Supervisory Control for Avoidance of Concurrency Bugs

Stéphane Lafortune
University of Michigan

ExCAPE Summer School

12 June 2013
Acknowledgements

- **Yin Wang**, Facebook
- Terence Kelly, HP Labs
- **Hongwei Liao**, GE Research
- Jason Stanley, Hyoun Kyu Cho, Scott Mahlke, University of Michigan
- Ahmed Nazeem, Spyros Reveliotis, Georgia Tech

- Gadara Project
Concurrency Bugs

• Multicore architectures in computer hardware
  Multithreaded programming: Notoriously difficult
  “After 25 years of research, we’re no closer to solving the ‘parallel programming problem’”
  --- Tim Mattson (Intel)

• Data sharing in multithreaded programming:
  – Races
  – Atomicity violations
  – Deadlock
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
Control Engineering Contribution

• Formal approach to model, analyze, and control multithreaded programs: controller synthesis

• Model-based approach using Supervisory Discrete Control of Discrete Event Systems
  – Provably prevent “all” Circular Mutex Wait deadlocks (correctness)
  – Maximally permissive control (optimality)
  – Application scenarios:
    • Rapid prototype development
    • Post-release bugs

• Modeling formalism: Petri nets
  – Graphical model of program, no explicit state space enumeration
  – Capture system concurrency and resource allocation
  – Structural properties can be exploited for controller synthesis
Control Engineering and Computer Science

- Continuous control applied to computer systems problems; [Hellerstein et al., 2004]

- Discrete event control and software systems
  - Iordache & Antsaklis
  - Marchand, Rutten, et al.
  - Benveniste, et al.
  - Tripakis et al.
  - Dingel, Rudie, et al.
Outline

• Introduction ✓

• *The Gadara Approach: Main Features*

• Synthesis of Control Logic for Deadlock Avoidance: Technical Details

• Case Studies and Control Logic Implementation

• Discussion and Conclusion
Gadara Project Architecture

C program source code → compile → control flow graph → translation → Petri net → control logic synthesis → control logic

Instrumented binary

offline
compile

online
instrumentation

observe
control
observe
control
observe
control
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, &ctrlplace);
        pthread_mutex_lock(&forkB);
    } else {
        /* grab B first */
        gadara_lock(&forkB, &ctrlplace);
        pthread_mutex_lock(&forkA);
    }
    eat();
gadara_replenish(&ctrlplace);
    pthread_mutex_unlock(&forkA);
    pthread_mutex_unlock(&forkB);
    ...
}

An example of the Gadara process
Discrete Event Systems: Key Features

• **Discrete State Space**
  - \{on, off, broken\}
  - \{empty, 1, 2, 3, ...\}
  - \{ready_to_send, waiting_for_ack, ...\}

• **Event-driven dynamics**
  - \{start, stop, break, repair\}
  - \{arrival, departure\}
  - \{send_packet, receive_ack, timeout\}

• **System trajectories:** “traces” (or “strings”) of events
  - often, an infinite set [language]
DES Modeling Formalisms (in control engineering)

- Automata (labeled transition systems)
  - simple, intuitive; analytical power
  - extensions to timed and hybrid
  - lack of structure and scalability
- Petri Nets
  - more structure; analysis more difficult in general but more powerful for special classes
An Automaton

Notion of marked state (double circle)
Modeling: Petri Nets Preliminaries

- Petri net structure
  A Petri net is a bipartite graph: two types of nodes

In general, each arc has an arc weight. A special case: ordinary net
Modeling: Petri Nets Preliminaries

- **Petri net dynamics**

![Petri net diagram]

- Transition enabled
- Fire
Modeling: Petri Nets Preliminaries

- Marking (State): $|P|X_1$ vector

\[
M_0 = \begin{bmatrix}
1 \\
0 \\
0 \\
0
\end{bmatrix}
\]

- Incidence matrix of PN structure: $|P| \times |T|$ matrix, denoted by $D$
How are DES models obtained?

• First principles modeling (domain knowledge)
• Abstraction of continuous system (bisimulation or other property)
• Software: directly from source code
Concurrent Software Failures

- Locks are used to manage concurrent accesses to shared data and avoid races
- Enforcement of atomicity conditions requires delaying lock release until after critical section
- Holding on to locks can lead to deadlock
  - Circular wait among a set of threads
  - **Mutual exclusion locks: Mutex**
    - CMW: Circular-Mutex-Wait deadlocks
  - Other kinds of locks: Reader-Writer
Petri Nets and Locks

the PN that models lock acquisition & release

Kavi et al., *IJoPP* 2002
Modeling: Programs Modeled by Petri Nets (from enhanced Control Flow Graph)

• **Place**
  A set of lines of code of program.

• **Transition**
  Lock allocation or release; and program *control flow*

• **Token**
  Thread execution or available lock;
  Multiple threads are represented by multiple tokens.
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);
...
}
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);
  ...
}
Dining Philosophers

start
if

lock(A)
lock(B)

else

lock(B)
lock(A)

eat()
unlock(B)
unlock(A)

if

lock(A)
lock(B)
unlock(B)
unlock(A)

else

lock(B)
lock(A)

No transition enabled.
Deadlock!
Typical Approaches

• Programmers use coarse-grain locking and manage lock acquisition ordering

• Recent CS literature:
  – *Rx*: Restart the program from a recent checkpoint
  – *Healing*: Add more locks to limit concurrency, by heuristics
  – *Dlmmunix*: delay lock allocation if it has caused deadlock before
Modeling: A Deadlock Example in Linux Kernel

```c
/** Thread 1 ***/
spin_lock(&kin->lock);
  ...
    if (!f1.f14_src){
    read_lock(&inetdev_lock);
      ...
        read_lock(&kin_dev->lock);
          ...
            if (...){
              read_unlock(&kin_dev->lock);
                /* UNLOCK(C), devinet.c, inet_select_addr(), 796 */
                 ...
                  break;
                }
              read_unlock(&inetdev_lock);
                /* UNLOCK(C), devinet.c, inet_select_addr(), 800 */
              ...
            read_unlock(&inetdev_lock);
                /* UNLOCK(B), devinet.c, inet_select_addr(), 803 */
          ...
        spin_unlock(&kin->lock);
            /* UNLOCK(A), igmp.c, igmp_timer_expire(), 289 */
    ...
/** Thread 2 ***/
read_lock(&kin_dev->lock);
  for (...){
    ...
      spin_lock_bh(&kin->lock);
        ...
          spin_unlock_bh(&kin->lock);
        read_unlock(&kin_dev->lock);
            /* LOCK(C), igmp.c, igmp_beard_query(), 338 */
          ...
    ...
/** Thread 3 ***/
read_lock(&inetdev_lock);
  if (!lin_dev){
    read_unlock(&inetdev_lock);
      return addr;
    }
  read_lock(&kin_dev->lock);
    /* LOCK(B), devinet.c, inet_select_addr(), 759 */
  ...
  read_unlock(&kin_dev->lock);
    /* UNLOCK(C), devinet.c, inet_select_addr(), 776 */
```
CMW Deadlock: Petri net viewpoint
Analysis of this model, which is restricted to the three threads of interest, reveals two total-deadlock markings that are reachable from the initial marking:

(i) The first total-deadlock marking is $M_1$, where there is one token in $p_{12}$, one in $p_{22}$, and one in $p_{33}$, while all other places are empty. At marking $M_1$, all three threads are involved in the deadlock.

(ii) The second total-deadlock marking is $M_2$, where there is one token in $p_{14}$, one in $p_{22}$, and one in $p_{03}$, while all other places are empty. At marking $M_2$, only Threads 1 and 2 are involved in the deadlock.
CMW Deadlocks: Automata viewpoint

Non-blockingness: no "red" states reachable
Supervisory Control for CMW deadlock avoidance

- This is an instance of SSCP from Stravos Tripakis’ lecture this morning:
  - DES Plant $G$ is the reachability graph of the Petri net under consideration;
  - The only marked state is the initial state
  - Must find maximally-permissive non-blocking supervisor
  - There are uncontrollable events
  - The solution is called the **Supremal Controllable Sublanguage** in Supervisory Control Theory; [Ramadge & Wonham, 1987]
  - Complexity is worst-case quadratic in the number of states of $G$
Uncontrollable transitions
Supervisory Control for CMW deadlock avoidance

• Working with Automata models, or equivalently with the Reachability Graph of the Petri net, presents two challenges:
  – Off-line Computational Burden: many reachable states to explicitly represent
  – Run-time Overhead: control action may need to be updated at each event occurrence:
    \[ \text{Supervisor: } E^* \to \text{Power}(E_c) \]
Supervisory Control for CMW deadlock avoidance

• We have developed two approaches to address the preceding challenges:
  – ICOG: structure-based control logic synthesis
  – MSCL: classification-based control logic synthesis
Outline

• Introduction ✓
• The Gadara Approach: Main Features ✓
• Synthesis of Control Logic for Deadlock Avoidance: Technical Details
• Case Studies and Control Logic Implementation
• Discussion and Conclusion
Petri net Control: Enforcing Linear Specification by Control Place

• Supervision Based on Place Invariants (SBPI)
  [Yamalidou et al. 1996], [Moody et al. 1998], [Iordache et al. 2006]

(1) Control specification: \( l^T M \leq b \)

(2) Control synthesis: a new control (aka monitor) place \( p_c \) with

\[
D_{p_c} = -l^T D \quad \text{(connectivity to the original net)}
\]

\[
M_0(p_c) = b - l^T M_0 \quad \text{(initial tokens)}
\]

(3) The supervision is \textbf{maximally permissive}. 
SBPI-based Control

• Take the linear inequality:
  – Sum[green places] greater than or equal to one
  – Results in red “control” (or monitor) place
  – Deadlock avoided!
• How was
  – $\text{Sum[green places]}$ greater than of equal to one obtained?
For a program to be deadlock-free, we want its Petri net model to be live (e.g., reversible: always able to return to its initial state)

The key is to map liveness of the Petri net, a behavioral property, to a structural property of the net

Key notion: Siphon
Analysis: Siphon – Structural Property

Definition and Property

■ Siphon
   A set of places whose

   \[ \{\text{Input Transitions}\} \subseteq \{\text{Output Transitions}\} \]

■ Property
   If a siphon becomes empty, it remains empty forever

→ Write a linear inequality that guarantees the siphon is not empty!
- Green places form siphon
- Deadlock when siphon is empty
Siphon Based Control

- Siphon is a set of places that can lose tokens permanently
  - structural property
  - related to deadlock
- Using SBPI, synthesize control place to prevent empty siphon
  - linear algebra
  - maximally permissive
• Control logic is
  – fine-grained
  – highly concurrent
  – easy to implement
What issues arise?

• Establish precise connection between liveness of Petri net and certain kinds of bad siphons

• Build a control strategy based on using SBPI to avoid bad siphons

• Iterations may be necessary:
  – Convergence must be established

• Uncontrollability of some transitions must be handled
Outline

• Introduction ✓
• The Gadara Approach: Main Features ✓
• Synthesis of Control Logic for Deadlock Avoidance: Technical Details
• Case Studies and Control Logic Implementation
• Discussion and Conclusion
Modeling: Definition of Gadara Nets

- Ordinary net: all arcs have weight 1

Condition 1
- Three types of places:
  1. $P_0$: idle places
  2. $P_R$: resource places
  3. $P_S$: operation places
Modeling: Definition of Gadara Nets

Condition 2
• Set of transitions

Condition 3
• Each process subnet is strongly connected state machine: single arc in/out of each transition

Condition 4
• Branch selection is not constrained by resources.
Modeling: Definition of Gadara Nets

Condition 5
• All resources are conservative in the net; notion of $P$-semiflow.

Condition 6
• Initial marking.

Condition 7
• Any operation place is involved with at least one type of resource.
The (ordinary) Gadara net $N_G$ is live iff $R(N_G, M_0)$ contains no empty siphons.

**Implications:**

- Program is deadlock-free (Goal)
- $\Leftrightarrow$ Gadara net is live (Behavioral Property)
- $\Leftrightarrow$ Gadara net cannot reach a problematic siphon (Structural Property)
Control: Challenges

• The need to iterate
  – Synthesized control place $\rightarrow$ generalized resource place
  – Can introduce new siphons, if coupled with existing resources
  – The new siphons are not considered in previous iterations

- Non-ordinary nets
  - Ordinary net $\rightarrow$ add control place $\rightarrow$ potentially non-ordinary net
  - The problem of optimal control in non-ordinary net based on siphons (structural analysis) is not well-resolved yet \cite{Li:2009}
Controlled Gadara Net

- Control place is a generalized resource place, with its own semiflow
Why iterations are necessary: Automata viewpoint

Initial and Marked state

Blocking states

Deadlock state

Livelock states

Nonblocking: no "red" states reachable
Why the controlled net may not remain ordinary

Supervision Based on Place Invariants (SBPI)

(1) Control specification: \( l^T M \leq b \)

(2) Control synthesis: a new control (aka monitor) place \( p_c \) with

\[
D_{p_c} = -l^T D
\]

(connectivity to the original net)

\[
M_0(p_c) = b - l^T M_0
\]

(initial tokens)

(3) The supervision is maximally permissive.
Control: Liveness Property - General Case

1. The Gadara net $N_G$ is live iff $R(N_G, M_0)$ contains no empty siphons.

2. The Controlled Gadara net $N_G^c$ is live iff $R(N_G^c, M_0)$ contains no Resource-Induced Deadly Marked siphons.

Implications (Still Apply)

- Program is deadlock-free (Goal)
- $\Leftrightarrow$ Gadara net is live (Behavioral Property)
- $\Leftrightarrow$ Gadara net cannot reach a problematic siphon (Structural Property)
Control: The Iterative Control Methodology – ICOG

Main Properties of ICOG:

- Iterative control based on structural analysis
- Optimal: Maximally-permissive and Liveness-enforcing (MPLE)
- Finite convergence

Diagram:

1. Input the Gadara net
2. Identify unsafe states using structural analysis
3. Are there any new unsafe states?
   - NO: Output the controlled Gadara net
   - YES: Run the UCCOR Algorithm to synthesize monitor places that remove the new unsafe states

Update
Look up
Global Bookkeeping Set $\Phi$
The UCCOR Algorithm: Controlling the identified bad siphons

• **Covering:** A generalized marking allowing for "don’t care" components
  • Enables a compact representation of a set of states that contains the identified deadlock

Some Important Properties:

- Optimal: MPLE
- Applies to non-ordinary Gadara nets
- Unit arc weights for synthesized control places
Control: The Customized ICOG-O Methodology

Main Properties of ICOG-O:

- Iterative control based on structural analysis
- Optimal: Maximally-permissive and Liveness-enforcing (MPLE)
- Finite convergence
- No redundant control logic and remains *ordinary*

```
Input the Gadara net¹ → Identify unsafe states using structural analysis² → Are there any new unsafe states³? → Run the UCCOR-O Algorithm to synthesize monitor places that remove the new unsafe states⁴ → Output the controlled Gadara net⁵
```

MIP formulation

2013/06/12 - ExCAPE Summer School
MSCL: Alternative Approach

• Minimizing the overhead of the control logic is essential in this application domain
• ICOG does not guarantee that the number of control places is minimized
• MSCL – Marking Separation using linear CLassifier: an alternative control logic synthesis method that leverages SBPI, but in a single overall application with provably minimum number of monitor places
  – Relies on classification theory to separate safe and unsafe states; requires (partial) enumeration of such states first
  – Details omitted here
Outline

• Introduction ✓
• The Gadara Approach: Main Features ✓
• Synthesis of Control Logic for Deadlock Avoidance: Technical Details ✓
• Case Studies and Control Logic Implementation
• Discussion and Conclusion
Linux kernel deadlock
Linux kernel deadlock – MSCL
Linux kernel deadlock – SSCP approach
Summary so far

• CMW-deadlock avoidance in multithreaded programs using DES control theory
• Challenges: modeling, analysis, control, implementation
• Modeling: at compile time
• Analysis and control: exploit Gadara net structure

• *Implementation of control places by code instrumentation*
Gadara Project Architecture

C program source code

compile

control flow graph

translation

Petri net

compile

instrumentation

control logic

observe

control

observe

control

observe

control

Instrumented binary

program

control logic
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);
    ...
}
OpenLDAP (Lightweight Directory Access Protocol)

OpenLDAP main()
OpenLDAP Deadlock

ldap_pvt_thread_rdwr_wlock(&bdb->bi_cache.c_rwlock); /* LOCK(A) */
...
ldap_pvt_thread_mutex_lock( &bdb->bi_cache.lru_mutex ); /* LOCK(B) */
...
ldap_pvt_thread_rdwr_wunlock(&bdb->bi_cache.c_rwlock); /*UNLOCK(A)*/
...
if ( bdb->bi_cache.c_cursize>bdb->bi_cache.c_maxsize ) {
  for (...; ...; ...) {
    ...
    ldap_pvt_thread_rdwr_wlock(&bdb->bi_cache.c_rwlock); /* LOCK(A) */
    ...
    ldap_pvt_thread_rdwr_wunlock(&bdb->bi_cache.c_rwlock); /*UNLOCK(A)*/
  }
}
...
else {
...
}
ldap_pvt_thread_mutex_unlock( &bdb->bi_cache.lru_mutex ); /*UNLOCK(B)*/
OpenLDAP deadlock
OpenLDAP Deadlock

ldap_pvt_thread_rwlock_lock(&bdb->bi_cache.c_rwlock);  /* LOCK(A) */
...
ldap_pvt_thread_mutex_lock( &bdb->bi_cache.lru_mutex );  /* LOCK(B) */
...
ldap_pvt_thread_rdwr_wunlock(&bdb->bi_cache.c_rwlock);  /*UNLOCK(A)*/
...
if ( bdb->bi_cache.c_cursize>bdb->bi_cache.c_maxsize ) {
  for (...; ...; ...) {
    ...
    ldap_pvt_thread_rwlock_lock(&bdb->bi_cache.c_rwlock);  /* LOCK(A) */
    ...
    ldap_pvt_thread_rwlock_wunlock(&bdb->bi_cache.c_rwlock);  /*UNLOCK(A)*/
  }
  replenish(&ctrl_place);
  ...
} else {
  replenish(&ctrl_place);
  ...
}
ldap_pvt_thread_mutex_unlock( &bdb->bi_cache.lru_mutex );  /*UNLOCK(B)*/

Max Permissiveness  ↓
Max Concurrency
Comparing With the Human Fix

```c
ldap_pvt_thread_rdwr_wlock(&bdb->bi_cache.c_rwlock);  /* LOCK(A) */
...
ldap_pvt_thread_mutex_lock( &bdb->bi_cache.lru_mutex );  /* LOCK(B) */
...
ldap_pvt_thread_rdwr_wunlock(&bdb->bi_cache.c_rwlock);  /*UNLOCK(A) */
...
if ( bdb->bi_cache.c_cursize>bdb->bi_cache.c_maxsize ) {
    for (,..., ..., ...) {
        ldap_pvt_thread_rdwr_wlock(&bdb->bi_cache.c_rwlock);  /* LOCK(A) */
        ldap_pvt_thread_rdwr_wunlock(&bdb->bi_cache.c_rwlock);  /* UNLOCK(A) */
    }
}
else {
    ...
    ldap_pvt_thread_mutex_unlock( &bdb->bi_cache.lru_mutex );  /* UNLOCK(B) */
}
```

- Simple fix?
- Newly added code violated the lock ordering again a year later.
- The final fix introduced new locks and rewrote the code completely.
- Control solution finds and fixes both deadlocks automatically.

• Simple fix?
• Newly added code violated the lock ordering again a year later.
• The final fix introduced new locks and rewrote the code completely.
• Control solution finds and fixes both deadlocks automatically.
Outline

• Introduction ✓
• The Gadara Approach: Main Features ✓
• Synthesis of Control Logic for Deadlock Avoidance: Technical Details ✓
• Case Studies and Control Logic Implementation ✓
• Discussion and Conclusion
Challenges for Large Scale Software

• Modeling
  – Model accuracy: conservative model (superset)
  – 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: constraint transformation
  – scalability: decomposition & pruning
  – completeness: other synchronization primitives
Practical Lock/Unlock Pairing

• Challenges
  – Infeasible paths
  – Spanning function boundaries
  – Pointers

• Lock/Unlock pairing
  – Path sensitive program analysis using a SAT solver
  – Combines best effort static analysis and dynamic checking

```c
if (x)
  lock(A);
...
if (x)
  unlock(A);
```
Software Development Status

- Rudimentary compiler tools
  - Static analysis
  - Instrumentation

- Relatively complete control tool suite
  - Random net benchmark (generator and verifier)
  - A few control synthesis algorithms
  - GUI
Evaluation of ICOG-O (MILP): Stress Test

<table>
<thead>
<tr>
<th>SS</th>
<th>US</th>
<th>time (s)</th>
<th>iters</th>
</tr>
</thead>
<tbody>
<tr>
<td>786,430</td>
<td>487,990</td>
<td>46.05</td>
<td>102</td>
</tr>
<tr>
<td>727,240</td>
<td>295,290</td>
<td>2.17</td>
<td>48</td>
</tr>
<tr>
<td>532,630</td>
<td>233,800</td>
<td>46.05</td>
<td>61</td>
</tr>
<tr>
<td>373,700</td>
<td>136,260</td>
<td>18.45</td>
<td>91</td>
</tr>
<tr>
<td>354,270</td>
<td>64,488</td>
<td>25.92</td>
<td>29</td>
</tr>
<tr>
<td>336,250</td>
<td>200,370</td>
<td>8.35</td>
<td>83</td>
</tr>
<tr>
<td>320,180</td>
<td>118,470</td>
<td>18.35</td>
<td>91</td>
</tr>
<tr>
<td>290,970</td>
<td>50,002</td>
<td>3.54</td>
<td>51</td>
</tr>
<tr>
<td>285,700</td>
<td>64,386</td>
<td>0.34</td>
<td>13</td>
</tr>
<tr>
<td>271,780</td>
<td>64,488</td>
<td>46.21</td>
<td>29</td>
</tr>
<tr>
<td>247,920</td>
<td>84,502</td>
<td>26.41</td>
<td>57</td>
</tr>
<tr>
<td>226,330</td>
<td>28,242</td>
<td>0.50</td>
<td>20</td>
</tr>
<tr>
<td>176,960</td>
<td>26,788</td>
<td>0.42</td>
<td>13</td>
</tr>
<tr>
<td>176,920</td>
<td>22,392</td>
<td>0.44</td>
<td>20</td>
</tr>
</tbody>
</table>

SS: Safe states; US: Unsafe states (not explicitly generated by ICOG-O)
Evaluation (cont.):
Replacing the MIP formulation for deadlock detection with a SAT formulation

<table>
<thead>
<tr>
<th>SS</th>
<th>USS</th>
<th>Time (s)</th>
<th>Monit</th>
<th>Iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>6,090,459,834</td>
<td>5,005,028,660</td>
<td>8,832.9</td>
<td>2,013</td>
<td>22,245</td>
</tr>
<tr>
<td>2,310,010,077</td>
<td>1,849,976,998</td>
<td>1,819.4</td>
<td>1,167</td>
<td>4,948</td>
</tr>
<tr>
<td>661,807,580</td>
<td>660,550,680</td>
<td>862.7</td>
<td>1,235</td>
<td>6439</td>
</tr>
<tr>
<td>431,175,853</td>
<td>427,807,294</td>
<td>71,781.1*</td>
<td>7,155*</td>
<td>30,000*</td>
</tr>
<tr>
<td>182,819,144</td>
<td>165,869,820</td>
<td>1,036.9</td>
<td>1,073</td>
<td>6,929</td>
</tr>
<tr>
<td>90,139,807</td>
<td>73,488,022</td>
<td>117.8</td>
<td>586</td>
<td>2,062</td>
</tr>
<tr>
<td>59,503,951</td>
<td>51,035,234</td>
<td>274.0</td>
<td>662</td>
<td>4,578</td>
</tr>
<tr>
<td>15,567,136</td>
<td>10,212,724</td>
<td>6.0</td>
<td>175</td>
<td>395</td>
</tr>
</tbody>
</table>
Detecting Unsafe States Using SAT

Main Features:
- Exploit structural properties of Gadara nets
- No state space enumeration
- More scalable than MIP-based method in general
- Compatible with ICOG-O

Key Ideas:
- Enforce total deadlock in the net
- Exploit the semiflow equations

Binary variable $p$ for each operation and resource place:

$$\bigvee_{p\in P_R} \neg p$$

$$\bigwedge_{t\in T} \neg t \quad \text{where } t = \bigwedge_{p\in \bullet t} p$$

$$\left( \bigvee_{p\in S_r} p \right) \bigwedge f(S_r)$$

where $f(S_r) = \bigwedge_{\{p_i, p_j\} \in S_r} \neg (p_i \land p_j)$

Restrict to CMW deadlocks

Enforce total deadlock

Enforce resource semiflows

~ state equation
Theoretical Side: Latest Results

• Extension of Gadara methodology to handle certain types of atomicity violations
• Non-linear classifiers for:
  – RW-locks
Lessons Learned

• Solving software problems using control theory is relevant and rewarding

• Bridging research work from different communities is the key to success but also very challenging
  – Theoretical solutions are often preferred, although ad hoc ones typically prevail in the literature
  – All solutions are evaluated by the same standard of practicality
Conclusion

• Control theory provides a new principled foundation to handle software failures
• Cross-pollinating advances both control theory and relevant CS areas
• Many open issues and research opportunities
  – Collaboration welcome!
References

• Gadara Project: please refer to
  – http://gadara.eecs.umich.edu/papers.html
  – CS papers: Wang et al.: OSDI’08 and POPL’09; Cho et al., CGO’13
  – Control engineering papers:
    • Nazeem et al.: IEEE TAC (2002)
• See above papers for pointers to relevant literature
• stephane@umich.edu; hwliao@umich.edu (H. Liao);
  yinw@umich.edu (Y. Wang); netforce@umich.edu (H.K. Cho)