Verifying Synchronous Distributed Applications

Dr. Sagar Chaki
June 11, 2014

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Joint work with Dr. James Edmondson
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

Program in Domain Specific Language

- Distributed Application
- Safety Specification

Verification

- Debug Application, Refine Specification
- Failure
- Success

Code Generation

Binary

- Run on Physical Device
- Run within simulator
Verification

Program in Domain Specific Language

- Distributed Application
- Safety Specification

Sequentialization (assuming synchronous communication)

Single-Threaded C Program

Software Model Checking (CBMC, BLAST etc.)

Failure  Success

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

- Distributed Application
- Safety Specification

Add synchronizer protocol

C++/MADARA Program

Compile (g++, clang, MSVC, etc.)

Binary

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.)
Case Study: Synchronous Collision Avoidance
Example: Synchronous Collision Avoidance

(0,0)

(0,3)

(3,0)

(3,3)

Reserve

Reserve

Reserve

X

Y
Example: Synchronous Collision Avoidance

(0,0)  (0,3)  (3,0)  (3,3)

Reserve  Reserve  Reserve

Y  X
Example: Synchronous Collision Avoidance

Reservation Contention Resolved based on Node ID. No collision possible if no over-booking.
Collision Avoidance Protocol

- **NEXT**
  - If time to move to next coordinate

- **REQUEST**
  - If no other node is locking the next coordinate

- **MOVE**
  - Reached the next coordinate
  - If no other node “with higher id” is trying to lock the next coordinate

- **WAITING**
  - Moving to the next coordinate
Synchronous Collision Avoidance Code

```c
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 { … }
```

```c
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
        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);
    }
} 
```
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;
}
Tool Usage

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 ...
Results: Collision Avoidance

- 2-nodes-4x4-dbl
- 2-nodes-4x4
- 2-nodes-7x7
- 2-nodes-10x10
- 4-nodes-4x4
- 4-nodes-7x7
- 4-nodes-10x10

X-axis: Time (in seconds)
Y-axis: Number of Collisions

Legend:
- Light blue: 2-nodes-4x4-dbl
- Light blue: 2-nodes-4x4
- Red: 2-nodes-7x7
- Green: 2-nodes-10x10
- Purple: 4-nodes-4x4
- Turquoise: 4-nodes-7x7
- Orange: 4-nodes-10x10

Graph shows the comparison of collision avoidance across different node configurations and grid sizes.
Demonstration: Synchronous Collision Avoidance
Future Work

Improving scalability and verifying with unbounded number of rounds

Verifying for unbounded number of nodes (parameterized verification)
  • Paper to appear 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
Biographical Sketch of Presenter

Dr. Sagar Chaki is a Principal Researcher at the Carnegie Mellon Software Engineering Institute. He received a B.Tech in Computer Science & Engineering from the Indian Institute of Technology, Kharagpur in 1999, and a Ph.D. in Computer Science from Carnegie Mellon University in 2005. These days, he works mainly on model checking software for real-time and cyber-physical systems, but he is generally interested in rigorous and automated approaches for improving software quality. He has developed several automated software verification tools, and has co-authored over 50 peer reviewed publications. More details about Dr. Chaki can be found at http://www.contrib.andrew.cmu.edu/~schaki/.
Contact Information Slide Format

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

Web
www.sei.cmu.edu
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