Formal Verification by Model Checking

Natasha Sharygina
Carnegie Mellon University

Guest Lectures at the Analysis of Software Artifacts Class, Spring 2005

Outline

Lecture 1: Overview of Model Checking

Lecture 2: Complexity Reduction Techniques

Lecture 3: Software Model Checking

Lecture 4: State/Event-based software model checking

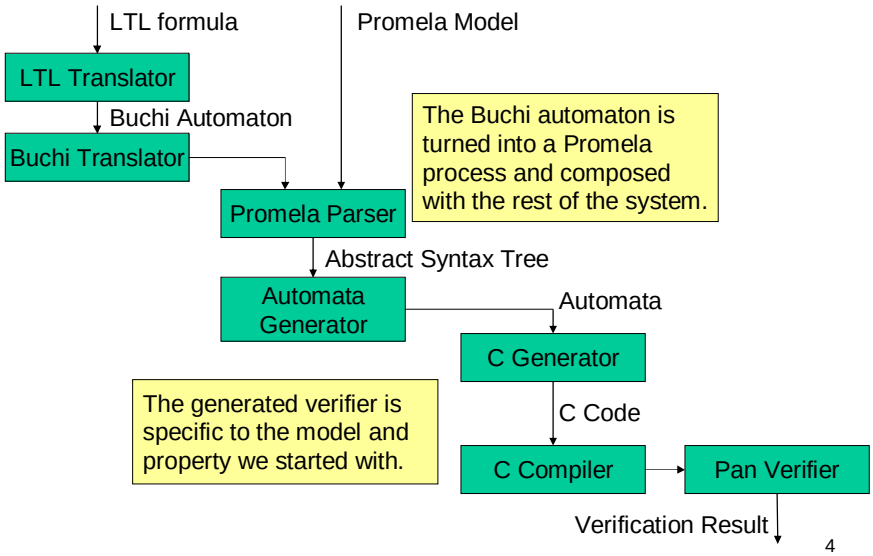
Lecture 5: Component Substitutability

Lecture 6: Model Checking Practicum (Student Reports on the Lab exercises)

Summary of the last lecture: How does Spin work?

- We already saw:
  - The Algorithm
  - The Promela Language
- We need to see how we does the tool work.

High Level Organization

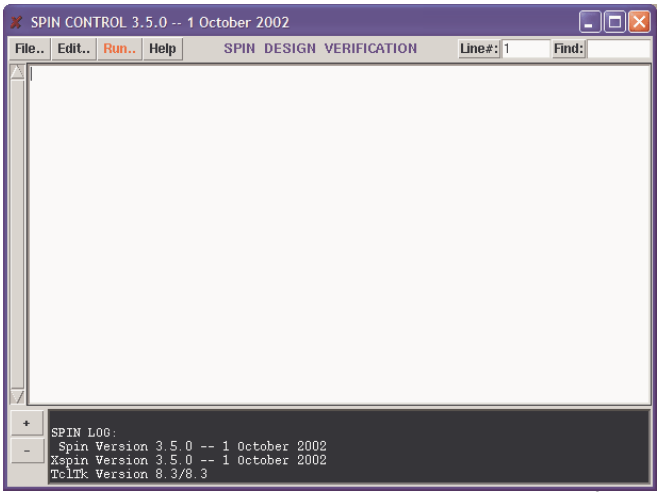


Command Line Tools

- Spin
  - Generates the Promela code for the LTL formula
    - ~$ spin –f “[]<>p”
    - The proposition in the formula must correspond to the model declarations
  - Generates the C source code
    - ~$ spin –a source.pro
    - The property must be included in the source
- Pan
  - Performs the verification
    - Has many compile time options to enable different features
    - Optimized for performance

Xspin

- GUI for Spin



Simulator

- Spin can also be used as a simulator
  - Simulated the Promela program
- It is used as a simulator when a counterexample is generated
  - Steps through the trace
  - The trace itself is not “readable”
- Can be used for random and manually guided simulation as well

Comments

- DFS does not necessarily find the shortest counterexample
- There might be a very short counterexample but the verification might go out of memory
- If we don’t finish we might still have some sort of a result (coverage metrics)

Today's Lecture

Advanced Techniques for Model Checking Software

(ComFoRT project)

Objectives

- C program and high-level designs verification
  - Sequential and concurrent
- Properties involve both data (states) and communication (events)
  - Specified as (State/Event) LTL formulas
  - Safety and Liveness
- Communication via shared actions
  - Synchronous communication
  - Asynchronous execution

Bluetooth L2CAP Spec

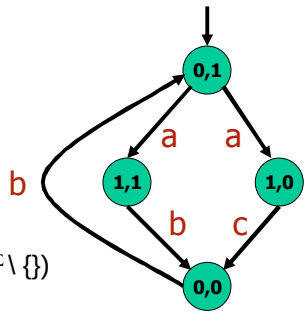
"When an L2CAP_ConnectRsp event is received in a W4_L2CAP_CONNECT_RSP state, an L2CAP process may send out an L2CA_ConnectInd event, disable the RTX timer, and move to state CONFIG."

…

Spec involves both states and events

Labelled Kripke Structures

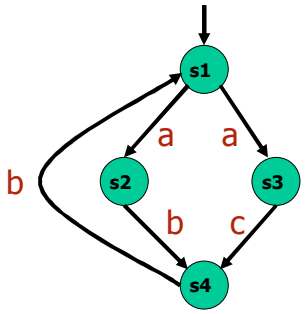
- Directed graph with labels on edges and states, (S,Init,P,L,T,Σ,E)
  - Every state is labeled with a set of atomic propositions, P, true in the state
  - Every LKS comes with an alphabet of actions, Σ



- State labeling function : L: S → 2^P
- Transition labeling function : E : T → ($2^\Sigma \setminus \{\}$)
  - Assumption: LKSs are deadlock-free
    [see deadlock detection algorithm, MEMOCODE'04]

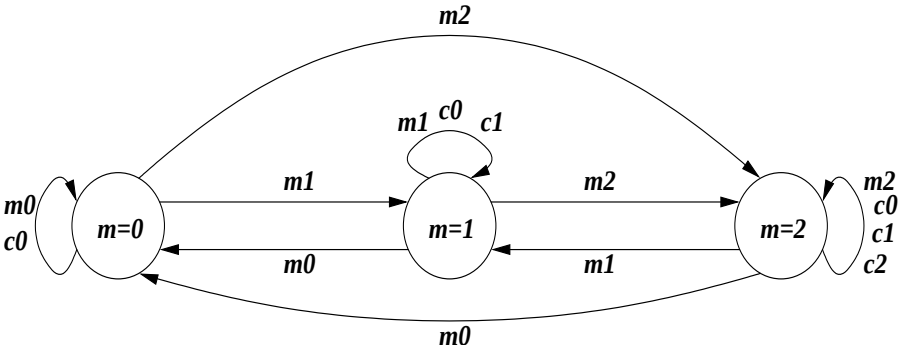
Traces and Languages

- Trace: infinite alternating sequence of states and actions
  - (s1 a s2 b s4 b s1…)
- Language: set of all traces
  - $\mathcal{L}(M) = \{s1\ a\ (s2\ b + s3\ c)\ s4\ b\}^{\omega}$



M

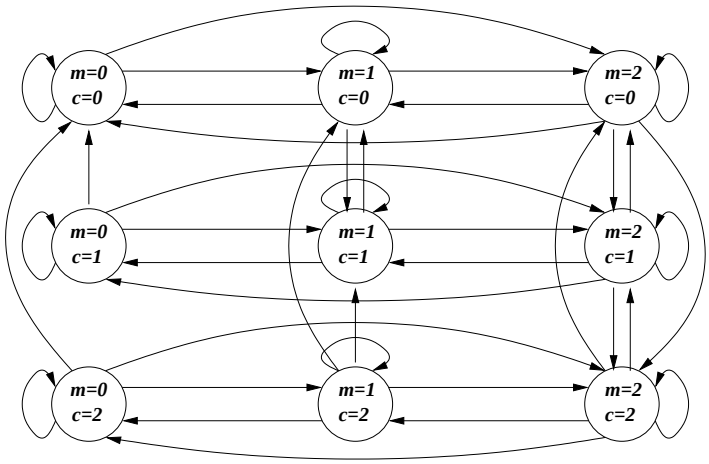
Surge Protector : State/Event



State/Event model of the Surge Protector

(example is given for m: [0..2], c: [0..2])

Surge Protector : State Only



Kripke structure of the Surge Protector
(example is given for m: [0..2], c: [0..2])

State/Event LTL

Given LKS M = (S,Init,P,L,T,Σ,E), and $p \in P$, $a \in \Sigma$,

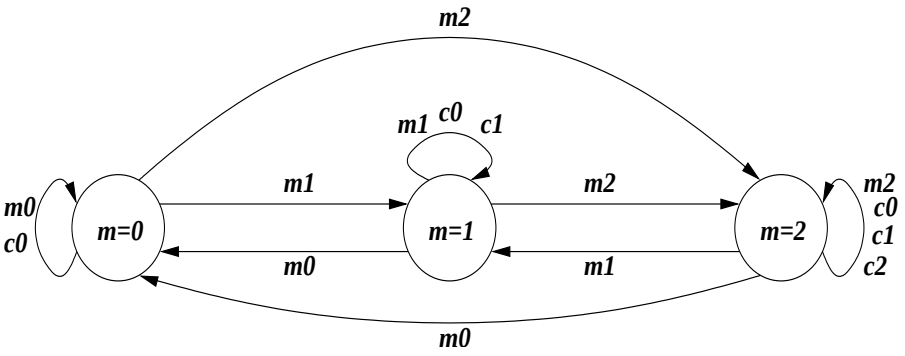
$$\varphi ::= p \mid a \mid \sim\varphi \mid \varphi \,\&\, \varphi \mid \mathbf{X}\varphi \mid \mathbf{G}\varphi \mid \mathbf{F}\varphi \mid \varphi\, \mathbf{U}\varphi$$

$\pi = (s1\ a1\ s2\ a2\ \ldots)$ is a path and π^i is the suffix of π starting at state s_i

$\pi \models p$ iff s_1 is the first state of π and $p \in L(s_1)$
$\pi \models a$ iff a is the first action of π
$\pi \models \sim\varphi$ iff $\sim(\pi \models \varphi)$
$\pi \models \mathbf{X}\varphi$ iff $\pi^2 \models \varphi$
$\pi \models \varphi_1\, \mathbf{U}\, \varphi_2$ iff there is some $i \geq 1$ such that $\pi^i \models \varphi_2$
and for all $1 \geq j \geq i-1$, $\pi^j \models \varphi_1$

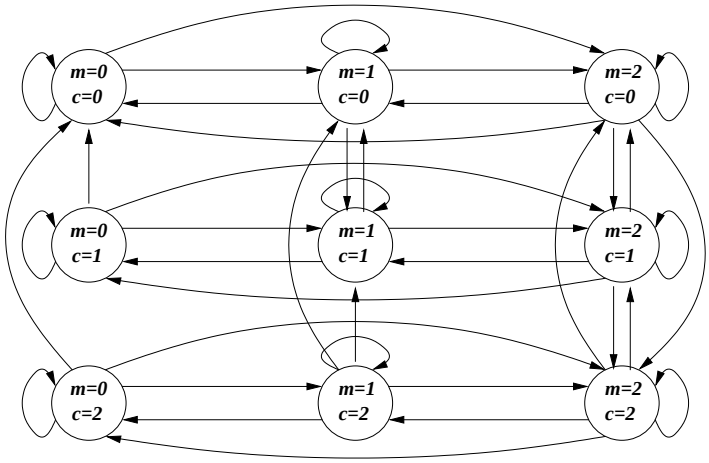
M $\models \varphi$ iff, for every path $\pi \in$ L(M), $\pi \models \varphi$

Surge Spec : State/Event



G ((c2 → m=2) & (c1 → (m=1 V m=2)))

Surge Spec : State Only



G (((c=0 V c=2) & **X** (c=1)) → (m=1 V m=2)) &
G (((c=0 V c=1) & **X** (c=2)) → m=2)

Surge Protector Verification

- Changes of current beyond threshold are disallowed
- State/Event Formula: **G** ((c2 → m=2) & (c1 → (m=1 V m=2)))
- State Formula: **G** (((c=0 V c=2) & **X** (c=1)) → (m=1 V m=2))
  & **G** (((c=0 V c=1) & **X** (c=2)) → m=2)
- Event Formula: **G** (m0 → ((~c1 **W** (m1 V m2))) &
  G (m0 → ((~c2 **W** m2)) &
  G (m1 → ((~c2 **W** m2))

Automata-based Verification

Given: M – LKS over Σ, P
φ – SE/LTL formula

How to check: M ⊨ φ

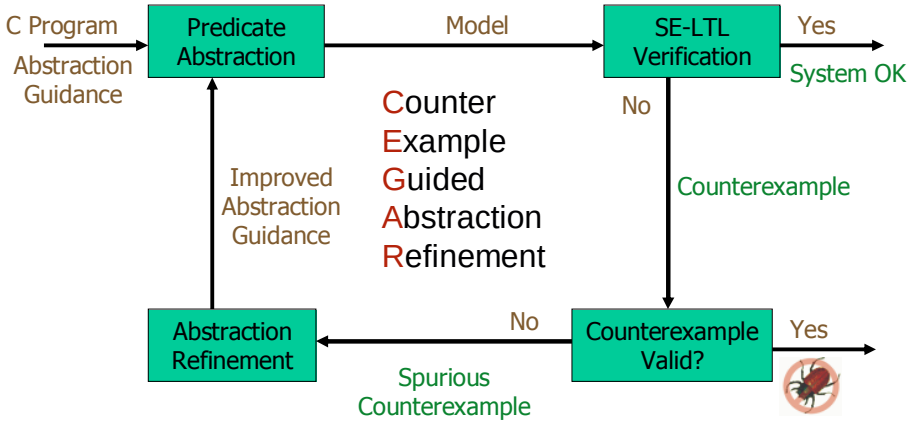
Possible Approach:
1. Convert M into a conventional state-only Kripke structure
2. Convert φ into a state-only LTL formula
3. Check whether M ⊨ φ

Inefficient!

What we do:
1. Interpret φ as an LTL formula over Σ U P
2. Compute B~φ (using Wring [Somenzi, Bloem'00])
3. Construct M ⊗ B~φ (result is a Buchi automaton)
4. Theorem: We have L (M ⊗ B~φ) = {} iff M ⊨ φ

No extra cost in time or space!

ComFoRT



Verification Results

Current range	State Formula				Event Formula				State/Event Formula			
	Aut. Size		Time		Aut. Size		Time		Aut. Size		Time	
	St	Tr	BC	MC	St	Tr	BC	MC	St	Tr	BC	MC
2	4	5	0.25	0.383	6	10	0.245	0.32	3	4	0.184	0.252
4	14	23	0.49	1.141	20	41	1.597	1.77	5	8	0.243	0.391
6	32	57	2.43	4.818	42	92	12.08	12.66	7	12	0.614	0.962
8	58	107	17.5	24.60	72	163	372.8	374.17	9	16	2.622	3.133
10	92	173	196	214.0	X	X	X	X	11	20	33.56	34.5
12	X	X	X	X	X	X	X	X	13	24	534.9	536.4
13	X	X	X	X	X	X	X	X	X	X	X	X

Parallel Composition

- Components synchronize on shared actions
  - proceed independently on local actions
- Propositions of components are disjoint (no shared variables)

Operational Semantics

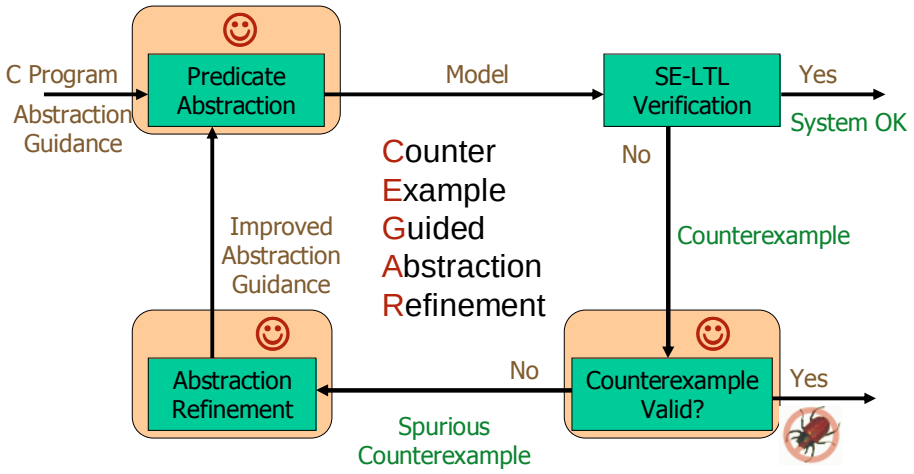
$$\frac{M_1 \xrightarrow{a} M_1' \qquad M_2 \xrightarrow{a} M_2'}{(M_1 \,||\, M_2) \xrightarrow{a} (M_1' \,||\, M_2')}$$

Synchronization on Shared action

$$\frac{M_1 \xrightarrow{a} M_1' \qquad a \notin \Sigma(M_2)}{(M_1 \,||\, M_2) \xrightarrow{a} (M_1' \,||\, M_2)}$$

Asynchronous Execution

Compositional Verification



Case Studies

- MicroC/OS-II
  - Real-time OS for embedded applications
  - Widely used (cell phones, medical devices, routers, washing machines…)
  - 6000+ LOC
  - Verified locking discipline
    - Locks and Unlocks alternate and locks are eventually released
  - Found four bugs
    - Missing unlock and return (three known – one unknown)

Results

Name	St-B	Tr-B	St-Mdl	T-BA	T-Mdl	T-Ver	T-Tot	Mem
SS	25	47	7951	0.690	48.8	6.84	56.9	39.3
SE	20	45	4331	0.497	18.8	2.92	22.8	24.2
SS	25	47	7574	0.699	43.6	1.65	46.4	38.1
SE	18	40	3691	0.407	15.3	1.089	17.3	21.2
SS	25	47	24.8 M	0.874	65.6	X	X	851
SE	20	45	13.6M	0.655	33.1	2.17	2207	162
SS	25	47	32.6M	0.836	66.0	X	X	347
SE	18	40	15.9M	0.713	34.6	4149	4185	321
BUG	8	14	873	0.205	3.41	0.261	3.88	X

Case Studies

- IPC Module
  - Deployed by a world leader in robotics engineering systems
  - 1500+ LOC
  - 4 components
  - Over 30 billion states after predicate abstraction
- Discovered synchronization bug in a matter of hours
  - Process can incorrectly block while writing to a queue
  - Undetected despite seven years of testing/industrial use

Case Studies

- Controller for a metal casting plant, used by Alcoa
  - ~30,000 LOC
  - Verified proper sequencing of various stages of casting:
    - Sequencing happens in prescribed order
    - The next stage eventually gets sequenced
- No bugs found… yet.

Key Contributions

- State/Event-based modeling and specification
- Efficient (direct) model checking algorithm
- CEGAR loop for software systems
  - Safety and liveness properties
- Compositional software verification
  - Component-wise abstractions
  - Component-wise counterexample validation
  - Component-wise refinements

Related Work

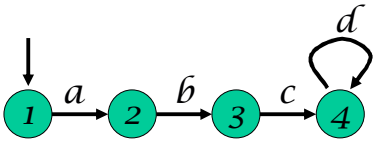
- Modal mu-calculus [Kozen 83]
- Doubly labeled transition systems [De Nicola and Vaandrager 95]
- CTL-based verification of doubly labeled transition systems [Gnesi et al. 96]
- State/event framework for three-valued logic verification [Huth et al. 01]
- SLAM [Ball et al. 00-…]
- BLAST [Henzinger et al. 02-…]

Deadlock Detection

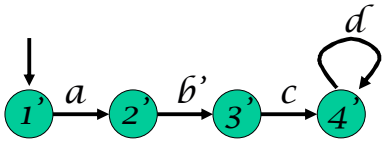
- Deadlock for concurrent blocking message-passing programs

- Need for an automated procedure

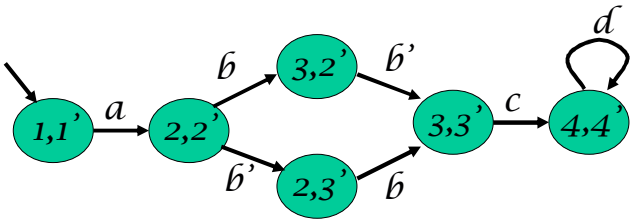
Example



$\mathcal{M}_1$ $\Sigma = \{a,b,c,d\}$



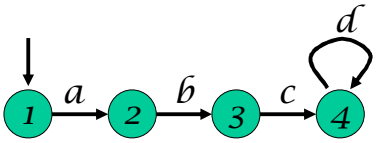
$\mathcal{M}_2$ $\Sigma = \{a,b',c,d\}$



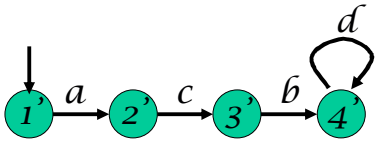
$\mathcal{M}_1 \parallel \mathcal{M}_2$

33

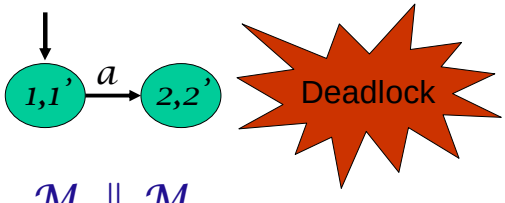
Deadlock



$\mathcal{M}_1$ $\Sigma = \{a,b,c,d\}$



$\mathcal{M}_2$ $\Sigma = \{a,b,c,d\}$



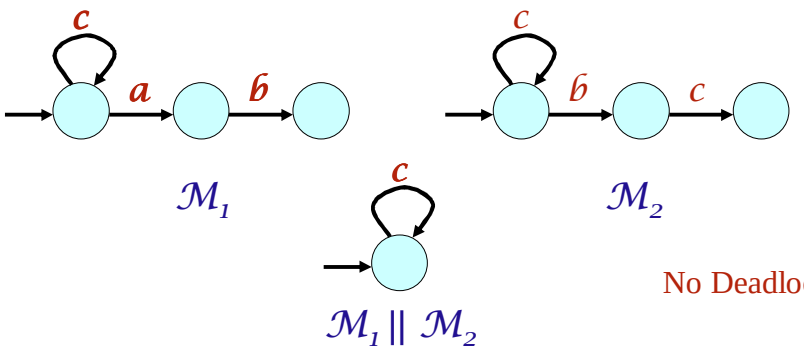
$\mathcal{M}_1 \parallel \mathcal{M}_2$

Deadlock ⇔ a reachable state cannot perform any actions at all

34

Deadlock and Composition

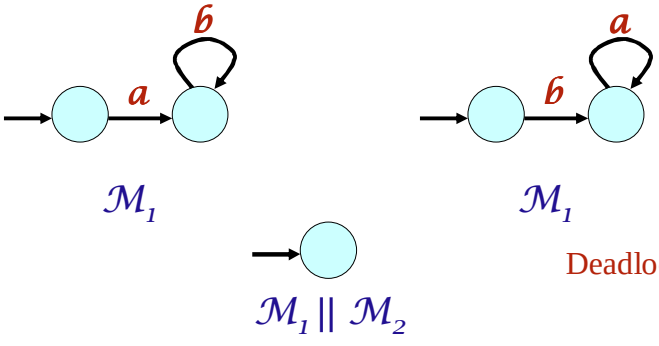
Deadlock



No Deadlock

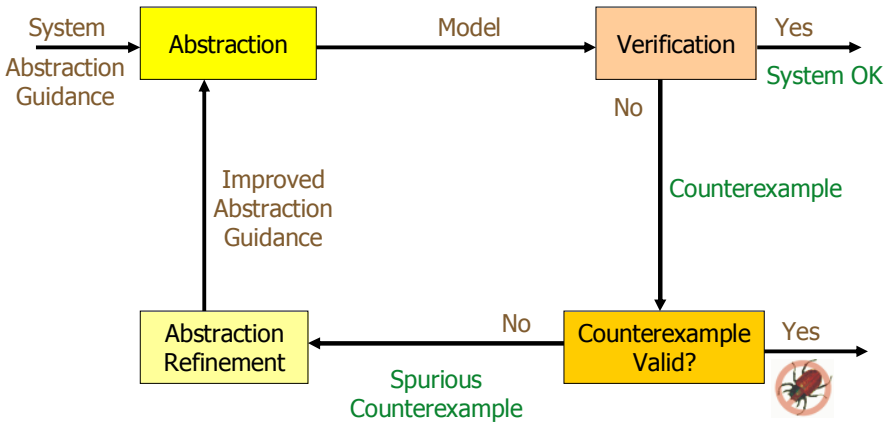
Deadlock and Composition

No Deadlock

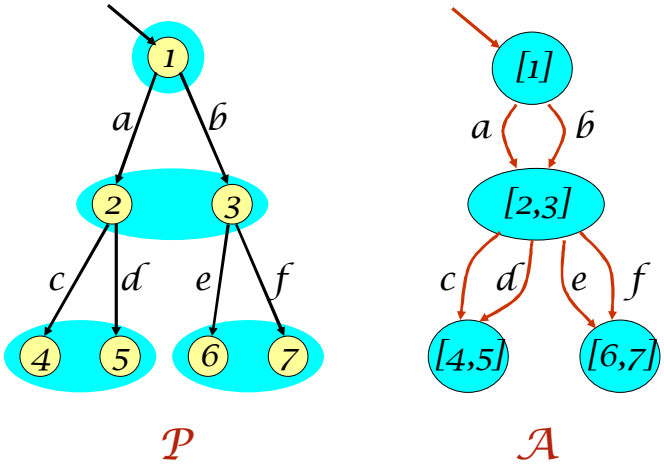


Deadlock

Iterative Refinement



Conservative Abstraction

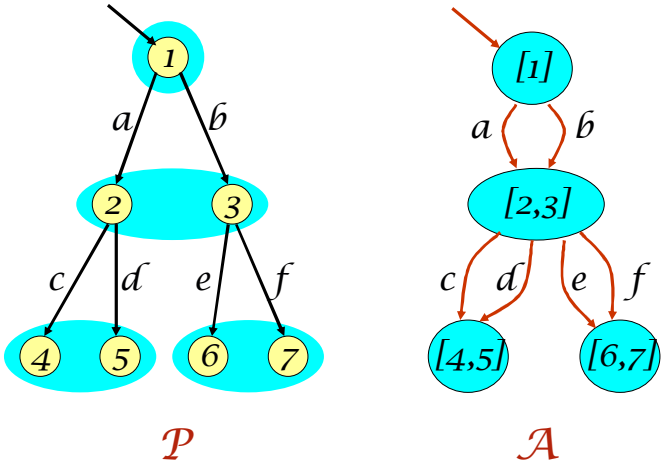


$\mathcal{P}$ $\mathcal{A}$

Conservative Abstraction

- Every trace of $\mathcal{P}$ is a trace of $\mathcal{A}$
  - Preserves safety properties: $\mathcal{A} \models \phi \Rightarrow \mathcal{P} \models \phi$
  - $\mathcal{A}$ over-approximates what $\mathcal{P}$ can do
- Some traces of $\mathcal{A}$ may not be traces of $\mathcal{P}$
  - May yield spurious counterexamples - $\langle a, e \rangle$
- Eliminated via abstraction refinement
  - Splitting some clusters in smaller ones
  - Refinement can be automated

Original Abstraction



$\mathcal{P}$ $\mathcal{A}$

Formal Verification by Model Checking

Natasha Sharygina
Carnegie Mellon University

Guest Lectures at the Analysis of Software Artifacts Class, Spring 2005

41

Outline

Lecture 1: Overview of Model Checking

Lecture 2: Complexity Reduction Techniques

Lecture 3: Software Model Checking

Lecture 4: State/Event-based software model checking

Lecture 5: Component Substitutability

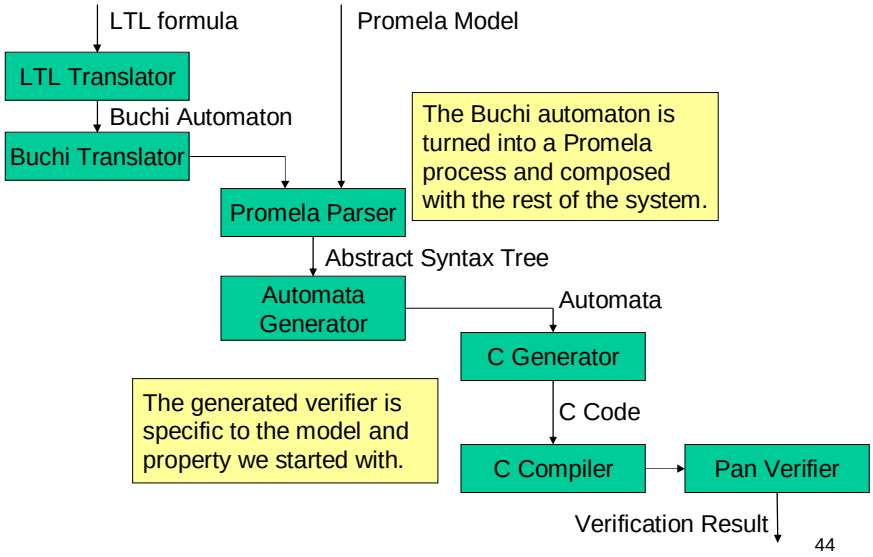
Lecture 6: Model Checking Practicum (Student Reports on the Lab exercises)

42

Summary of the last lecture: How does Spin work?

- We already saw:
  - The Algorithm
  - The Promela Language
- We need to see how we does the tool work.

High Level Organization



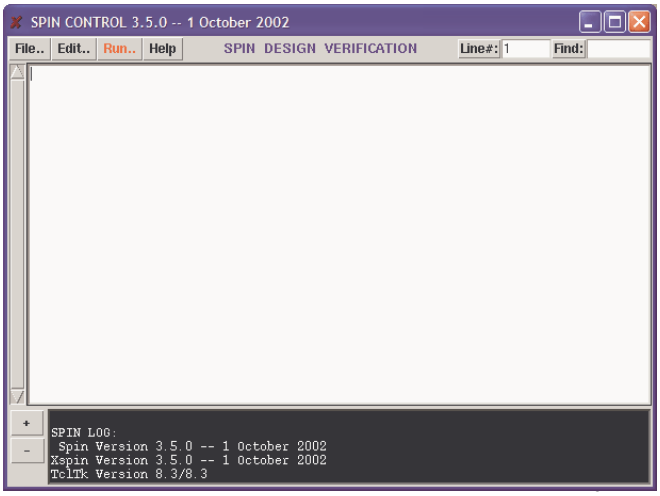
Command Line Tools

- Spin
  - Generates the Promela code for the LTL formula
    - ~$ spin –f “[]<>p”
    - The proposition in the formula must correspond to the model declarations
  - Generates the C source code
    - ~$ spin –a source.pro
    - The property must be included in the source
- Pan
  - Performs the verification
    - Has many compile time options to enable different features
    - Optimized for performance

45

Xspin

- GUI for Spin



Simulator

- Spin can also be used as a simulator
  - Simulated the Promela program
- It is used as a simulator when a counterexample is generated
  - Steps through the trace
  - The trace itself is not “readable”
- Can be used for random and manually guided simulation as well

Comments

- DFS does not necessarily find the shortest counterexample
- There might be a very short counterexample but the verification might go out of memory
- If we don’t finish we might still have some sort of a result (coverage metrics)

Today's Lecture

Advanced Techniques for Model Checking Software

(ComFoRT project)

Objectives

- C program and high-level designs verification
  - Sequential and concurrent
- Properties involve both data (states) and communication (events)
  - Specified as (State/Event) LTL formulas
  - Safety and Liveness
- Communication via shared actions
  - Synchronous communication
  - Asynchronous execution

Bluetooth L2CAP Spec

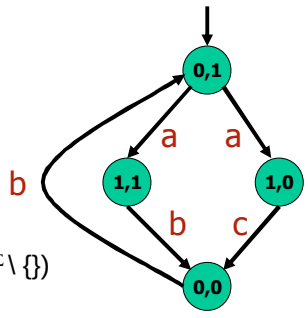
"When an L2CAP_ConnectRsp event is received in a W4_L2CAP_CONNECT_RSP state, an L2CAP process may send out an L2CA_ConnectInd event, disable the RTX timer, and move to state CONFIG."

…

Spec involves both states and events

Labelled Kripke Structures

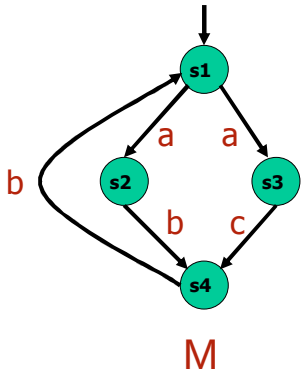
- Directed graph with labels on edges and states, (S,Init,P,L,T,Σ,E)
  - Every state is labeled with a set of atomic propositions, P, true in the state
  - Every LKS comes with an alphabet of actions, Σ



- State labeling function : L: S → 2^P
- Transition labeling function : E : T → ($2^\Sigma \setminus \{\}$)
  - Assumption: LKSs are deadlock-free
    [see deadlock detection algorithm, MEMOCODE'04]

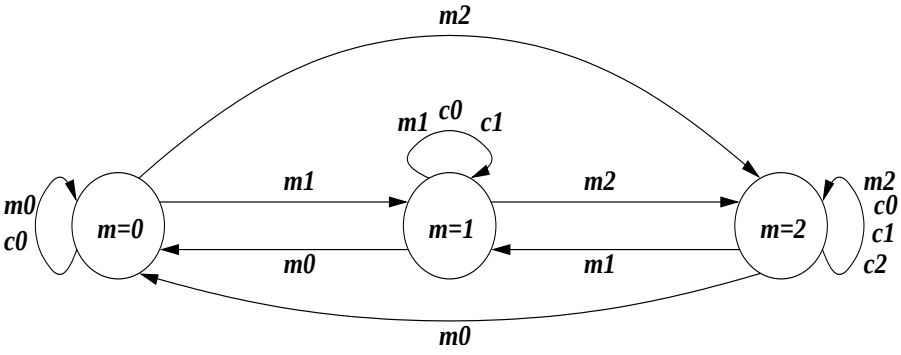
Traces and Languages

- Trace: infinite alternating sequence of states and actions
  - (s1 a s2 b s4 b s1…)
- Language: set of all traces
  - $\mathcal{L}(M) = \{s1\ a\ (s2\ b + s3\ c)\ s4\ b\}^{\omega}$



M

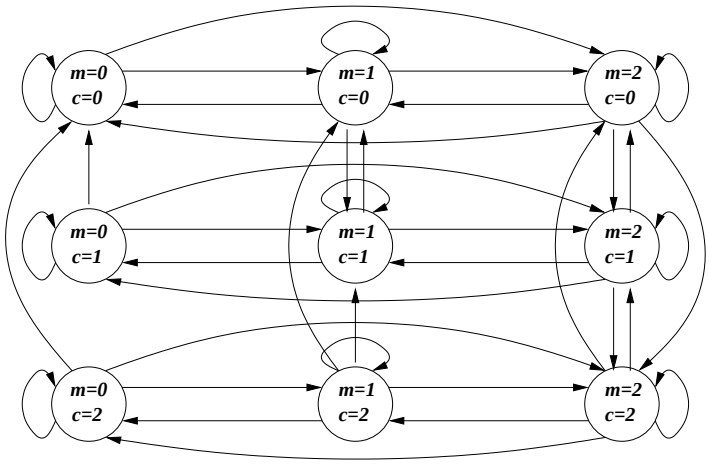
Surge Protector : State/Event



State/Event model of the Surge Protector

(example is given for m: [0..2], c: [0..2])

Surge Protector : State Only



Kripke structure of the Surge Protector
(example is given for m: [0..2], c: [0..2])

State/Event LTL

Given LKS M = (S,Init,P,L,T,Σ,E), and $p \in P$, $a \in \Sigma$,

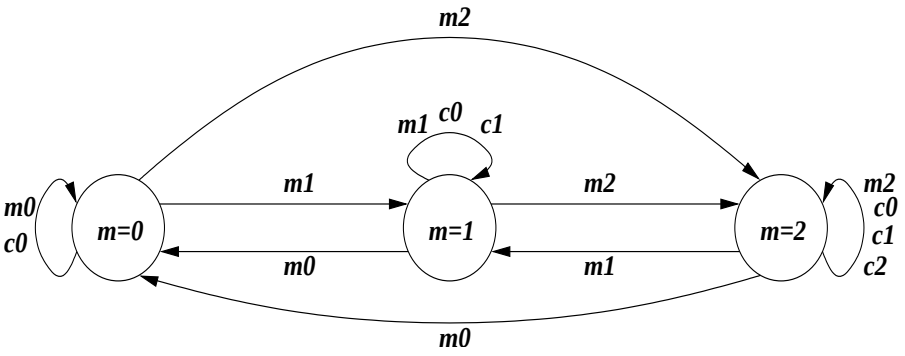
$$\varphi ::= p \mid a \mid \sim\varphi \mid \varphi \,\&\, \varphi \mid \mathbf{X}\varphi \mid \mathbf{G}\varphi \mid \mathbf{F}\varphi \mid \varphi\, \mathbf{U} \varphi$$

$\pi = (s1\ a1\ s2\ a2\ \ldots)$ is a path and π^i is the suffix of π starting at state s_i

$\pi \models p$ iff s_1 is the first state of π and $p \in L(s_1)$
$\pi \models a$ iff a is the first action of π
$\pi \models \sim\varphi$ iff $\sim(\pi \models \varphi)$
$\pi \models \mathbf{X}\varphi$ iff $\pi^2 \models \varphi$
$\pi \models \varphi_1\, \mathbf{U}\, \varphi_2$ iff there is some $i \geq 1$ such that $\pi^i \models \varphi_2$
and for all $1 \geq j \geq i - 1$, $\pi^j \models \varphi_1$

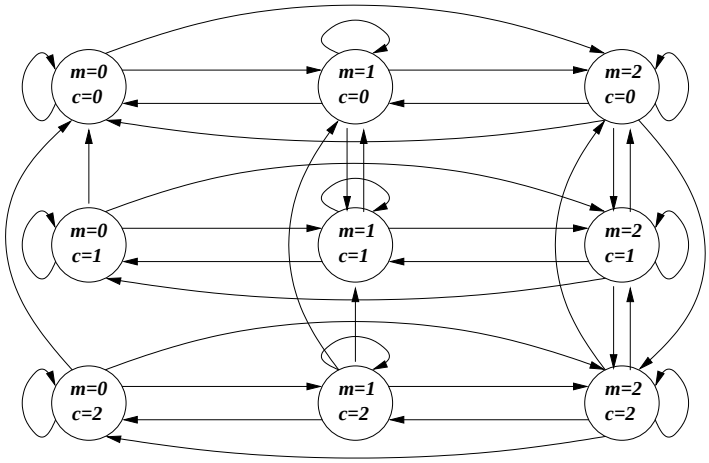
M $\models \varphi$ iff, for every path $\pi \in$ L(M), $\pi \models \varphi$

Surge Spec : State/Event



G ((c2 → m=2) & (c1 → (m=1 V m=2)))

Surge Spec : State Only



G (((c=0 V c=2) & **X** (c=1)) → (m=1 V m=2)) &
G (((c=0 V c=1) & **X** (c=2)) → m=2)

Surge Protector Verification

- Changes of current beyond threshold are disallowed
- State/Event Formula: **G** ((c2 → m=2) & (c1 → (m=1 V m=2)))
- State Formula: **G** (((c=0 V c=2) & **X** (c=1)) → (m=1 V m=2))
  & **G** (((c=0 V c=1) & **X** (c=2)) → m=2)
- Event Formula: **G** (m0 → ((~c1 **W** (m1 V m2))) &
  G (m0 → ((~c2 **W** m2)) &
  G (m1 → ((~c2 **W** m2))

Automata-based Verification

Given: M – LKS over Σ, P
φ – SE/LTL formula

How to check: M ⊨ φ

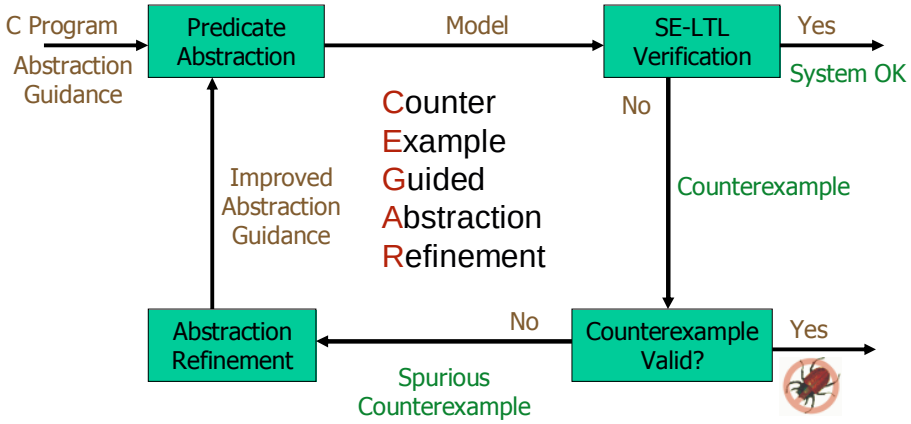
Possible Approach:
1. Convert M into a conventional state-only Kripke structure
2. Convert φ into a state-only LTL formula
3. Check whether M ⊨ φ

Inefficient!

What we do:
1. Interpret φ as an LTL formula over Σ U P
2. Compute B~φ (using Wring [Somenzi, Bloem'00])
3. Construct M ⊗ B~φ (result is a Buchi automaton)
4. Theorem: We have L (M ⊗ B~φ) = {} iff M ⊨ φ

No extra cost in time or space!

ComFoRT



Verification Results

Current range	State Formula				Event Formula				State/Event Formula			
	Aut. Size		Time		Aut. Size		Time		Aut. Size		Time	
	St	Tr	BC	MC	St	Tr	BC	MC	St	Tr	BC	MC
2	4	5	0.25	0.383	6	10	0.245	0.32	3	4	0.184	0.252
4	14	23	0.49	1.141	20	41	1.597	1.77	5	8	0.243	0.391
6	32	57	2.43	4.818	42	92	12.08	12.66	7	12	0.614	0.962
8	58	107	17.5	24.60	72	163	372.8	374.17	9	16	2.622	3.133
10	92	173	196	214.0	X	X	X	X	11	20	33.56	34.5
12	X	X	X	X	X	X	X	X	13	24	534.9	536.4
13	X	X	X	X	X	X	X	X	X	X	X	X

Parallel Composition

- Components synchronize on shared actions
  - proceed independently on local actions
- Propositions of components are disjoint (no shared variables)

Operational Semantics

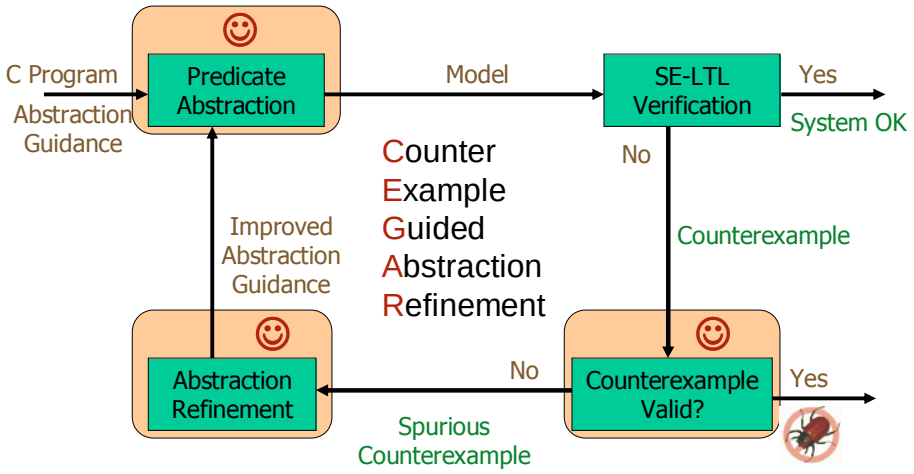
$$\frac{M_1 \xrightarrow{a} M_1' \quad M_2 \xrightarrow{a} M_2'}{(M_1 \,||\, M_2) \xrightarrow{a} (M_1' \,||\, M_2')}$$

Synchronization on Shared action

$$\frac{M_1 \xrightarrow{a} M_1' \quad a \notin \Sigma(M_2)}{(M_1 \,||\, M_2) \xrightarrow{a} (M_1' \,||\, M_2)}$$

Asynchronous Execution

Compositional Verification



Case Studies

- MicroC/OS-II
  - Real-time OS for embedded applications
  - Widely used (cell phones, medical devices, routers, washing machines…)
  - 6000+ LOC
  - Verified locking discipline
    - Locks and Unlocks alternate and locks are eventually released
  - Found four bugs
    - Missing unlock and return (three known – one unknown)

Results

Name	St-B	Tr-B	St-Mdl	T-BA	T-Mdl	T-Ver	T-Tot	Mem
SS	25	47	7951	0.690	48.8	6.84	56.9	39.3
SE	20	45	4331	0.497	18.8	2.92	22.8	24.2
SS	25	47	7574	0.699	43.6	1.65	46.4	38.1
SE	18	40	3691	0.407	15.3	1.089	17.3	21.2
SS	25	47	24.8 M	0.874	65.6	X	X	851
SE	20	45	13.6M	0.655	33.1	2.17	2207	162
SS	25	47	32.6M	0.836	66.0	X	X	347
SE	18	40	15.9M	0.713	34.6	4149	4185	321
BUG	8	14	873	0.205	3.41	0.261	3.88	X

Case Studies

- IPC Module
  - Deployed by a world leader in robotics engineering systems
  - 1500+ LOC
  - 4 components
  - Over 30 billion states after predicate abstraction
- Discovered synchronization bug in a matter of hours
  - Process can incorrectly block while writing to a queue
  - Undetected despite seven years of testing/industrial use

Case Studies

- Controller for a metal casting plant, used by Alcoa
  - ~30,000 LOC
  - Verified proper sequencing of various stages of casting:
    - Sequencing happens in prescribed order
    - The next stage eventually gets sequenced
- No bugs found… yet.

Key Contributions

- State/Event-based modeling and specification
- Efficient (direct) model checking algorithm
- CEGAR loop for software systems
  - Safety and liveness properties
- Compositional software verification
  - Component-wise abstractions
  - Component-wise counterexample validation
  - Component-wise refinements

Related Work

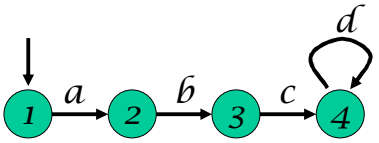
- Modal mu-calculus [Kozen 83]
- Doubly labeled transition systems [De Nicola and Vaandrager 95]
- CTL-based verification of doubly labeled transition systems [Gnesi et al. 96]
- State/event framework for three-valued logic verification [Huth et al. 01]
- SLAM [Ball et al. 00-…]
- BLAST [Henzinger et al. 02-…]

Deadlock Detection

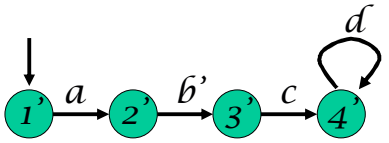
- Deadlock for concurrent blocking message-passing programs

- Need for an automated procedure

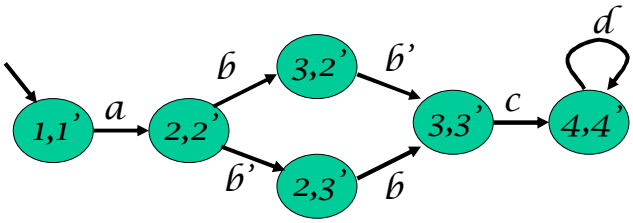
Example



$\mathcal{M}_1$ $\Sigma = \{a,b,c,d\}$

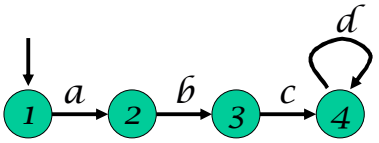


$\mathcal{M}_2$ $\Sigma = \{a,b',c,d\}$

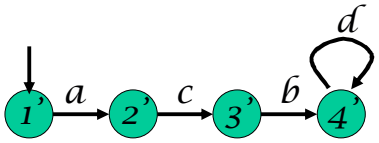


$\mathcal{M}_1 \parallel \mathcal{M}_2$

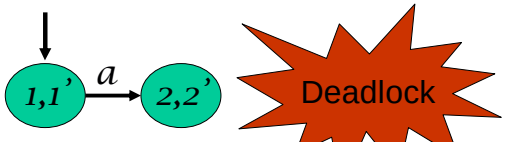
Deadlock



$\mathcal{M}_1$ $\Sigma = \{a,b,c,d\}$



$\mathcal{M}_2$ $\Sigma = \{a,b,c,d\}$

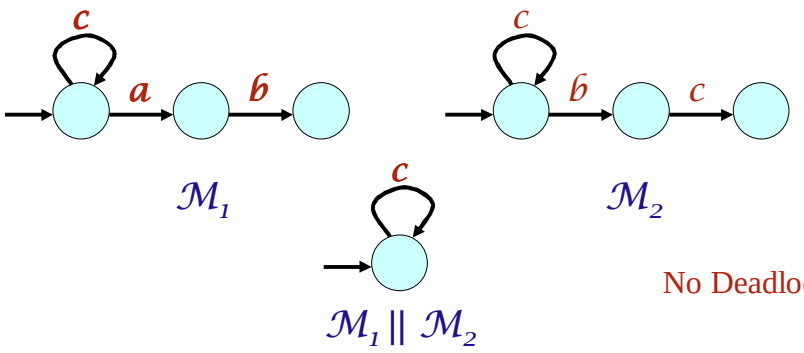


$\mathcal{M}_1 \parallel \mathcal{M}_2$

Deadlock $\Leftrightarrow$ a reachable state cannot perform any actions at all

Deadlock and Composition

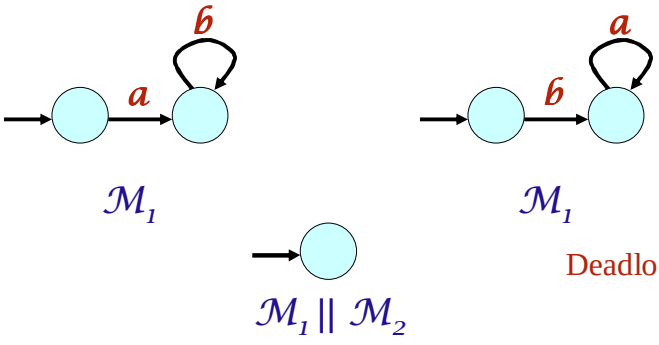
Deadlock



No Deadlock

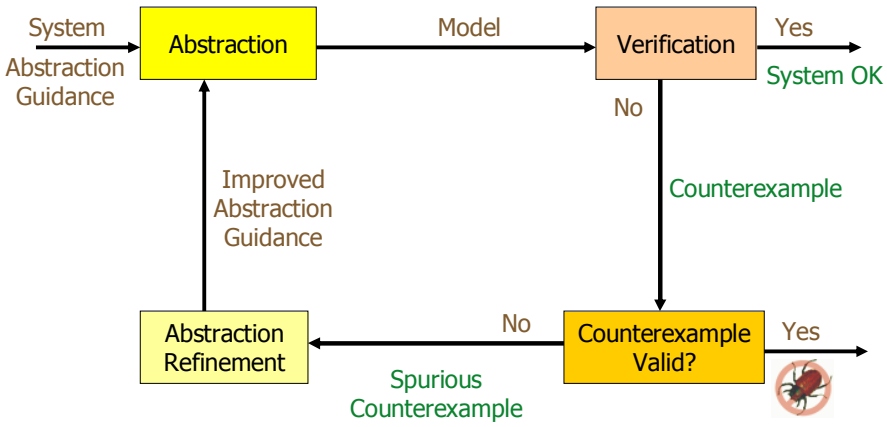
Deadlock and Composition

No Deadlock

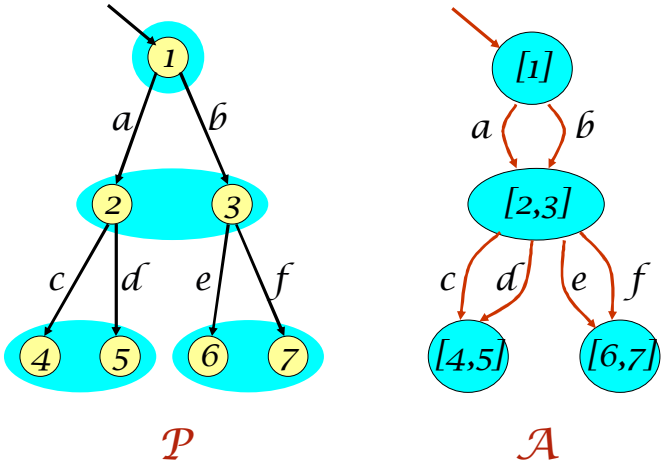


Deadlock

Iterative Refinement



Conservative Abstraction

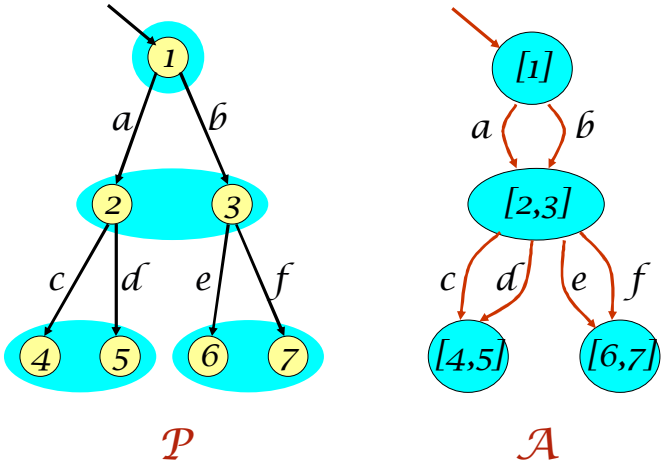


$\mathcal{P}$ $\mathcal{A}$

Conservative Abstraction

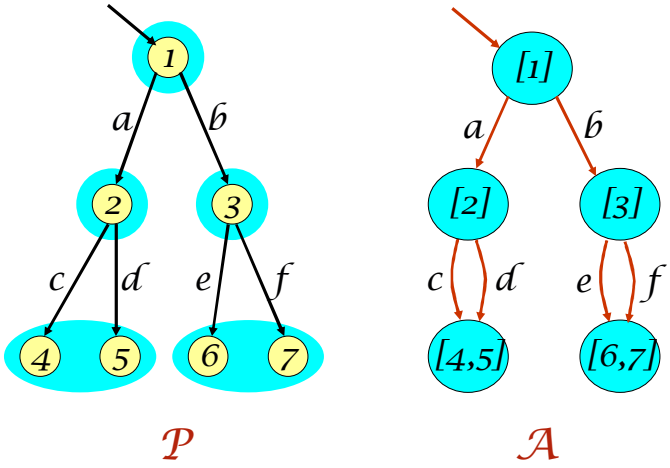
- Every trace of $\mathcal{P}$ is a trace of $\mathcal{A}$
  - Preserves safety properties: $\mathcal{A} \models \phi \Rightarrow \mathcal{P} \models \phi$
  - $\mathcal{A}$ over-approximates what $\mathcal{P}$ can do
- Some traces of $\mathcal{A}$ may not be traces of $\mathcal{P}$
  - May yield spurious counterexamples - $\langle a, e \rangle$
- Eliminated via abstraction refinement
  - Splitting some clusters in smaller ones
  - Refinement can be automated

Original Abstraction



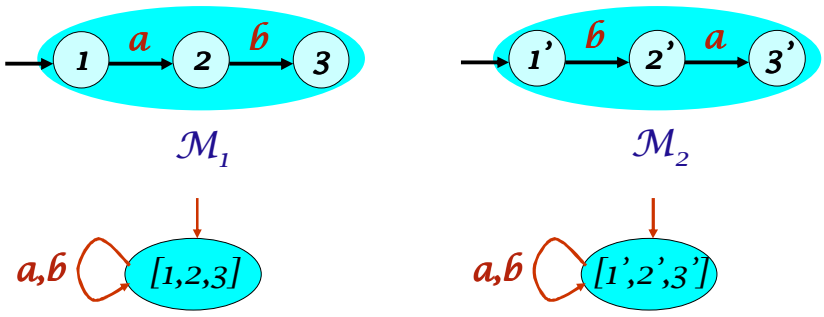
$\mathcal{P}$ $\mathcal{A}$

Refined Abstraction



Deadlock : Problem

- Deadlock is not preserved by abstraction



Deadlock Detection : Insight

- Deadlock ⇔ a reachable state cannot perform any actions at all
  - Deadlock depends on the set of actions that a reachable state cannot perform
- In order to preserve deadlock $\mathcal{A}$ must over-approximate not just what $\mathcal{P}$ can do but also what $\mathcal{P}$ refuses

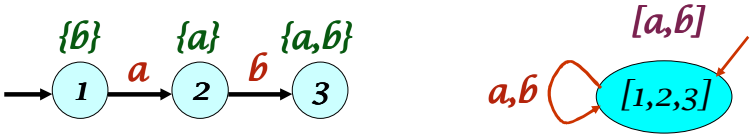
Refusal & Deadlock

- $\mathcal{R}ef(s)$ = set of actions s cannot perform

- $\mathcal{M}$ deadlocks iff there is a reachable state s such that $\mathcal{R}ef(s) = \Sigma$
  - Denote by $\mathcal{DL}ock(\mathcal{M})$
- $\mathcal{R}ef([s_1 .. s_n]) = \mathcal{R}ef(s_1) \cap .. \cap \mathcal{R}ef(s_n)$

Abstract Refusal

- $\mathcal{AR}([s_1 .. s_n]) = \mathcal{R}ef(s_1) \cup .. \cup \mathcal{R}ef(s_n)$



- $\mathcal{AR}([M_1] .. [M_n]) = \mathcal{AR}([M_1]) \cup .. \cup \mathcal{AR}([M_n])$

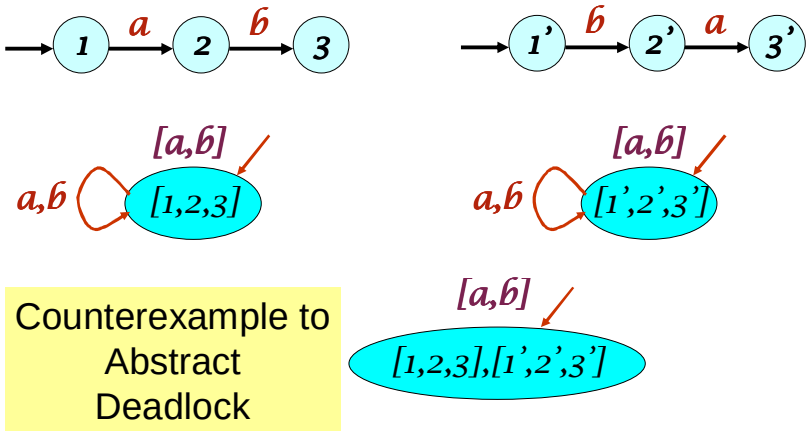
85

Abstract Deadlock

- $\mathcal{M}$ abstractly deadlocks iff there is a reachable state s such that $\mathcal{AR}(s) = \Sigma$
  - Denote by $\mathcal{ADL}ock(\mathcal{M})$

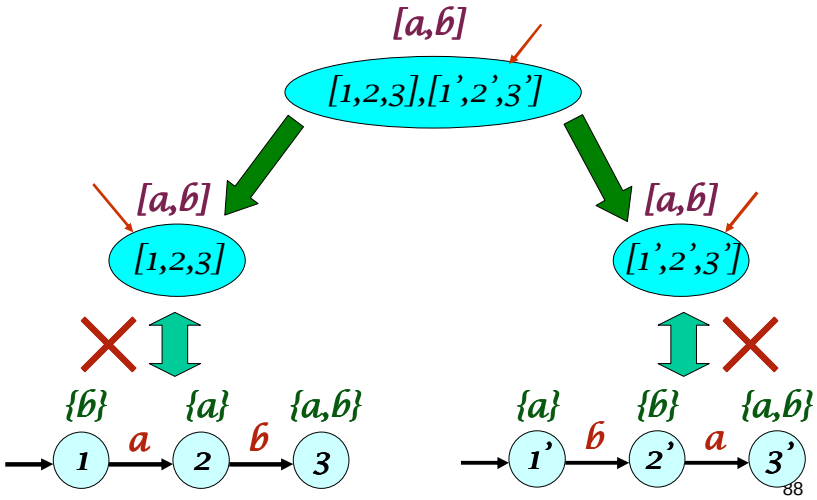
$$\neg\ \mathcal{ADL}ock([M_1] \parallel .. \parallel [M_n]) \Rightarrow \neg\ \mathcal{DL}ock(M_1 \parallel .. \parallel M_n)$$

Iterative Deadlock Detection

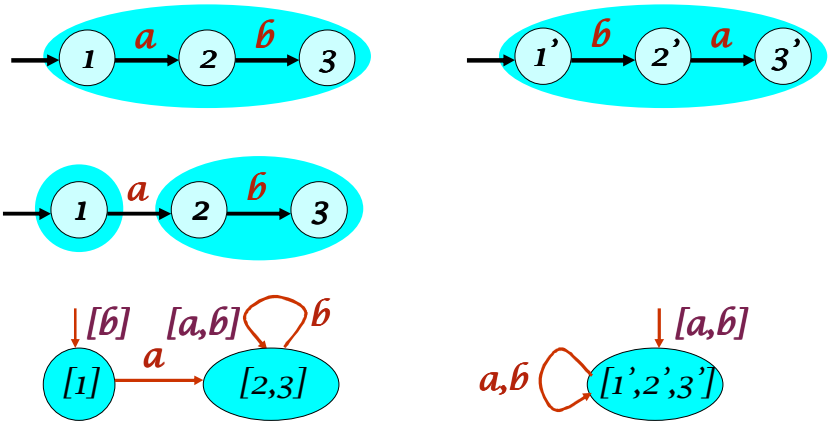


Counterexample to Abstract Deadlock

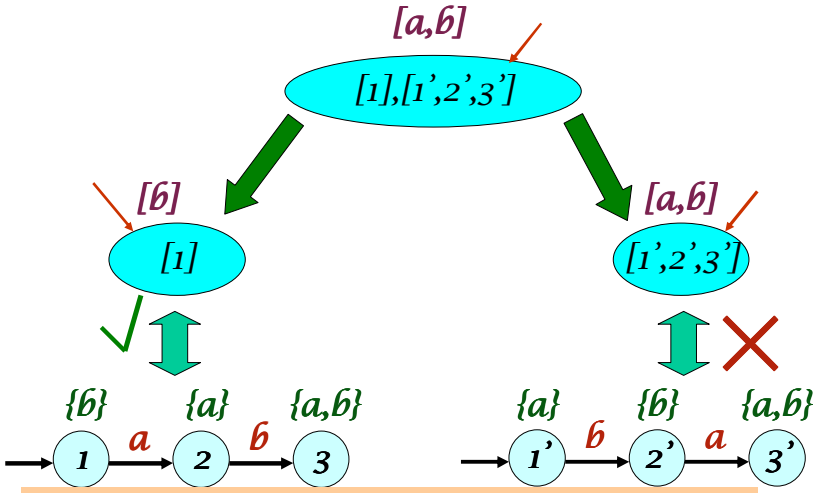
Counterexample Validation



Refinement

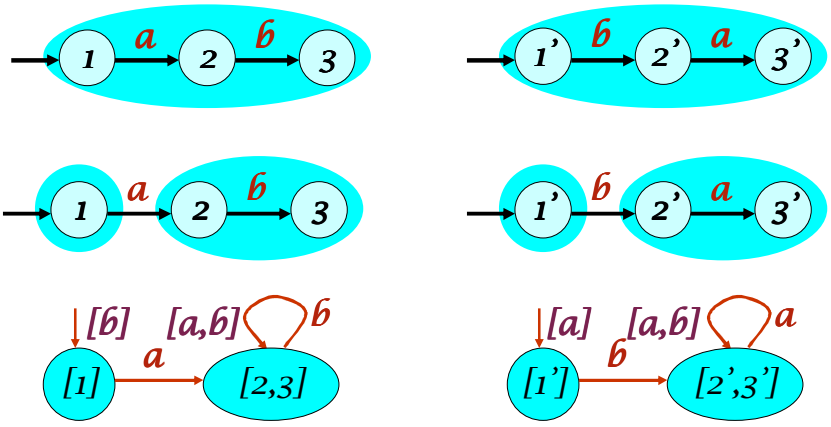


Counterexample Validation

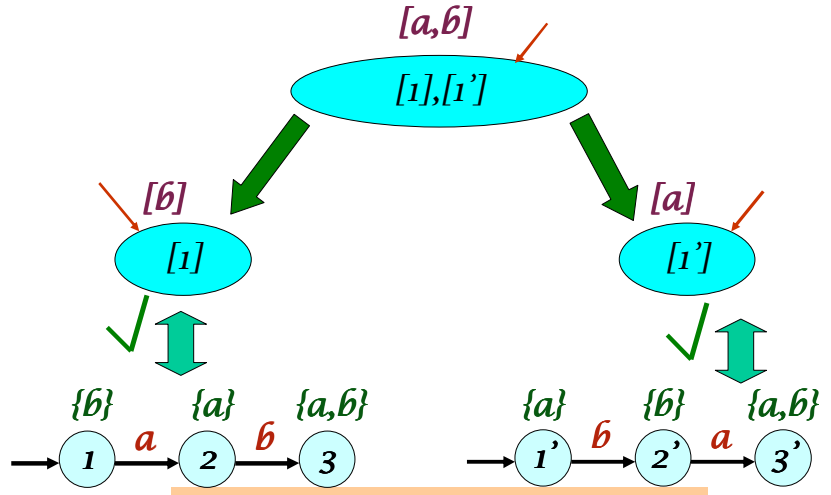


Another spurious counterexample

Refinement

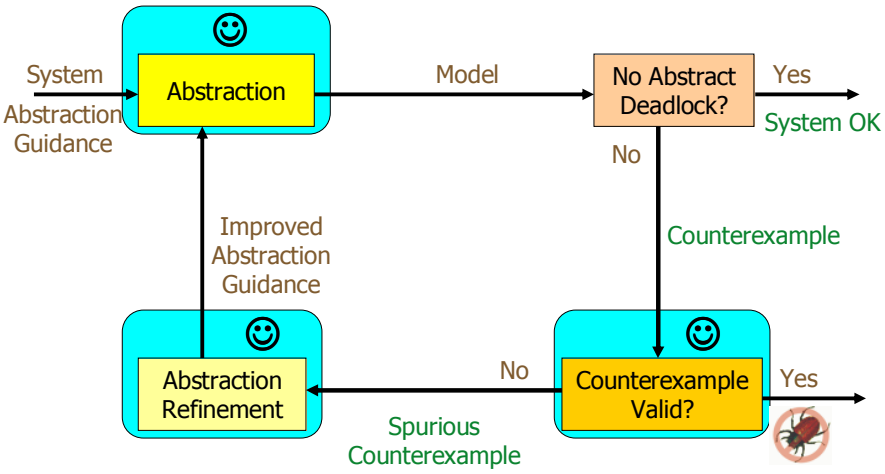


Counterexample Validation



Real Deadlock Detected

Iterative Deadlock



Case Studies

- MicroC/OS-II
  - Real-time OS for embedded applications
  - Widely used (cell phones, medical devices, routers, washing machines…)
  - 6000+ LOC
  - Verified locking discipline
    - Locks and Unlocks alternate and locks are eventually released
  - Found four bugs
    - Missing unlock and return (one bug - deadlock)

Case Studies

- ABB IPC Module
  - Deployed by a world leader in robotics
  - 15000+ LOC
  - 4 components
  - Over 30 billion states after predicate abstraction
- Discovered synchronization bug in a matter of hours
  - Process can incorrectly block while writing to a queue
  - Undetected despite seven years of testing/industrial use

Results

	Plain			IterDeadlock			
Name	St	T	Mem	St	It	T	Mem
ABB	*	*	162	1973	861	1446	33.3
SSL	25731	44	43.5	16	16	31.9	40.8
µCD-3	*	*	58.6	4930	120	221.8	15
µCN-6	*	*	219.3	71875	44	813	30.8
DPN-6	*	*	203	62426	48	831	26.1
DPD-10	38268	87.6	17.3	44493	51	755	18.4

* indicates out of time limit (1500s)

Ongoing and Future Work

- Shared memory
- Assume-Guarantee reasoning
- Industrial size examples
- Symbolic implementation
- Branching-time state/event logic (completed)

Lab Assignment

- Spit into groups of 4-5 people
- Design, implementation and verification of the current surge protector
  - In PROMELA/SPIN
  - In ComFoRT
- Comparative validation
- Presentations on March 31, 2005

Lab Assignment (2)

- Questions about ComFoRT
  - Natasha Sharygina: nys@sei.cmu.edu - *theory*
  - Sagar Chaki: chaki@sei.cmu.edu – *tool support*

Collaboration Opportunities

- Research and development projects on verification of software (ComFoRT project)
- As part of the PACC (Predictable Assembly from Certifiable Components) project at the SEI
- Joint work with Prof. Ed Clarke

Collaboration Opportunities

- Independent studies
- M.S. and Ph.D. Research (jointly with your current advisors)
- Internships

If interested contact me and we can discuss options