Verifying Periodic Programs with Priority Inheritance Locks

Sagar Chaki[1], Arie Gurfinkel[1],
Ofer Strichman[2]

FMCAD, October 22, 2013

[1]Software Engineering Institute, CMU
[2]Technion, Israel Institute of Technology

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

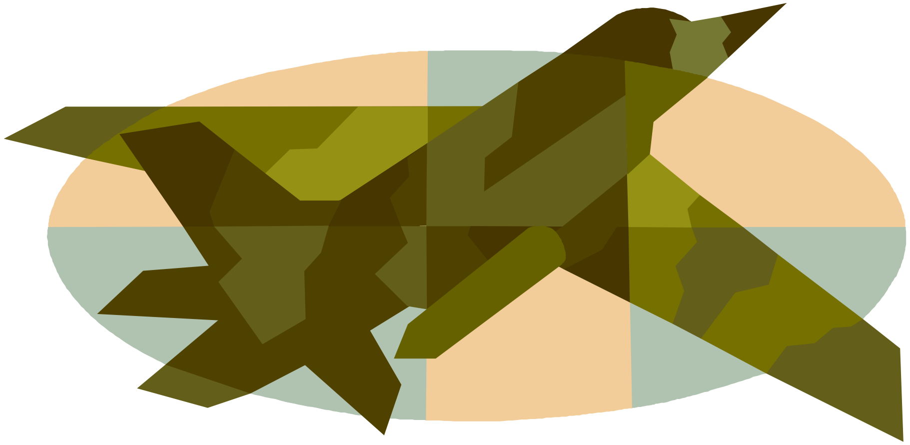
DM-0000695

Periodic Embedded Real-Time Software

Avionics Mission System*

Rate Monotonic Scheduling (RMS)

Task	Period
weapon release	10ms
radar tracking	40ms
target tracking	40ms
aircraft flight data	50ms
display	50ms
steering	80ms



Domains: Avionics, Automotive
OS: OSEK, VxWorks, RTEMS
We call them **periodic programs**

*Locke, Vogel, Lucas, and Goodenough. "Generic Avionics Software Specification". SEI/CMU Technical Report CMU/SEI-90-TR-8-ESD-TR-90-209, December, 1990

Context: Time-Bounded Verification [FMCAD'11, VMCAI'13]

Periodic Program

- Collection of periodic tasks
  - Execute concurrently with preemptive priority-based scheduling
  - Priorities respect RMS
  - Communicate through shared memory

Time-Bounded Verification

- Assertion A violated within X ms of a system's execution from initial state I?
  - A, X , I are user specified
  - Time bounds map naturally to program's functionality (e.g., air bags)

Locks

- **CPU**-locks, **priority ceiling protocol** locks [FMCAD'11, VMCAI'13]
- **priority inheritance protocol** locks

Main focus of this paper

Periodic Program (PP)

An N-task periodic program PP is a set of tasks $\{\tau_1, \ldots, \tau_N\}$

A task τ is a tuple $\langle I, T, P, C, A \rangle$, where

- I is a task identifier = its priority
- T is a task body (i.e., code)
- P is a period
- C is the worst-case execution time
- A is the *release time:* the time at which task becomes first enabled

Semantics of PP bounded by time X is given by an asynchronous concurrent program:

```
||  k_i = 0;
    while (k_i < J_i && Wait(τ_i, k_i))
      T_i ();
      k_i = k_i + 1;
```

parallel execution w/ priorities

blocks τ_i until time $A_i + k_i \times P_i$

$J_i = X / P_i$

Priority Inheritance Protocol (PIP)

Ensure **mutual exclusion** when accessing shared resources

Works by dynamically **raising and lowering** thread **priorities**

- *Lock:*
  - o If lock, is available, grab it.
  - o Otherwise, block; the thread holding the lock “inherits” my priority
- *Unlock:* Release lock. Return to normal priority.

Provably avoids the **priority inversion** problem

- High-priority task is blocked on a lock held by low-priority task

However, incorrect usage leads to **deadlocks**

- In contrast to **priority ceiling** locks and **CPU locks** **[FMCAD’11, VMCAI’13]**

Our Contributions

Time-bounded verification of reachability properties of PP with PIP locks

- Based on **sequentialization** [LR08], but supports PIP locks
- **Challenge:** # sequentialization rounds needed for completeness cannot be statically determined
- **Insight:** whether more rounds needed can be statically determined
- **Solution:** Iterative-deepening search with fixed point check

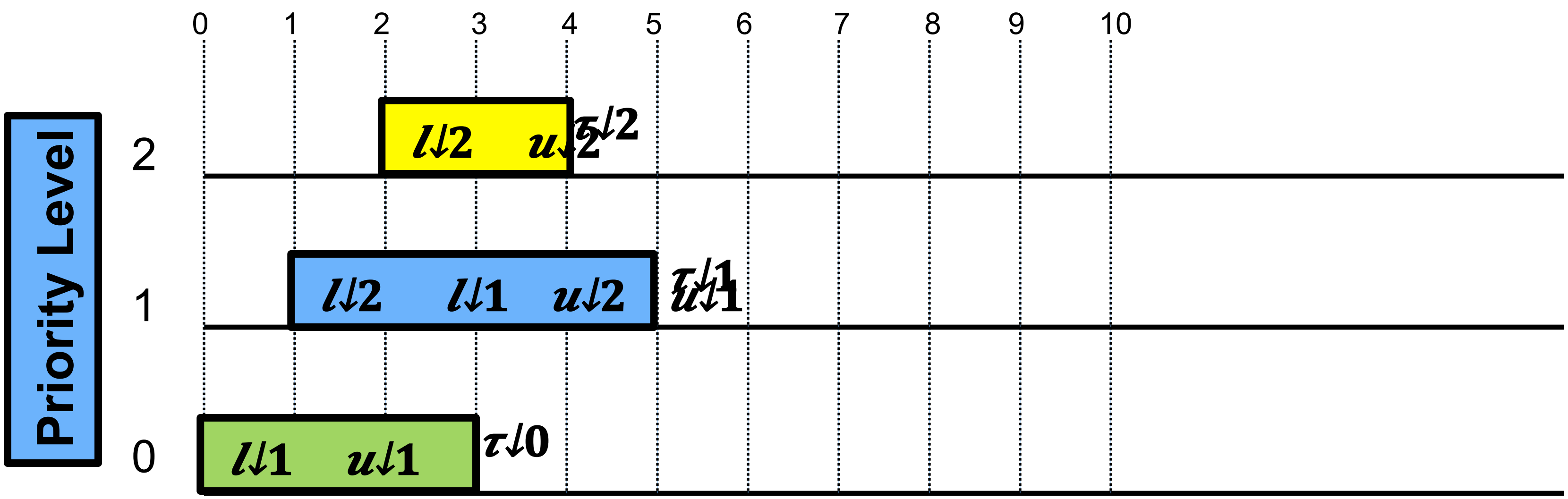
Deadlock detection in PPs with PIP locks

- Builds dynamically the **Task-Resource Graph**
- Aborts if a cycle in that graph is detected

Implementation and Empirical Evaluation

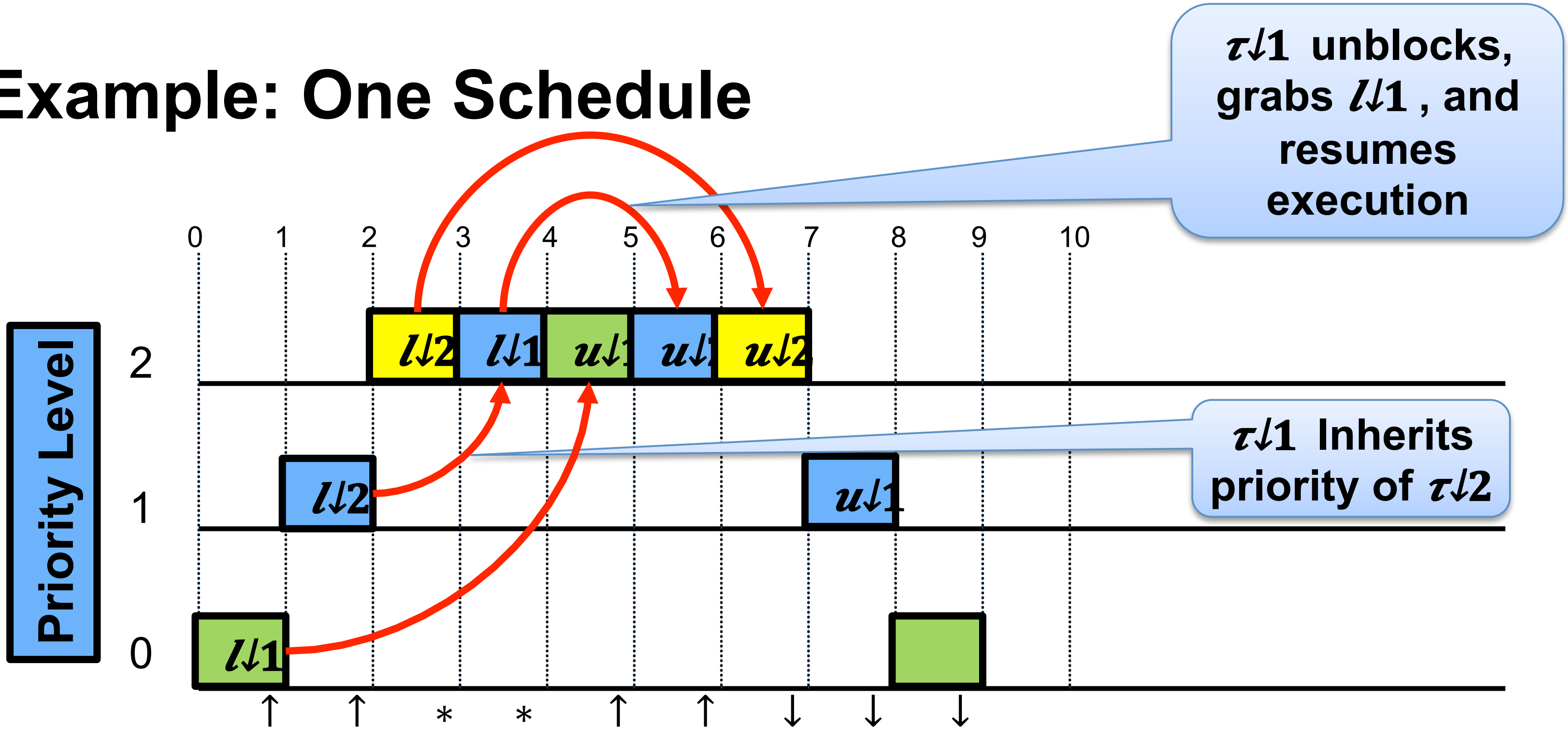
Example: A Periodic Program

Two PIP locks: 1 and 2
$l\downarrow i = acquring\ lock\ i$
$u\downarrow i = releasing\ lock\ i$



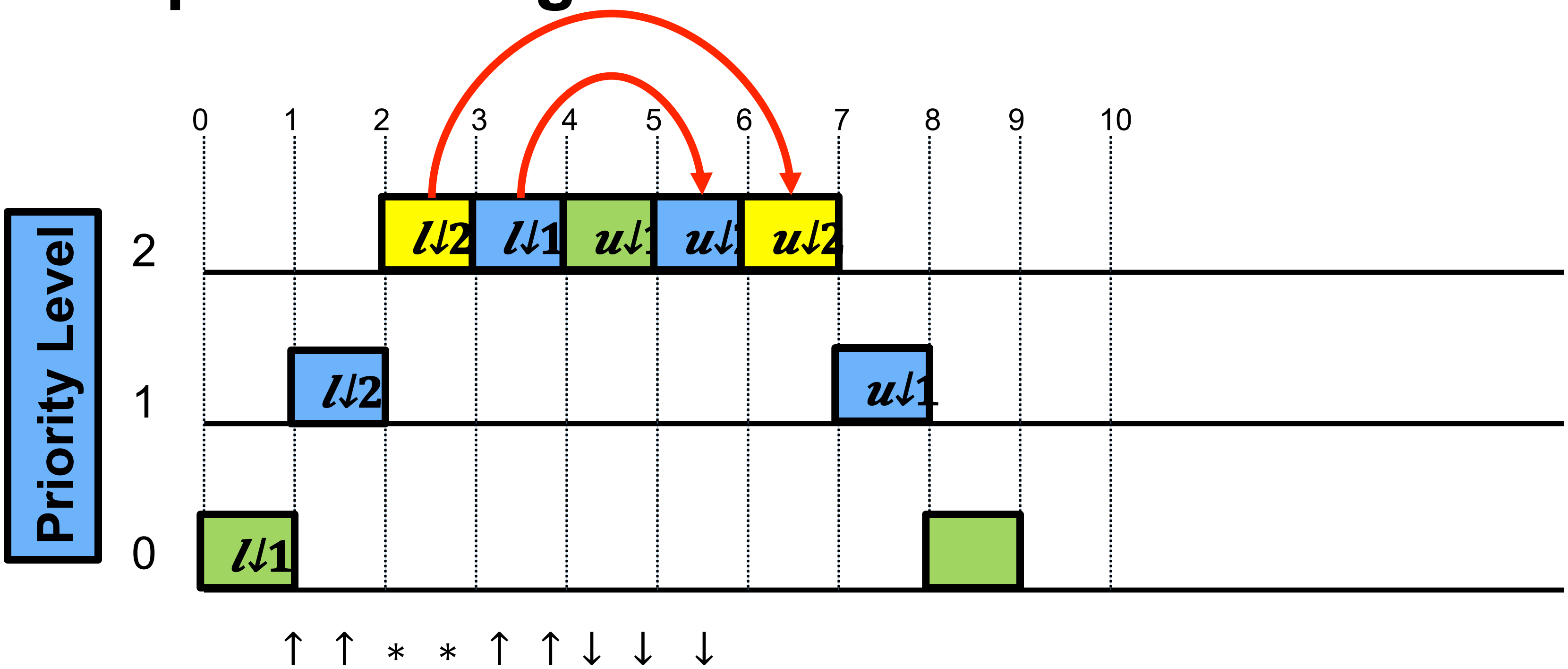
Task	Prio (I_1)	WCET (C_i)	Period (P_i)	Arrival Time (A_i)
τ_2	2	2	10	2
τ_1	1	4	20	1
τ_0	0	3	40	0

Example: One Schedule



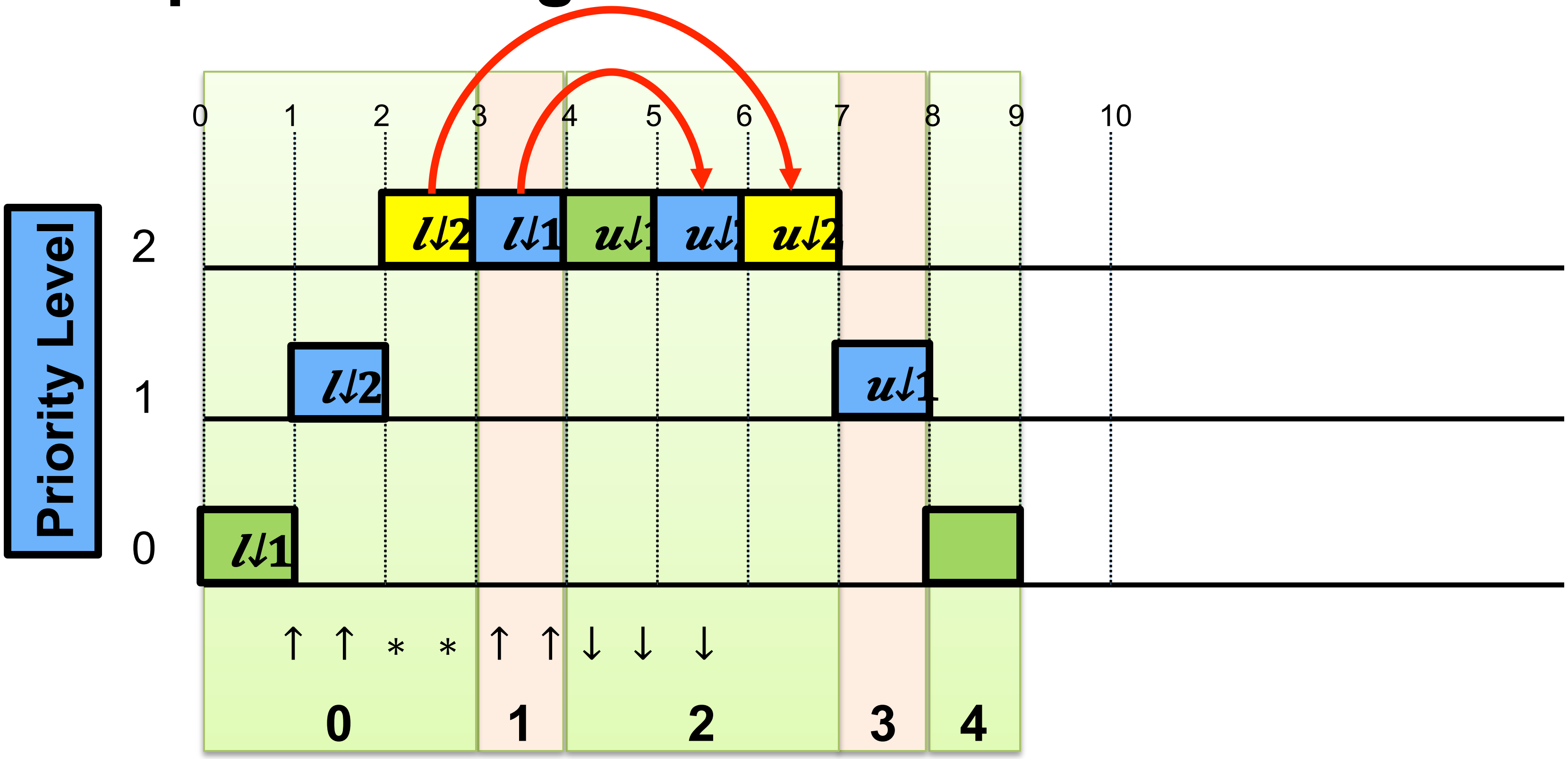
- Note: A scheduling point is either a preemption (↑), a block (*), or a job end (↓)

Example: Viewing as a Round-Based Schedule



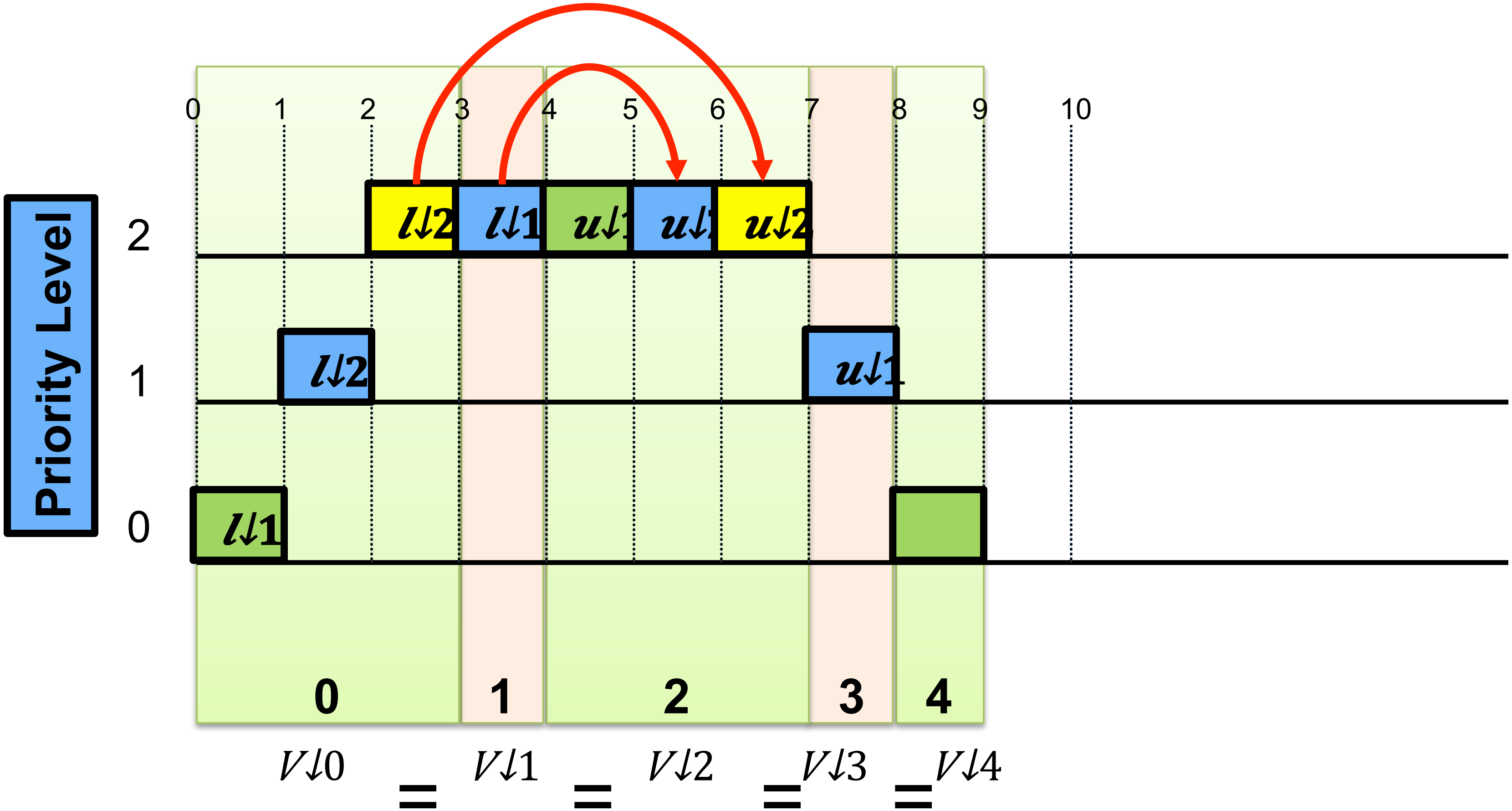
- Note: A scheduling point is either a preemption (↑), a block (*), or a job end (↓)
- Define: A round ends if the scheduling point is either a block, or a job end
- Define: A round continues if the scheduling point is a preemption

Example: Viewing as a Round-Based Schedule



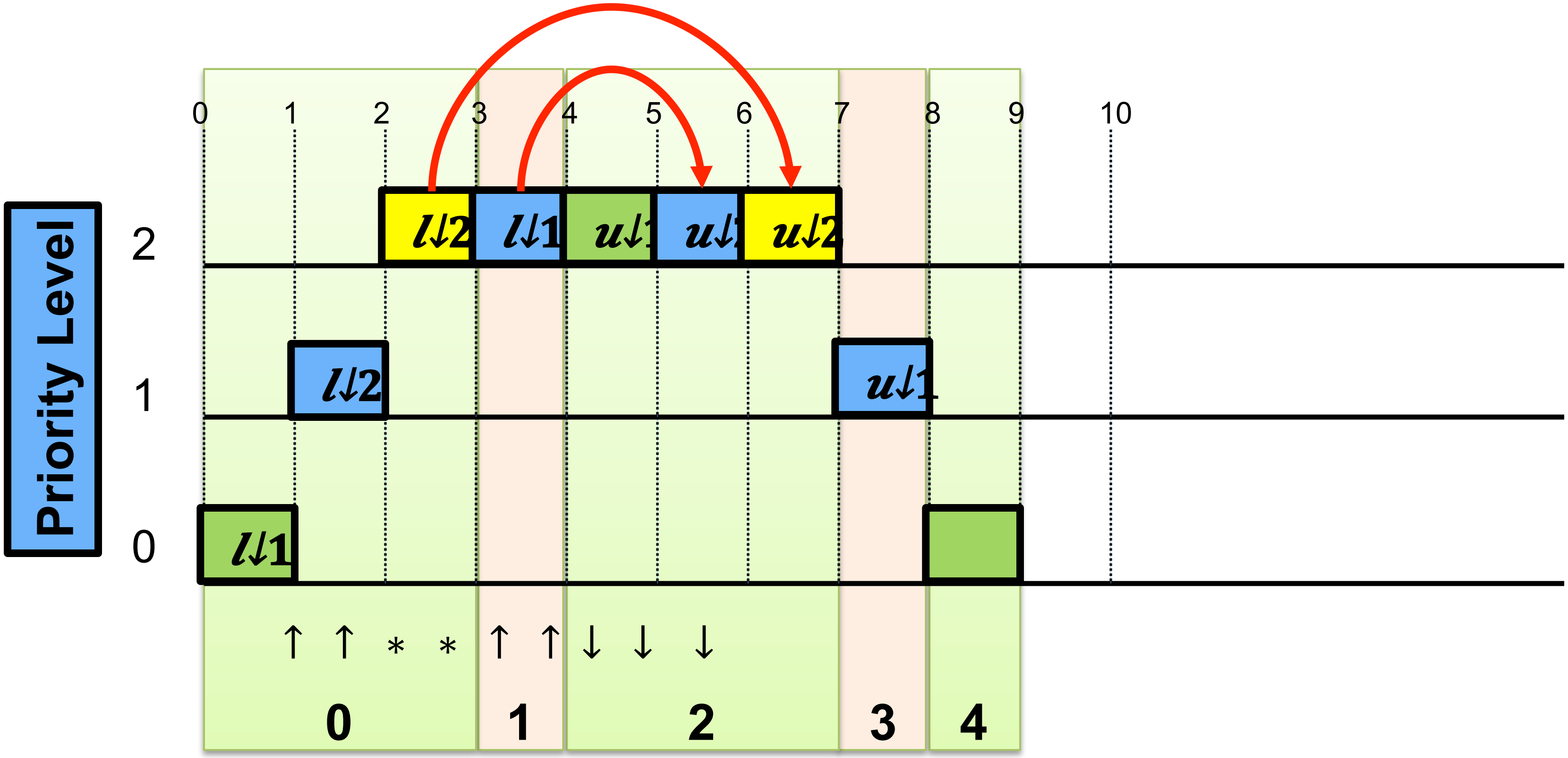
- Note: A scheduling point is either a preemption (↑), a block (*), or a job end (↓)
- Define: A round ends if the scheduling point is either a block, or a job end
- Define: A round continues if the scheduling point is a preemption

Sequentialization With PIP locks and fixed #Rounds



1. Create fresh variables for each round
2. Distribute jobs across rounds
3. Execute jobs using variables for the round it is in
4. Equate ending value at round i to beginning value at round $i+1$
5. Building on prior work [VMCAI13] – adding PIP locks non-trivial

Complete Algorithm: Iteratively Increase #Rounds



- **Challenge:** Different schedules have different number of rounds
  - #Rounds = #Jobs + #Blocks
  - #Blocks depends on the execution and preemption
- **Solution:** Start with a small number of rounds (equal to #Jobs)
  - Add more rounds iteratively till counterexample found, or fixed-point reached

Overall Algorithm

function PIPVERIF(C)
    $R := |\mathsf{J}|$
    loop
        $x :=$ VERIFROUNDS(C, R)
        if $x =$ INCROUNDS **then** $R := R + 1$
        else return x

Aborts if a job blocks but all R rounds already allocated

$$[\![S_a(C,R)]\!] = \emptyset \iff b^{R-|\mathsf{J}|} \bullet a \notin [\![C]\!]$$

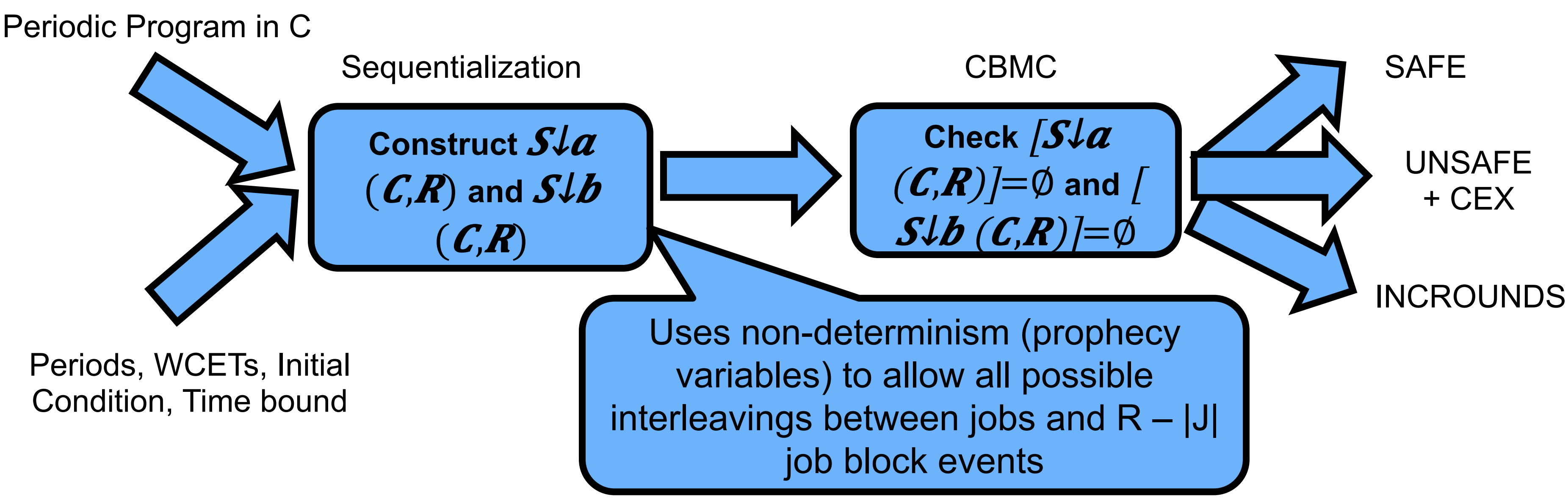
$$[\![S_b(C,R)]\!] = \emptyset \iff b^{R+1-|\mathsf{J}|} \notin [\![\mathring{C}]\!]$$

Implementing *VerifRounds*(*C*,*R*)

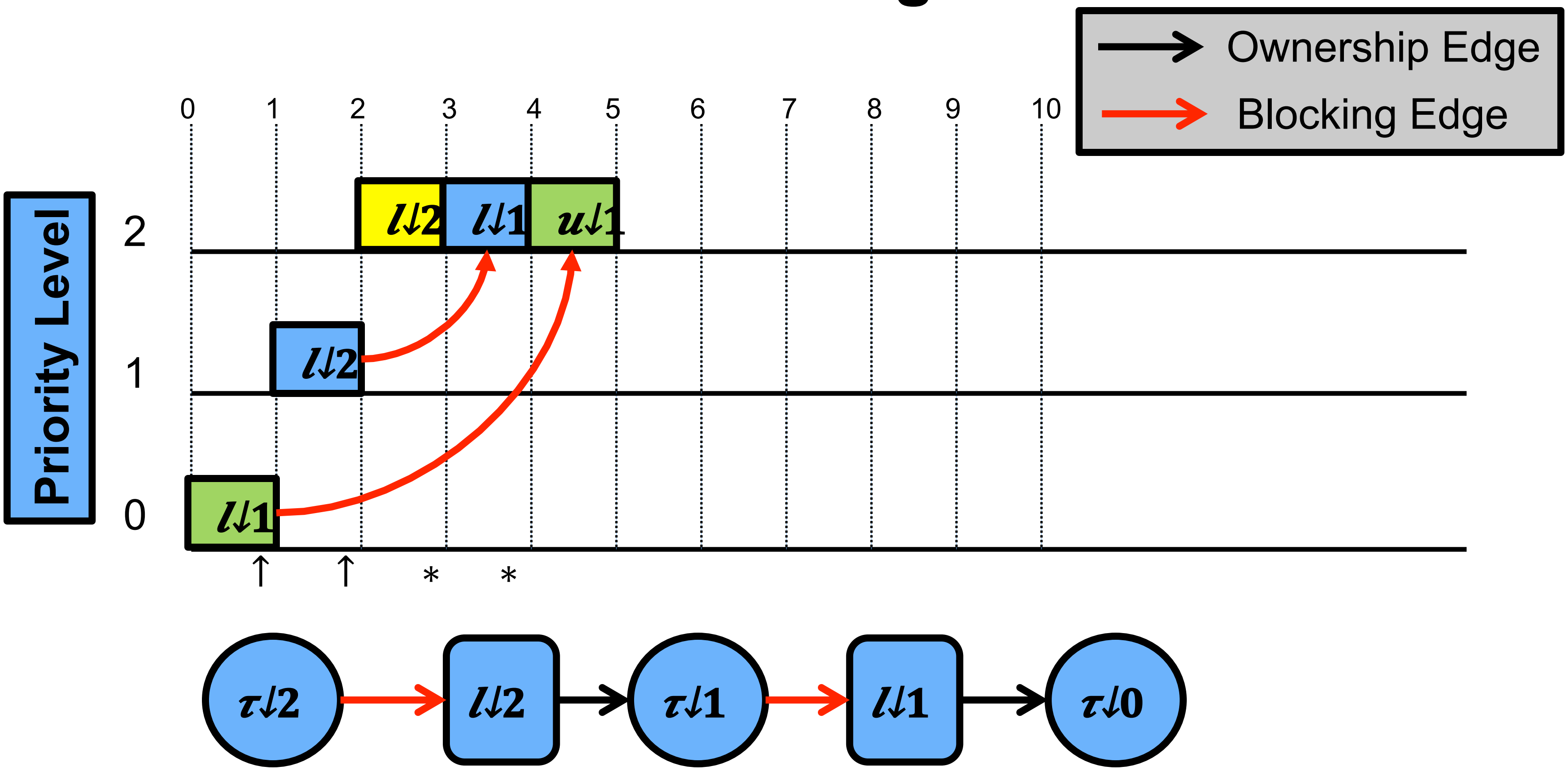
Supports C programs w/ tasks, priorities, priority ceiling protocol, shared variables

Works in two stages:

1. *Sequentialization* – reduction to sequential program w/ *prophecy* variables
2. *Bounded program analysis*: bounded C model checker (CBMC, HAVOC, …)

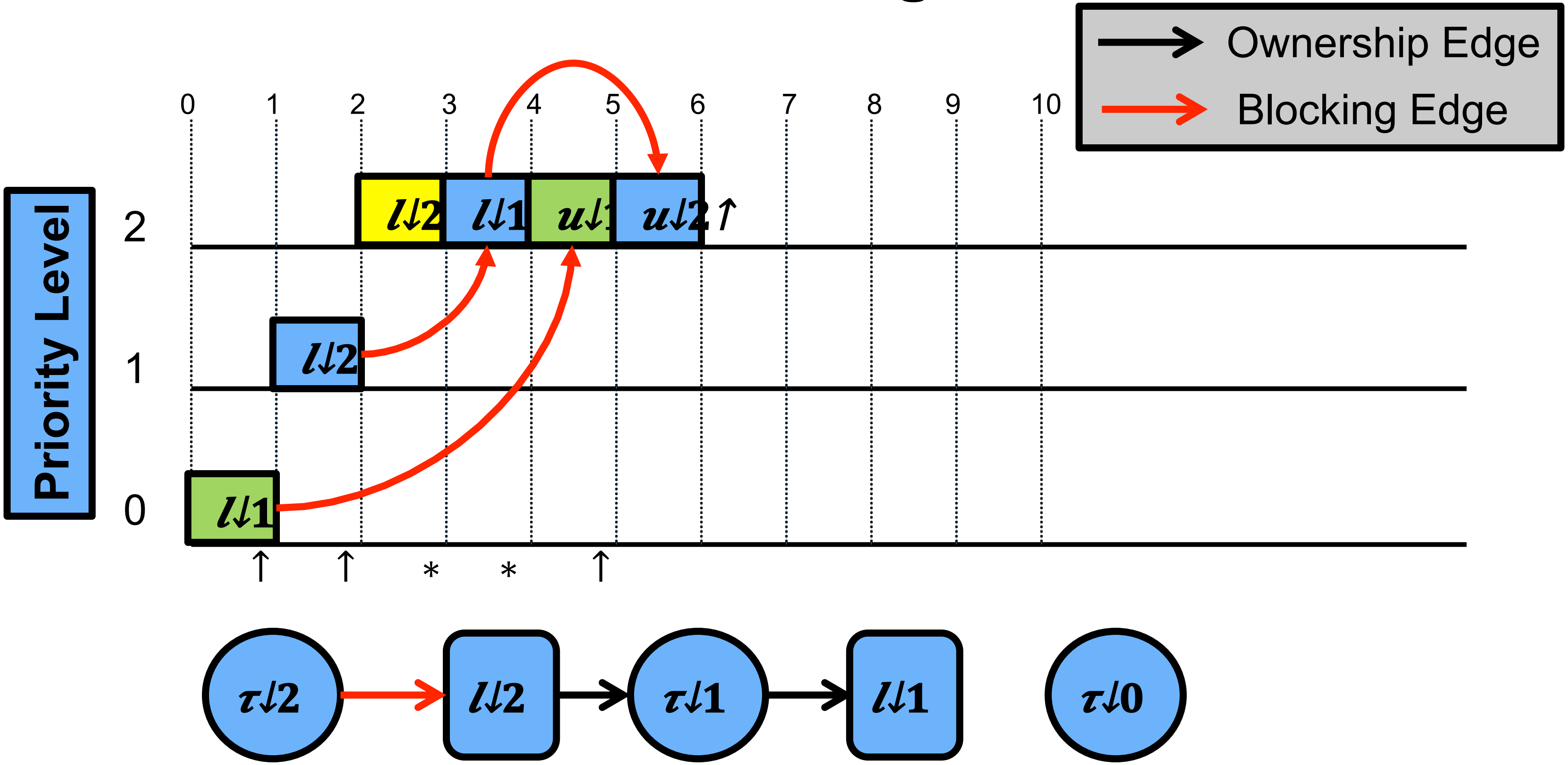


Deadlock Detection: Encoding TRG



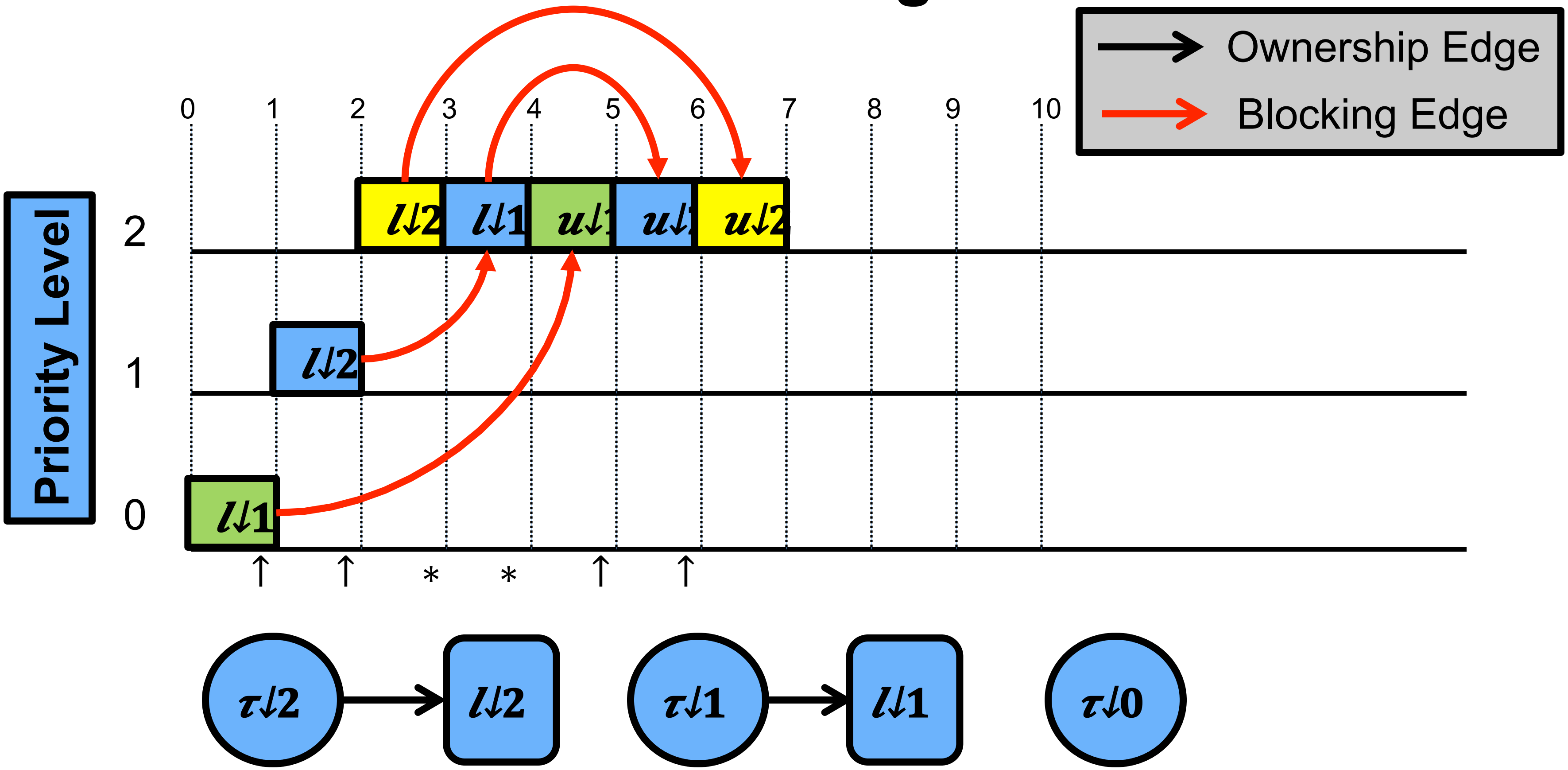
- **TRG:** Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
- Transitive closure of TRG maintained and updated dynamically
- Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)

Deadlock Detection: Encoding TRG



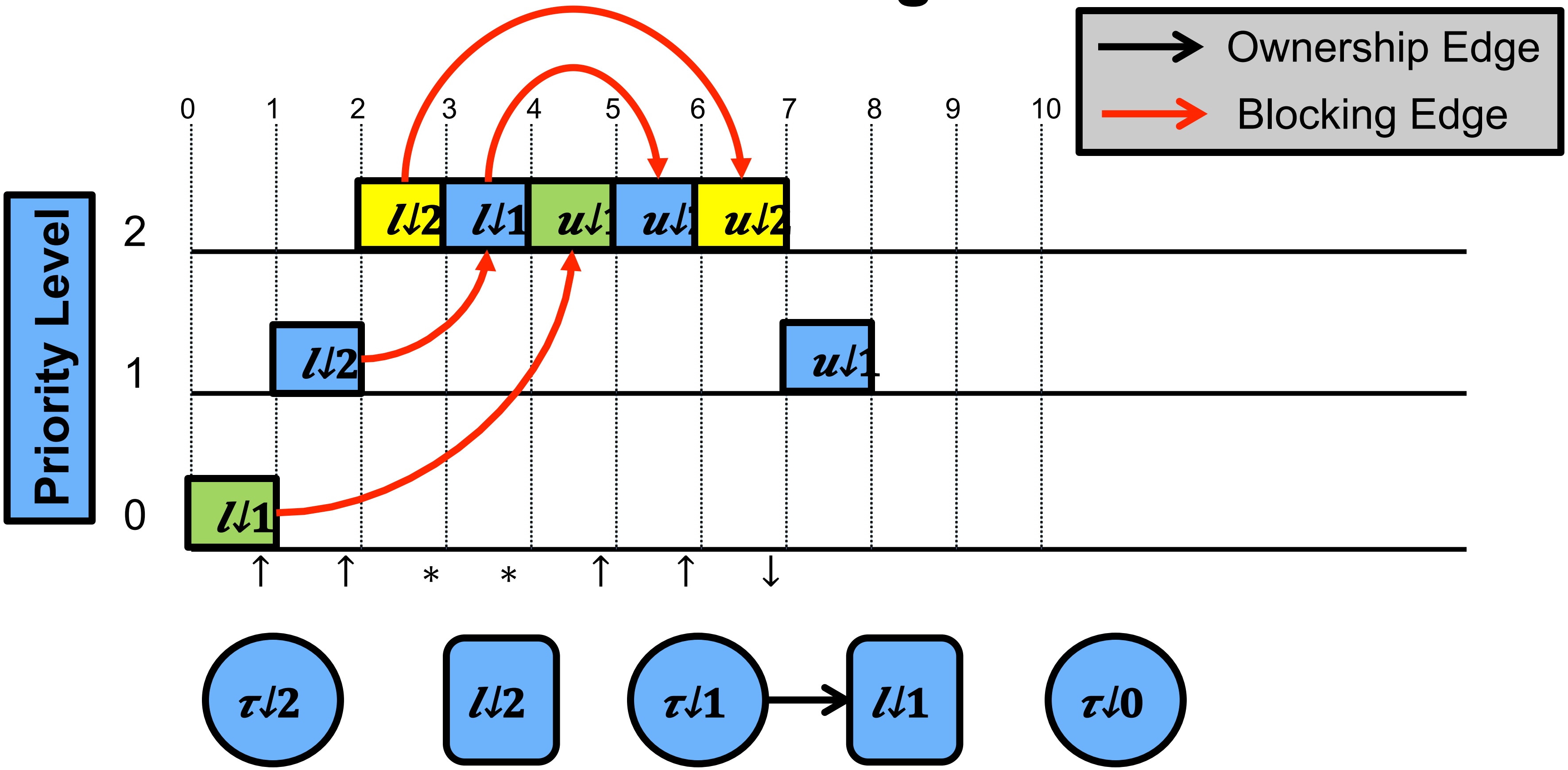
- **TRG:** Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
- Transitive closure of TRG maintained and updated dynamically
- Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)

Deadlock Detection: Encoding TRG



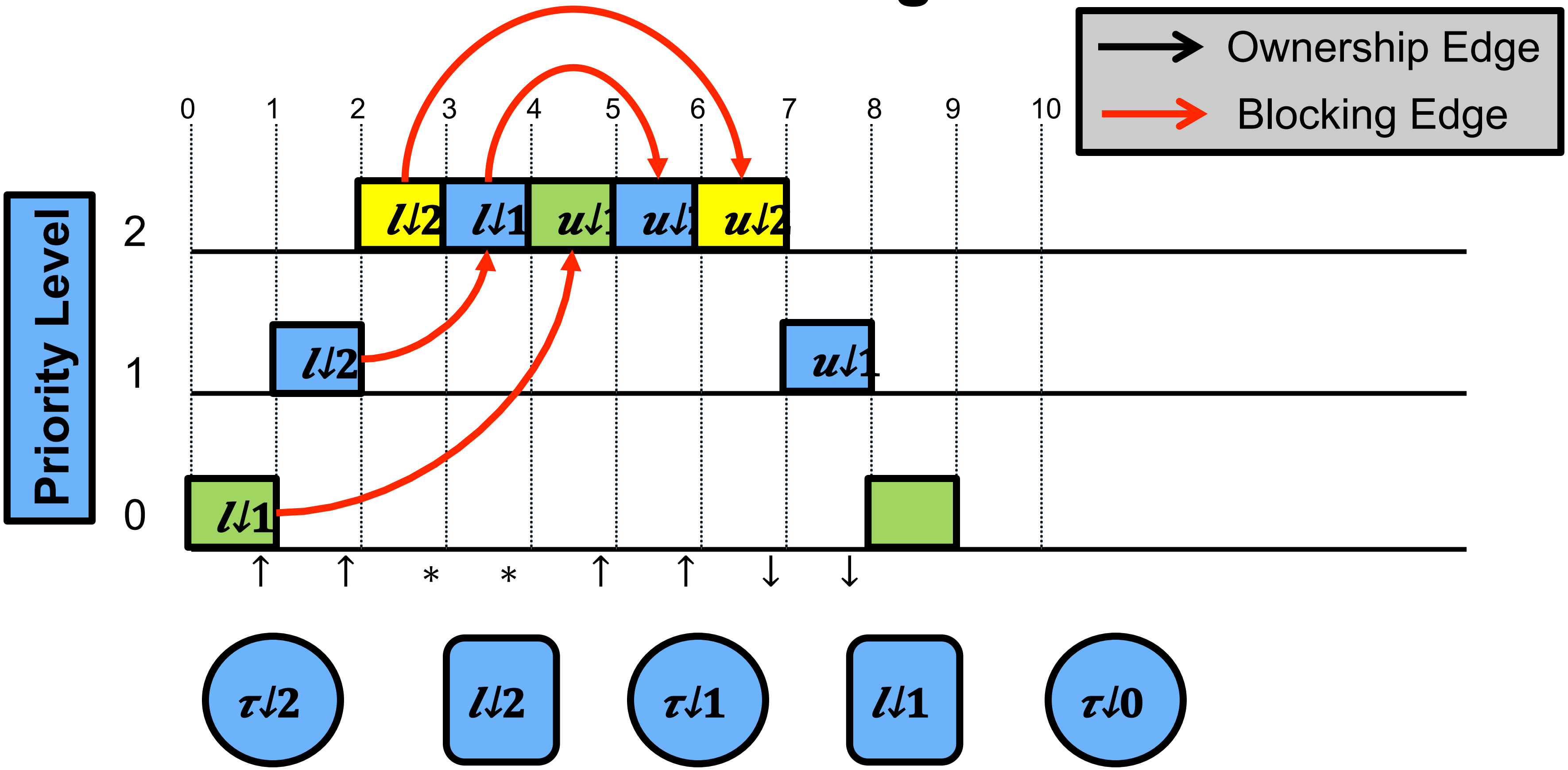
- **TRG:** Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
- Transitive closure of TRG maintained and updated dynamically
- Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)

Deadlock Detection: Encoding TRG



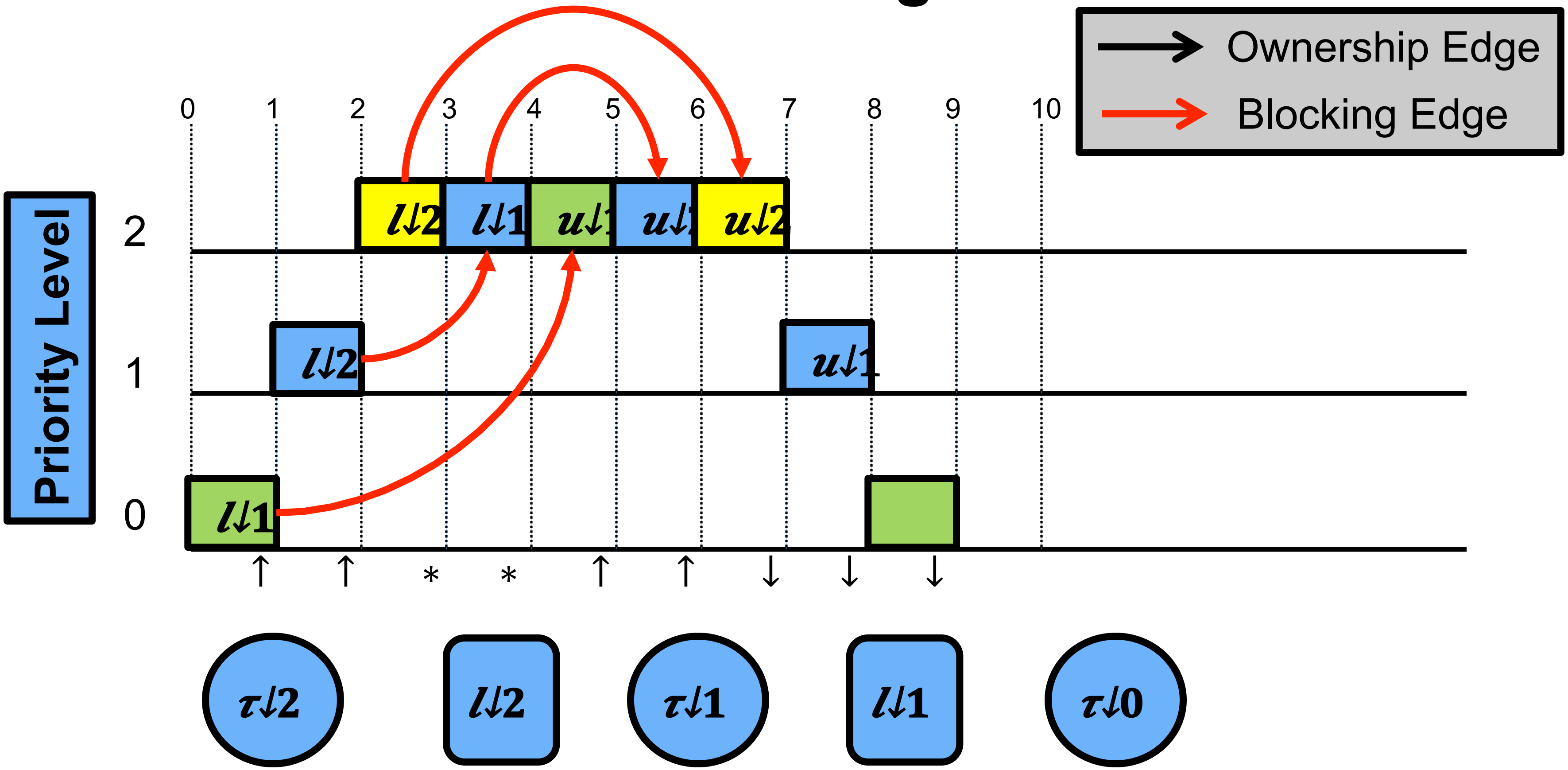
- **TRG:** Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
- Transitive closure of TRG maintained and updated dynamically
- Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)

Deadlock Detection: Encoding TRG



- **TRG:** Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
- Transitive closure of TRG maintained and updated dynamically
- Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)

Deadlock Detection: Encoding TRG



- **TRG:** Node = task/lock; Edge = blocking/ownership; Cycle = deadlock
- Transitive closure of TRG maintained and updated dynamically
- Program aborts if TRG becomes cyclic (i.e., transitive closure has self-loop)

NXTway-GS: a 2 wheeled self-balancing robot

Original: nxt (2 tasks)

- *balancer* (4ms)
  – Keeps the robot upright and responds to BT commands
- *obstacle* (50ms)
  – monitors sonar sensor for obstacle and communicates with *balancer* to back up the robot

Ours: aso (3 tasks)

- *balancer* as above but no BT
- *obstacle* as above
- *bluetooth* (100ms)
  – responds to BT commands and communicates with the *balancer*

Verified consistency of communication between tasks

Experimental Results

File	T	J	Rn	Vars	Cls	SAT	Result
nxt.bug1a.c	29	15	15	1.4M	4.3M	26	UNSAFE
nxt.bug1b.c	58	15	15	2.5M	7.5M	54	UNSAFE
nxt.bug1c.c	61	15	15	2.6M	8.1M	57	UNSAFE
nxt.ok1.c	746	15	17	2.9M	9.0M	714	SAFE
aso.bug1a.c	73	15	15	2.7M	8.3M	68	UNSAFE
aso.bug1b.c	64	15	15	2.6M	8.0M	59	UNSAFE
aso.bug1c.c	33	15	15	1.7M	5.1M	29	UNSAFE
aso.ok1.c	4148	15	19	3.5M	10.9M	4,088	SAFE
aso.bug2a.c	43	15	15	1.6M	4.9M	39	UNSAFE
aso.bug3a.c	48	15	15	1.7M	5.1M	45	UNSAFE
aso.bug3b.c	35	15	15	1.5M	4.6M	32	UNSAFE
aso.bug3c.c	55	15	15	1.6M	4.9M	52	UNSAFE
aso.ok3.c	879	15	16	1.8M	5.5M	866	SAFE
aso.bug4a.c	63	15	15	2.0M	6.1M	58	UNSAFE
aso.bug4b.c	908	15	16	2.1M	6.4M	898	UNSAFE
aso.ok4.c	3047	15	17	2.2M	6.7M	3,027	SAFE

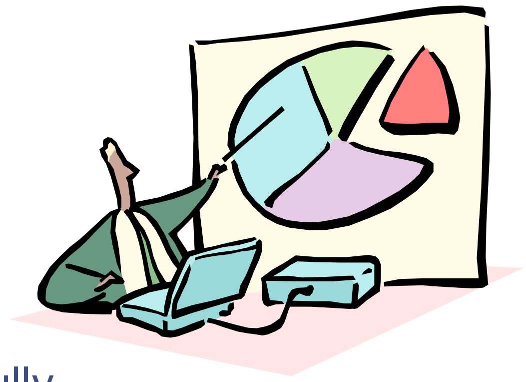
Related, Ongoing and Future Work

Related Work

- Sequentialization of Periodic Programs with CPU locks and priority ceiling protocol locks (FMCAD'11, VMCAI'13)
- Sequentialization of Concurrent Programs (Lal & Reps '08, and others)
- Sequentialization of Periodic Programs (Kidd, Jagannathan, Vitek '10)
- Verification of periodic programs using SPIN (Florian, Gamble, & Holzmann '12)
- Verification of Time Properties of (Models of) Real Time Embedded Systems
- Model Checking Real-Time Java using JPF (Lindstrom, Mehlitz, and Visser '05)

Ongoing and Future Work

- Verification without the time bound
- Memory Consistency based Sequentialization
- Abstraction / Refinement
- Modeling physical aspects (i.e., environment) more faithfully
- **More Examples**



Contact Information

Presenter
Sagar Chaki
SSD
Telephone: +1 412-268-5800
Email: chaki@sei.cmu.edu

Web:
www.sei.cmu.edu
http://www.sei.cmu.edu/contact.cfm

U.S. mail:
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
USA

Customer Relations
Email: info@sei.cmu.edu
Telephone: +1 412-268-5800
SEI Phone: +1 412-268-5800
SEI Fax: +1 412-268-6257

QUESTIONS?