Chapter 1

Partial-order reduction

Course “Model checking”
Martin Steffen
Autumn 2021

Section

Introduction

Chapter 1 "Partial-order reduction"
Course "Model checking"
Martin Steffen
Autumn 2021

State space explosion problem

- model checking in general "intractable"
- fundamental limitation: combinatorial explosion
- state space: exponential in problem size
  - in particular in *number of processes*

“Asynchronous” systems and interleaving

- remember: synchronous and asynchronous product (in connection with LTL model checking)
- asynchronous: software and asynchonous HW
- synchronous: often HW, global clock
- interleaving (of steps, actions, transitions . . .)

Where does the name come from?

- partial-order semantics
- what is *concurrent* (or parallel) execution?
- "causal" order
- *"true"* concurrency vs. *interleaving* semantics
- "math" fact: PO equivalent set of all linearizations
- "reality" fact: POR only "approximates" that math-fact
- perhaps better name for POR: "COR":

commutativity-based reduction

Exploiting "equivalences"

Instead if checking all "situations",

- figure which are equivalent (also wrt. to the property)
- check only one (or at least not all) representatives per equivalence class

(Labelled) transition systems

- basically unchanged,
  - assume initial states
  - states labelled with sets 2^P
  - state-labelling function V
  - transitions are labelled as well (from L), $\alpha, \beta \ldots$
- alternatively multiple transition relations: instead of $\xrightarrow{\alpha}$, we also see α as relation

$$(S, S_0, L, \rightarrow, V)$$

Determinism and enabledness

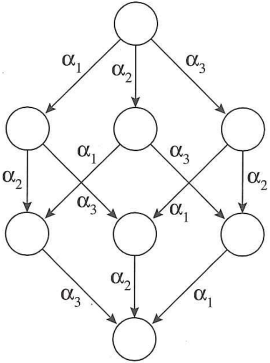
- remember: $\xrightarrow{\alpha}$ deterministic
- in that case: also write $s' = \alpha(s)$ for $s \xrightarrow{\alpha} s'$ (or $\alpha(s, s')$)

Enabledness

$\xrightarrow{\alpha}$ enabled in s, if $s \xrightarrow{\alpha}$

Concurrency in asynchronous systems

- independent transitions
- arbitrary orderings or linearizations (= interleavings)
- [actions themselves assumed atomic / indivisible]



- raw math calculation: n transition relations
  - $n!$ different orderings
  - 2^n states

Reducing the state space

- goal: pruning the state space

Super-unrealistic:

1. generate explititly the state space by DFS
2. then prune it (remove equivalent transitions & states)
3. then model check the property

Reducing the state space

- goal: pruning the state space

unrealistic (but for presentation reasons)

1. generate explictly the reduced state space (using modified DFS)
2. then model check the property

Modified DFS: ample set

- standard DFS: basically *recursion* (probably with explicit stack)
- exploration: explore "successor states", i.e.,

  follow all enabled transitions

- graph exploration (not tree): check for *revisits*

Modified DFS: ample set

- standard DFS: basically *recursion* (probably with explicit stack)
- exploration: explore "successor states", i.e.,

follow all enabled transitions

- graph exploration (not tree): check for *revisits*

Modification/improvement

Don't explore *all* enabled transitions.

follow enough enabled transition

Modified DFS

```
hash(s0);
set on_stack(s0);
expand_state(s0);

procedure expand_state(s);
  work_set(s) := ample(s);
  while work_set(s) ≠ ∅
  do
      let α ∈ work_set(s);
      work_set(s) := work_set(s) \{α}
      s' := α(s)
      if    is_new(s')
      then  hash(s')
            set on_stack(s');
            expand_state(s');
      end if
  end while
  set completed(s)
end procedure;
```

Ample sets

General requirements on *ample*

1. pruning with ample does not change the outcome of the MC run (correctness)
2. pruning should, however, cut out a *significant* amount
3. calculating the ample set: not too much *overhead*

With a little help of the programmer ...

- for instance: Spin
- Spin: early adoptor of POR
- reduce the amount of interleavings

atomic	**D_step**
atomic block executed indivisibly	deterministic code fragment executed indivisibly.

- `D_step` more strict than `atomic` (eg. wrt. `goto` statements)

Section

Independence and invisibility

Chapter 1 “Partial-order reduction”
Course “Model checking”
Martin Steffen
Autumn 2021

2 relations between relations

- we have labelled transitions (resp. multiple relations)
- 2 important conditions for POR
  - one connects *two relations*
  - one connects one relation with the property to verify

Independence

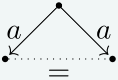
roughly: the order of 2 independent transitions does not matter.

Invisible

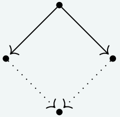
Taking a transition does not change the satisfaction of relevant formulas

Determinism, confluence, and commuting diamond property

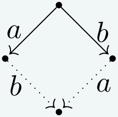
Determinism



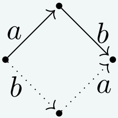
Diamond prop.



Comm. d-prop.



"Swapping" or commuting



and vice versa

Independence

- assume: transition relations $\xrightarrow{\alpha_i}$ *deterministic*
- write $\alpha_i(s)$ for $s \xrightarrow{\alpha_i}$

Definition (Independence)

An *independence relation* $I \subseteq \rightarrow \times \rightarrow$ is a symmetric, antireflexive relation such that the following holds, for all states $s \in S$ and all $(\xrightarrow{\alpha_1}, \xrightarrow{\alpha_2}) \in I$

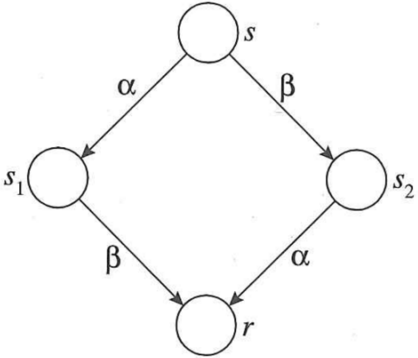
Enabledness If $\alpha_1, \alpha_2 \in enabled(s)$, then $\alpha_1 \in enabled(\alpha_2(s))$

Commutativity: if $\alpha_1, \alpha_2 \in enabled(s)$, then

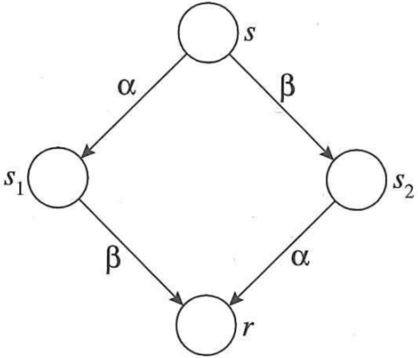
$$\alpha_1(\alpha_2(s)) = \alpha_2(\alpha_1(s))$$

- *dependence relation*: $D = (\rightarrow \times \rightarrow) \setminus I$

Is that all?



Is that all?



2 issues

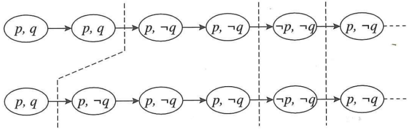
1. The checked property might be sensitive to the choice between s_1 and s_2 (and not just depend on s and r
2. s_1 and s_2 may have other successors not shown in the diagram.

Visibility

- $V : S \to 2^P$
- $\xrightarrow{\alpha}$ is invisible wrt. to a set of $P' \subseteq P$ if for all $s_1 \xrightarrow{\alpha} s_2$ and all $p' \in P'$

$$s_1 \models p \quad \text{iff} \quad s_2 \models p$$

Blocks and stuttering



stuttering equivalent paths

- block: finite sequence of intentically labelled states
- stuttering (in this form): important for *asynchronous* systems

Stutter invariance

An LTL formula φ is *invariant under stuttering* iff for all pairs of paths π_1 and π_2 with $\pi_1 \sim_{st} \pi_2$,

$$\pi_1 \models \varphi \quad \text{iff} \quad \pi_2 \models \varphi$$

Next-free LTL

- $\bigcirc$ breaks stutter invariance
- $LTL_{-\bigcirc}$: "next-free" fragment of LTL (often also LTL_{-X})

Theorem (Stuttering)

- *Any $LTL_{-\bigcirc}$ property is invariant under stuttering*
- *Any LTL property which is invariant under stuttering is expressible in $LTL_{-\bigcirc}$*

Section

POR for $LTL_{-\bigcirc}$

Chapter 1 “Partial-order reduction”
Course “Model checking”
Martin Steffen
Autumn 2021

POR for LTL$_{-\bigcirc}$

- general useful and fuitful setting for POR
- of course: one may look more specific for specific formulas
- in that setting:

Correctness of POR

Ample sets prune the (DFS) search. Goal:

$$T, s \models \varphi \qquad \text{iff} \qquad T^{\text{✂}}, s \models \varphi$$

Path representatives

each path π_1 in T starting in s is represented by an equivalent path π_2 in $T^{\text{✂}}$, starting in s

Conditions on selecting ample sets

4 conditions for selecting ample set

- each pruned path can be “reordered” to an which is explored (using independence). Includes a condition covering end-states
- make sure that the reordering (pre-poning) does not change the logical status (stutting, visibility)
- “fairness”: make use not to prune “relevant” transitions by letting the search cycle in irrelevant ones.

Reordering conditions (C_0, C_1)

C_0: stop at a dead end, only

$$ample(s) = \emptyset \text{ iff } enabled(s) = \emptyset$$

C_1

Along every path in T starting at s, the following condition holds: a transition dependent on a transition in $ample(s)$ cannot be executed without a transition from $ample(s)$ occuring *first*.

Form of paths

- consequence of **C**$_1$: two forms of paths

with prefix $\beta_0\beta_1 \ldots \beta_m\alpha$

- $\alpha \in ample(s)$
- $\beta_i \bowtie ample(s)$

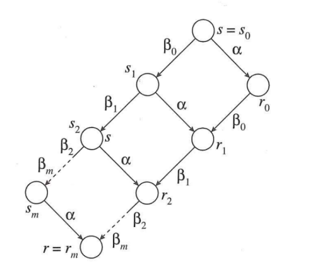
without such prefix:

- infinite $\beta_0\beta_1\beta_2 \ldots$
- $\beta_i \bowtie ample(s)$

Commutation

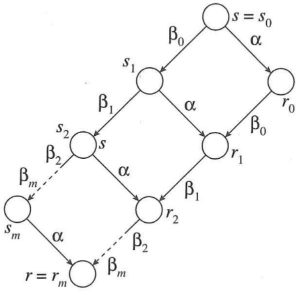
path $\vec{\beta}\alpha$ in T, starting in s

- $\alpha \in ample(s)$, $\beta_i \notin ample(s)$



- $\pi_1 = \vec{\beta}\alpha$
- $\pi_2 = \alpha\vec{\beta}$

Does it make a difference **how** to go from s to r?



- π_1 and π_2 (and intermediate mixures): "interchangable"
- start and end point equal
- but: does it matter which one is taken
  - wrt. the logical property, i.e.,
  - does it matter which intermediate states are visited?

$$s_i \xrightarrow{\alpha} r_i$$

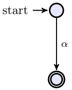
Invisibility of transitions

- remember: invisibility of transitions (by sets of atomic propositions)

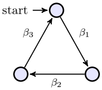
C_2 (invisibility)

If s is not fully expanded, then every $\alpha \in ample(s)$ is *invisible*.

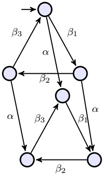
Is that all?



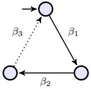
start
α



start
β_3
β_1
β_2



β_3
β_1
α
β_2
α
β_3
β_1
α
β_2



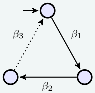
β_3
β_1
β_2

Is that all?

Is that all?

Is that all?

$T^{\text{✂}}$

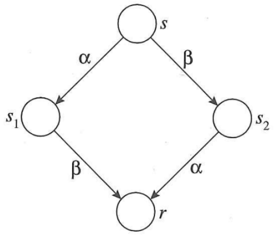


Cycle condition C_3

C_3

A cycle is not allowed if it contains a state in which some transition α is enabled but never included in $ample(s)$ for any state s on the cycle.

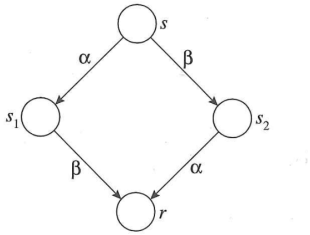
Remember the 2 issues



1. satisfaction depends in chosing path via s_1 or s_2?
2. forgotten successors?

- assume: s_1 is omitted ($\beta \in ample(s)$, but not α)

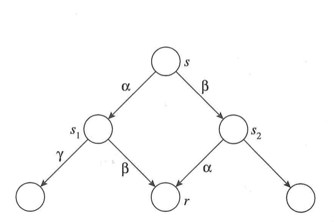
Remember the 2 issues



1. satisfaction depends in chosing path via s_1 or s_2?
2. forgotten successors?

- assume: s_1 is omitted ($\beta \in ample(s)$, but not α)

issue 2



the conditions imply

1. $ss_2r \sim_{st} ss_1r$
2. $ss_1s_1' \sim_{st} ss_2rr'$

Complexity

- checking conditions on-the-fly
- $\mathbf{C}_0$: easy
- $\mathbf{C}_1$: tricky
  - refers to T, not $T^{✂}$
  - checking $\mathbf{C}_1$: equivalent to reachability checking
- strengthen $\mathbf{C}_3$:

sufficient for $\mathbf{C}_3$

- at least one state along each cycle must be fully expanded

- since we do DFS: watch out for "back edges": $\mathbf{C}_3'$: If s is not fully expanded, then no transition in $ample(s)$ may reach a state that is on the search *stack*

General remarks on heuristics

- dependence and independence $\bowtie$ "theoretical" relation between (deterministic) relations
- "use case": capturing steps of concurrent programs
  - processes with program counter (control points)
  - different ways of
    - synchronization
    - sharing memory
    - communication
- calculating (approx. of) ample sets: dependent on the programming model

Notions, notations, definitions

- we write now α for $\xrightarrow{\alpha}$
- fixed, finite set of procecesses i (called P_i)
- T_i: those transitions that "belong to" P_i
- some more easy definitions
  - $pc_i(s)$: value of program counter of i in state s
  - $pre(\alpha)$:
    - transition whose execution *may* enable α
    - can be over-approximative
  - $dep(\alpha)$: transitions interdependent with α
  - $current_i(s)$
  - $T_i(s)$

When are transitions (inter)dependent

- note: dependence is *symmtetric*! (good terminology?)

Shared variables

pairs of transitions, that *share* a variables which is changed (or written?) by at least one of them

Same process

pairs of transitions belonging to the *same process* are interdependent. In particular $current_i(s)$

Message passing

- 2 sends to the same channel or message queue
- 2 receives from the same channel
- Note send and receive indepenent (also on the same channel).
- side remark: rendezvouz is seen/ can be seen a joint step of 2 processes

Transitions that may enable α ($pre\alpha$)

$$pre(\alpha) \supseteq \{\beta \mid \alpha \notin enabled(s), \beta \in enabled(s), \alpha \in enabled(\beta(s))\}$$

- assume α is an action from P_i
- $pre(\alpha)$ includes
  - "local predecessor" of i ("program order")
  - shared variables: if enabling conditions of α involves shared variables: the set contains *all other transitions* that can change these shared variables
  - message passing: if α is a send (reps. receive), the $pre(\alpha)$ contains transitions of other processes that receive (resp. send) on the channel

Ample

```
function ample (s) =
 for all P_i such that T_i(s) ≠ ∅ // try to focus on one P_i
   if
        check_C1(s, P_1) ∧
        check_C2(T_i(s)) ∧
        check_C3'(s, T_i(s))
   then
         return T_i(s)
   if
 end for all      // too bad, cannot focus on any but
 return enabled(s) // fully expanded can't be wrong
end
```

Check C_2

```
function check_C2(X) =
  for all α ∈ X
  do  if   visible(α)
      then false
      else true
```

Check C_3'

```
function check_C3' (s, X) =
  for all α ∈ X
   do
      if    on_stack(α(s))
      then  false
      else  true
```

Check C_1

```
function check_C1 (s, P_i) =
  for all P_j ≠ P_i
   do
       if          dep(T_i(s)) ∩ T_j ≠ ∅
          ∨
                   pre(current_i(s) \ T_i(s)) ∩ T_j ≠ ∅
       then return false
   end forall;
  return true
```

References I

Bibliography

[1] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. (1992). Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170.

[2] Clarke, E. M. and Emerson, E. A. (1982). Design and synthesis of synchronisation skeletons using branching time temporal logic specifications. In Kozen, D., editor, *Proceedings of the Workshop on Logic of Programs 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 244–263. Springer Verlag.

[3] Queille, J. P. and Sifakis, J. (1982). Specification and verification of concurrent systems in CESAR. In Dezani-Ciancaglini, M. and Montanari, U., editors, *Proceedings of the 5th International Symposium on Programming 1981*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Verlag.