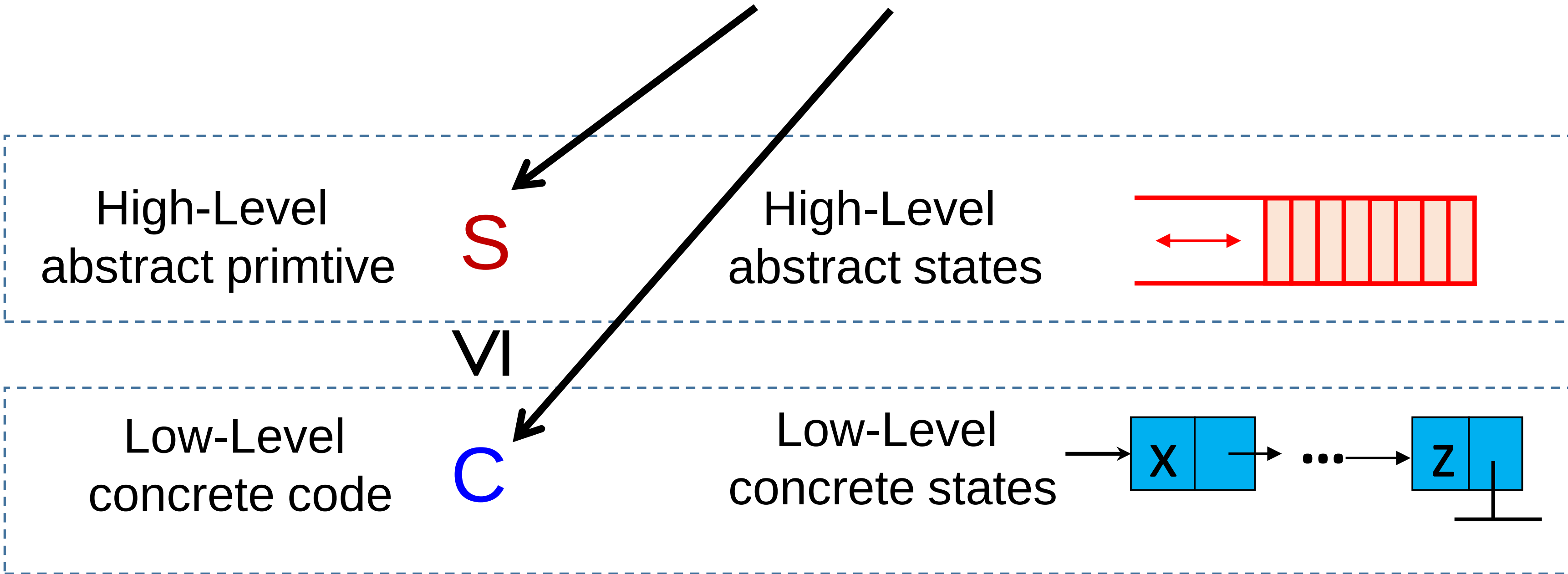


Program Logic for Simulation

Remaining high-level code that needs to be refined

- Judgement

$$I \vdash \{p * [|S|]\}\ C\ \{q * [|end|]\}$$



Program Logic for Simulation

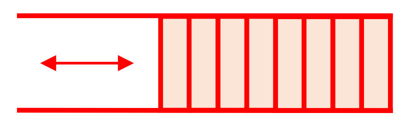
- Judgement

No remaining high-level code (refinement is done)

$$I \vdash \{p * [|S|]\}\ C\ \{q * [|end|]\}$$

High-Level abstract primtive S

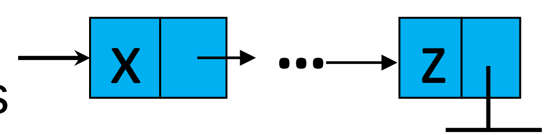
High-Level abstract states



$\leq$

Low-Level concrete code C

Low-Level concrete states

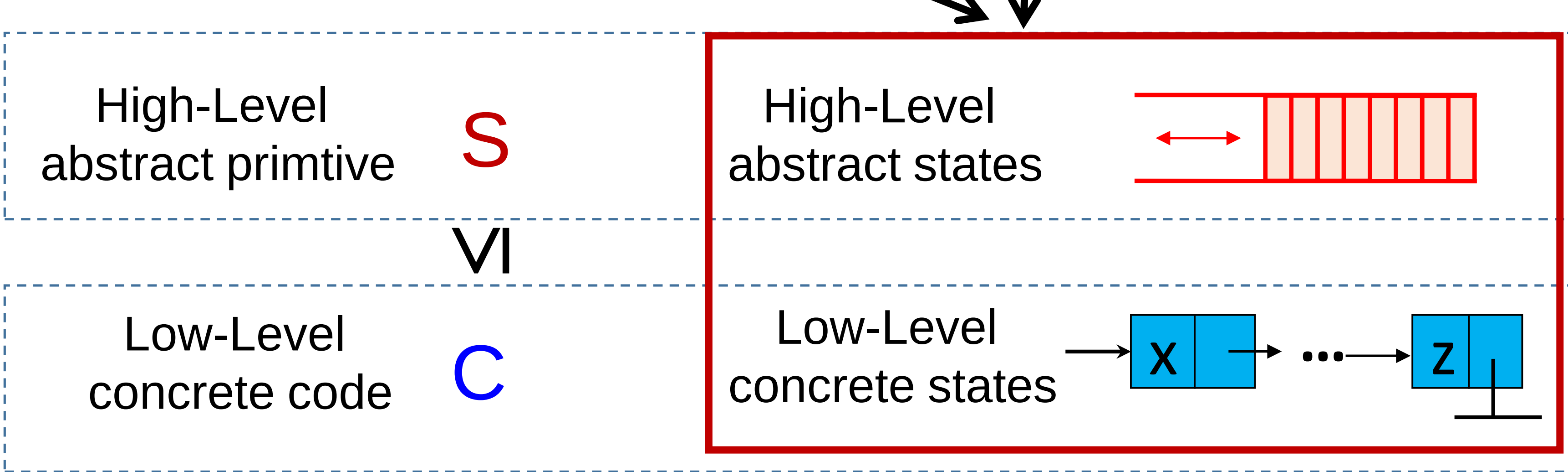


Program Logic for Simulation

- Judgement

Relational assertions for pre-/post- condition

$$I \vdash \{p * [|S|]\}\ C\ \{q * [|end|]\}$$

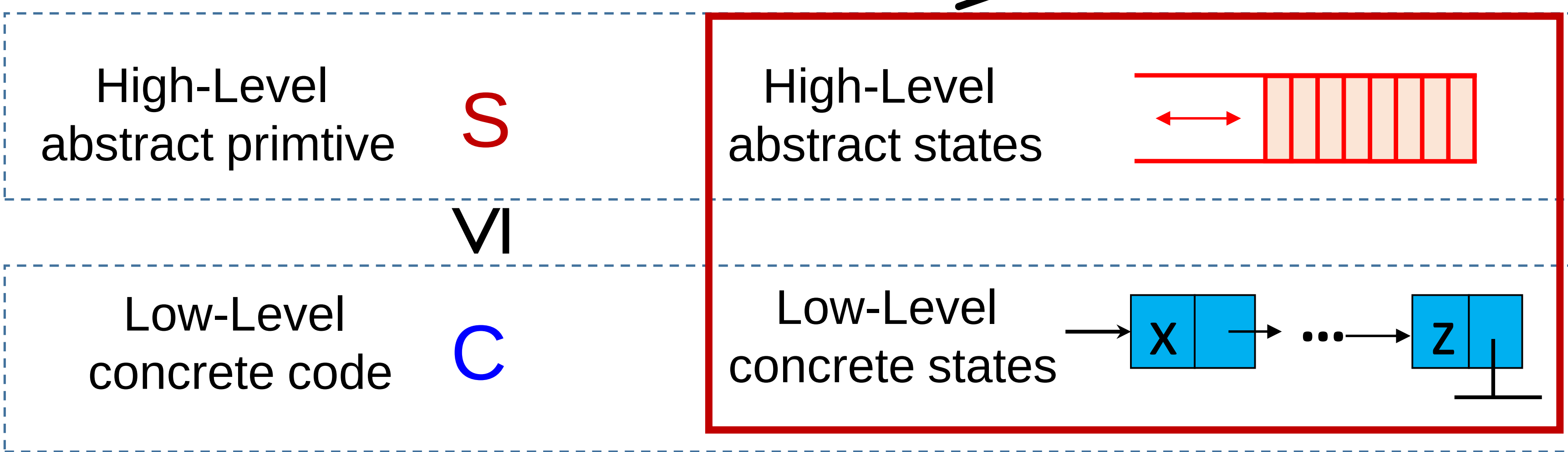


Program Logic for Simulation

- Judgement

Relational Invariants

$$I \vdash \{p * [\![S]\!]\}\ C\ \{q * [\![\mathrm{end}]\!]\}$$



Soundness

If

$$I \vdash \{p * [|S|]\}\ C\ \{q * [|end|]\}$$

then C is simulated by S, …

An Example

```
void Add() {
  OS_ENTER_CRITICAL();
    Count ++ ;
  OS_EXIT_CRITICAL();
}
```

$\leq$

< CNT++ >

$$I ::= \exists \mathrm{v}.\ \mathrm{Count} \rightarrow \mathrm{v}\ *\ \mathrm{CNT} = \mathrm{v}$$

An Example

{ [< CNT++ >] }

OS_ENTER_CRITICAL();

{ [< CNT++ >] * *I* }

{ [< CNT++ >] * Count→ v * CNT=v }

Count ++ ;

{ [< CNT++ >] * **Count→ v+1** * CNT=v}

{ [< end>] * Count→ v+1 * CNT=v+1}

{ [< end>] * *I* }

OS_EXIT_CRITICAL();

{ [<end>] }

Ownership transfer

Unfold *I*

Execute high-level code

Fold *I*

Ownership transfer

An Example

{ [< CNT++ >] }

OS_ENTER_CRITICAL();

Count ++ ;

OS_EXIT_CRITICAL();

{ [<end>] }

The code refines
<CNT++>

An Example

{ [< CNT++ >] }

OS_ENTER_CRITICAL();

{ [< CNT++ >] * Count→ v * CNT=v }

Count ++ ;

{ [< CNT++ >] * **Count→ v+1** * CNT=v}

Execute high-level code

{ [< end>] * Count→ v+1 * CNT=v+1}

OS_EXIT_CRITICAL();

{ [<end>] }

An Example

{ [< CNT++ >] }

OS_ENTER_CRITICAL();

Abstract consequence rule:

$$\frac{p*[S] \Longrightarrow r*[S'] \qquad \vdash \{ r * [S'] \}\ C\ \{ q \}}{\vdash \{ p * [S] \}\ C\ \{ q \}}$$

Count ++ ;

{ [< CNT++ >] * **Count→ v+1** * CNT=v}

{ [<end>] * Count→ v+1 * CNT=v+1}

Execute high-level code

OS_EXIT_CRITICAL();

{ [<end>] }

$p \Longrightarrow q$ **iff** $\forall(\sigma, \Sigma, S) \models p,$

$\exists (\Sigma', S'). (\Sigma, S) \rightarrow^* (\Sigma', S')$

$\wedge (\sigma, \Sigma', S') \models q,$

An Example

{ [< CNT++ >] }

OS_ENTER_CRITICAL();

{ [< CNT++ >] * *I* }

Ownership transfer

Count ++ ;

Interrupt reasoning

{ [< end>] * *I* }

OS_EXIT_CRITICAL();

{ [<end>] }

Ownership transfer

Interrupt Reasoning

Program invariant [O'Hearn CONCUR'04]

There is always a partition of resource among concurrent entities, and each concurrent entity only accesses its own part.

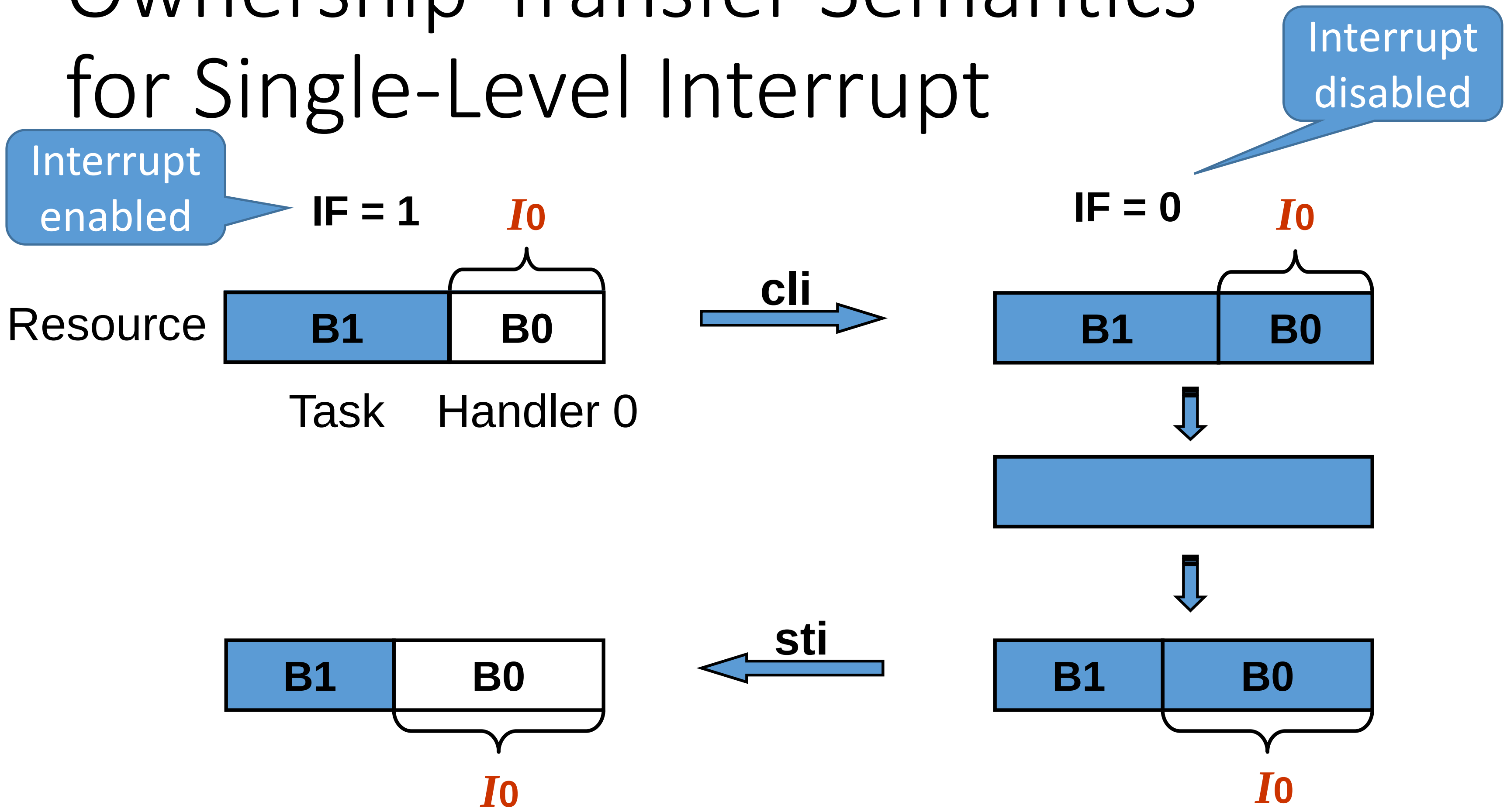
Tasks and interrupt handlers

But note:

The partition is dynamic: ownership of resource can be dynamically transferred.

Interrupt operations can be modeled as operations that trigger resource ownership transfer. [Feng et al. PLDI'08]

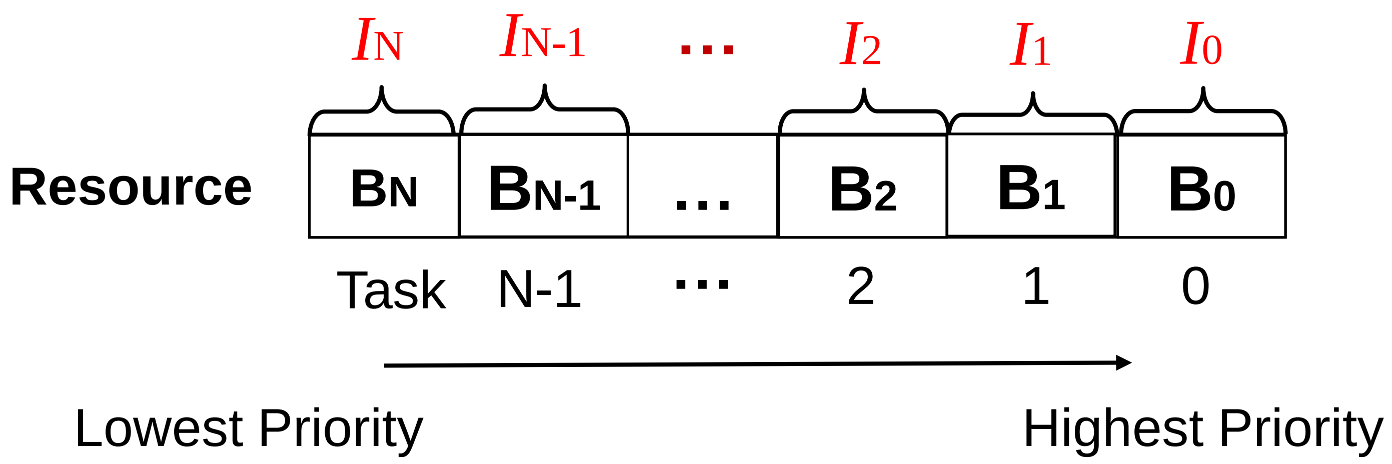
Ownership-Transfer Semantics for Single-Level Interrupt



$$I_0 \vdash \{p\}\ \mathbf{cli}\ \{p * I_0\}$$

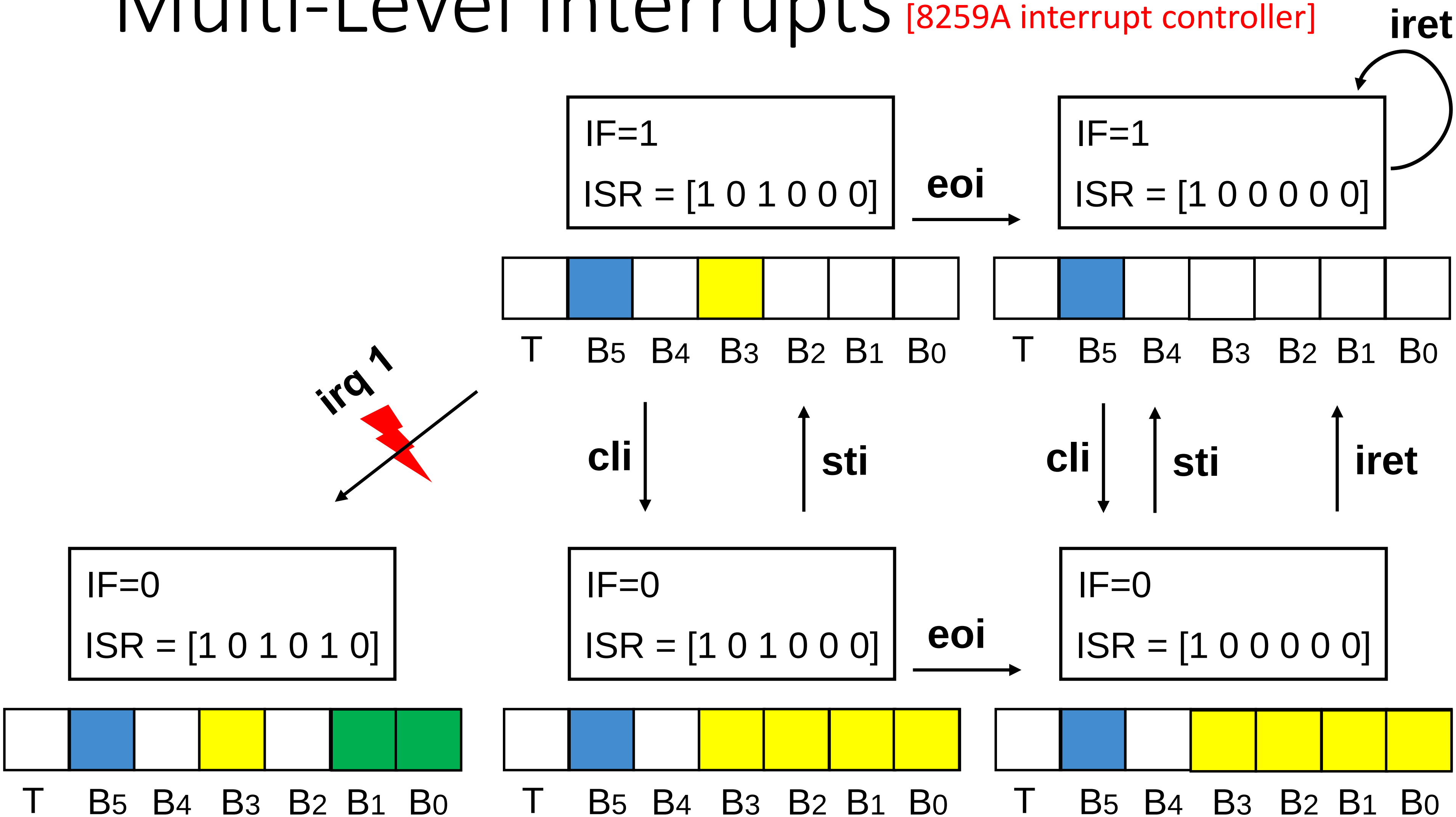
$$I_0 \vdash \{p * I_0\}\ \mathbf{sti}\ \{p\}$$

Memory Model for Multi-Level Interrupts

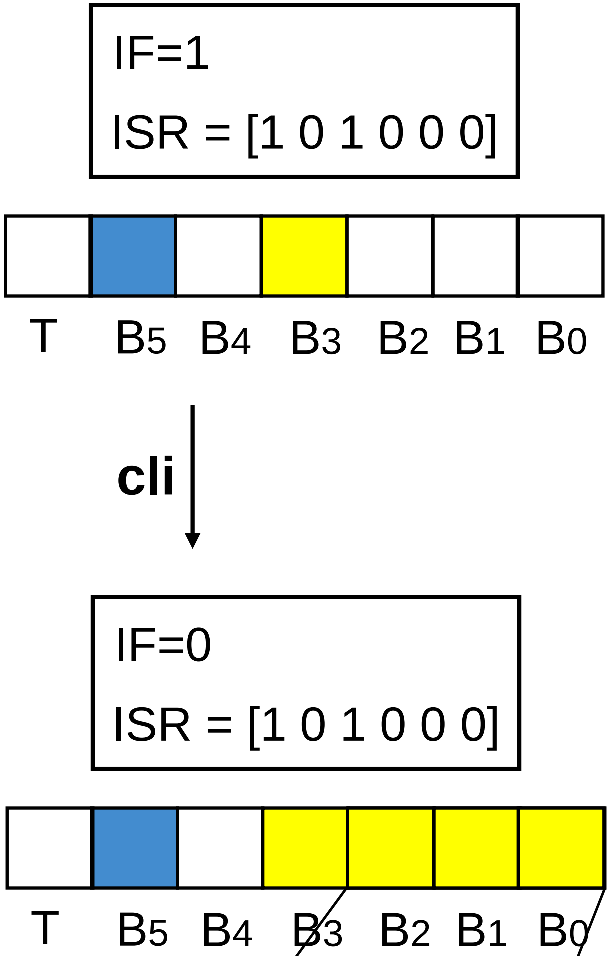


- Higher-priority handler has priority to select its required resource
- N blocks are assigned to N interrupt handlers
- Each well-formed resource block is specified by a resource invariant

Ownership-Transfer Semantics for Multi-Level Interrupts [8259A interrupt controller]



Inference Rules for Interrupt Operations



$$\frac{ISR(k) = 1}{I \vdash \{ [ISR, 1, k] * p \}\ \textbf{cli}\ \{ [ISR, 0, k] * p * \boldsymbol{I\,[0\ldots k-1]} \}}$$

Top Rule for Proving $O \subseteq_{ctxt} S$

Verifying internal functions

Verifying kernel APIs

Verifying interrupt handlers

$$\frac{\chi, I \vdash \eta_i : \Gamma \qquad \Gamma, \chi, I \vdash \eta_a : \varphi \qquad \Gamma, \chi, I \vdash \theta : \varepsilon \qquad \text{Side conditions}}{O \subseteq_{ctxt} S}$$

abstract primitivesfor kernel APIs

kernel APIs

(η_a, θ, η_i) $\qquad$ $(\varphi, \varepsilon, \chi)$

interrupt handlers

internal functions

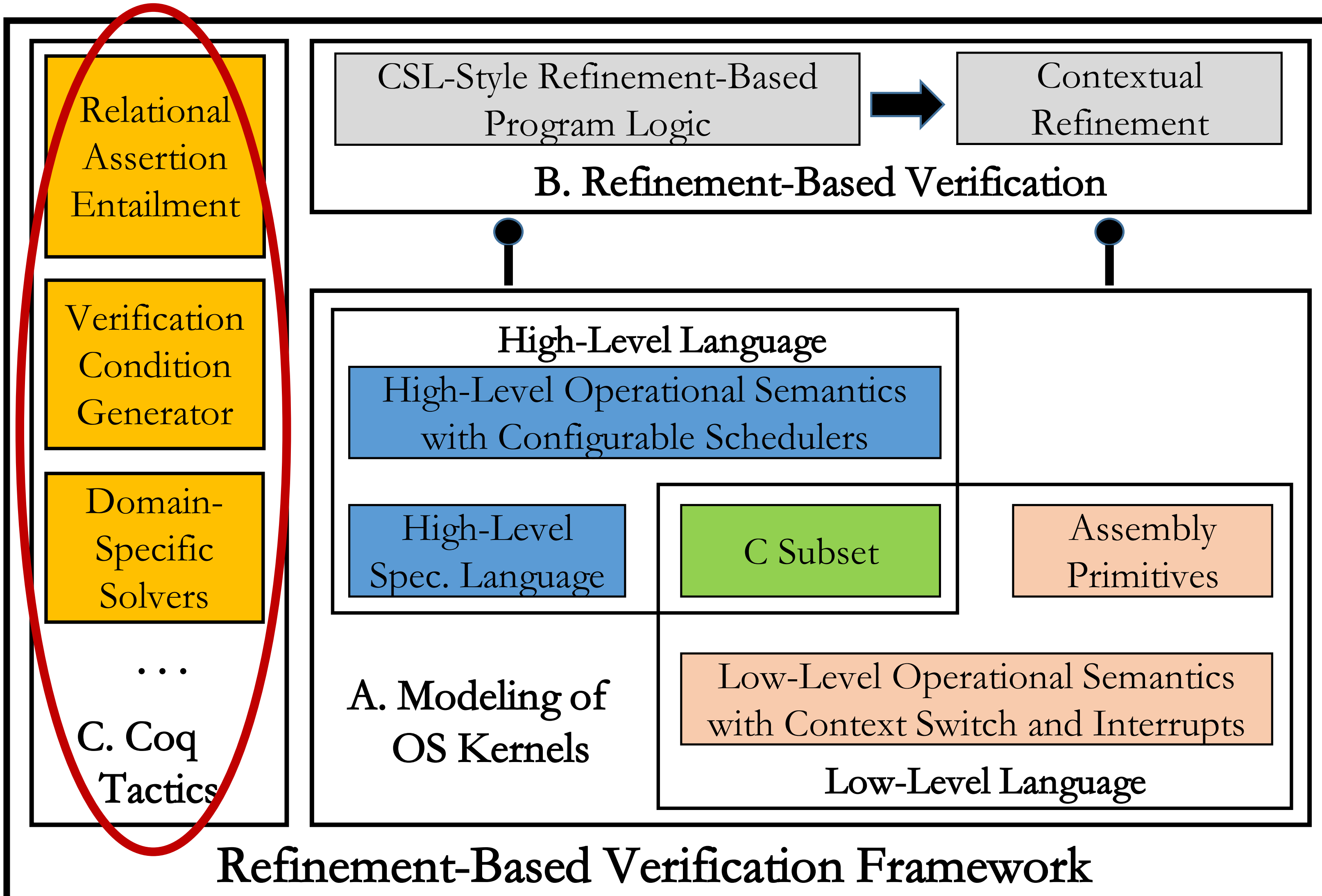
abstract primitives for

abstract scheduler

Outline

- OS Correctness Specification
- Verification Framework
  - System modeling
  - CSL-R: Program logic for refinement & multi-level interrupts
  - Coq tactics
- Verifying μC/OS-II

Our Verification Framework



Coq Tactics for Automation Support

- Verification condition generator : **hoare forward**
  - Automatically select and apply the inference rules

- Assertion entailment prover : **sep auto**
  - Automatically prove “p => q”

- Domain specific solvers : **mauto …**

$$\forall x. x < 64 \rightarrow x \gg 3 < 8 \; ; \quad \forall x. x < 8 \rightarrow (x \ll 3) \& 7 = 0$$

Coq Tactics for Automation Support

- To reduce the proof efforts
- To hide the underlying details of the verification framework
- To prove domain specific propositions

The ratio of Coq scripts to the verified C is around 27:1

lots of space for improvement

Outline

- OS Correctness Specification
- Verification Framework
  - System modeling
  - CSL-R: Program logic for refinement & multi-level interrupts
  - Coq tactics
- Verifying µC/OS-II

μC/OS-II

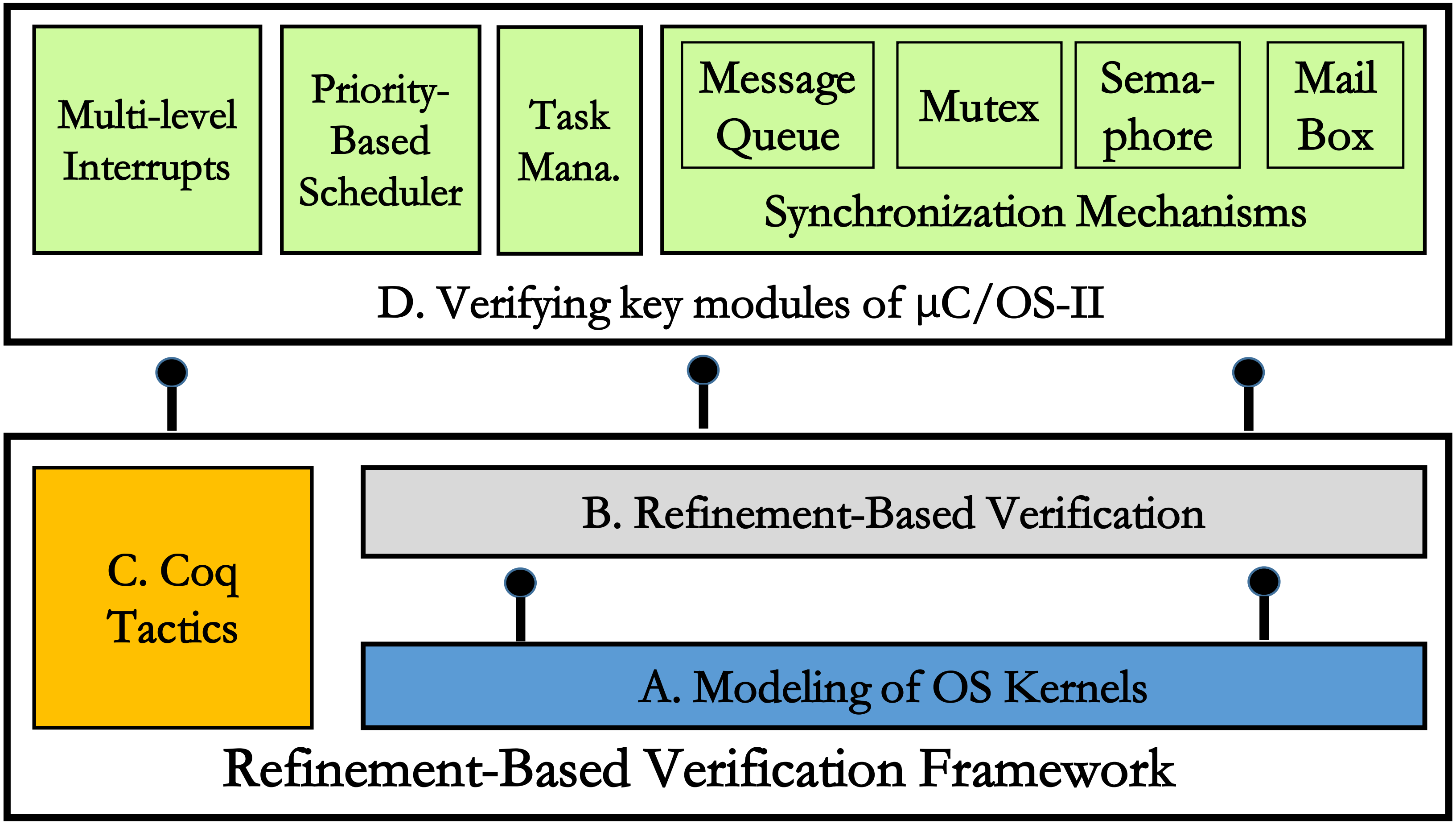
- A commercial preemptive real-time multitasking OS kernel developed by Micrium.

- 6,316 lines of C & 316 lines of assembly code.

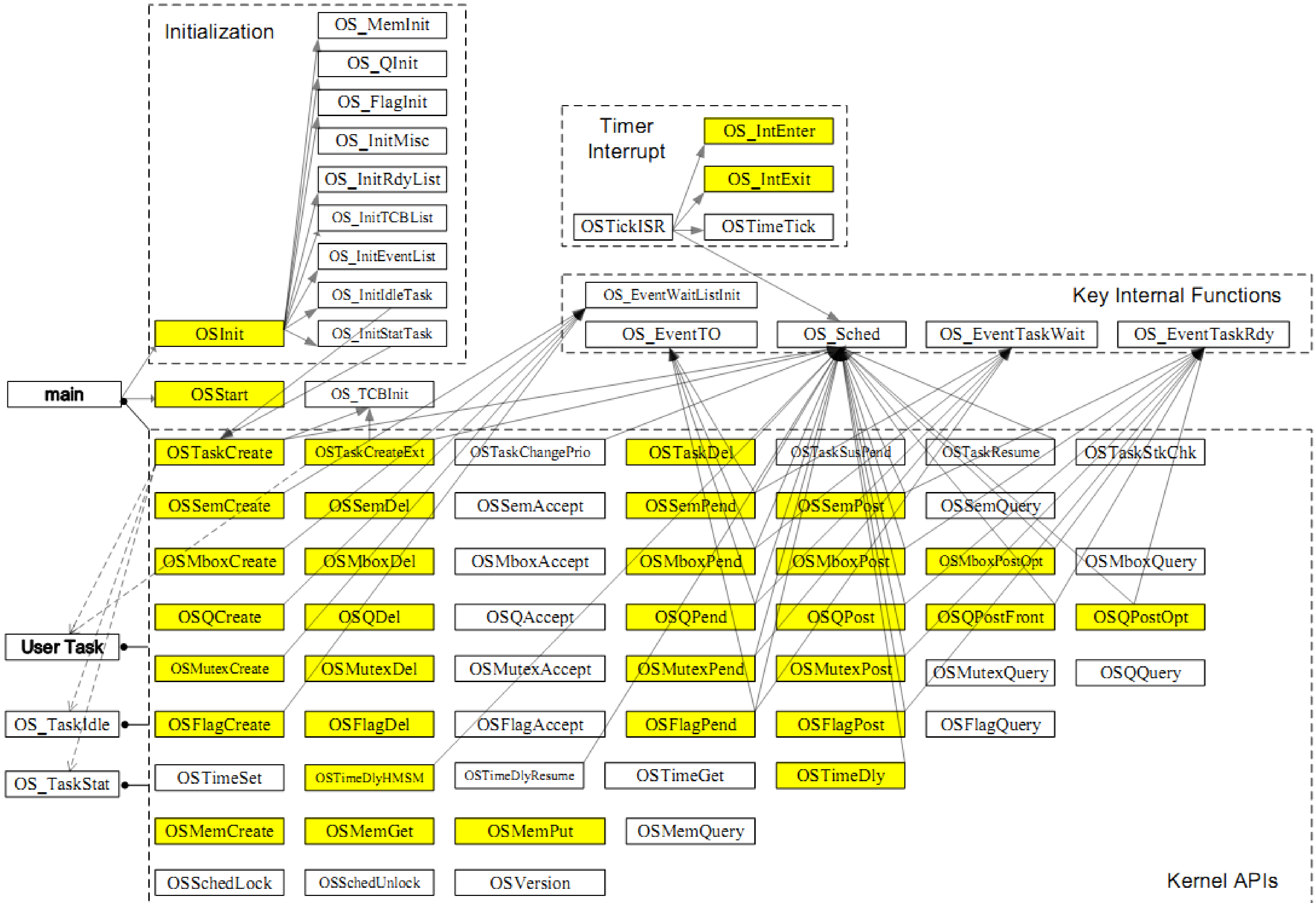
- Multitasking & Multi-Level interrupts & Preemptive priority-based scheduling & Synchronization mechanism

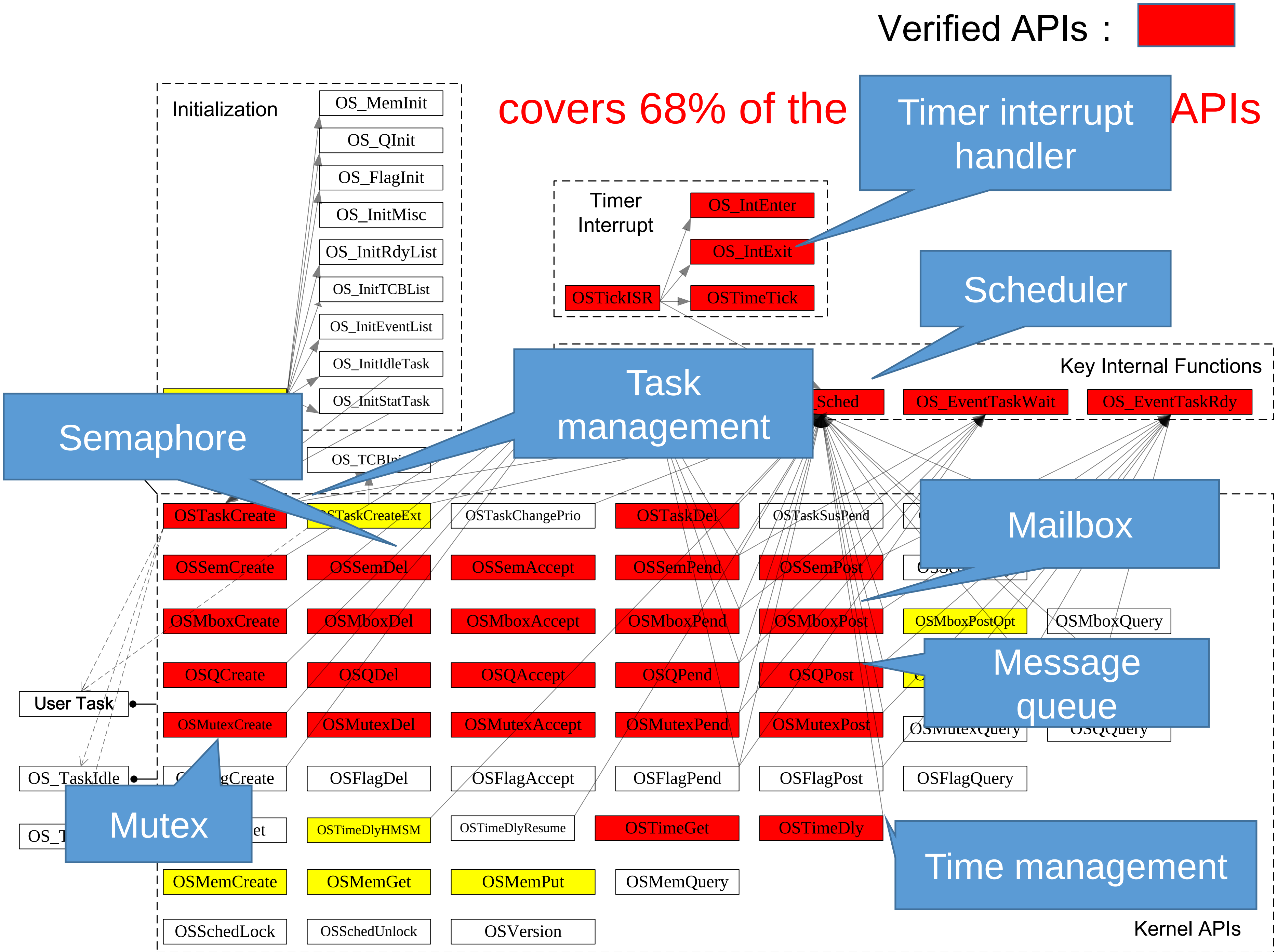
- Deployed in many real-world safety critical applications
  - Avionics and medical equipments, *etc*.

Verifying µC/OS-II



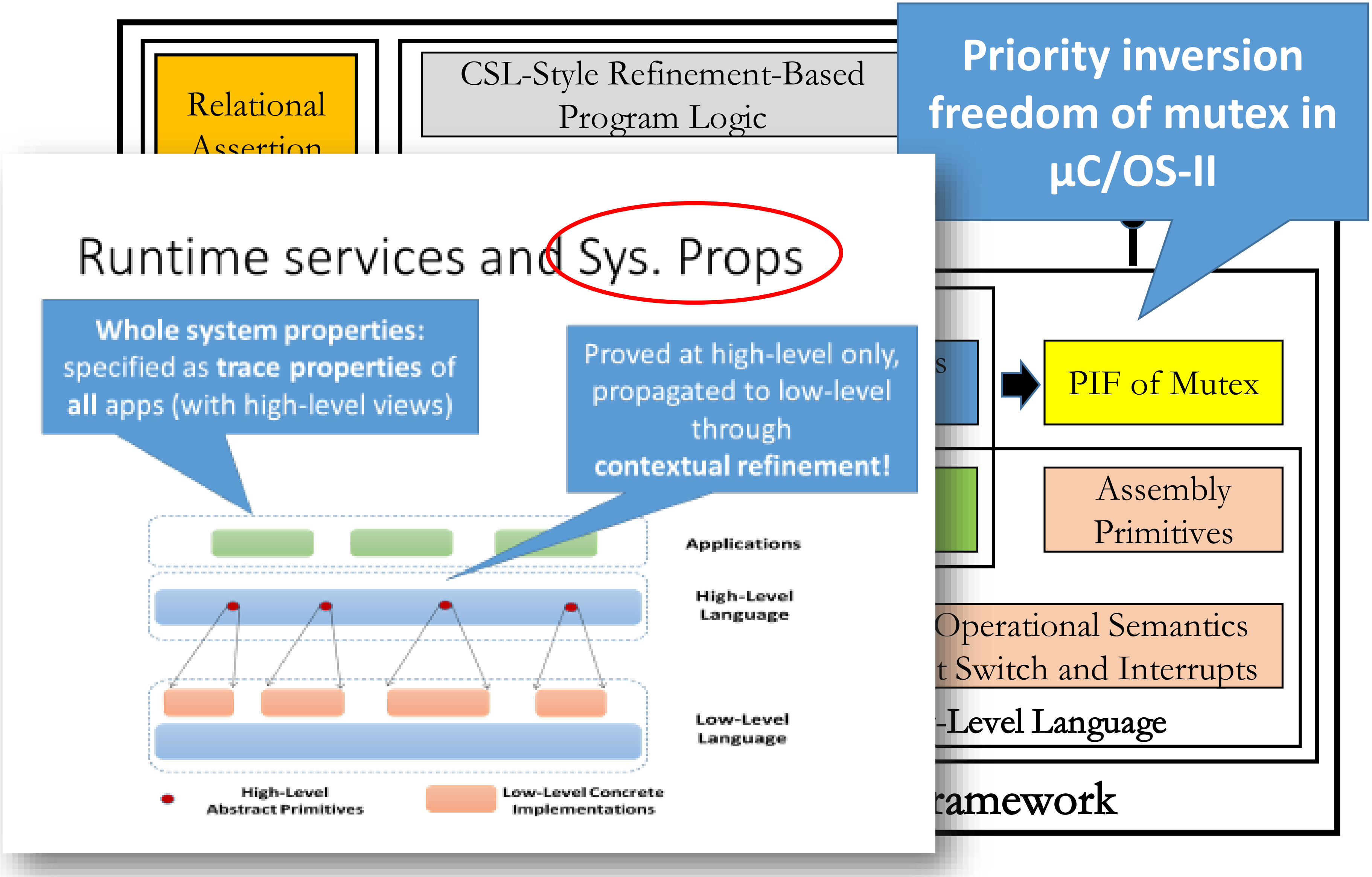
Frequently Used APIs :





Verified APIs :
covers 68% of the APIs
Timer interrupt handler
Scheduler
Task management
Semaphore
Mailbox
Message queue
Mutex
Time management
Initialization
OS_MemInit
OS_QInit
OS_FlagInit
OS_InitMisc
OS_InitRdyList
OS_InitTCBList
OS_InitEventList
OS_InitIdleTask
OS_InitStatTask
Timer Interrupt
OS_IntEnter
OS_IntExit
OSTickISR
OSTimeTick
Key Internal Functions
OS_EventTaskWait
OS_EventTaskRdy
User Task
OSTaskCreate
OSTaskChangePrio
OSTaskDel
OSTaskSusPend
OSSemCreate
OSSemDel
OSSemAccept
OSSemPend
OSSemPost
OSMboxCreate
OSMboxDel
OSMboxAccept
OSMboxPend
OSMboxPost
OSMboxPostOpt
OSMboxQuery
OSQCreate
OSQDel
OSQAccept
OSQPend
OSQPost
OSMutexCreate
OSMutexDel
OSMutexAccept
OSMutexPend
OSMutexPost
OSFlagDel
OSFlagAccept
OSFlagPend
OSFlagPost
OSFlagQuery
OSTimeDlyHMSM
OSTimeDlyResume
OSTimeGet
OSTimeDly
OSMemCreate
OSMemGet
OSMemPut
OSMemQuery
OSSchedLock
OSSchedUnlock
OSVersion
Kernel APIs

Proving Priority Inversion Freedom



Bugs found in μC/OS-II

- Priority Inversion Freedom in Mutex
  - Use a simplified priority ceiling protocol

Limitation of Mutex

µC/OS-II®
The Real-Time Kernel

CPUs Download How to Buy

Mutual exclusion semaphores with built-in priority ceiling protocol to prevent priority inversions

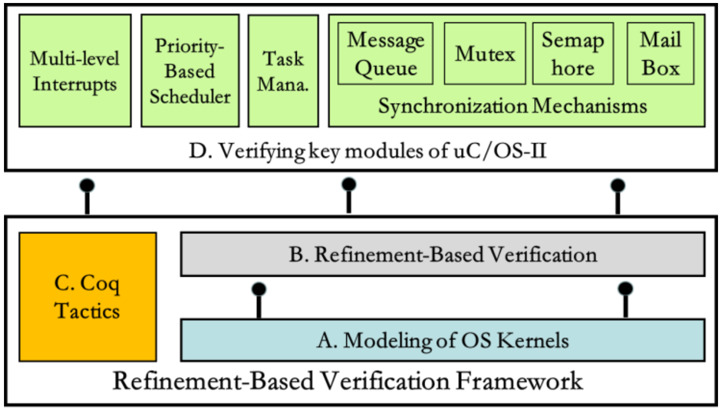
- Delivered with complete, clean, consis… 100% ANSI C source code with in-depth documentation.
- Mutual exclusion semaphores with built-in priority ceiling protocol to prevent priority inversions
- Timeouts on ‘pend’ calls to prevent deadlocks
- Up to 254 application tasks (1 task per priority level), and unlimited number of kernel objects
- Highly scalable (6K to 24K bytes code space, 1K+ bytes data space)
- Very low interrupt disable time
- Third party certifiable

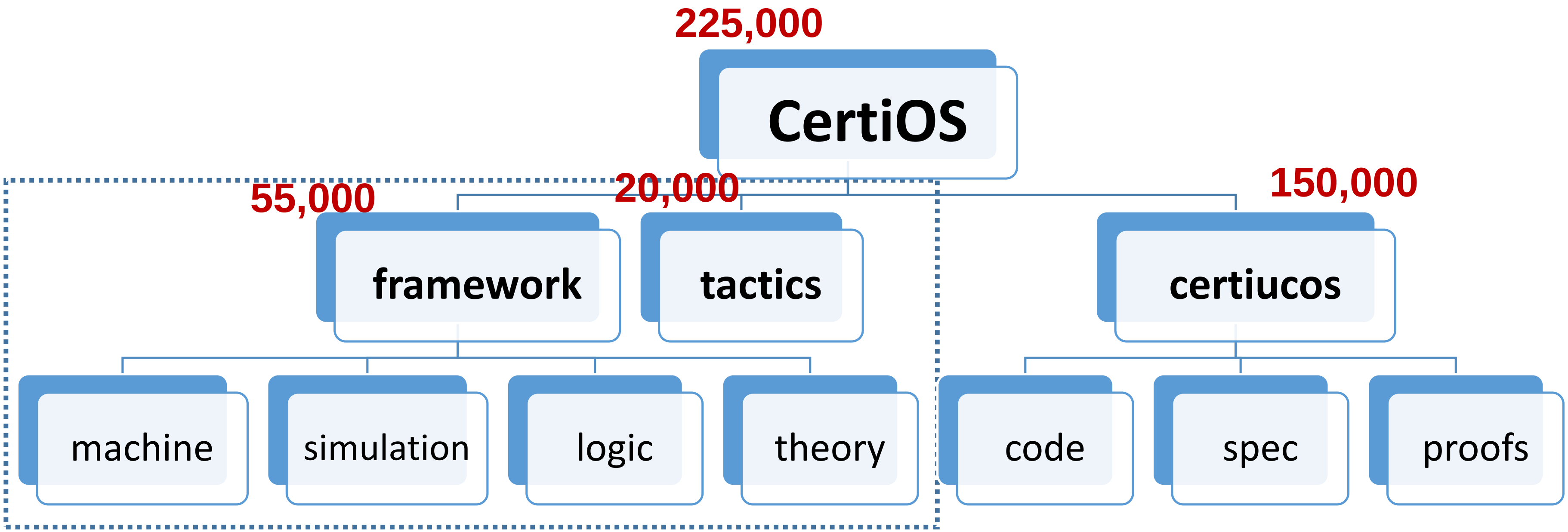
Bugs found in μC/OS-II

- Priority Inversion Freedom in Mutex
  - Use a simplified priority ceiling protocol
  - **May cause priority inversion with nested use of mutex!**
  - Fixed in μC/OS-III
- Concurrency bug (atomicity violation)

  INT8U **OSTaskChangePrio** (INT8U oldprio, INT8U newprio)
  - **May lead to access of invalid pointers**
  - Found in μC/OS-II v2.52 (the version we verified)
  - Fixed in μC/OS-II v2.9

Coq Implementations





Time cost: 6 person-years

Components	Cost (py)
Basic framework design and impl.	4
First module: message queue （750 lines of C）	1
the other modules （3000 loc）	1
Total	6

Debugging and fixing framework, specifications, tactics, etc.

Verification can be much faster with stable framework, tools and libraries

http://staff.ustc.edu.cn/~fuming/research/certiucos/

Conclusion

- Contextual refinements:
  a natural correctness formulation for OS kernels
- Verification framework for preemptive kernels
  - CSL-R: Concurrency refinement + hardware interrupts
- Verification of μC/OS-II
  - Commercial system independently developed by third-party

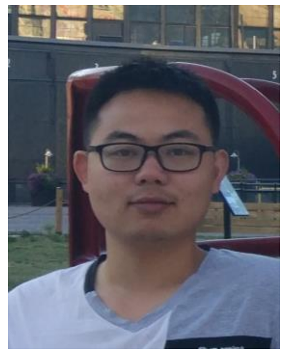
Limitations & Future Work

- No termination proofs
  - Relatively simple, can be done in logic or using tools
- Assembly and compiler are not verified
  - Ongoing work
- No separate addr. space and isolation
- No real-time properties
- More whole-system properties, in addition to PIF
- Improvements for automation (better tools and libs)

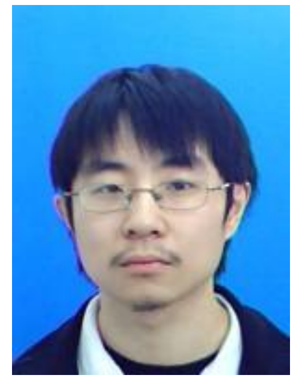
Acknowledgments: Group Members

Ming Fu

Fengwei Xu



Xiaoran Zhang

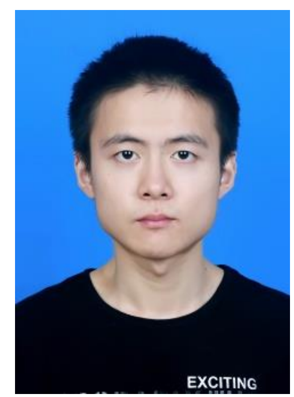


Zhaohui Li

Hui Zhang

Ding Ma

Haibo Gu



Alumni: Jingyuan Cao, Jiebo Ma

Thank you!