Synthesis of Provably-correct Software using Discrete Control Theory

Yin Wang, Terence Kelly
(HP Labs)

Stephane Lafortune, Scott Mahlke
(University of Michigan)

Software @ Scale, Berkeley
(08/18/2010)

Team

- University of Michigan
  - Students: Hongwei Liao, Hyoun Kyu Cho, Jason Stanley,
  - Faculty: Stephane Lafortune, Scott Mahlke
- HP Labs
  - Yin Wang, Terence Kelly
- Georgia Tech (recently)
  - Student: Ahmed Nazeem
  - Faculty: Spyros Reveliotis

Software Failures Are Costly

- Software bugs cost the U.S. economy an estimated $59.5 billion annually
  [National Institute of Standards and Technology, 2002]
- Post-release bugs are the worst
  - 100X increase in cost of removing defects
    [Barry Boehm, 1981]
  - on average, 11-25+ critical bugs are found within 12 months of the release, which cost $5.2—22 million per company annually
    [IDC white paper, 2008]
- Multicore era brings new challenges

The Multicore Challenge

The Free Lunch Is Over
A Fundamental Turn Toward Concurrency in Software

By Herb Sutter

The biggest sea change in software development since the OO revolution is

Patterson: Multicore is a Hail Mary pass

Like

David Patterson writes in this month's IEEE Spectru multicore; a move that he characterizes as a Hail M yet know will be caught

April 19, 2008

Multicore Parallel Programming: Can We Please do it Right This Time?

– IEEE Electronic Design Processes Workshop 2008

Steve Leibson

Tim Mattson, a principal engineer at Intel's Applications Research Laboratory, describe

With An 80-Core Chip On The Way, Software Needs To Start Changing

The big question is how -- and how soon -- the software industry will step up and produce applic that can take advantage of all those cores.

By Sharon Gaudin
InformationWeek

Cambrian Explosion of Research

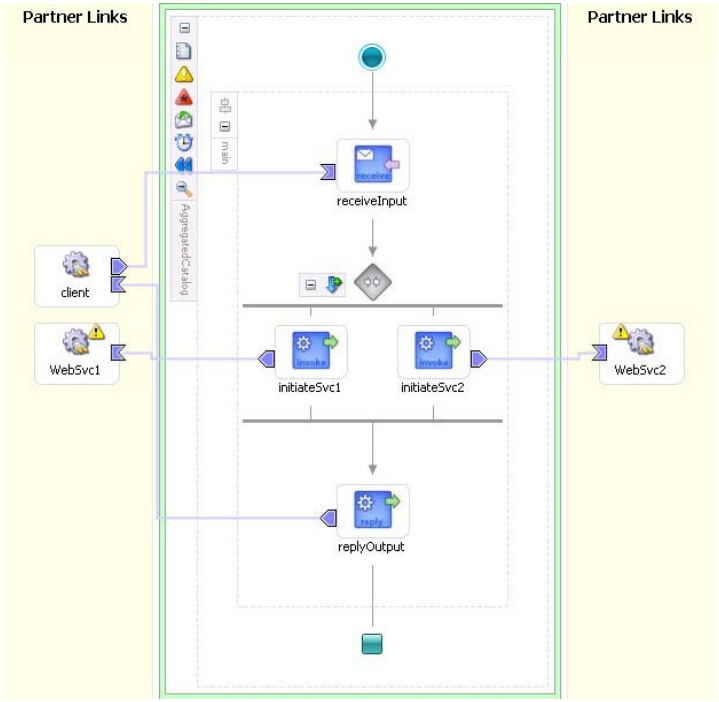
- New libraries, languages, features
  - Intel TBB, Erlang, Cilk++, atomic sections, Trans. Memory, OpenMP
- Tools
  - Static analysis, testing tools
    - Coverity™, Locksmith
    - Klee, CHESS, CheckFence
  - Runtime analysis
    - Eraser, Intel Thread Checker™
  - Post-mortem analysis
    - Triage, CrashRpt

A New Angle---Control

- Goal: Controlling software execution to avoid failure
- Approach: Offline control synthesis + code instrumentation
- Foundation: Discrete Control Theory
- Paradigm: Control Engineering
- Application scenarios:
  - Rapid prototype development
  - Post-release bugs

Two Examples

- Workflow control [EuroSys 07]
  - High-level scripting language
  - *Safe execution of flawed workflows*
  - Ongoing effort at HP Labs

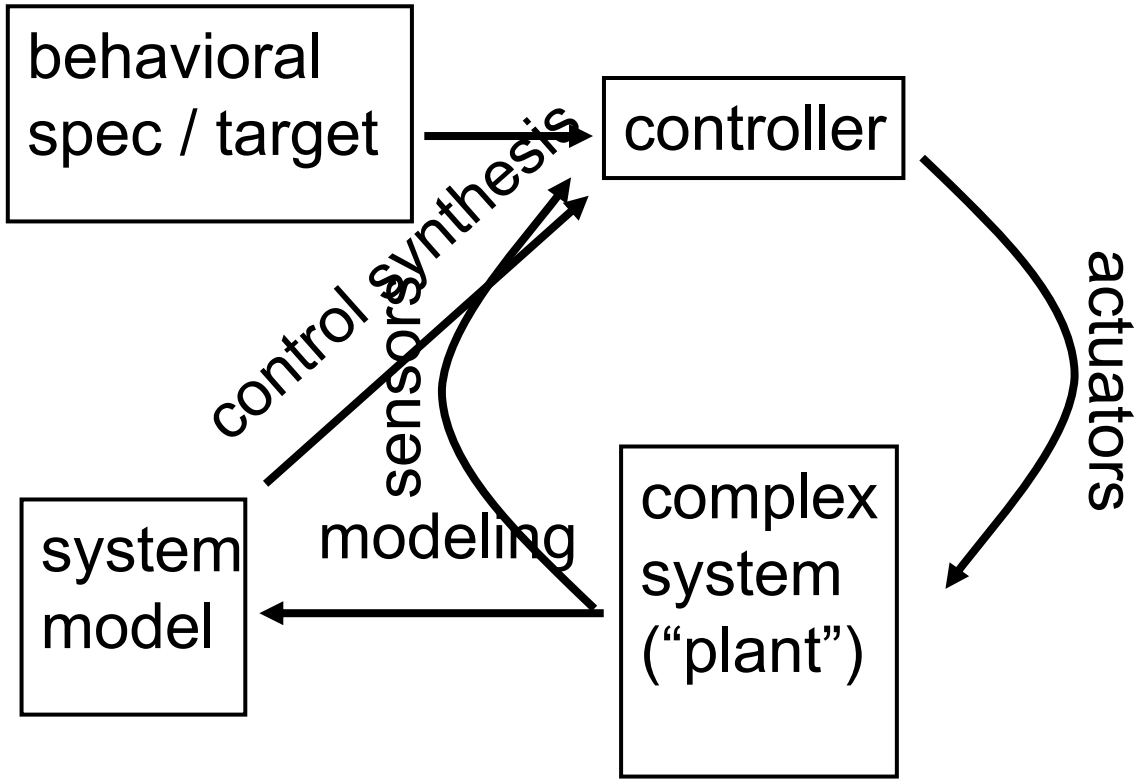


- Gadara: Deadlock avoidance in multi-threaded software [OSDI 08, POPL 09, IEEE Computer 09]

Outline

- Motivation
- Control Engineering
- Gadara Walkthrough
- Discussion

Using the Control Engineering Toolkit



"closed-loop" system provably satisfies spec: correct by construction

Control Engineering

- 100+ years of remarkable success
- Cornerstone for industrial civilization
- Pervasive in everyday life
  - power grid
  - automobile, airplane, spaceship
- Applications in computer systems for quantitative measurements, e.g., performance
  [Hellerstein, Tilbury, et al. 2004]
- Can this paradigm work for the synthesis of failure-free software?

Discrete Control Theory

- Analogue of conventional control
  - discrete vs. continuous state spaces (not discrete time)
  - event-driven vs. time-driven dynamics
- Modeling formalisms
  - finite automata [workflow control]
  - Petri nets [deadlock avoidance]
- Control synthesis
  - 25 years of research
  - well understood & automated for many models & specs

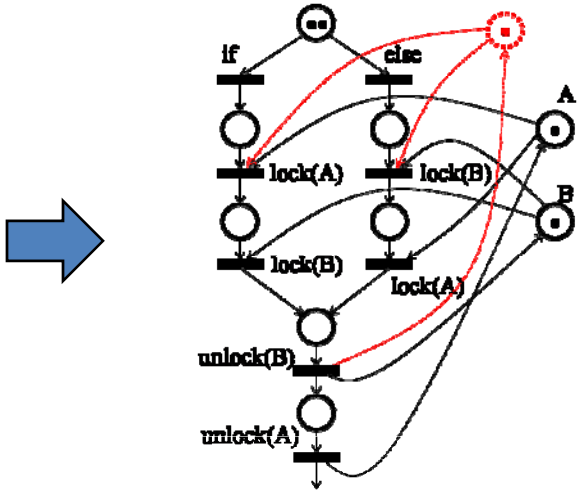
Outline

- Motivation
- Control Engineering
- Gadara Walkthrough
- Discussion

Gadara: Approach

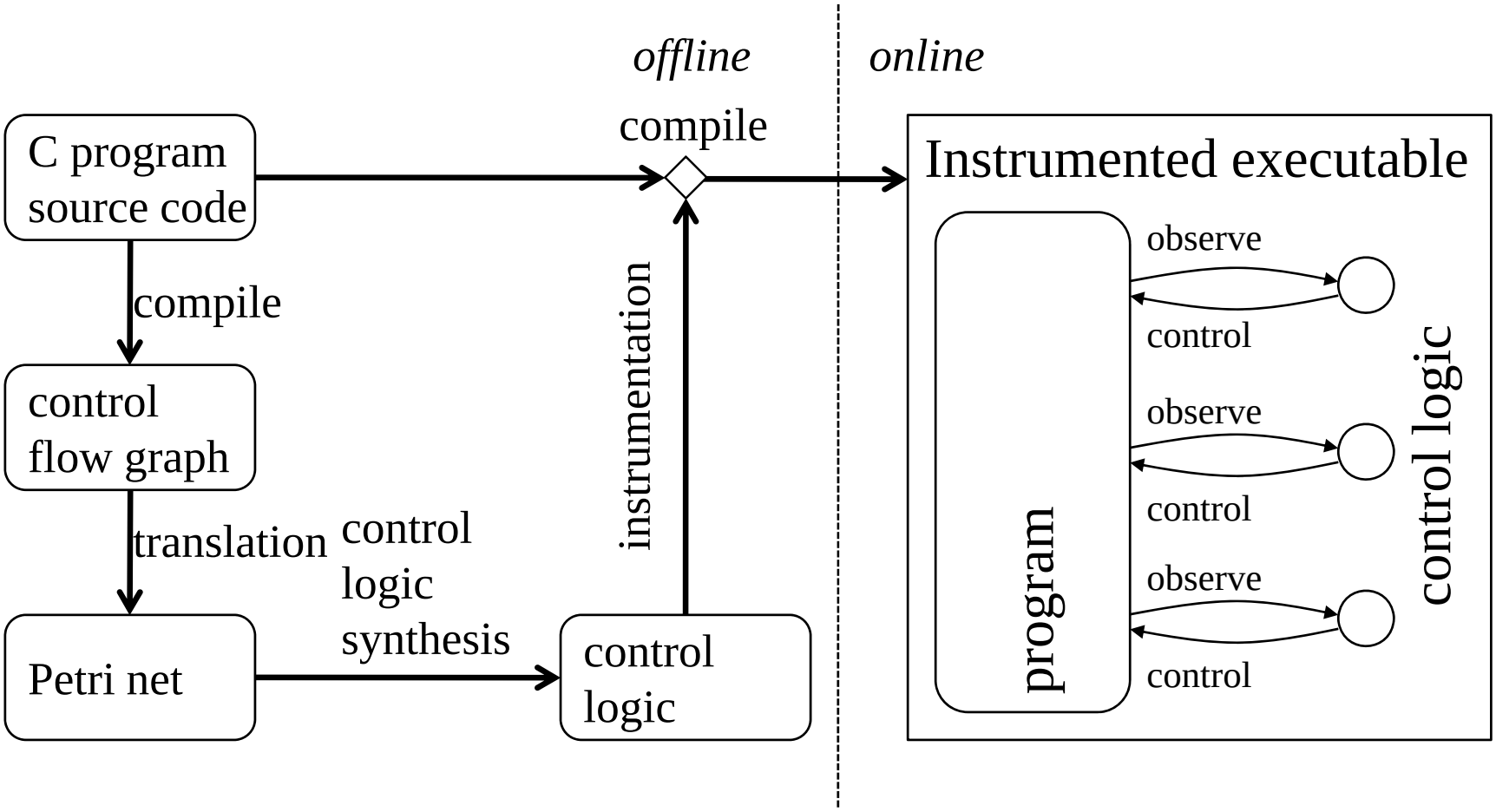
- Model building (Petri net)
- Control logic synthesis
- Source instrumentation

```
void * philosopher(void *arg) {
  ...
  if (RAND_MAX/2 > random()) {
    /* grab A first */
    pthread_mutex_lock(&forkA);
    pthread_mutex_lock(&forkB);
  } else {
    /* grab B first */
    pthread_mutex_lock(&forkB);
    pthread_mutex_lock(&forkA);
  }
  eat();
  pthread_mutex_unlock(&forkA);
  pthread_mutex_unlock(&forkB);
  ...
}
```




```
void * philosopher(void *arg) {
  ...
  if (RAND_MAX/2 > random()) {
    /* grab A first */
    gadara_lock(&forkA, &ctriplace);
    pthread_mutex_lock(&forkB);
  } else {
    /* grab B first */
    gadara_lock(&forkB, &ctriplace);
    pthread_mutex_lock(&forkA);
  }
  eat();
  gadara_replenish(&ctriplace);
  pthread_mutex_unlock(&forkA);
  pthread_mutex_unlock(&forkB);
  ...
}
```

Architecture



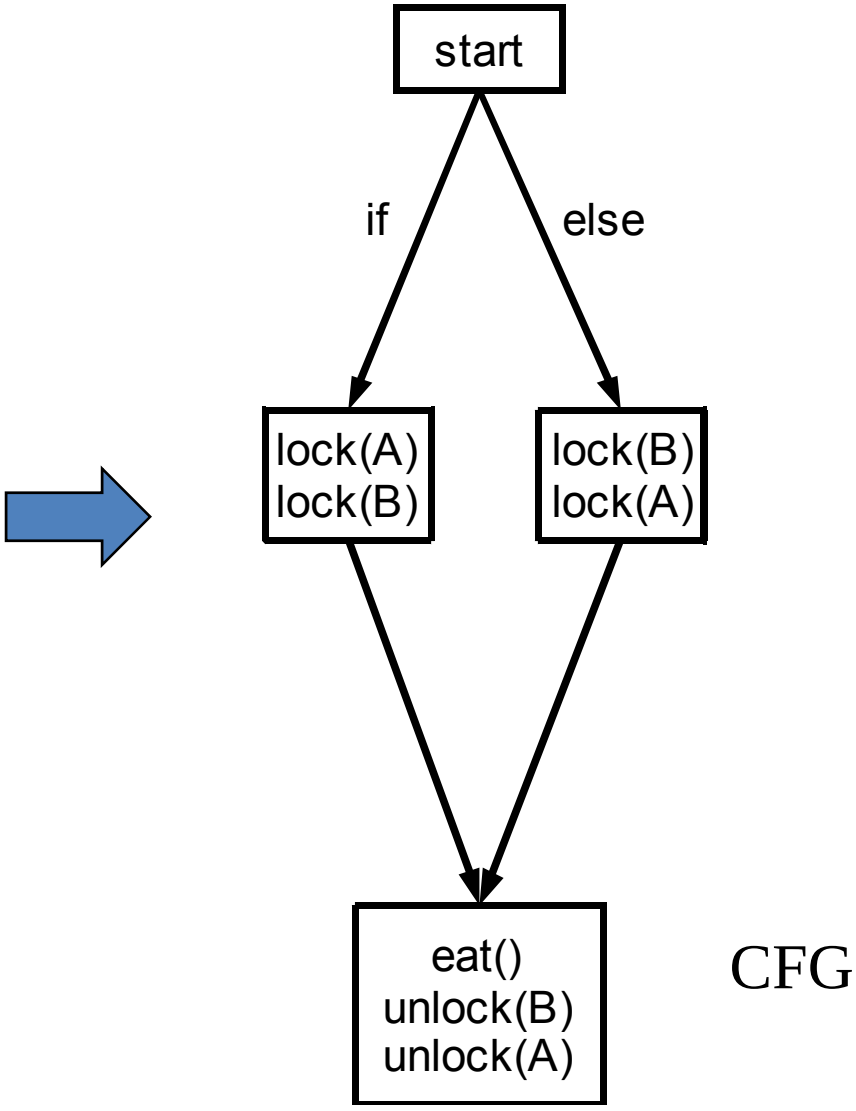
Dining Philosophers

```
void * philosopher(void *arg) {
   …
   if (RAND_MAX/2 > random()) {
      /* grab A first */
      pthread_mutex_lock(&forkA);
      pthread_mutex_lock(&forkB);
   } else {
      /* grab B first */
      pthread_mutex_lock(&forkB);
      pthread_mutex_lock(&forkA);
   }
   eat();
   pthread_mutex_unlock(&forkB);
   pthread_mutex_unlock(&forkA);
   ...
}
```

```
int main(int argc, char *argv[]) {
   ...
   pthread_create(&p1, NULL,
      philosopher, NULL);
   pthread_create(&p2, NULL,
      philosopher, NULL);
   ...
}
```

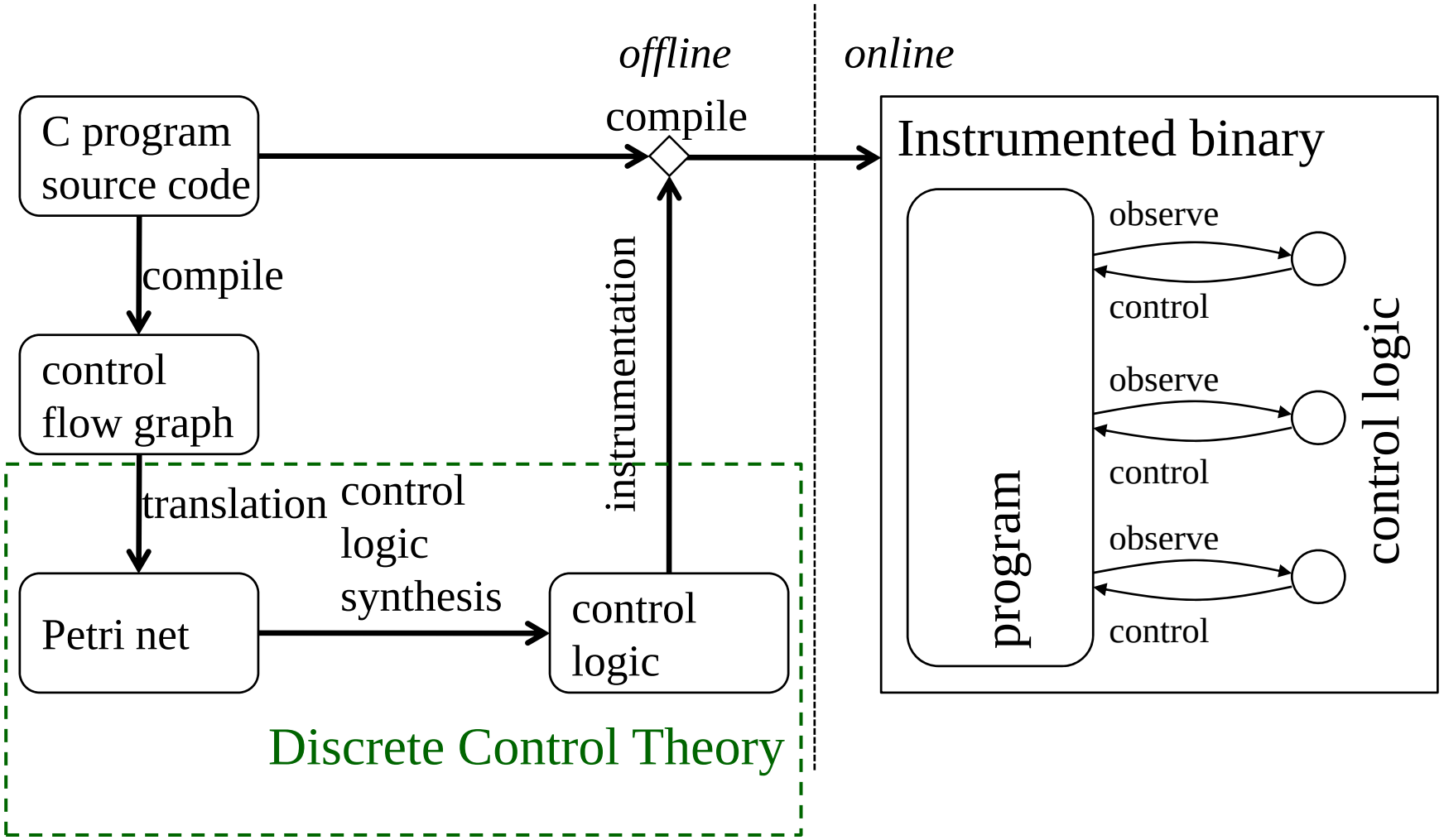
Dining Philosophers

```
void * philosopher(void *arg) {
   …
   if (RAND_MAX/2 > random()) {
      /* grab A first */
      pthread_mutex_lock(&forkA);
      pthread_mutex_lock(&forkB);
   } else {
      /* grab B first */
      pthread_mutex_lock(&forkB);
      pthread_mutex_lock(&forkA);
   }
   eat();
   pthread_mutex_unlock(&forkB);
   pthread_mutex_unlock(&forkA);
   …
}
```



CFG

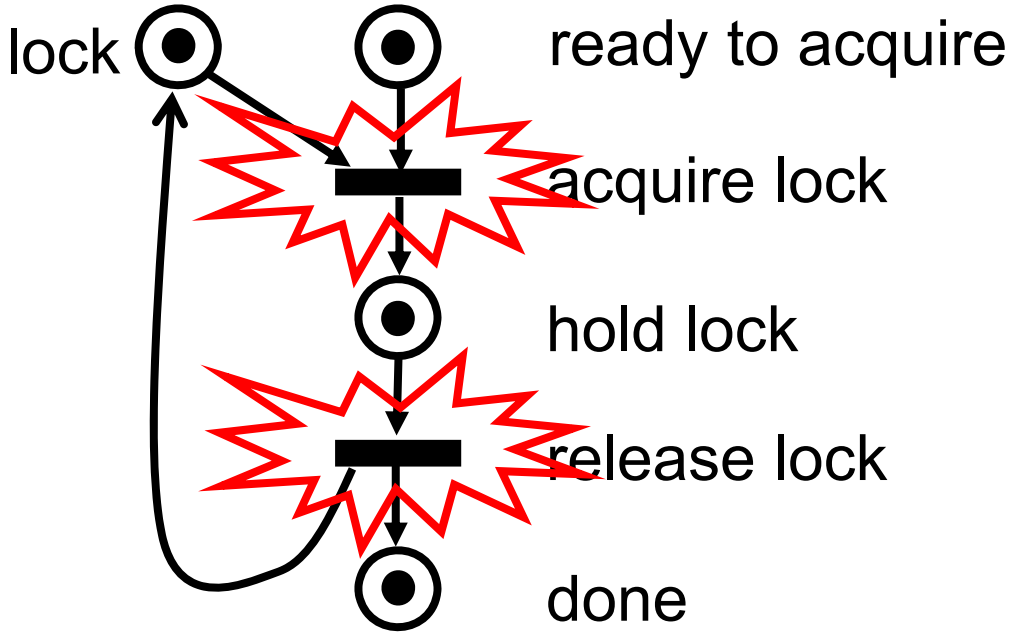
Architecture



Petri Net Basics

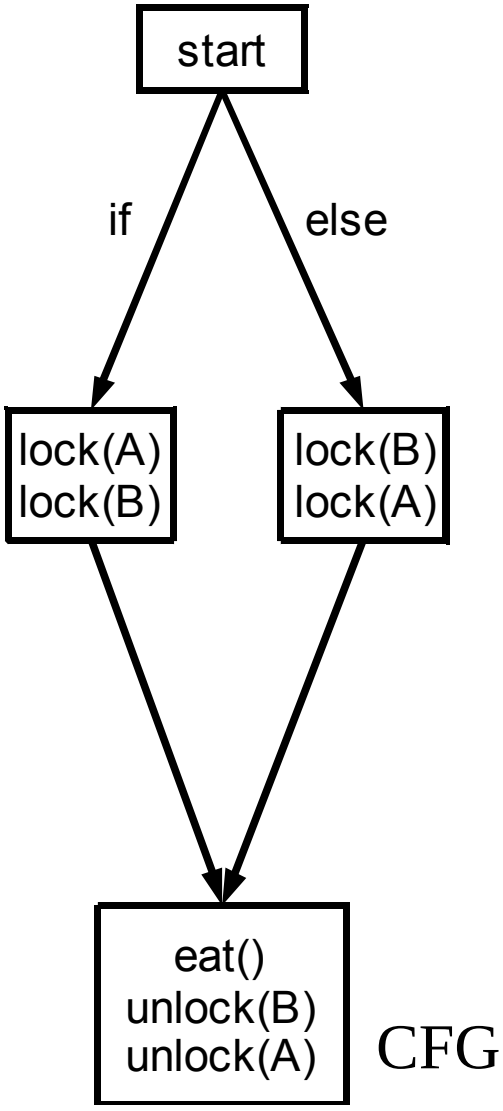
- bipartite graph: two kinds of nodes
- tokens represent states and dynamics

the PN that models lock acquisition & release

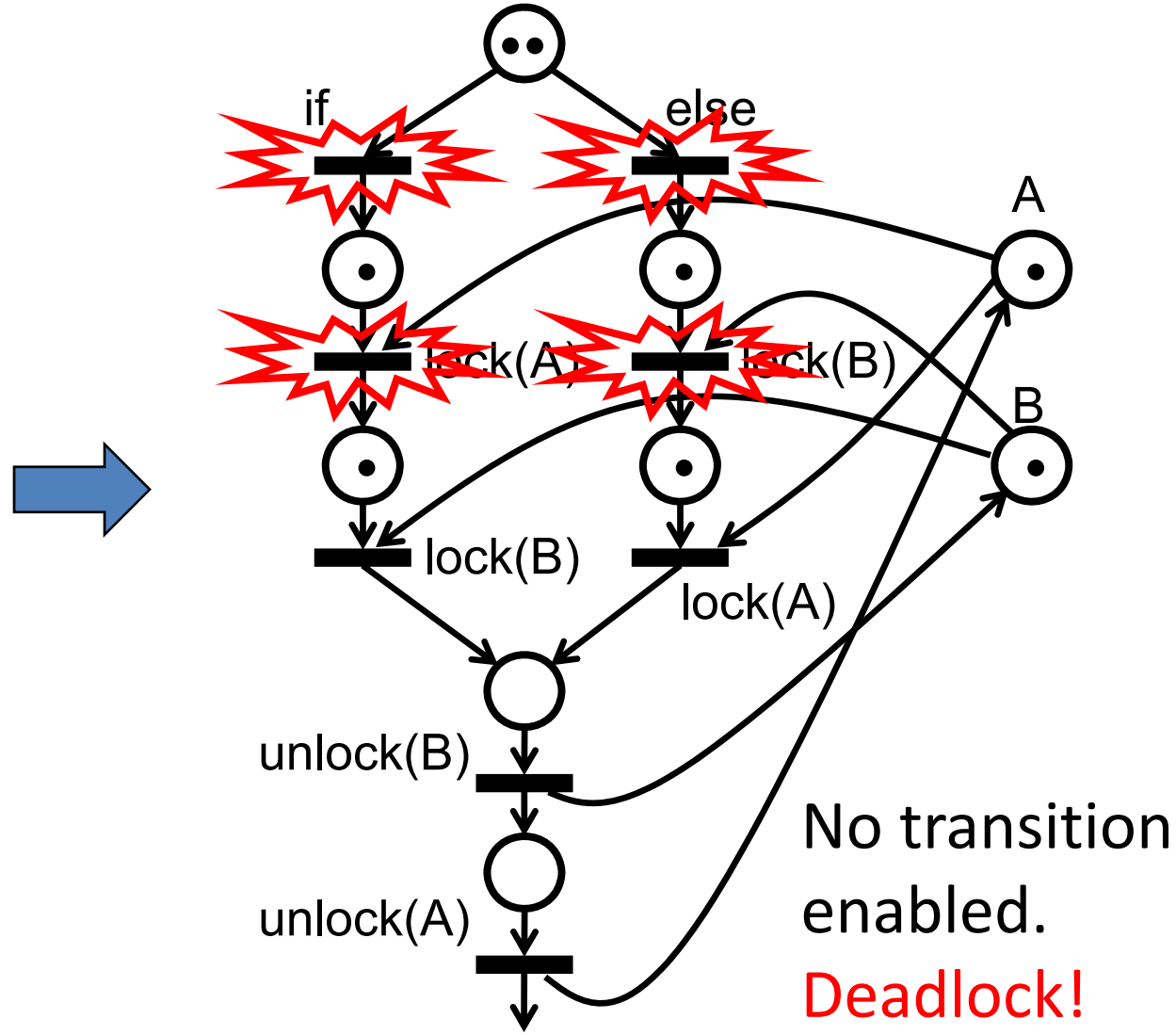


Murata, *Proc. IEEE* 1989
Kavi et al., *IJoPP* 2002

Dining Philosophers



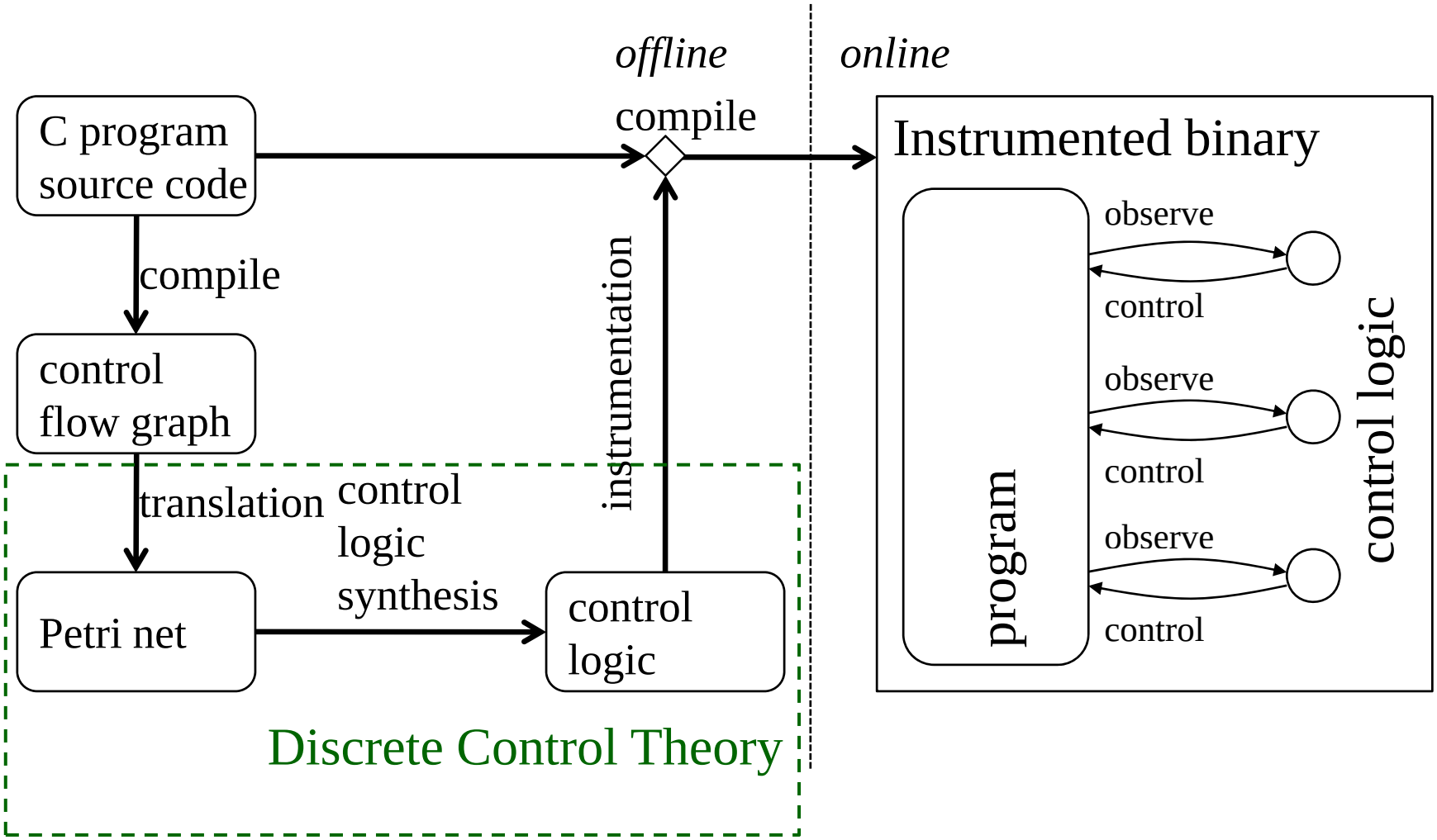
CFG



No transition enabled.

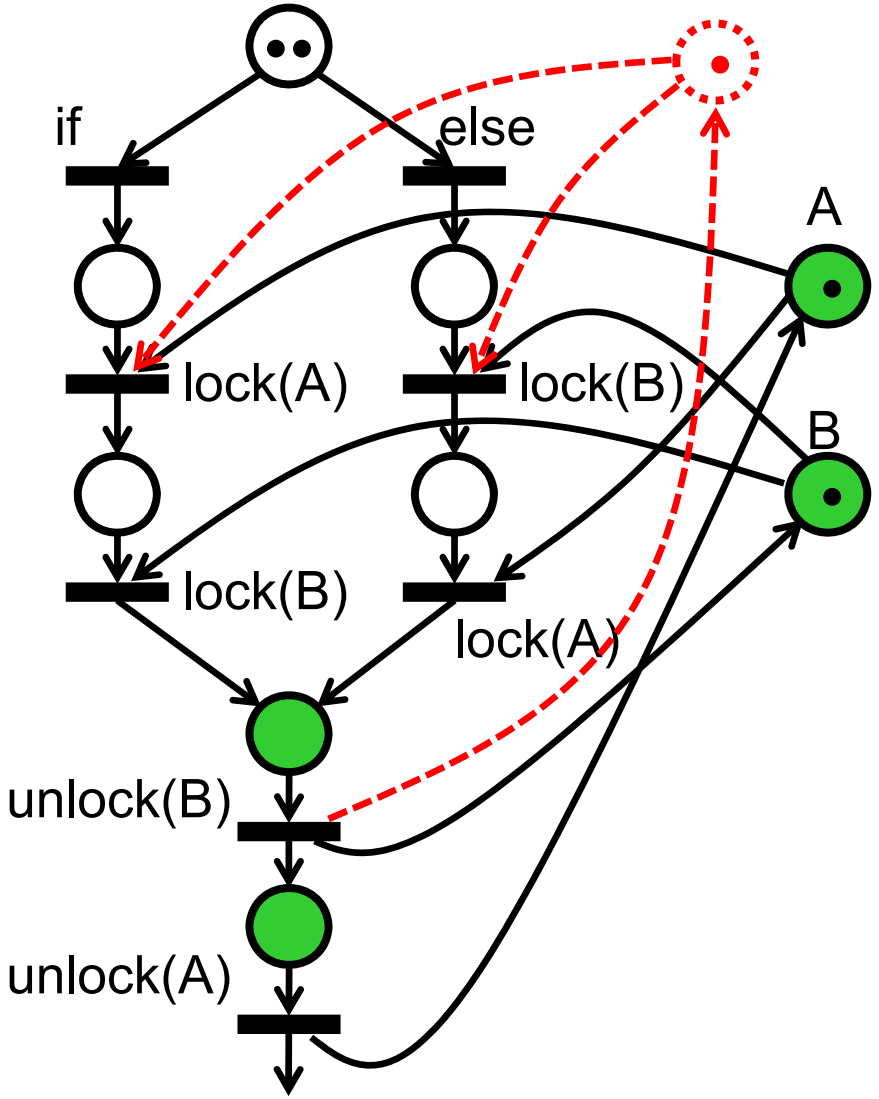
Deadlock!

Architecture

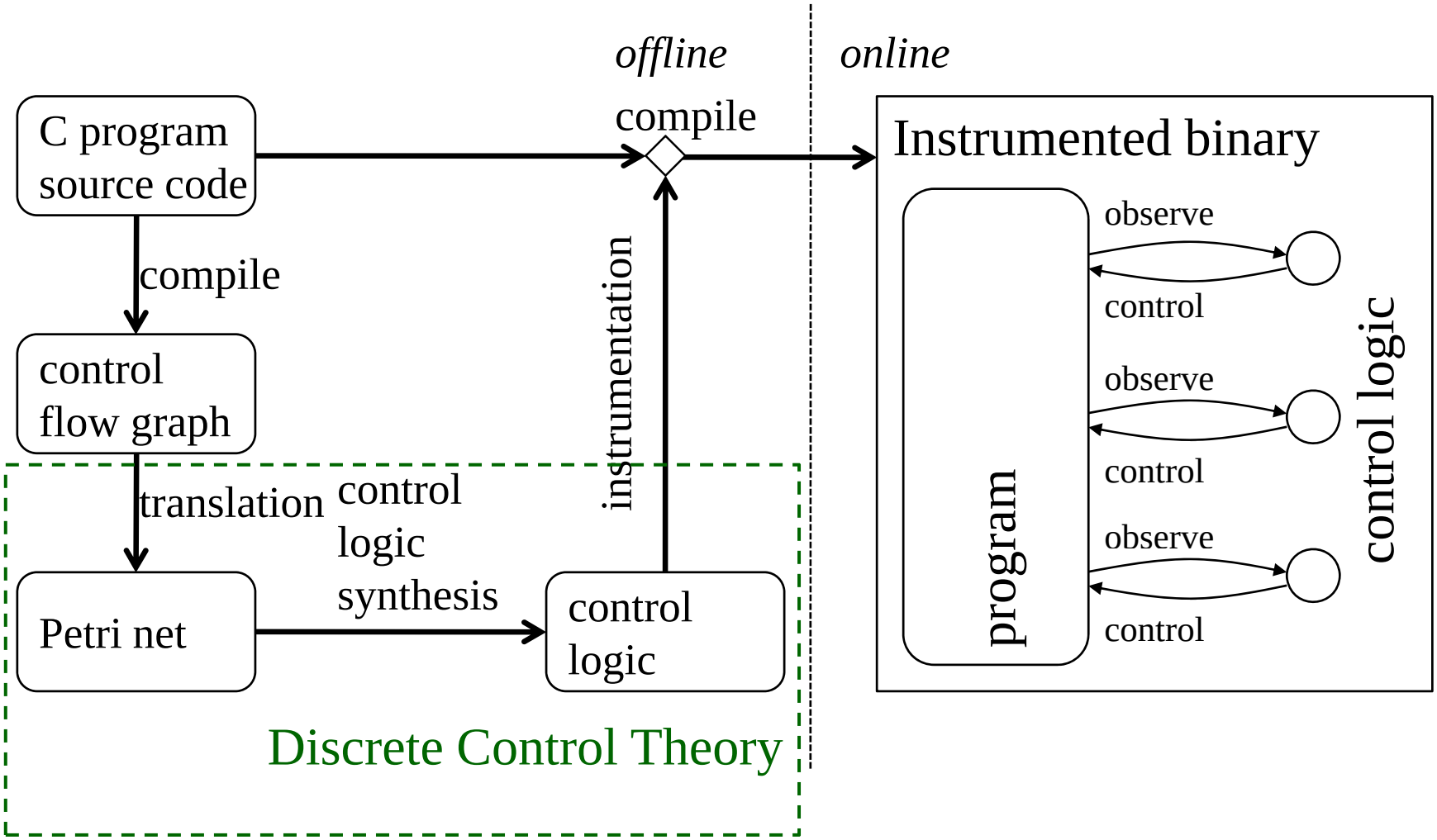


Siphon Based Control

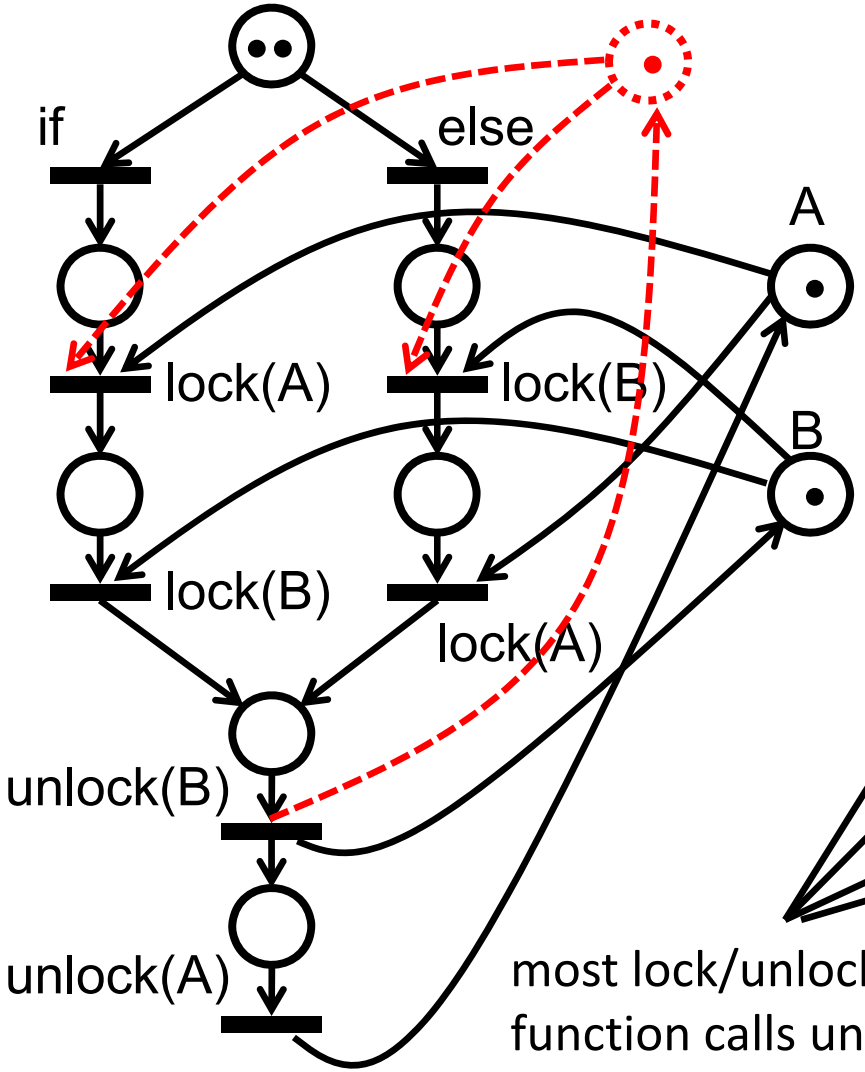
- Siphon is a set of places that can lose tokens permanently
  - structural property
  - related to deadlock
- Synthesize control place to prevent empty siphon
  - linear algebra
  - maximally permissive
- Control logic is
  - fine-grained
  - highly concurrent
  - easy to implement



Architecture



Dining Philosophers: instrumentation




```
void * philosopher(void *arg) {
  ...
  if (RAND_MAX/2 > random()) {
    /* grab A first */
    pthread_mutex_lock(&forkA);
    pthread_mutex_lock(&forkB);
  } else {
    /* grab B first */
    pthread_mutex_lock(&forkB);
    pthread_mutex_lock(&forkA);
  }
  eat();
  replenish(&ctrlplace);
  pthread_mutex_unlock(&forkB);
  pthread_mutex_unlock(&forkA);
  ...
}
```

most lock/unlock function calls unaffected, incur no overhead

Challenges for Large Scale Software

- Modeling
  - language features
    - Handles: function pointer, recursion
    - Ignores: setjmp, longjmp, exception/signal
  - data flow ambiguity: local annotations
  - dynamically selected locks: type analysis
- Control logic synthesis
  - uncontrollability: report at compile time
  - scalability: decomposition & pruning
  - completeness: other synchronization primitives

Performance Evaluation

- Pub-Sub benchmark [OSDI 08]
  - injected deadlocks in common-case logic
  - outperforms Intel STM compiler
  - negligible response time overhead under moderate load
  - 18% throughput reduction with overload workload
- OpenLDAP v2.2.20 [OSDI 08]
  - known & unknown deadlocks in corner-case logic
  - negligible overhead with default configuration
  - at most 11% overhead with bizarre pessimistic configuration
- BIND v9.3.0a0 [Wang, Ph.D Thesis 2009]
  - real workload (trace replay of HP **named** log)
  - 15% overhead with overload query workload

Discrete Control: Benefits

- Provably correct controlled behavior
- Maximal permissiveness
  - Maximal concurrency
- Minimal instrumentation [WODES 2010]
- Offline synthesis + online control
  - Optimal control logic synthesized offline
  - Light-weight control at runtime

Discrete Control: Extensions

- Control specification
  - Linear specification: $l^T M > b$
  - Forbidden state
- Uncontrollable transitions
  - Branches, loops
- Unobservable transitions
  - Library interposition, System calls
- Distributed systems

Conclusions

- Discrete Control Theory provides a principled foundation for the synthesis of provably-correct software
- Gadara eliminates deadlocks from real programs with acceptable overhead
- Useful in several situations
  - rapid prototype development
  - post-release bug fixing

Outline

- Motivation
- Control Engineering
- Gadara Walkthrough
- Discussion

Lessons Learned

- Modeling
  - The difficulty can never be overestimated
  - Identify the right level of abstraction
- Control synthesis
  - Leverage existing literature and inspire the community
  - Fully exploit the features of the class of models
- Implementation
  - Experimental science
  - Practicality is the top priority

Other Applications Under Investigation

- Lock synthesis for atomic sections
  - Yu Liu (SUNY), Scott Smith (JHU)
- Distributed diagnosis in sensor networks
  - Matt Welsh (Harvard)
- Enforcing correct interleaving in concurrent software
  - Satish Narayanasamy (U. of Michigan)
- Controlled simulation of embedded systems
  - Stefan Resmerita (Toyota)

Discussion

- Will new parallel languages or language features make Gadara and other tools unnecessary?
- To what extent can tools, e.g., testing, static analysis, runtime analysis, and control synthesis, help eliminate software bugs?
- Is software synthesis practical, how much can we synthesize automatically?
- Can we automate model building according to the class of control specifications?

Thank You