Checking Safety Properties of Concurrent Programs

Huimin Lin

joint work with Yi Lv, Hong Pan, Peng Wu

Institute of Software, Chinese Academy of Sciences

SERE 2012 June 22, 2012

Concurrent software systems

- Safety-critical
  - Nuclear reactor control
  - Medical treatment systems
  - Train control systems
  - ...

- Complicated behaviors
  - Complexity grows exponentially in the number of components

- Safety properties are difficult to ensure

Concurrent programs vs. Sequential programs

Sequential programs: Input ⟹ Output

* Terminating

* Deterministic: same input, same output

* Failure reproducible: outcome of test is meaningful

Concurrent programs: Interacting with environment

* Nonterminating

* Nondeterministic: same initial configuration, different execution traces

* Failure not reproducible: outcome of test is meaningless

Model checking [Clarke & Emerson & Sifakis, 1981]

Automatically verify if a concurrent system has desired behaviors

A concurrent system is modeled as a transition system M

A property is specified as a formula ϕ in some temporal logic (CTL, LTL, μ-calculus ...)

For finite state systems, it is decidable if $M \models \phi$

– in case the answer is “No”, a counter-example (an execution trace) can be generated leading to the faulty state

Traditionally, model checking research tends to focus on “system skeletons” in which data aspects are largely ignored

– message-exchanging are simplified to synchronization on signals

– temporal logics (CTL, LTL, μ-calculus ...) are propositional

However, ignoring data aspects in concurrent software-rich systems is dangerous: systems with serious safety bugs may get model checked

The purpose of this talk: to introduce a methodology for model checking concurrent programs, where data are treated as “first-class citizens”

Outline

1. Symbolic transition graphs

2. First-order μ-calculus

3. A local model checking algorithm

4. Case studies

5. Summary and future directions

Symbolic Transition Graphs (STGs): A model for concurrent programs

Each note n is associated with a set of (free) variables $fv(n)$

Edges are of the form $n \overset{b,\ \overline{x}:=\overline{e},\ \alpha}{\longmapsto} m$

b: Boolean expression

x: data variable

e: data expression

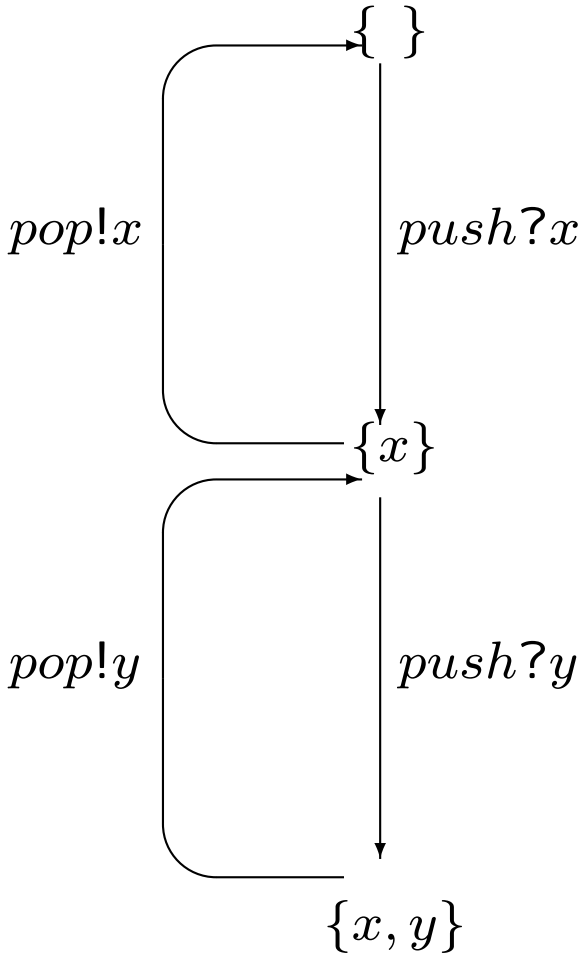
α: communication action

$$\begin{array}{rl} c?x & \text{input} \\ c!e & \text{output} \\ \tau & \text{internal communication} \end{array}$$

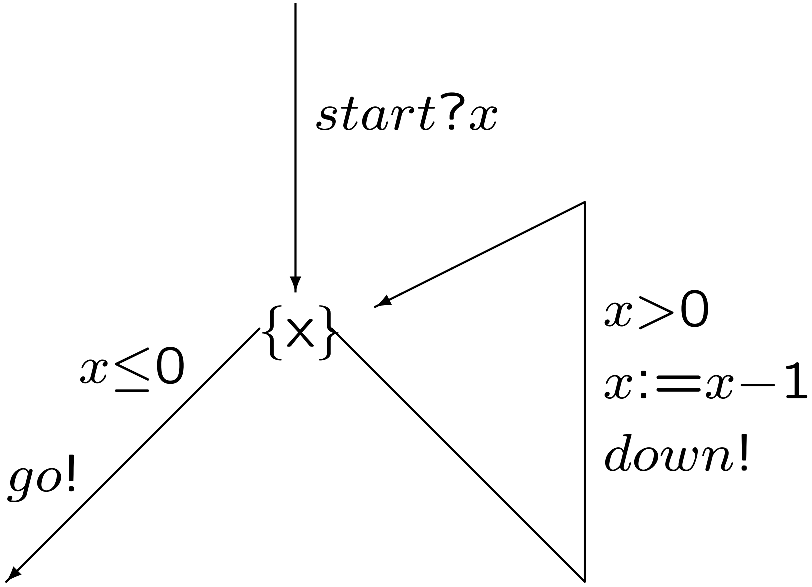
Satisfying some consistency constraints:

$$\begin{array}{l} fv(b,\overline{e}) \subseteq fv(n) \\ fv(\alpha) \subseteq \{\overline{x}\} \\ fv(m) \subseteq \{\overline{x}\} \cup bv(\alpha) \end{array}$$

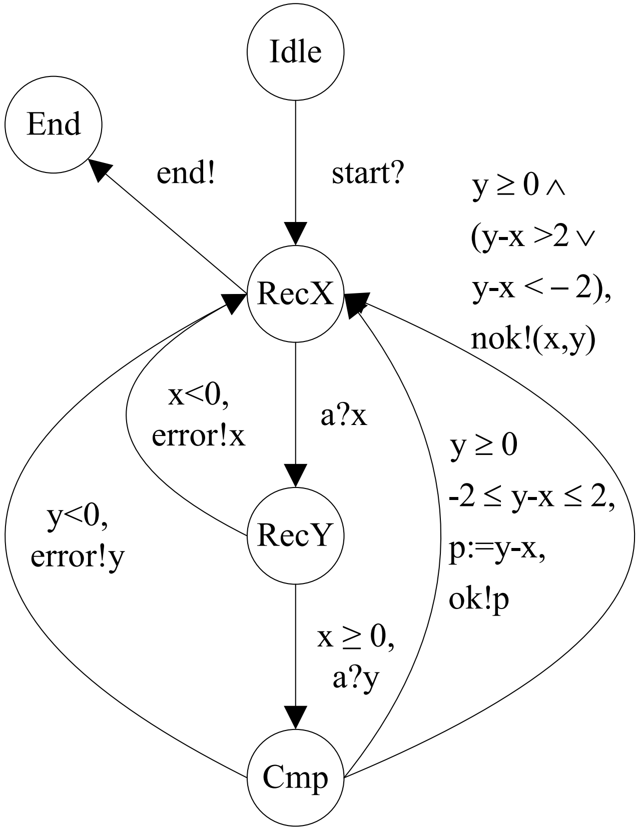
Example: Stack of capacity 2



Example: Counting machine



Example


Idle
End
end!
start?
y ≥ 0 ∧
(y-x >2 ∨
y-x < – 2),
nok!(x,y)
RecX
x<0,
error!x
a?x
y ≥ 0
-2 ≤ y-x ≤ 2,
p:=y-x,
ok!p
y<0,
error!y
RecY
x ≥ 0,
a?y
Cmp

Compared with flow-chart programs

No conditional nodes: guards are carried on edges

Guards on different out-going edges of the same node may be true at the same time: nondeterminism

Parallel composition (with channel restriction)

$(\mathcal{G}\|\mathcal{H})\lceil_R$: product of $\mathcal{G}$ and $\mathcal{H}$ with channel names in R restricted

$$\frac{n \overset{b,\theta,\alpha}{\longmapsto} n'}{< n, m > \overset{b,\theta\cup\{\overline{y}:=\overline{y}\},\alpha}{\longmapsto} < n', m >} \quad \begin{array}{l} chan(\alpha)\cap R=\emptyset \\ fv(m) = \{\overline{y}\} \end{array}$$

$$\frac{m \overset{b,\theta,\alpha}{\longmapsto} m'}{< n, m > \overset{b,\theta\cup\{\overline{y}:=\overline{y}\},\alpha}{\longmapsto} < n, m' >} \quad \begin{array}{l} chan(\alpha)\cap R=\emptyset \\ fv(n) = \{\overline{y}\} \end{array}$$

$$\frac{n \overset{b_1,\theta_1,c?x}{\longmapsto} n', \qquad m \overset{b_2,\theta_2,c!e}{\longmapsto} m'}{< n, m > \overset{b_1\wedge b_2,\theta_1\cup\theta_2\cup\{x:=e\theta_2\},\tau}{\longmapsto} < n', m' >}$$

Labeled transition systems induced from STGs

State: $(m,\ \rho)$ (m an STG node, $\rho : \mathit{Var} \to \mathit{Val}$)

$$\frac{m \overset{b,\overline{x}:=\overline{e},\tau}{\longmapsto} n}{(m,\rho) \xrightarrow{\tau} (n, \rho\{\overline{x} \mapsto \rho(\overline{e})\})} \quad \rho \models b$$

$$\frac{m \overset{b,\overline{x}:=\overline{e},c!e'}{\longmapsto} n}{(m,\rho) \overset{c!\rho(e'[\overline{e}/\overline{x}])}{\longrightarrow} (n, \rho\{\overline{x} \mapsto \rho(\overline{e})\})} \quad \rho \models b$$

$$\frac{m \overset{b,\overline{x}:=\overline{e},c?y}{\longmapsto} n}{(m,\rho) \xrightarrow{c?y} (n, \rho\{\overline{x} \mapsto \rho(\overline{e})\})} \quad \rho \models b$$

First-order μ-Calculus (finite part)

Syntax

$$\phi \;::=\; p \quad | \quad \phi \wedge \psi \quad | \quad \phi \vee \psi \quad | \quad \forall x \phi \quad | \quad \exists x \phi \quad |$$

$$\langle \alpha \rangle \phi \quad | \quad [\alpha]\phi$$

Modalities: $\langle c!e \rangle$, $[c!e]$, $\langle c?x \rangle$, $[c?x]$

Semantics

Formulas are interpreted over labeled transition systems

$(m, \rho) \models \langle c!e \rangle \phi$ if for some (n, ρ') $s.t.$ $(m, \rho) \xrightarrow{c!\rho(e)} (n, \rho')$, $(n, \rho') \models \phi$

$(m, \rho) \models [c!e] \phi$ if for all (n, ρ') $s.t.$ $(m, \rho) \xrightarrow{c!\rho(e)} (n, \rho')$, $(n, \rho') \models \phi$

$(m, \rho) \models \langle c?x \rangle \phi$ if for some (n, ρ') $s.t.$ $(m, \rho) \xrightarrow{c?y} (n, \rho')$,
$(n, \rho'\{y \mapsto v\}) \models \phi[v/x]$ for all $v \in \mathit{Val}$

$(m, \rho) \models \langle c?x \rangle \phi$ if for all (n, ρ') $s.t.$ $(m, \rho) \xrightarrow{c?y} (n, \rho')$,
$(n, \rho'\{y \mapsto v\}) \models \phi[v/x]$ for all $v \in \mathit{Val}$

...

The logic defined so far is finite: we can say “a process will send m along channel c within three steps of computation”, by writing

$$\langle c!m\rangle \mathit{true} \vee \langle -\rangle\langle c!m\rangle \mathit{true} \vee \langle -\rangle\langle -\rangle\langle c!m\rangle \mathit{true}$$

But we cannot say “a process will eventually send m along channel c”.

The standard approach to describing infinite behaviour is to use *fixpoint* formulas. For instance, in propositional μ-calculus, the above property can be written as a least fixpoint formula:

$$\mu Z.\langle c!m\rangle \mathit{true} \vee \langle -\rangle Z$$

By fixpoint unwinding, this intuitively means

$$\langle c!m\rangle \mathit{true} \vee \langle -\rangle(\langle c!m\rangle \mathit{true} \vee \langle -\rangle(\langle c!m\rangle \mathit{true} \vee \langle -\rangle(\langle c!m\rangle \mathit{true} \vee \cdots)))$$

However, for first-order μ-calculus, there is a potential problem.

Consider the formula

$$A = \exists x.\mu Z.\langle c!x\rangle \mathit{true} \vee \exists x.\langle -\rangle Z$$

which is equivalent to

$$B = \exists x.\mu Z.\langle c!x\rangle \mathit{true} \vee \exists y.\langle -\rangle Z$$

But unwinding them will result in semantically different formulas:

$$A \equiv \exists x.\langle c!x\rangle \mathit{true} \vee \exists x.\langle -\rangle(\langle c!x\rangle \mathit{true} \vee \cdots)$$

is not equivalent to

$$B \equiv \exists x.\langle c!x\rangle \mathit{true} \vee \exists y.\langle -\rangle(\langle c!x\rangle \mathit{true} \vee \cdots)$$

Because in the first formula the out-most quantifier binds only the x in the first $\langle c!x\rangle$ subformula of A, while in the second formula the out-most quantifier binds *all* occurrences of x in the body of B.

The problem lies in the fact that the (propositional) variable Z in the body of the fixpoint formula

$$\mu Z.\langle c!x\rangle \mathit{true} \vee \exists x.\langle - \rangle Z$$

stands for the entire formula which has x free. But this cannot be seen from the syntax.

First-order μ-Calculus (fixpoints)

Predicates: functions from data expressions to propositions

Predicate variables: $X, Y \ldots$, each has an *arity*

$$
\begin{array}{lcl}
\Lambda & ::= & (\overline{x})\phi \quad | \quad X \quad | \quad \mu X.\Lambda \quad | \quad \nu X.\Lambda \\
\phi & ::= & p \quad | \quad \phi \wedge \psi \quad | \quad \phi \vee \psi \quad | \quad \forall x \phi \quad | \quad \exists x \phi \quad | \\
 & & \langle \alpha \rangle \phi \quad | \quad [\alpha]\phi \quad | \quad \Lambda(\overline{e})
\end{array}
$$

In $\mu X.\Lambda$ and $\nu X.\Lambda$, X is bound with scope Λ

Now the formula A can be written as

$$(\mu Y.(x)(\langle c!x\rangle true \vee \exists x.\langle - \rangle Z(x))(x)$$

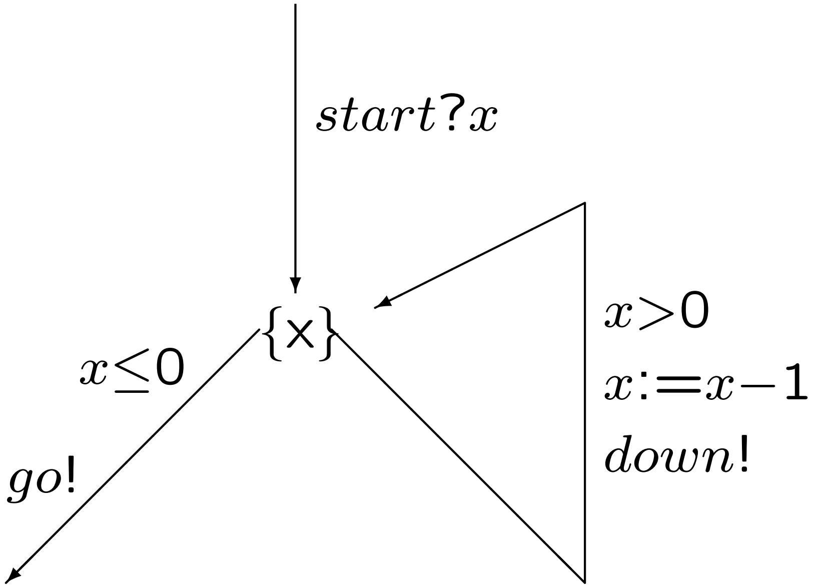
Changing bound variable x to y will result in

$$(\mu Y.(x)(\langle c!x\rangle true \vee \exists y.\langle - \rangle Z(y))(x)$$

which is equivalent to the original formula.

Understanding fixpoints

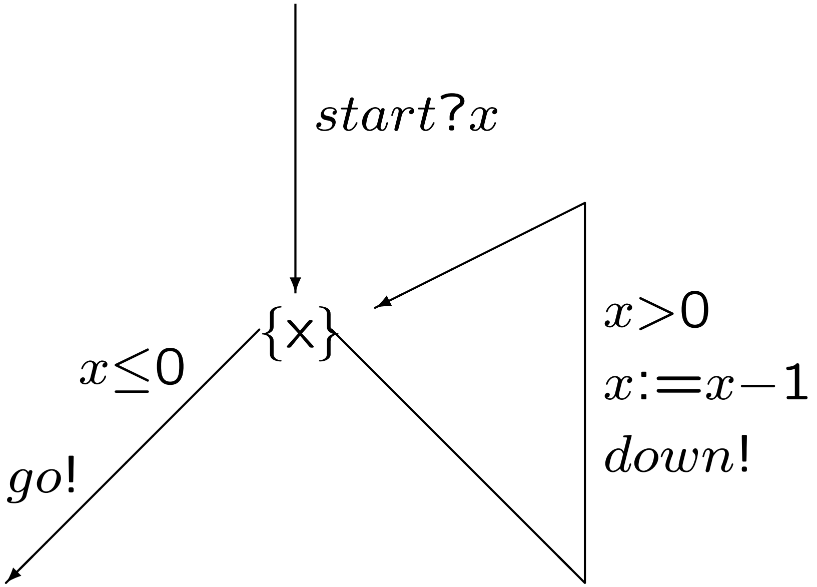
Counting machine



Counting will eventually finish: $[start?x]\mu Z.(\langle go!\rangle true \vee [down!]Z)$

Understanding fixpoints

Counting machine



Counting will eventually finish: $[start?x]\mu Z.(\langle go!\rangle true \vee [down!]Z)$

"μ for finite looping"

Intuitively

$$\mu Z.\phi = \phi^0 \vee \phi^1 \vee \phi^2 \vee \cdots$$

where $\phi^0 = false$, $\phi^{i+1} = \phi[\phi_i/Z]$

Understanding fixpoints

Can push a value: $\langle push?x\rangle true$

Understanding fixpoints

Can push a value: $\langle push?x\rangle true$

Can pop a value immediately after it is pushed: $[push?x]\langle pop!x\rangle true$

Understanding fixpoints

Can push a value: $\langle push?x\rangle true$

Can pop a value immediately after it is pushed: $[push?x]\langle pop!x\rangle true$

This behavior can repeat forever: $\nu Z.[push?x]\langle pop!x\rangle Z$

Understanding fixpoints

Can push a value: $\langle push?x\rangle true$

Can pop a value immediately after it is pushed: $[push?x]\langle pop!x\rangle true$

This behavior can repeat forever: $\nu Z.[push?x]\langle pop!x\rangle Z$

"ν for infinite looping"

Intuitively, $\nu Z.\phi = \phi^0 \wedge \phi^1 \wedge \phi^2 \wedge \cdots$ where $\phi^0 = true$, $\phi^{i+1} = \phi[\phi_i/Z]$.

Formally

$\mu X.\Lambda$: the least solution of the equation $X = \Lambda$

That is, $\mu X.\Lambda$ is the least fixpoint of Λ

Terminating computation: a feature of sequential programs

Formally

$\mu X.\Lambda$: the least solution of the equation $X = \Lambda$

That is, $\mu X.\Lambda$ is the least fixpoint of Λ

Terminating computation: a feature of sequential programs

$\nu X.\Lambda$: the greatest solution of the equation $X = \Lambda$

That is, $\nu X.\Lambda$ is the greatest fixpoint of Λ

Nonterminating computation: a feature of concurrent programs

(better notations: $\mu X.X = \Lambda$, $\nu X.X = \Lambda$)

Mixed fixpoints

It is always the case that whenever a value is received via channel r, it will eventually be sent along channel s:

$$\nu X.[r?x]\mu Y.(\langle - \rangle Y(x) \vee \langle s!x \rangle X)$$

Alternative nesting of ν:s and μ:s affects the complexity of model checking algorithms

Alternative nesting of ν:s and μ:s affects the complexity of model checking algorithms

Suppose ϕ has X and Y free

The alternation depth of $\nu X.\mu Y.\nu Z.\phi$ is 3

Alternative nesting of ν:s and μ:s affects the complexity of model checking algorithms

Suppose ϕ has X and Y free

The alternation depth of $\nu X.\mu Y.\nu Z.\phi$ is 3

The alternation depth of $\nu X.\mu Y.\mu Z.\phi$ is 2

Alternative nesting of ν:s and μ:s affects the complexity of model checking algorithms

Suppose ϕ has X and Y free

The alternation depth of $\nu X.\mu Y.\nu Z.\phi$ is 3

The alternation depth of $\nu X.\mu Y.\mu Z.\phi$ is 2

All known model checking algorithms for μ-calculus are exponential in the alternation depth of the formulas

A local model checking algorithm (for finite data domains)

Given a node $p \equiv (m, \rho')$ in a labeled transition system (induced from a STG) and a closed formula ϕ, check if $p \models \phi$

Each state s has five attributes:

$s.depth$	alternation depth of the formula
$s.status$	current status, `FRESH` or `VISITED`(b)
$s.\sigma$	true if the type of the formula is ν, false otherwise
$s.D$	set of states whose current values depend on s
$s.instack$	true if s is in the stack

The algorithm employs a “multi-stack” (an array of stacks indexed by alternation depths)

`pushStack`(s): push s onto `stack`$[s.depth]$
`top`$()$: return the element last pushed onto `stack`
`pop`$()$: remove from the stack the element last pushed onto `stack`

```
modelCheck$(s)$ = { $s.status$ := VISITED$(s.\sigma)$; push$(s)$;
                    while stack is non-empty do close(top());
                    return $s.status$ }
```

```
check$(s)$ = case $s.status$ of
      FRESH $\Longrightarrow$ {$s.status$ := VISITED$(s.\sigma)$; push$(s)$; return(DEFERRED, $s.D$)}
    | VISITED(b) $\Longrightarrow$ return(VALUE$(b)$, $s.D$)
```

```
close($s$ as $(p, \phi)$) =
let checkAnd($W$) =
    let $(B_i, D_i)$ = check($s_i$) for each $s_i \in W$
    in  if some $B_i$ = DEFERRED then return(DEFERRED)
        else if some $B_i$ = VALUE($false$) then {add $s$ to $D_i$; return(VALUE(*false*))}
        else {add $s$ to each $D_i$; return(VALUE(*true*))}
    $B$ = case $\phi$ of
          $be \Rightarrow \rho(be)$
        | $\phi_1 \wedge \phi_2 \Rightarrow$ checkAnd($\{(p, \phi_1), (p, \phi_2\}$)
        | $\phi_1 \vee \phi_2 \Rightarrow$ checkOr($\{(p, \phi_1), (p, \phi_2\}$)
        | $[c?x]\phi' \Rightarrow$ checkAnd($\{\, (p_i\{y \mapsto v\}, \phi'[v/x])) \mid p \overset{c?y}{\rightarrow} p_i, v \in \mathit{Val}\,\}$)
        | $\langle c?x \rangle \phi' \Rightarrow$ checkOr($\{\, (p_i\{y \mapsto v\}, \phi'[v/x])) \mid p \overset{c?y}{\rightarrow} p_i, v \in \mathit{Val}\,\}$)
        | ...
in  if $B$ = VALUE($b$) then
        {pop(); if $s.status$ = VISITED($b'$) and $b' \neq b$
                then {$s.status$ := VISITED($b$); restore($s.D, b$)}}
    else ...
```

```
$\texttt{restore}(D, b)$ = for each $s \in D$
                              if $s.status = \texttt{VISITED}(b')$ and $b' \neq b$ then
                                 { $s.status$ := $\texttt{VISITED}(s.\sigma)$;
                                   if $\neg s.instack$ then push($s$);
                                   $\texttt{restore}(s.D, b)$; $s.D$ := $\emptyset$ }
```

Input variables are instantiated "on-the-fly"

Example: Check $c?x.P$ against $\langle c?y \rangle \phi$, where $x, y \in \{1, 2, \ldots, \ 10\}$

Instantiating before running the algorithm would translate them to

$$c_1.P[1/x] + c_2.P[2/x] + \cdots + c_{10}.P[10/x] \qquad \text{and}$$
$$\langle c_1 \rangle \phi[1/x] \vee \langle c_2 \rangle \phi[2/x] \vee \cdots \vee \langle c_{10} \rangle \phi[10/x]$$

resulting in $10 \times 10 = 100$ comparisons during the execution time.

Input variables are instantiated “on-the-fly”

Example: Check $c?x.P$ against $\langle c?y\rangle\phi$, where $x, y \in \{1, 2, \ldots,\ 10\}$

Instantiating before running the algorithm would translate them to

$$c_1.P[1/x] + c_2.P[2/x] + \cdots + c_{10}.P[10/x] \qquad \text{and}$$
$$\langle c_1\rangle\phi[1/x] \vee \langle c_2\rangle\phi[2/x] \vee \cdots \vee \langle c_{10}\rangle\phi[10/x]$$

resulting in $10 \times 10 = 100$ comparisons during the execution time.

With the “on-the-fly” instantiation strategy, only 10 pairs need to be further checked:

$$(c_1.P[1/x], \langle c_1\rangle\phi[1/x]),\ \ldots,\ (c_{10}.P[10/x], \langle c_{10}\rangle\phi[10/x])$$

For safety properties, it is sufficient to use the *alternation-free* fragment of μ-calculus

Alternation-free:

in $\nu X. \cdots \mu Y.\phi$, X does not occur free in ϕ

in $\mu Y. \cdots \nu X.\phi$, Y does not occur free in ϕ

For alternation-free μ-calculus, the time complexity of the algorithm is linear in the size of the formula

Case study: German2004 cache coherence protocol

Distributed hardware architecture

Directory-based: client nodes request shared or exclusive access to a memory address line from the “home” node

Each node can act as either the home or a client, according to how the memory address is distributed

Properties to check

Abbreviation: Always(ϕ) $\equiv \nu X.\phi \wedge [-]X$

Cache consistency:

Always($\forall i, j \; i \neq j \Rightarrow (C_i$.state $=$ EXCLUSIVE $\Rightarrow C_j$.state $=$ INVALID))

Data consistency:

Always($\forall i$ Dir[i] $\neq$ EXCLUSIVE $\Rightarrow$ Mem $=$ latest)

Always($\forall i \; C_i$.state $\neq$ INVALID $\Rightarrow C_i$.datum $=$ latest)

The protocol had been shown to be correct by other groups

However, when we model check it against these properties, only cache consistency got through, data consistency didn't

Previous work either ignored data completely, or restricted the data domain size to 1

We set data domain size to 2, and the bug shows up!

By analyzing the counter example generated by our model checker, the problem was fixed

Question: what about data domain of size more than 2?

Data Independence

Data independent domains: May have infinite many elements, but are not subject to operations other than equality or inequality tests

Example: data transmitted in communication protocols

Data Independence

Data independent domains: May have infinite many elements, but are not subject to operations other than equality or inequality tests

Example: data transmitted in communication protocols

Can a data independent domain be collapsed to singleton?

No.

Consider

$$P\colon \xrightarrow{\;in?x\;} \xrightarrow{\;in?y\;} \xrightarrow{\;out!x\;} \xrightarrow{\;out!y\;}$$

P is data independent.

Does $P \models \langle in?x\rangle\langle in?y\rangle\langle out!y\rangle\langle out!x\rangle true$?

No.

Consider

$$P\colon \xrightarrow{in?x} \xrightarrow{in?y} \xrightarrow{out!x} \xrightarrow{out!y}$$

P is data independent.

Does $P \models \langle in?x\rangle\langle in?y\rangle\langle out!y\rangle\langle out!x\rangle true$?

Valid on any singleton domain, but not on domains with two or more elements

Algorithm: only need to modify the input modality part

Use schematic values, linearly ordered: $\{v_1,\ v_2,\ \ldots\}$

$\mathit{nextVal}(s)$: returns the least schematic value not in s

B = case operator(n) of

$$\begin{array}{ll} & \ldots \\ | & [c?x]\phi \Rightarrow \texttt{checkAnd}(\{(p_i\{y \mapsto v\}, \phi[v/x]) \mid \{n \rightarrow n', p \overset{c?y}{\rightarrow} p_i, v \in U\}) \end{array}$$

$$\text{where } U = \begin{cases} C \cup \mathit{nextVal}(p, [c?x]\phi) & \text{if } x \text{ is data independent} \\ \mathit{Val} & \text{otherwise} \end{cases}$$

We set the data domain type to be “data independent”, and verified the modified protocol again, both control consistency and data consistency got model checked

Other abstraction techniques used

Parameter abstraction

Bisimulation abstraction

...

Other case studies

Conference protocol

FLASH cache coherence protocol

More abstraction techniques used:

– Environment abstraction

– State clustering (domain specific)

– Parameter truncating (domain specific)

– ...

Summary

- Symbolic transition graphs as a model for concurrent programs
- A first-order μ-calculus, where fixpoint formulas are based on predicates, and its graphical representation
- An algorithm for model checking concurrent systems with data against properties expressed as formulas in the predicate μ-calculus
- Case studies showing data aspects should not be ignored in model checking

Future directions

- New abstraction techniques (domain-specific)
- Working directly on the code
  - Bounded model checking
  - Creating models from source code
  - Library function calls
- Combine model checking and testing
- From yes/no to “likehood”