Model-Driven Verifying Compilation of Synchronous Distributed Applications

Sagar Chaki, James Edmondson
October 1, 2014

MODELS'14, Valencia, Spain

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-0001691

Outline

- Motivation
- Approach
- Sequentialization : SEQSEM & SEQDBL
- Examples
- Experimental Results
- Synchronizer Protocol : 2BSYNC
- Tool Overview & Demo
- Future Work

Motivation

Distributed algorithms have always been important

- File Systems, Resource Allocation, Internet, …

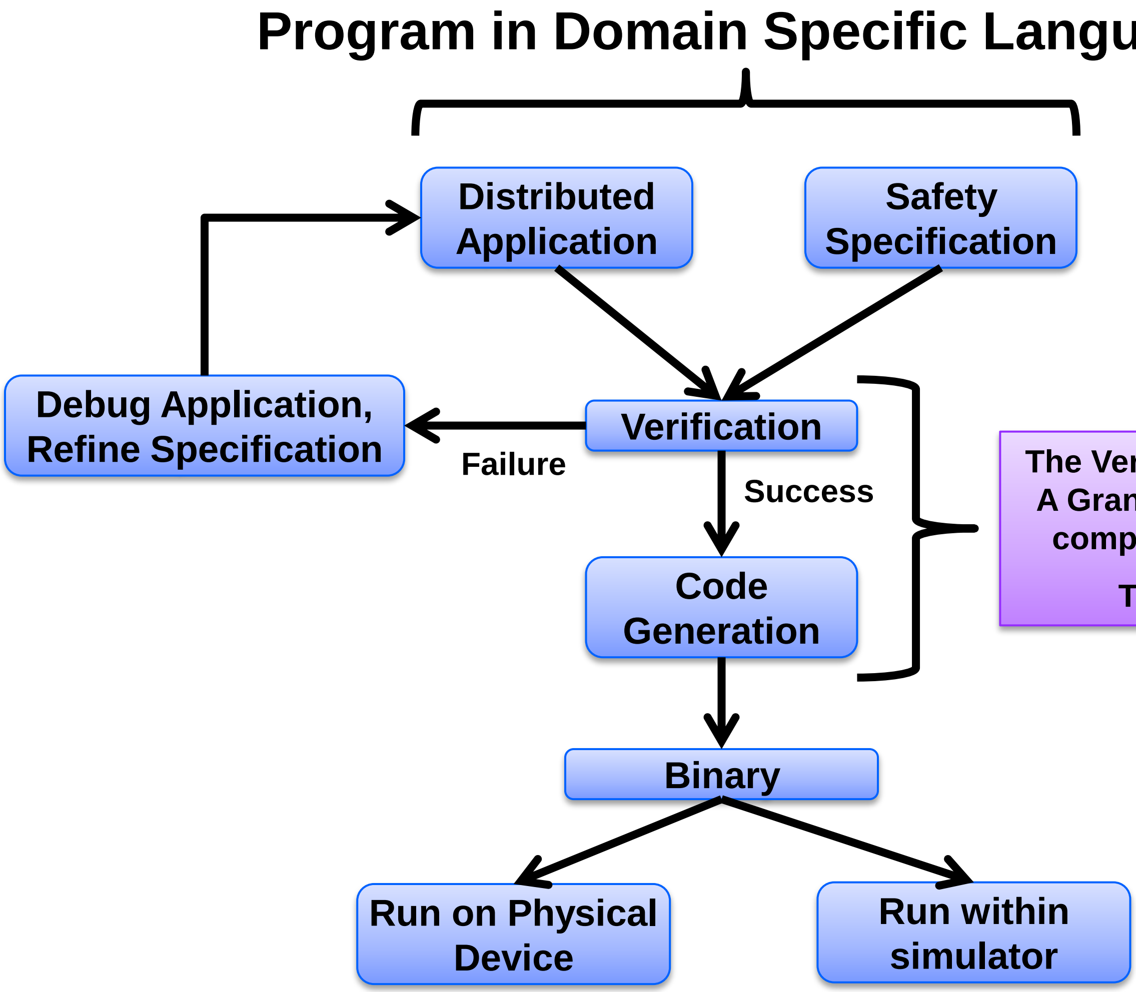
Increasingly becoming safety-critical

- Robotic, transportation, energy, medical

Prove correctness of distributed algorithm implementations

- Pseudo-code is verified manually (semantic gap)
- Implementations are heavily tested (low coverage)

Approach : Verification + Code Generation

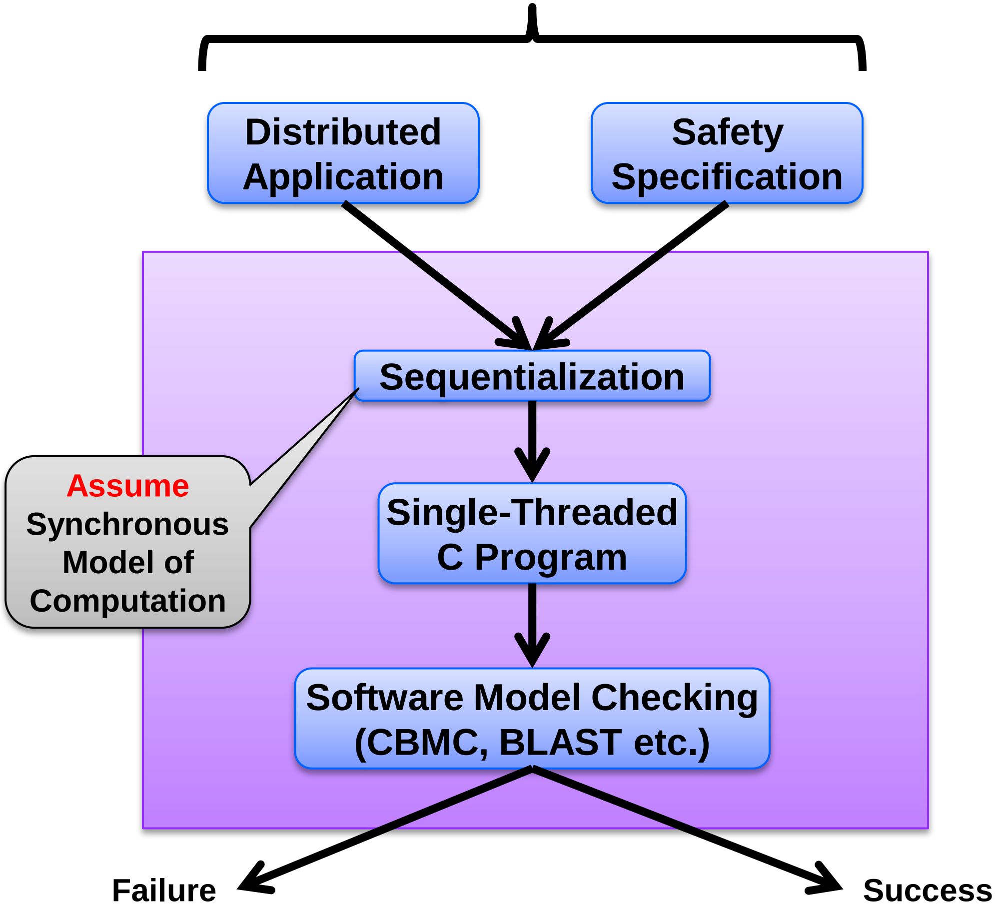


The Verifying Compiler: A Grand Challenge for computing research

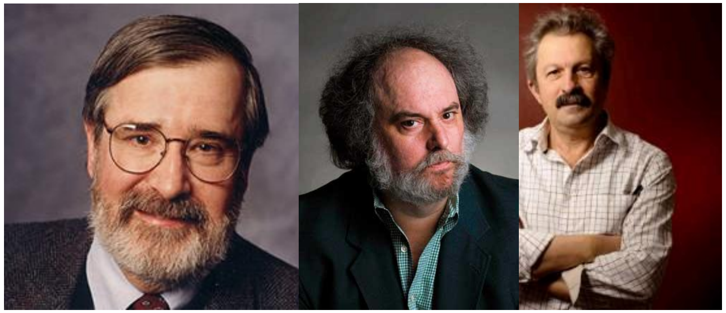
Tony Hoare

Verification

Program in Domain Specific Language



Model Checking



Automatic verification technique for finite state concurrent systems.

- Developed independently by Clarke and Emerson and by Queille and Sifakis in early 1980's.
- ACM Turing Award 2007

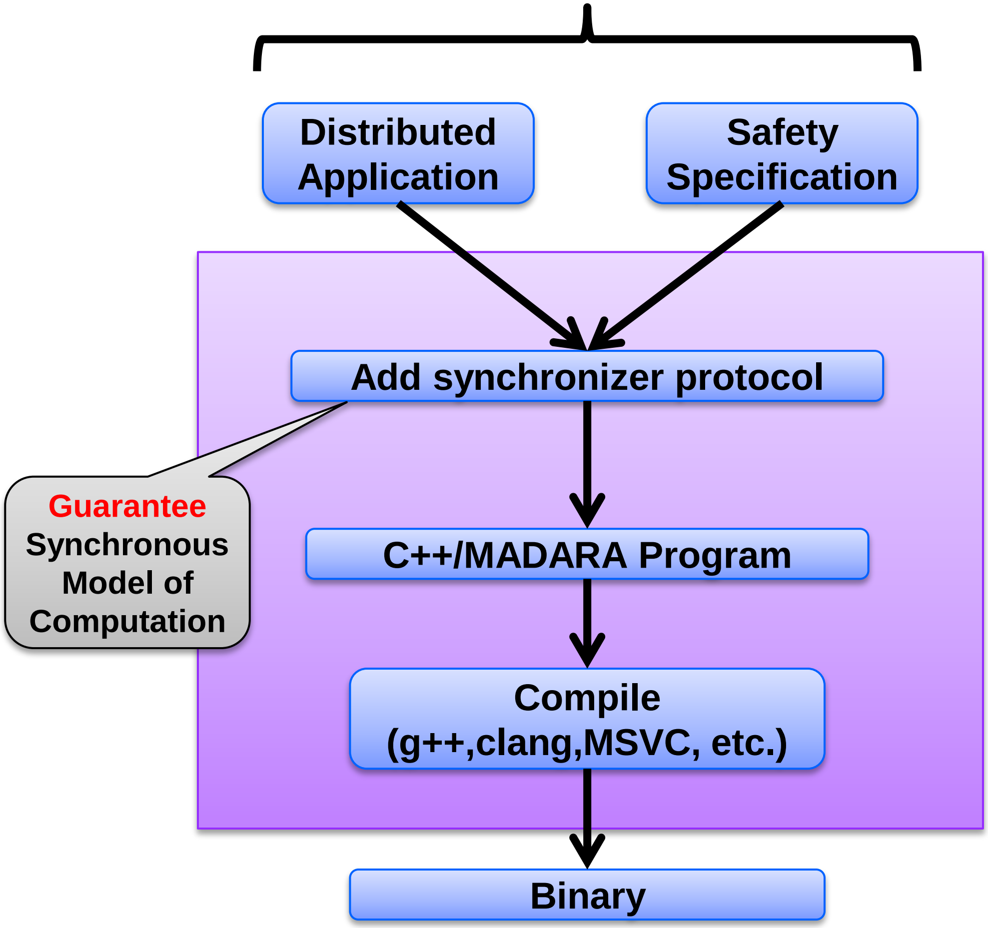
Specifications are written in propositional temporal logic. (Pnueli 77)

- Computation Tree Logic (CTL), Linear Temporal Logic (LTL), …

Verification procedure is an intelligent exhaustive search of the state space of the design

Code Generation

Program in Domain Specific Language



MADARA Middleware

A database of facts: $DB = Var \mapsto Value$

Node i has a local copy: DB_i

- update DB_i arbitrarily
- publish new variable mappings
  - Immediate or delayed
  - Multiple variable mappings transmitted atomically

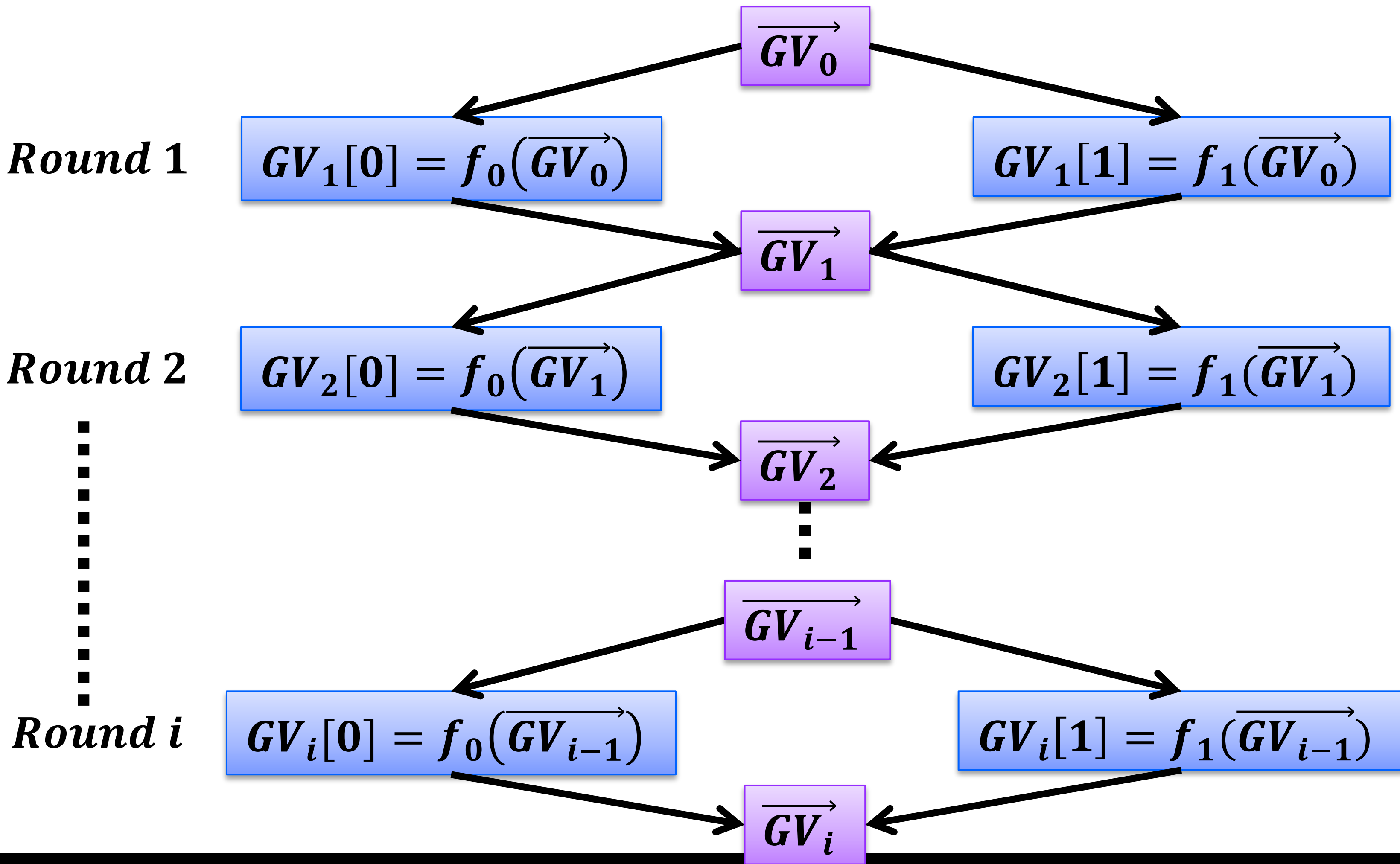
Implicit "receive" thread on each node

- Receives and processes variable updates from other nodes
- Updates ordered via Lamport clocks

Portable to different OSes (Windows, Linux, Android etc.) and networking technology (TCP/IP, UDP, DDS etc.)

Synchronous Distributed Application (SDA)

Node 0 $= f_0()$ **Shared Variables:** $\overrightarrow{GV} = GV[0], GV[1]$ **Node 1** $= f_1()$



SDA Verification

Program with n nodes : $P(n)$

- Each node has a distinct $id \in [1, n]$
- Array GV has n elements, $GV[i]$ writable only by node with id i
- Each element of GV is drawn from a finite domain

In each round, node with id id executes function ρ whose body is a statement

$$
\begin{aligned}
stmt &\coloneqq skip \mid lval = exp && (assignment) \\
&\mid ITE(exp, stmt, stmt) && (if, then, else) \\
&\mid ALL(IV, stmt) && (iterate\ over\ nodes : use\ to\ check\ existence) \\
&\mid \langle stmt^{+} \rangle && (iteration\ of\ statements) \\
lval &\coloneqq GV[id][w] && (lvalues) \\
exp &\coloneqq \top \mid \bot \mid lval \mid GV[iv][w] \mid id \mid IV \mid \diamond(exp^{+}) && (expressions)
\end{aligned}
$$

Initial states and "ERROR" states of the program are define

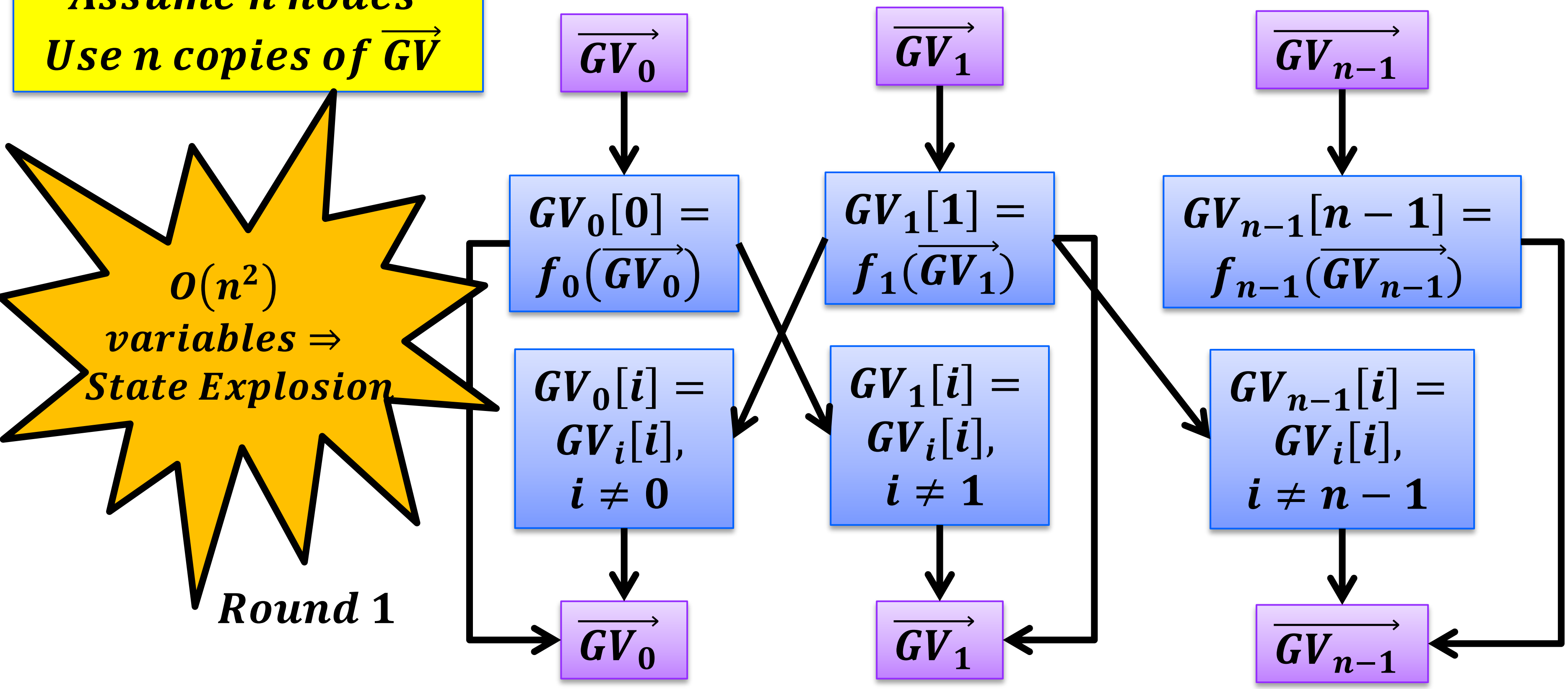
- State ≡ value assigned to all variables

Verification ≡ decide if there is an execution of the program that starts in an initial state and ends in an ERROR state

Semantic Sequentialization: SEQSEM

Node 0 $= f_0()$ Shared Variables: $\overrightarrow{GV} = GV[0], GV[1]$ Node 1 $= f_1()$

Assume n nodes
Use n copies of $\overrightarrow{GV}$



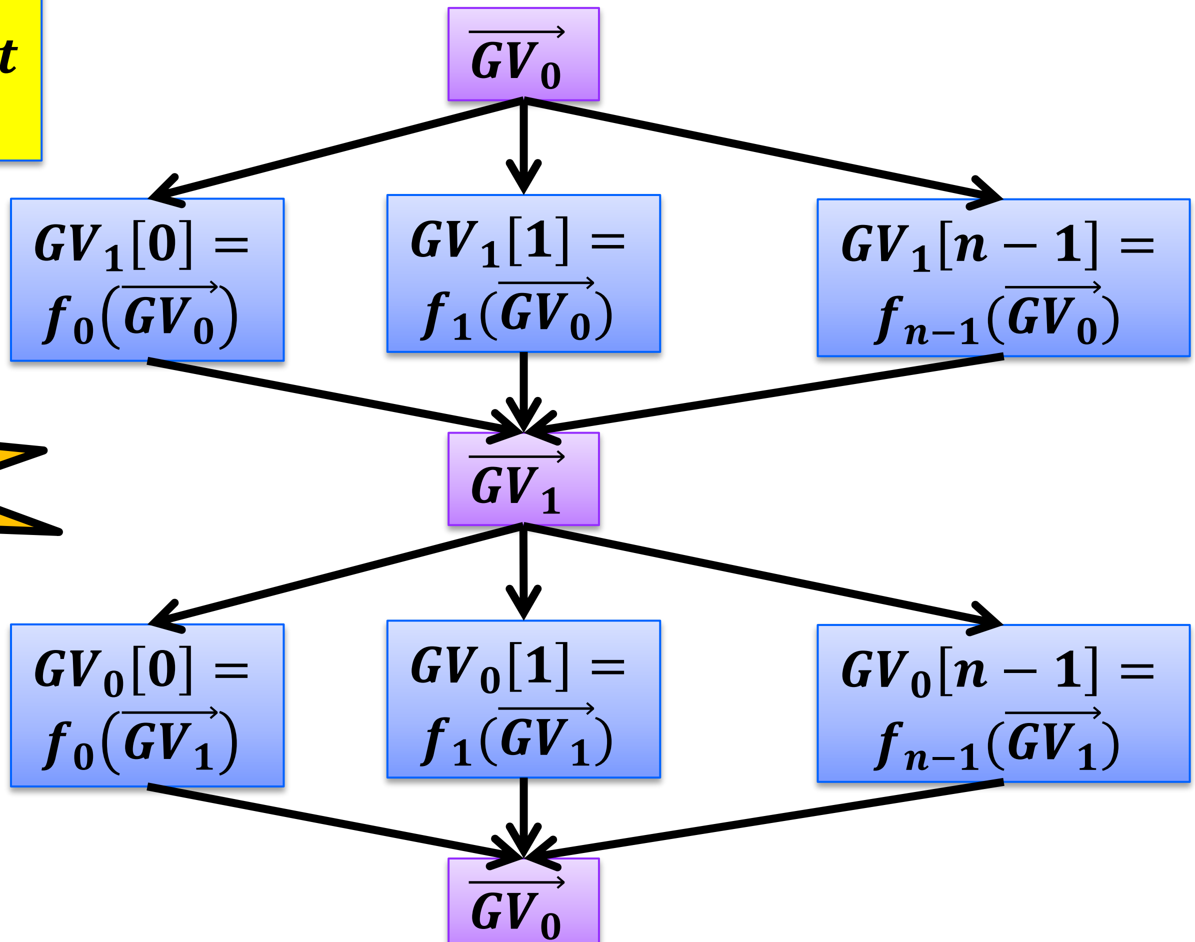
Operations have independence ⇒ reordered sequentially.

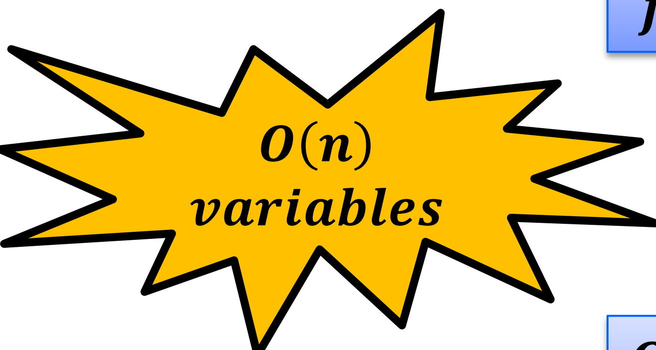
Rounds are repeated in a loop.

Double Buffering Sequentialization: SEQDBL

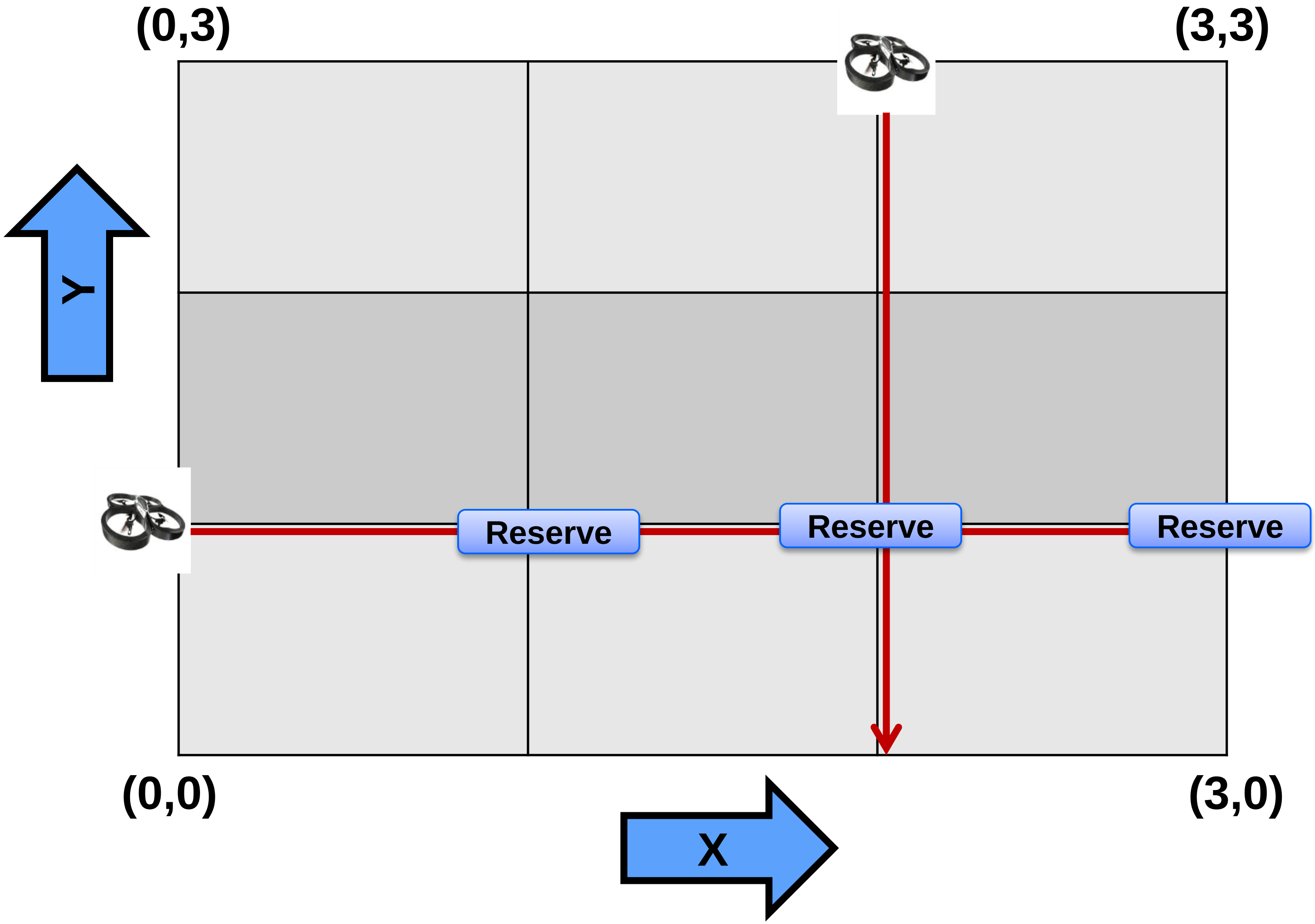
Node 0 $= f_0()$ **Shared Variables:** $\overrightarrow{GV} = GV[0], GV[1]$ **Node 1** $= f_1()$

Use 2 copies of $\overrightarrow{GV}$
Use each copy as input in alternate rounds

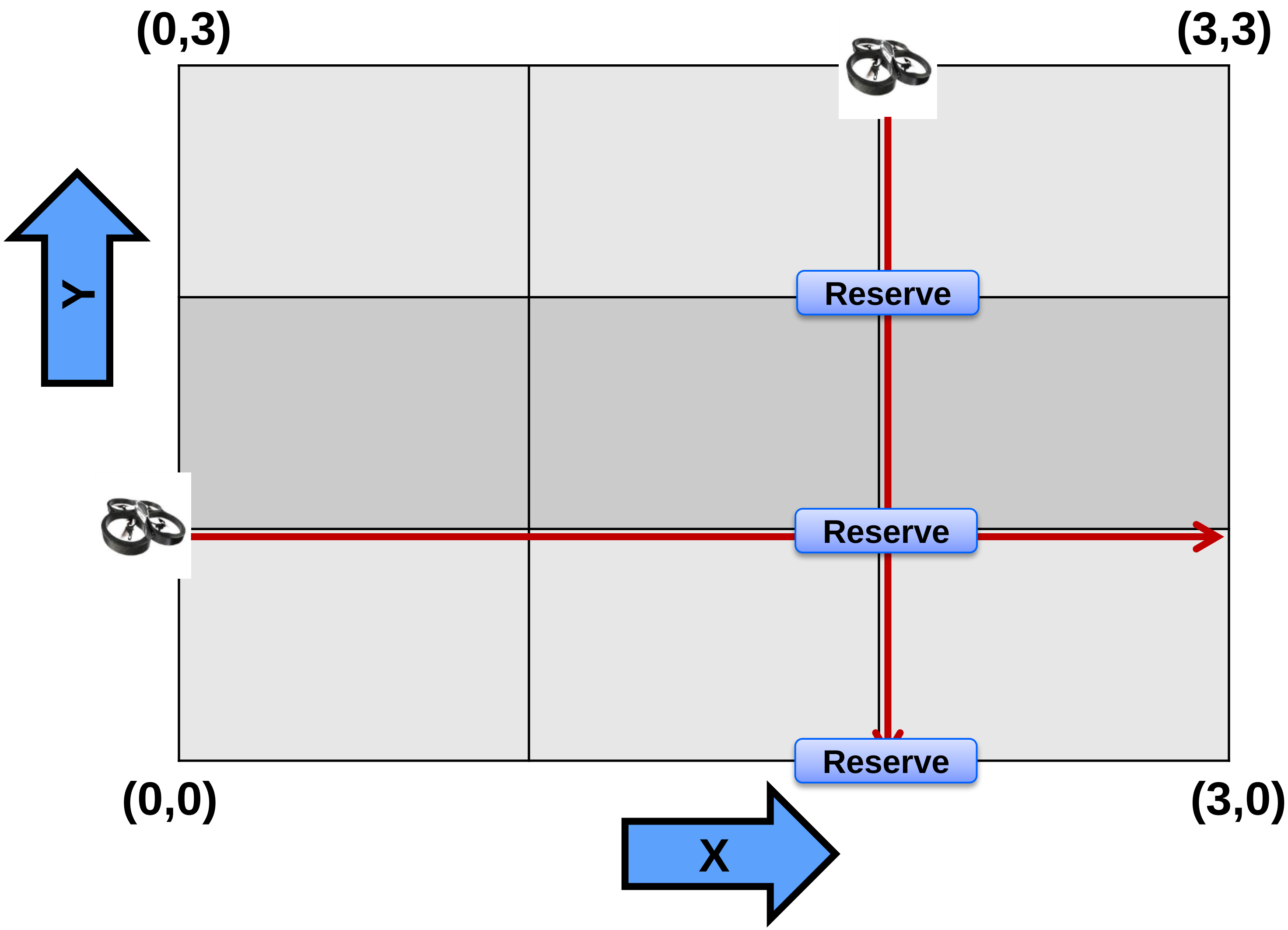




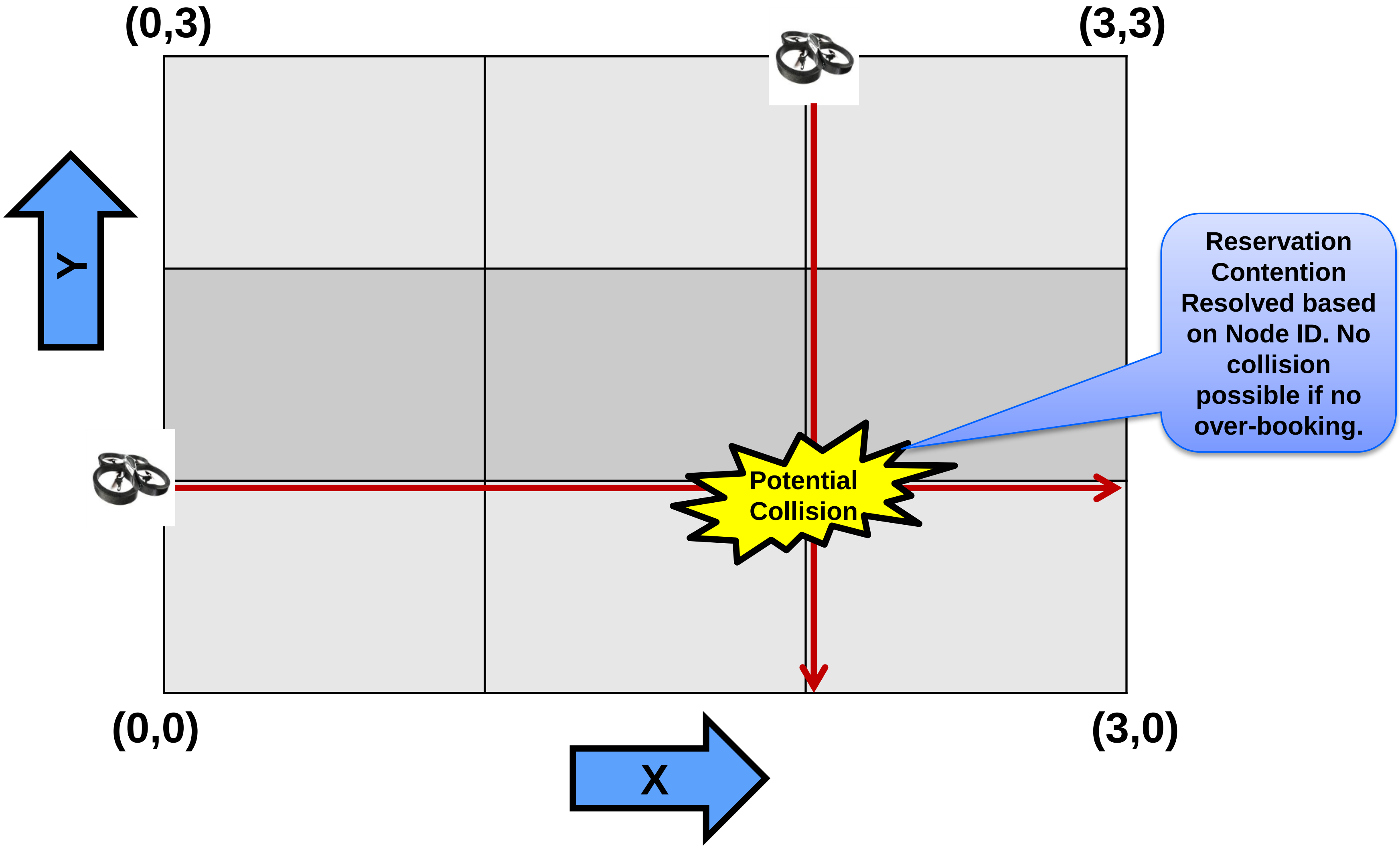
Example: 2D Synchronous Collision Avoidance



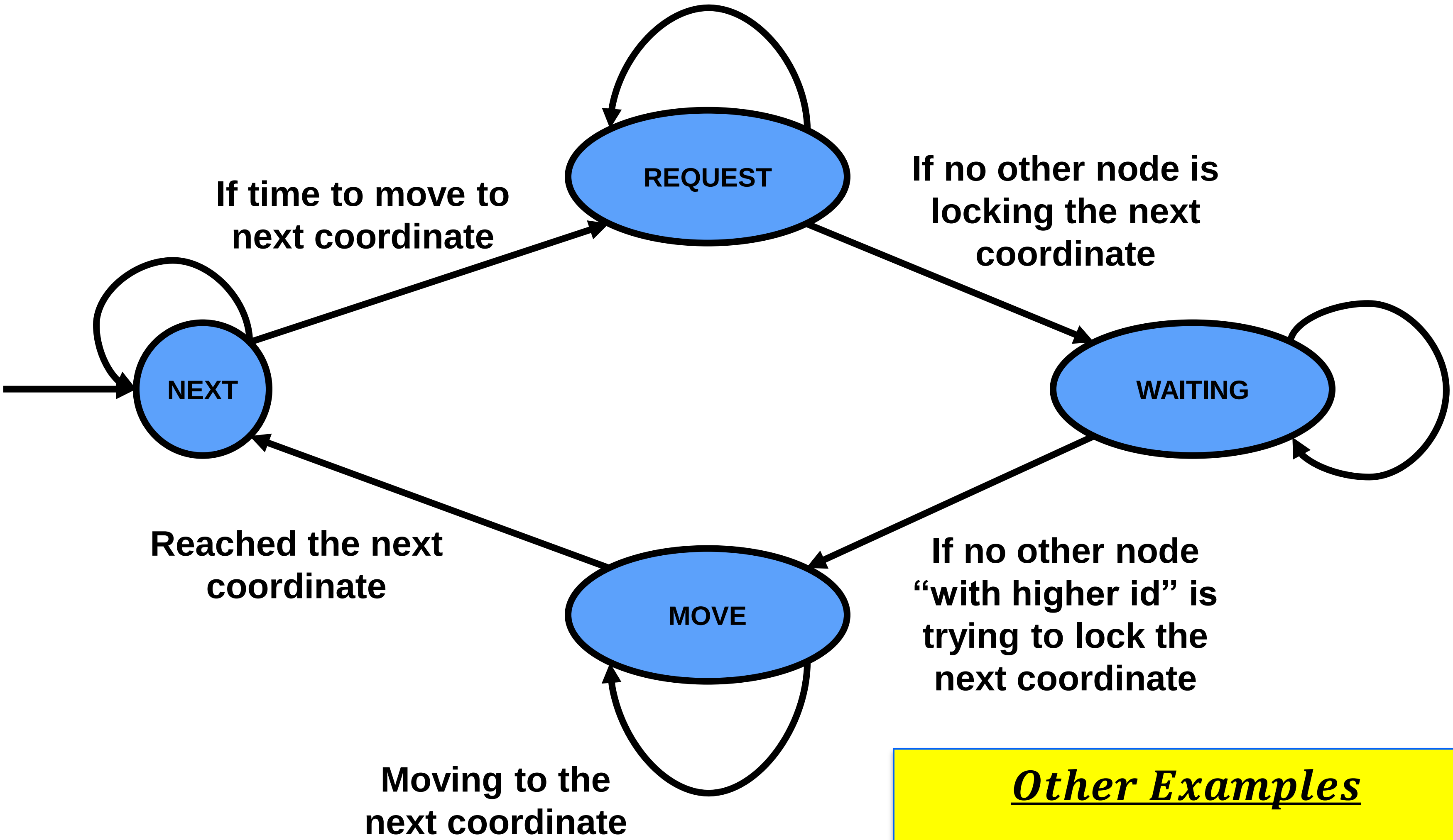
Example: 2D Synchronous Collision Avoidance



Example: 2D Synchronous Collision Avoidance



2D Collision Avoidance Protocol



<u>Other Examples</u>

3D Collision Avoidance

Mutual Exclusion

Results: 3D Collision Avoidance

3DCOLL-OK-4x4						
R	T_S	T_D	T_S	T_D	T_S	T_D
	$n = 2$		$n = 4$		$n = 6$	
10	13	10	59	40	219	96
20	37	31	351	123	1014	480
30	48	48	406	202	–	–
	μ=2.213 σ=0.715					

SEQDBL is better

$$T_S, T_D = model\ checking\ time\ with\ SEQSEM, SEQDBL$$

$$\mu, \sigma = Avg, StDev\ of\ \frac{T_S}{T_D}$$

$$n = \#of\ nodes \quad R = \#of\ rounds \quad G \times G = grid\ size$$

Results: 2D Collision Avoidance

2DCOLL-OK-4x4						
R	T_S	T_D	T_S	T_D	T_S	T_D
	$n=2$		$n=4$		$n=6$	
10	17	25	87	262	280	831
20	123	271	1474	2754	–	–
30	863	1301	–	–	–	–
	μ=0.446 σ=0.118					

2DCOLL-OK-7x7						
R	T_S	T_D	T_S	T_D	T_S	T_D
	$n=2$		$n=4$		$n=6$	
10	74	146	395	1016	1707	–
20	1726	3096	–	–	–	–
30	–	–	–	–	–	–
	μ=0.598 σ=0.202					

Depends on the example

$$T_S, T_D = model\ checking\ time\ with\ SEQSEM, SEQDBL$$

$$\mu, \sigma = Avg, StDev\ of\ \frac{T_S}{T_D}$$

$$n = \#of\ nodes \quad R = \#of\ rounds \quad G \times G = grid\ size$$

Results: Mutual Exclusion

MUTEX-OK						
R	T_S	T_D	T_S	T_D	T_S	T_D
	$n=6$		$n=8$		$n=10$	
60	406	396	1116	1051	2388	2268
80	850	806	2268	1967	4525	4249
100	1404	1381	3584	3452	7092	6764
	μ=1.040 σ=0.038					

MUTEX-BUG1					
T_S	T_D	T_S	T_D	T_S	T_D
$n=6$		$n=8$		$n=10$	
184	175	517	439	1068	959
402	372	1013	925	2203	1812
734	686	1726	1566	3513	3287
μ=1.056 σ=0.060					

SEQSEM and SEQDBL similar

MUTEX-BUG2					
T_S	T_D	T_S	T_D	T_S	T_D
$n=6$		$n=8$		$n=10$	
233	216	637	553	1292	1167
500	462	1218	1112	2602	2139
890	838	2056	1860	4216	3742
μ=1.065 σ=0.056					

T_S, T_D = model checking time with SEQSEM, SEQDBL

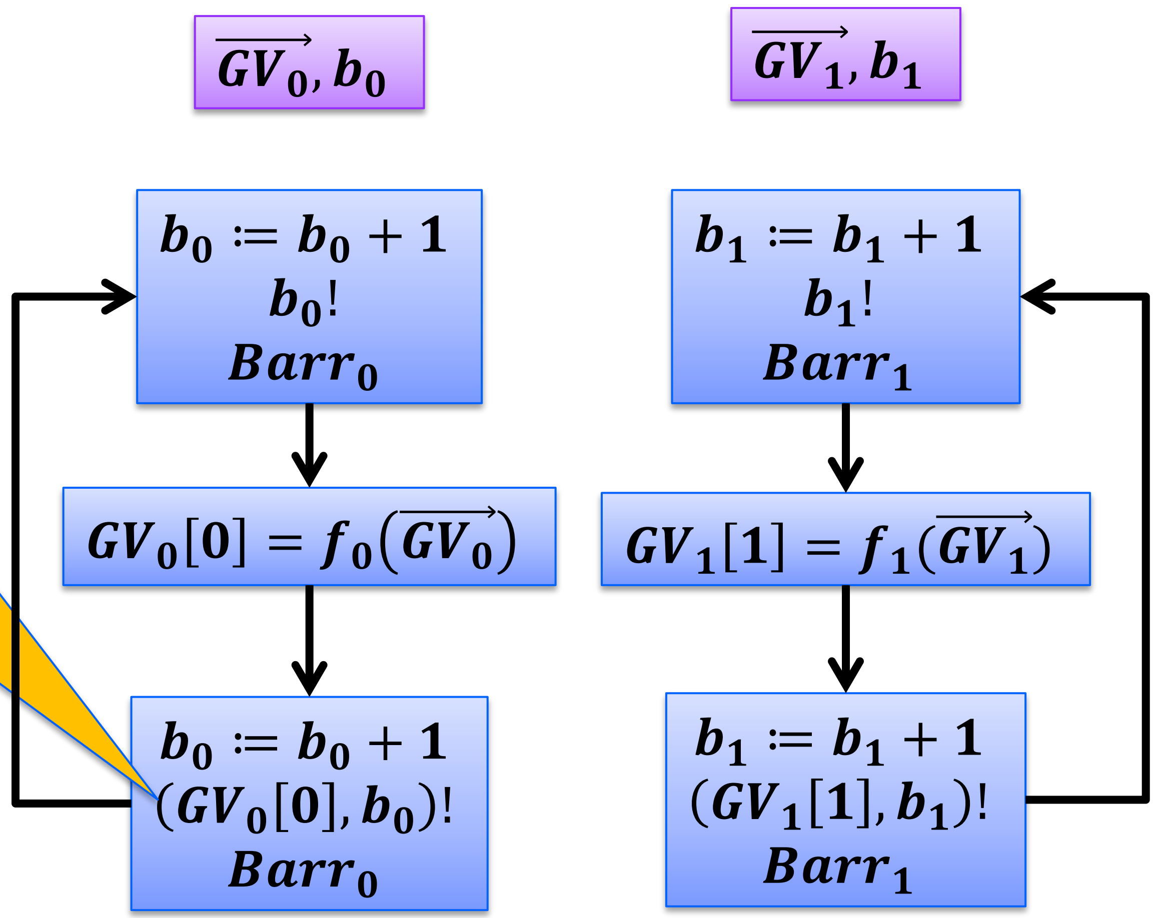
μ, σ = Avg, StDev of $\frac{T_S}{T_D}$

n = #of nodes R = #of rounds $G \times G$ = grid size

Synchronizer Protocol: 2BSYNC

Node 0 $= f_0()$ **Shared Variables:** $\overrightarrow{GV} = GV[0], GV[1]$ **Node 1** $= f_1()$

Use barrier variables**: b_0, b_1 **Initialized to 0



Atomic Send. Either both $GV_0[0]$ and b_0 are received, or none is received. Can be implemented on existing network stack, e.g., TPC/IP

$$Barr_0 \equiv while(b_1 < b_0)\ \ skip;$$

Implicit thread receiving messages and updating variables

Proof of correctness in paper

Tool Overview

Project webpage (http://mcda.googlecode.com)

- Tutorial (https://code.google.com/p/mcda/wiki/Tutorial)

Verification

- daslc --nodes 3 --seq --rounds 3 --seq-dbl --out tutorial-02.c tutorial-02.dasl
- cbmc tutorial-02.c (takes about 10s to verify)

Code generation & simulation

- daslc --nodes 3 --madara --vrep --out tutorial-02.cpp tutorial-02.dasl
- g++ …
- mcda-vrep.sh 3 outdir ./tutorial-02 …

DEMO

Future Work

Improving scalability and verifying with unbounded number of rounds

Verifying for unbounded number of nodes (parameterized verification)

- Paper at SPIN'2014 Symposium

Asynchronous and partially synchronous network semantics

Scalable model checking

- Abstraction, compositionality, symmetry reduction, partial order reduction

Fault-tolerance, uncertainty, …

- Combine V&V of safety-critical and mission-critical properties

QUESTIONS?

Contact Information Slide Format

Sagar Chaki
Principal Researcher
SSD/CSC
Telephone: +1 412-268-1436
Email: chaki@sei.cmu.edu

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

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

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

Synchronous Collision Avoidance Code

```
MOC_SYNC;

CONST X = 4; CONST Y = 4;
CONST NEXT = 0;
CONST REQUEST = 1;
CONST WAITING = 2;
CONST MOVE = 3;

EXTERN int
MOVE_TO (unsigned char x,
         unsigned char y);

NODE uav (id) { … }

void INIT () { … }

void SAFETY { … }
```

```
NODE uav (id)
{
  GLOBAL bool lock [X][Y][#N];
  LOCAL int state,x,y,xp,yp,xf,yf;
  void NEXT_XY () { … }
  void ROUND () {
    if(state == NEXT) { …
      state = REQUEST;
    } else if(state == REQUEST) { …
      state = WAITING;
    } else if(state == WAITING) { …
      state = MOVE;
    } else if(state == MOVE) { …
      state = NEXT;
    } } }
```

```
INIT
{
  FORALL_NODE(id)
    state.id = NEXT;
    //assign x.id and y.id non-deterministically
    //assume they are within the correct range
    //assign lock[x.id][y.id][id] appropriately

  //nodes don't collide initially
  FORALL_DISTINCT_NODE_PAIR (id1,id2)
    ASSUME(x.id1 != x.id2 || y.id1 != y.id2);
}

SAFETY {
  FORALL_DISTINCT_NODE_PAIR (id1,id2)
    ASSERT(x.id1 != x.id2 || y.id1 != y.id2);
}
```

Synchronous Collision Avoidance Code

```
if(state == NEXT) {
  //compute next point on route
  if(x == xf && y == yf) return;
  NEXT_XY();
  state = REQUEST;
} else if(state == REQUEST) {
  //request the lock but only if it is free
  if(EXISTS_OTHER(idp,lock[xp][yp][idp] != 0)) return;
  lock[xp][yp][id] = 1;
  state = WAITING;
} else if(state == WAITING) {
  //grab the lock if we are the highest
  //id node to request or hold the lock
  if(EXISTS_HIGHER(idp, lock[xp][yp][idp] != 0)) return;
  state = MOVE;
}
```

```
else if(state == MOVE) {
  //now we have the lock on (xp,yp)
  if(MOVE_TO()) return;
  lock[x ][y][id] = 0;
  x = xp; y = yp;
  state = NEXT;
}
```